UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS

SCHOOL OF MATHEMATICS

MultiCrossover Genetic Algorithms for Combinatorial Optimisation Problems

by

Lai Soon LEE

Thesis for the degree of
Doctor of Philosophy in Operational Research

April 2006

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF MATHEMATICS

Doctor of Philosophy

MULTICROSSOVER GENETIC ALGORITHMS FOR COMBINATORIAL OPTIMISATION PROBLEMS

by Lai Soon LEE

The usual strategy within a genetic algorithm (GA) is to generate a pair of offspring during crossover. We hypothesise that generating multiple offspring during the crossover can improve the performance of a GA. This thesis reports on the development and evaluation of a new strain of GA, called the MultiCrossover Genetic Algorithms (MXGAs) for solving combinatorial optimisation problems (COPs) to investigate this hypothesis. The MXGA utilises a multicrossover operator that uses a simple yet effective standard crossover strategy to generate offspring. The proposed multicrossover first generates a candidate list of temporary offspring from a pair of selected parents through repeated applications of the proposed crossover strategy. Two distinct temporary offspring are generated each time the strategy is executed. The best and a selected temporary offspring are then chosen to be the offspring for the current generation. Various techniques are also introduced into the MXGA to further enhance the solution quality.

In this thesis, MXGAs are applied to three specific variants of COPs: single machine family scheduling problem, non-oriented two-dimensional rectangular single bin size bin packing problem with due dates, and symmetric travelling salesman problem with due dates. These problems are motivated by the dilemma faced by the manufacturing organisations which involves the trade-off between the manufacturer's efficiency and customers' satisfaction. The common characteristic of the problems studied is the inclusion of the customers' due dates. Schemes for obtaining a lower bound on the maximum lateness for the problems studied are also introduced. Extensive computational experiments are carried out to assess the effectiveness of the MXGAs compared to other local search methods such as tabu search, steepest descent and a standard genetic algorithm.

Contents

List of Figures

List of Tables

DECLARATION OF AUTHORSHIP

I, Lai Soon LEE, declare that the thesis entitled MultiCrossover Genetic Algorithms for Combinatorial Optimisation Problems and the work presented in it are my own. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- none of this work has been published before submission.

Signed:  .....

Date:.....24 April 2006.....

Acknowledgements

THANK YOU . . . There are many people that I am indebted to for making this thesis possible. I start by saying a big thank you, "TERIMA KASIH", "XIE XIE", "MAHALO", "GRACIAS", "MERCI", and "DANKE" to my supervisor, Professor Chris Potts and my advisor, Dr Julia Bennell for their guidance, motivation (mostly in the form of conferences in "*exotic*" places), concern, patience (especially when reading my thesis) and freedom to conduct the research. Special mention to Julia and Howard for providing a roof over Yvonne and my head for the crucial months *(for the record, we loved the flat!)*.

I would like to express my gratitude to my employer Universiti Putra Malaysia (UPM) and Ministry Of Science, Technology and Innovation (MOSTI) for supporting my Ph.D. studies and finance over the past three and a half years.

My special thanks and warm wishes to the clerical staff on level five for the day-to-day support, assistance and attention throughout my time at the University of Southampton.

To Jonathan and Christine, thanks for the sound advice with regards to the thesis and LaTeX. My dear "*editors*": Honora, Andrew and Christine, what do I do without you! My officemates: Naomi, Jenni, and Ebert who have made my time at the office memorable with laughter, concern and jokes. I will cherish these times very much. To Edgar, Alex and the "*Belgian Panda*" (Christophe), you all have made the stay at Southampton very pleasant and memorable! I also would like to mention my badminton team: Lelio, Bis, Vanessa, Hong, Eric and Stephen for many enjoyable hours of games that help me control the everyday pressures of Ph.D. life.

I would like to express my love to my family and loved ones for the support and motivation they have given to me throughout my time at the University of Southampton, especially for taking upon themselves many of my duties whilst I was here. Thanks to all my friends at home, who were always there for a laugh, and did not forget me despite the distance.

These acknowledgements would not be completed without a big *THANKS* to Yvonne. You know how important you are for me. You were always be my side, and without your love and support, this work would have never been finished.

Chapter 1

Introduction

1.1 Background

Combinatorial Optimisation Problems (COPs) appear in many diverse areas such as resource allocation, scheduling, cutting and packing, sequencing, and routing. The objective is that of assigning value to a set of decision variables such that a function of these variables is *optimised* (minimised or maximised), perhaps in the presence of some constraints.

A class of problems of particular interest in COPs is that of the '*hard*' problems. It is often easy to find a feasible solution to these type of problems. However, it is usually quite difficult to find a good solution as the solution space for all the feasible solutions is very large. This class includes problems famous for their difficulty such as the Machine Scheduling Problems (MSPs), Cutting and Packing (C&P) Problems , and the Travelling Salesman Problem (TSP).

COPs occur in many more areas of our lives than we might initially expect. For instance, scheduling course assignments to be completed based on their deadlines, placing clothes into a suitcase or shopping for daily groceries in a huge market. In this research, we focus on the business and industrial applications of the problems. We study specific problem variants of the three different areas of COPs mentioned above. Specifically these are: single machine family scheduling problem, non-

oriented two-dimensional rectangular single bin size bin packing problem with due dates, and symmetric travelling salesman problem with due dates.

The study of MSPs dates back to 1950s (e.g. Jackson [156] and Smith [256]). In general, a MSP is concerned with the allocation of scarce resources to activities with the objective of optimising one or more performance measures. Resources may be machines in an assembly plant, runways at an airport, nurses in a hospital, etc. Activities may be various operations in a manufacturing process, landings and take-offs at an airport, duties of nurses in a hospital, etc. There are also many different performance measures to optimise. One objective might be the minimisation of the maximum lateness, while another objective may be minimisation of the mean completion time. As in the examples given above, real world applications of the MSPs arise in the process industry, airline industry, hospital, etc. For some latest surveys on the real world applications, see the collection of Leung [185].

Although C&P has been studied since the mid-fifties, Gilmore and Gomory's articles in the 1960s ([112, 113, 114]) are the first to present techniques which could be practically applied to medium size real-world problems. C&P is concerned with finding a good arrangement of multiple small items in one or more larger objects. The usual objective of the allocation process is aimed at maximising the utilisation of the larger objects (and therefore minimising the wastage), or maximising the value of the small items packed. The C&P problems are encountered in many real-world applications such as wood, glass, metal and textile industries, newspaper paging and cargo loading. High material utilisation is of particular interest to industries with mass-production, since small improvements in utilisation can result in large savings of material and considerable reduction of the production cost.

The general form of the TSP was first stated by Karl Menger in 1930s, but it is not until 1954 when the first mathematical formulation for the TSP appears courtesy of Dantzig *et al.* [63]. A TSP specifies a number of cities and the distance between any pair of cities. The objective is to find the shortest round trip visiting each city exactly once. Although transportation applications are the most natural setting for the TSP, the simplicity of the model has led to many interesting

applications in other areas. Example include, computer wiring on circuit boards (Lenstra and Rinnooy Kan [183]), X-ray crystallography (Bland and Shallcross [33]), and hole drilling on metal sheets (Reinelt [241]).

From the theoretical point of view, it is possible to calculate the value of all the feasible solutions and select the best for all these problems. This strategy is best known as *complete enumeration.* However, from a practical point of view, sometimes it is impossible to follow such a strategy especially with large problems. In many cases, the number of feasible solutions grows exponentially as the problem size increases. For instance, a symmetric version of the TSP with 25 cities will contain over 3×10^{23} feasible solutions. As the problem size increases, the number of feasible solutions will increase exponentially. For a problem with 50 cities, there exist over 3×10^{62} feasible solutions! Clearly for problems over a certain size, it is impossible to follow a strategy of complete enumeration. It is in this situation that methods known as heuristics are used.

Heuristics seek good feasible solutions to COPs in circumstances where the complexity of the problem or the limited time available for its solution do not allow complete enumeration. A heuristic is a technique which seeks good solutions at a reasonable computational cost. However, a heuristic is not guaranteed to find the best solution. In fact, many heuristics give no guarantee on solution quality. Moreover, heuristics are often domain specific in that they are designed to deal with the needs of a specific problem.

To deal with these shortcomings, there has been increasing interest in techniques that have a more generic structure. In particular, local search methods and metaheuristics have become widely used. A local search technique can be summarized as an iterative search procedure. It starts from an initial feasible solution from the search space and then improves it by applying a series of local modifications until a local optimum is found. Metaheuristics provide a way of considerably improving the performance of simple heuristic procedures. The search strategies proposed by metaheuristic methodologies result in iterative procedures with the ability to search the solution space effectively.

1.2 Motivations, Objectives and Aims

In the last two decades, we have seen dramatic changes of the conditions under which manufacturing organisations have to operate and the objectives they have to meet. Next to efficiency, quality and delivery reliability have become key performance criteria. In particular, the ability to cut manufacturing lead times and to meet tight due dates determines a company's competitive position. For instance, one of the most common scheduling problems in batch production involves the trade-off between the machine efficiency and meeting customers' due dates. On one hand, scheduling large batches means that relatively little time is spent in set up (e.g. obtaining and returning tools, inspecting material, time for cooling, etc.) and by doing so, machine efficiency is high. However, long runs on a given batch of jobs, may mean that due dates for other jobs are missed. On the other hand, scheduling jobs based on priority of customers' due dates result in shorter runs which also mean a large amount of setup time is incurred. As a result, capacity may become inadequate to meet the demand on time. This problem becomes particularly difficult when the setup time between jobs from a different batch is significant.

The problems studied in this thesis are mainly motivated by the dilemma faced by the manufacturing organisations as mentioned above which involves the trade-off between the manufacturer's efficiency and customers' satisfaction. An efficient way of dealing with the problem needs to be developed to achieve a balance between these performance measures. With this in mind, we investigate three specific variant of hard problems mentioned in the previous section where the common interest between the problems to be solved is the inclusion of the customers' due dates. We solve the problems using some well-known local search methods with a particular focus on Genetic Algorithms (GAs).

One of the main objectives of this research is to develop a general framework for our proposed MultiCrossover Genetic Algorithms (MXGAs) for solving the problems. The proposed MXGA utilises a multicrossover operator that uses a

simple yet effective standard crossover strategy to generate offspring. Every time the proposed crossover strategy is executed, two temporary offspring are generated from the selected parents. The main feature of the multicrossover is that it first generates a candidate list of valid temporary offspring from a pair of selected parents through repeated applications of the proposed crossover strategy. Then, the best and a selected temporary offspring (using the probabilistic binary tournament selection mechanism) are chosen to be the offspring for the current generation. Various techniques will also be introduced into the proposed MXGA to further enhance the solution quality when compared with other local search methods. Detailed descriptions of the framework will be given in Section 4.9.

For the remainder of this section, we describe briefly the problems to be solved in this thesis. Detailed descriptions of the problems will be given in Chapter 5–7.

The Single Machine Family Scheduling Problem (SMFSP) is a scheduling problem in which a set of jobs that are partitioned into groups, called families, and processed by a single machine. Each job has a processing time on the machine and a due date by which it should ideally be completed. A setup time is required at the start of the schedule and also when the next job is from a different family. If a job is not completed on time, a cost is associated for each time period it is late. The objective is to minimise the maximum cost caused by late jobs.

The classical Two-Dimensional Bin Packing Problem (2DBPP) refers to the problem of packing a set of small two-dimensional items into one or more larger objects (i.e. bins). In this study, the small items are constrained to be rectangular and may be rotated by 90°. The bins are also rectangular and have fixed dimensions. We refer to this problem type as the non-oriented Two-Dimensional Rectangular Single Bin Size Bin Packing Problem (2DRSBSBPP). Each rectangle is placed into a bin without creating overlapping between the rectangles that have already been packed in the bin or overflowing the bin. The objective is to minimise the number of bins used to pack all the rectangles.

We consider a logical extension to the problem where each rectangle has a due date and each bin has a fixed processing time. This extension has practical applications in the wood and metal industries. In the metal industry for instance, suppose that the bins used in the problem are the metal sheets with fixed dimensions, and the rectangles placed in a bin are the rectangular shapes to be cut from a metal sheet. Each metal sheet requires a fixed processing time on a cutting machine to cut all the shapes. As each rectangular shape has a due date, metal sheets which contain shapes with small due dates are ideally cut earlier. On the other hand, by mixing the shapes with different due dates might increased the packing efficiency if the shapes with different due dates can be use to fill in the gaps between the shapes on the metal sheets. However, this approach may result in missing the due dates of the shape with small due dates. If a shape is not completed on time, a cost is associated for each time period it is late. The objective is to minimise the maximum cost caused by the lateness and the number of bins used.

The classical TSP specifies a number of cities and the distance between any pair of cities. The objective is to find the shortest round trip visiting each city exactly once. The TSP is *symmetric* if the distance between two cities is the same in both directions; otherwise it is *asymmetric*. We study an extension to the TSP where each city has a due date by which it should ideally be visited. This extension has important practical applications in bank or postal deliveries, school bus routing, etc. If a city is not visited on time, a cost is associated for each time period it is late. The objective is to minimise the maximum cost caused by the late visit and the shortest round trip of visiting each city exactly once.

1.3 Overview of the Thesis

The remainder of the thesis is organised as follows. Chapter 2 gives a brief introduction to COPs and the techniques, both exact and heuristic, that can be applied to solve them. We also describe some of the widely used local search methods in their basic form in Chapter 2. In Chapter 3, we introduce the MSPs, C&P and

TSP in more details. Some basic concepts of bicriteria objective function are introduced. We present a typology of MSP and a lower bounding scheme for a SMFSP with family setup times. Typologies of C&P are also given in Chapter 3. We discuss some of the well-known approaches, both exact and heuristic, that can be used to solve the 2DRSBSBPP. We also present the lower bounds for 2DRSBSBPP in both oriented and non-oriented cases. The remainder of Chapter 3 concentrates on reviewing some of the well-known heuristic and exact approaches used in solving the symmetric and asymmetric version of TSP. Chapter 4 gives detailed descriptions of the main components in a Standard GA (SGA) and the proposed MXGA. We address each main component of the SGA by giving brief summaries for the approaches used in each component. The general framework of the MXGA is then discussed in detail in the remainder of the chapter.

The next three chapters give accounts of MXGAs applied to the problems studied. The proposed MXGA for each problem is based on the general framework suggested in Chapter 4. The performance of the MXGAs is experimentally evaluated on standard and benchmark instances of these problems. Extensive computational comparisons are also conducted using some of the well-known local search algorithms such as tabu search (TS), steepest descent method (SDM) and SGA. For each problem, a substantial amount of effort has also been put into the developments of the TS and SDM to further improve the solution quality.

Chapter 5 tackles the SMFSP with the objective of minimising the maximum lateness of the jobs with the presence of the family setup times. We give a general introduction to the problem and review some approaches used for solving them. To the best of our knowledge, no research has been carried out on the application of the genetic algorithm for this specific problem type. Some variations of the MXGA, TS and SDM are investigated, and all experimental results are presented.

Chapter 6 concentrates on the non-oriented 2DRSBSBPP. We develop a new heuristic placement routine, called Lowest Gap Fill (LGF), that is effective in filling the gaps in a partial layout by dynamically selecting the best rectangle for placement. We compare the LGF placement routine with some well known

heuristics reported in the literature. A new variant of the 2DRSBSBPP, called 2DRSBSBPP with due date is introduced, where each rectangle has a due date and each bin has a fixed processing time. The objective is minimising the maximum lateness of the rectangles by packing them, without overlapping, and minimising the number of bins. We also derive a lower bounding scheme for the maximum lateness of the problem.

In Chapter 7, we study a new variant of the symmetric version of the time-constrained TSP, called the Symmetric TSP with due dates (STSPDD), where each city has a due date. The objective is to minimise the maximum lateness and the total tour length of the cities to be visited. We give a brief introduction to the time-constrained TSP and review some approaches used for the TSP with time windows. A lower bounding scheme for the maximum lateness of the STSPDD is derived.

The thesis concludes with Chapter 8 where the work on MXGAs are summarised and comments are given on some possible extensions for future work.

Chapter 2

Search and Optimisation Techniques

2.1 Introduction

In this chapter, we present some of the well-known search and optimisation techniques used for solving combinatorial optimisation problems. A *combinatorial optimisation problem* can be described as the search for a feasible solution with the best objective function value from a finite set of feasible solutions that optimises (minimises or maximises) a given objective function. The best objective function value is the smallest objective function value for a minimisation problem and the largest for a maximisation problem.

Search and optimisation techniques are too wide to cover in one chapter and are beyond the scope of this thesis. Therefore we only concentrate on some of the well-known techniques in their basic form from the literature. In Section 2.2, we introduce the concept of complexity theory. Section 2.3 gives brief overviews on two well-known exact approaches used for solving combinatorial optimisation problems: branch and bound and dynamic programming. To end this chapter, we introduce the ideas of heuristics and metaheuristics in Section 2.4, and present some of the widely used local search methods in their basic form in Section 2.5.

2.2 Complexity Theory

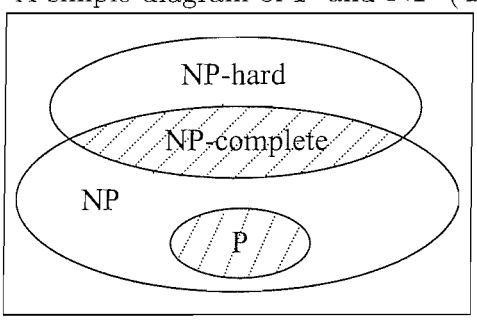
Complexity theory is part of the theory of computation dealing with the resources required during computation to solve a given problem. The most common resources are *time* (how many steps it takes to solve a problem) and *space* (how much memory it takes). In this section, we concentrate on the time complexity theory. The definitions, as well as most of the theory presented in this section, are extracted from Tovey [265] and Whitley and Watson [279]. Detailed descriptions can be found in Garey and Johnson [108], Papadimitriou [226] and Sipser [254].

The time complexity of a problem is the number of steps that it takes to solve an instance of the problem as a function of the size of the input length, using the most efficient algorithm. More formally, we use the *Big-O* notation: '$O(p\ (input\ length))$', where p is a function of the input length. For example, consider an instance of size n which can be solved in n^2 steps using an algorithm. We say the algorithm requires $O(n^2)$ time. Note that this function expresses the worst-case scenario of the problem at sufficiently large sizes. Suppose an algorithm solves a problem of size n in at most $2n^3 + 9n^2 + 99$ steps. For such function, we are concerned in the rate of growth as n increases exponentially. Therefore, the difference between $2n^3$ and n^3 are not important. We can also discard the lower order terms, as at large sizes it is the highest degree that determines the rate of growth. Thus, we say this algorithm requires $O(n^3)$ time.

The idea of complexity theory is that of classifying problems into two main classes, namely **P** and **NP**. The problem class **P** is the set of problems that can be solved by a deterministic Turing machine in polynomial time. A deterministic Turing machine is a mathematical model of an algorithm. This class corresponds to the problems which can be effectively solved in the worst case. The problem class **NP** is the set of problems that can be solved by a non-deterministic Turing machine in polynomial time. This class contains problems that people would like to be able to solve effectively such as the Boolean Satisfiability Problem and Travelling Salesman Problem (TSP).

It is clear that $\mathbf{P} \subseteq \mathbf{NP}$, and $\mathbf{P} \neq \mathbf{NP}$ is a widely believed conjecture although no proof has been established to date. Further research has gained insight into the class **NP** by dividing the class into subclasses. **NP-complete** class is a subclass of **NP** which has a property that all **NP** problems can be reduced to the **NP-complete** problem in polynomial time. In other words, a *decision problem* (i.e. problem where the answer is either 'YES' or 'NO') is called **NP-complete** if it is polynomially equivalent to the *satisfiability* problem, which is proved by Stephen Cook in 1971 to be **NP-complete**. More formally, a problem R is **NP-complete** if: (1) $R \in \mathbf{NP}$ and (2) R is **NP-hard**. The term **NP-hard** is used to describe the corresponding optimisation problem of a **NP-complete** decision problem. The significance of the class **NP-complete** is explained below. If problem A can be reduced to problem B in polynomial time and a polynomial algorithm for solving problem B is found, then problem A is also solved in polynomial time. This mean that if a polynomial time algorithm is found for any **NP-complete** problem then all problems in the class **NP** can be solved in polynomial time, and therefore $\mathbf{P} = \mathbf{NP}$. If, as believed, $\mathbf{P} \neq \mathbf{NP}$, then it has been shown that there must exists problems that are neither in **P** nor **NP-complete** (see Figure 2.1).

Figure 2.1: A simple diagram of **P** and **NP** (Tovey [265])



The **NP**-hardness of a problem suggests that it is impossible to find an optimal solution without the use of an essentially enumerative algorithm, for which computation times will increase exponentially with problem size. For this reason, heuristic methods have been developed to obtain good solutions for large problems in a reasonable amount of time. There is clearly a tradeoff between the computational investment in obtaining a solution and the quality of that solution.

2.3 Exact Approaches

We introduce the basics of branch and bound and dynamic programming with the help of Denardo [70] and Dowsland [77].

2.3.1 Branch and Bound

The origins of the Branch and Bound (B&B) idea go back to the work of Dantzig *et al.* [63] on the TSP in 1954. Four years later, Eastman [85] developed the first B&B algorithm based on a subtour elimination scheme. However, the term '*branch and bound*' itself was coined by Little *et al.* [191] in conjunction with their TSP algorithm in 1963. Many such procedures have since been proposed.

The main idea of a B&B approach is to partition the feasible solutions into disjoint sets, each belonging to a *branch* of a tree, and then *bound* the cost of each set in order to restrict the search to an optimal solution. The rationale behind the B&B is to reduce the size of the feasible solutions that need to be considered by repeatedly partitioning the problem into a set of smaller subproblems. Suppose we are dealing with a minimisation problem. By calculating a lower bound on the objective function values in a set, and if it is equal or worse than the best objective function value found so far, the optimal solution of the problem cannot lie in the subset and the subset is referred to as *fathomed*. No further work needs to be done on a fathomed subset.

The search of subset can be represented as a tree. The set of all feasible solutions is represented by the first node (i.e. *root* of the tree). The disjoint subsets are also represented as nodes, joined by edges to the first node. At each node, a lower bound is calculated and a decision is made either to partition the node further forming new nodes or to fathom the node. The sets of solutions represented by the nodes become smaller at each successive level of the tree. At the final level, each node represents a single solution. The algorithm terminates when all nodes are fathomed and the node with the best upper bound gives the optimal solution.

The efficiency of a B&B approach relies on the quality of the bounds and the search strategy used. It is usually worth deriving bounds that are as tight as possible. In the case of lower bounds, this is often achieved by exploiting as much information about the problem as possible. The tighter the lower bound at a node, the greater the chance of the node being fathomed. Sometime, an upper bound is also used in conjunction with the best objective function value found so far to prune the tree. In some cases, a problem may become easy to solve if some of the constraints are removed. This is a process known as *relaxation* and the solution to the relaxed problem often provides a valid bound to the solution of the original problem. Two common search strategies used in B&B in solving the combinatorial optimisation problem are known as *depth-first search* and *breadth-first search*. Detailed descriptions of the branching schemes can be found in Dowsland [77].

2.3.2 Dynamic Programming

Dynamic Programming (DP) was first introduced by Richard Bellman in 1953. The essence of DP is Bellman's *Principle of Optimality* [28] which states that:

> "*An optimal policy has the property that whatever the initial state and the initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*"

Similar to B&B algorithm, DP is a procedure that solves combinatorial optimisation problems by breaking down into subproblems. The method hinges on the ability to break the problem down into stages, which enable the subproblems in each stage to be solved by an efficient recursive algorithm from the solutions in the previous stage.

Any DP implementation has four main ingredients: *stages, state, decisions* and *policies*. At each stage, for each feasible state, a decision is make on how to achieve the next stage. The decisions are then combined into subpolicies that are themselves combined into an overall optimal policy. Stages are often referred to

the time periods from the start or end of the planning horizon, or in terms of expanding subsets of variables that may be included at each stage. States are commonly defined as the amount of produce in stock or yet to be produced, the size or capacity of an entity such as a stock sheet, container, or the destination already reached in a TSP. Depending on the level of complexity, DPs have been classified into four categories: deterministic, stochastic, adaptive and residual.

2.4 Heuristics and Metaheuristics

The basic concept of heuristic was first introduced by Polya [231] in 1945. This term is derived from the Greek word '*heuriskein*' meaning to find or discover. Reeves and Beasley [240] give the following definition:

> "*A heuristic is a technique which seeks good (i.e. near optimal) solutions at a reasonable computational cost without being able to guarantee either feasibility or optimality or even in many cases to state how close to optimality a particular feasible solution is.*"

Heuristic procedures can be divided into four basic strategies based on the classification given by Foulds [103]. Many heuristics comprise a combination of more than one of these strategies:

- *construction strategy* : begins with a partial solution and successively add new elements which are likely to be valuable parts for the final solution. This strategy is useful when it is relatively difficult to generate feasible solutions to the problem.
- *improvement strategy* : begins with a sub-optimal solution and progressively seek improvement via a series of modifications. This strategy is useful when it is relatively easy to generate starting solutions.
- *component analysis strategy* : divides the problem into component parts. Each component part is optimised through a heuristic or even algorithm and then recompiled in a beneficial way.

- *learning strategy* : uses a tree-search diagram similar to B&B approach to chart the progress. The choice of which branch to take is guided by learning from the outcome of earlier decisions.

The inability of the classical heuristics to continue the search upon becoming trapped in a local optimum leads to the consideration of techniques for guiding known heuristics to overcome local optimality. One might investigate the application of metaheuristics for solving optimisation problems.

Metaheuristics provide a way of considerably improving the performance of simple heuristic procedures. The search strategies proposed by metaheuristic result in iterative procedures with the ability to escape local optimal points. As such local optima often differ considerably in value from the global optimum, particularly if there are many. The practical impact of metaheuristics has been immense.

The term metaheuristic was coined by Fred Glover [116] in 1986 and has became widely applied in the literature. The formal definition of metaheuristics is based on a variety of definitions from different authors. Following Glover and Laguna [120]:

> *"A metaheuristic refers to a master strategy that guides and modifies other heuristics to produce solutions beyond those that are normally generated in a quest for local optimality."*

The following definition was given in Osman and Kelly [222] in 1996:

> *"A metaheuristic is an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search spaces using learning strategies to structure information in order to find efficiently near optimal solutions"*

The evolution of metaheuristics during the past two decades has taken an explosive upturn. Metaheuristics in their modern forms are based on a variety of interpretations of what constitutes '*intelligent*' search. A number of adaptive processes originating from different settings such as psychology (*learning*), biology (*evolution*), physics (*annealing*), and neurology (*nerve impulses*) have served as a staring point.

To summarise, the following definition by Voß *et al.* [271] seems to be most appropriate:

> *"A metaheuristic is an iterative master process that guides and modifies the operations of subordinate heuristics to efficiently produce high quality solutions. It may manipulate a complete (or incomplete) single solution or a collection of solutions at each iteration. The subordinate heuristics may be high (or low) level procedures, a simple local search, or just a construction method. The family of metaheuristics includes, but is not limited to, adaptive memory procedures, tabu search, ant systems, greedy randomised adaptive search, variable neighborhood search, evolutionary methods, genetic algorithms, scatter search, neural networks, simulated annealing, and their hybrids."*

2.5 Local Search Methods

A Local Search (LS) method can be summarized as an iterative search procedure. It starts from an initial feasible solution from the search space and then improves it by applying a series of local modifications until a local optimum is found. In other words, during the search a set of neighbouring solutions, $N(i)$, in the search space is defined for each feasible solution i, and the next solution j is searched among the solutions in $N(i)$. A neighbouring solution is generated using some suitable mechanism with some acceptance rule to decide on whether to replace the current solution.

The acceptance rule used in a LS method is usually dependent on a comparison of objective function values between the current solution and its neighbouring solution. If the former is larger (for a minimisation problem), then we consider the neighbour solution as an *improving* move. But if the latter is larger, then it is a *deteriorating* move. In the case where both are the same, it is a *neutral* move.

Depending on the strategy of choosing solutions from the neighbourhood of the current solution and the way in which the stopping criteria are defined, we get different LS methods. For an excellent general survey on LS methods, see the collections of Reeves [238], Aarts and Lenstra [1] and Burke and Kendall [41]. In the remainder of this section, we summarise the basics of some of the well-known LS methods.

2.5.1 Descent Method

In a Descent Method (DM), only *improving* moves are allowed. A potential move is rejected if it is found to be *deteriorating* or *neutral*. The procedure stops when no more improving moves can be made. As a result of this limitation, the method terminates at a local optimum and is quite often a fairly mediocre solution. Figure 2.2 shows the structure of a DM.

Figure 2.2: Algorithm of a Descent Method

S1: Choose an initial solution i in S.
S2: Find a j in $N(i)$ such that $f(j) \leq f(k)$ for any $k \in N(i)$.
S3: If $f(j) \geq f(i)$ then stop. Else set $i = j$ and go to **S2**.

The version of a DM called *steepest descent method* scans the entire neighbourhood of i in search of a neighbour solution j that gives the best $f(j)$ value over $j \in N(i)$. Due to its greedy nature, steepest descent is sometimes impractical because it is computationally too expensive, when $N(i)$ contains many elements or each element is too costly to retrieve or evaluate.

To allow further exploration after the finding of a local optimum, one can adopt an acceptance rule which allows non-improving moves. Consequently, the procedure can escape from a local optimum and continue its search. Some examples of procedures that allow deteriorating and neutral moves during search are Simulated Annealing (SA) and Tabu Search (TS).

2.5.2 Tabu Search

Tabu Search (TS) is a method that was originally proposed by Fred Glover [116] in 1986. The word *tabu* (or *taboo*) comes from Tongan, a language of Polynesia, where it was used by the aborigines of Tonga island to indicate things that cannot be touched because they are sacred.

The basic principle of the TS is to pursue LS whenever it encounters a local optimum by allowing non-improving moves. The action of cycling back to previously visited solutions is countered by the use of a short term memory function, called tabu list which records the recent history of the search. Looking at the procedure, one can say that basically TS is an extension of a classical LS. Glover did not see TS as a proper heuristic, but rather as a metaheuristic, a general strategy for guiding and controlling '*inner*' heuristics specifically tailored to solve the problems at hand.

The basic idea of the method is to explore the search space of all feasible solutions by a sequence of moves. A move involves jumping from one solution to another better solution that is available. However, based on the memory of the search history, certain moves may be forbidden or tabu. This might mean not permitting the search to return to a recently visited point in the search space or not allowing a recent move to be reversed. At each iteration, a best admissible move will be selected. A move is called admissible if it is not tabu or if an aspiration criterion is fulfilled. These are described in detail in Glover [117, 118] and Glover and Laguna [120].

General speaking, a tabu list is a short term memory function in which a fixed and fairly limited quantity of information is recorded. The tabu list implicitly keeps track of moves. These attributes will be forbidden from being embodied in moves selected to be used in at least one subsequent iteration because their inclusion might lead back to a previously visited solution. The goal is to permit '*good*' moves in each iteration without revisiting solutions already encountered.

Tabu lists are sometimes too powerful where they may prohibit attractive moves, even when there is no danger of cycling. This may stall the entire searching process. The opportunity for breaking the tabu conditions is defined by the aspiration criterion. The aspiration criterion can be considered a device that allows one to revoke a tabu by allowing a move, even if it is tabu, if the action will result in a solution with an objective value better than that of the current best known solution. Two general aspiration criteria are *influence* and *quality*. A move is thought to be influential if it substantially changes the structure of the current solution, thus moving in to new areas of the solution space. A solution attribute is defined as of sufficient quality to break the tabu conditions if it implies a shift towards the global optimum. The algorithm of a TS is shown in Figure 2.3.

Figure 2.3: Algorithm of a Tabu Search

Generate initial solution
Loop
    Identify *neighbourhood* set
    Identify *tabu* set
    Identify *aspiration* set
    Choose the *best* move
    Exit (when goal is satisfied or the stopping criterion is reached)
End Loop

Various extensions for the TS have been derived over the years such as the *diversification* and *intensification* techniques. At initialization, the goal is make a coarse examination of the solution space, or better known as '*diversification*'. This procedure forces the search into previously unexplored areas of the search space using a long term memory such as *frequency* memory. But as candidate locations are identified, the search becomes more focused to produce local optimal solutions

through the process of '*intensification*' using a intermediate term memory such as the *recency* memory. Detailed descriptions on various extensions for the TS can be found in Glover [117, 118].

2.5.3 Simulated Annealing

The term Simulated Annealing (SA) is derived from the analogous physical process of heating and then slowly cooling a substance to obtain a strong crystalline structure. SA is based on an idea that was first published by Metropolis *et al.* [209] in 1953. But interest in SA began 30 years later with the work of Kirkpatrick *et al.* [168] in 1983 and Černý [45] in 1985 for solving combinatorial optimisation problems. They showed that the Metropolis algorithm could be applied to optimisation problems by mapping the elements of the physical cooling process onto the elements of a combinatorial optimisation problem as shown in Table 2.1.

Table 2.1: Analogy of Simulated Annealing (Dowsland [75])

Thermodynamic Simulation	**Combinatorial Optimisation**
System states	Feasible solutions
Energy	Cost
Change of state	Neighbouring solution
Temperature	Control parameter
Frozen state	Heuristic solution

The SA process lowers the temperature in slow stages until the system '*freezes*' and no further changes occur. At each temperature, the simulation must proceed long enough for the system to reach a steady state or equilibrium. This is known as *thermalization*. The slower the cooling schedule, the more likely the algorithm is to find an optimal or near-optimal solution but with a longer run time. Thus effective use of this technique depends on finding a cooling schedule that produces good enough solutions without taking excessive time for the problem.

The idea was that an initial state of a thermodynamic system was chosen at energy E and temperature T, holding T constant. Then the initial T configuration is perturbed and the change in energy ΔE, is computed. If the change in energy is

negative, the new configuration is accepted. If the change in energy is positive, it is accepted with a probability given by the Boltzmann factor $e^{-\frac{\triangle E}{T}}$. This process is repeated for sufficient times to give a good sampling statistic for the current temperature. The temperature is then slowly decremented by some cooling function C, and the entire process repeated until a frozen state is achieved at $T = 0$. Improving and neutral moves are always accepted, while deteriorating moves are accepted according to a given probabilistic acceptance function. Good overviews of SA and its applications can be found in Dowsland [75] and the SA chapters in textbooks edited by Reeves [238], Aarts and Lenstra [1] and Burke and Kendall [41]. An algorithm stating SA is given in Figure 2.4.

Figure 2.4: Algorithm of a Simulated Annealing

Initialise i to i_0 and T to T_0
Loop - *Cooling*
  Loop - *Local Search*
    Derive a neighbour, j of i
    $\triangle E := E(j) - E(i)$
    If $\triangle E < 0$
    Then $i := j$
    Else derive random number $r \in [0, 1]$
      If $r < e^{-\frac{\triangle E}{T}}$
      Then $i := j$
      End If
    End If
  End Loop - *Local Search*
  Exit (when goal is satisfied or the stopping criterion is reached)
  $T = C(T)$
End Loop - *Cooling*

2.5.4 Genetic Algorithms

This subsection gives a brief overview of Genetic Algorithms (GAs) and does not cover the whole variety of GAs. Detailed descriptions of GAs can be found in the textbooks by Goldberg [121] and Davis [66] and will be discussed in Chapter 4. A comprehensive overview of GAs can also be found in Liepins and Hilliard [188],

Beasley *et al.* [26, 27], Whitley [277], Tomassini [264], Mitchell [212] and Dowsland [76]. Moreover, an annotated bibliography is given in Alander [4].

GAs were first conceived by John Holland in the 1960s and developed by Holland and his students and colleagues at the University of Michigan in 1970s. This led to Holland's book "*Adaptation in Natural and Artificial Systems*" [147] published in 1975. GA is a part of evolutionary computing. The idea of evolutionary computing was first introduced in the 1960s by Ingo Rechenberg in his work '*Evolution Strategy*'. GAs are inspired by Darwin's theory of evolution, based on the genetic processes of biological organisms. Over many generations, natural populations evolve according to the principles of natural selection and 'survival of the fittest', as stated by Charles Darwin in *"The Origin of Species"*. By mimicking this process, a GA is able to '*evolve*' solutions to real world problems, if they have been suitably encoded.

A GA mimics some of the processes of natural evolution and selection. In nature, each species must adapt successfully to an ever changing environment in order to maximise the likelihood of its survival. The knowledge gained by each species is encoded in its chromosomes which will undergo transformation when reproduction occurs. Over a period of time, changes to the chromosomes give rise to species that are more likely to survive, and so have a greater chance of passing their improved characteristics on to future generations. Of course, not all changes will be beneficial but those which are not tend to die out.

Holland's GA attempts to simulate nature's genetic algorithm in the following manner. The first step is to represent a feasible solution to a problem with a string of genes that can take on some value from a specified finite range or alphabet. This string of genes is known as a chromosome (individual). Then, an initial population of individuals, each representing a feasible solution to the given problem is constructed at random. Each individual is assigned a fitness value according to how good a solution to the problem it is. For each generation, the fitness of each individual in the population is measured (a high fitness value would indicate a better solution compared to a low fitness value). The fitter the individuals, the

more likely they are to be selected from the population using a selection mechanism to produce offspring for the next generation via a reproduction stage (crossover and mutation). These offspring will inherit good characteristics of both parents. After many generations of selection for the fitter individuals, the result is hopefully a population that is substantially fitter than the original. Figure 2.5 shows the structure of a GA.

Figure 2.5: Algorithm of a Genetic Algorithm

S1: [**Start**] Generate an initial population P_{pop}, of n chromosomes.
S2: [**Fitness**] Evaluate the fitness $g(x)$ of each chromosome x in the population.
S3: [**New Population**] Create a new population by repeating the following steps until the new population is complete.
  i. [**Selection**] Select 2 parent chromosomes from a population according to their fitness (the fitter, the better chance of being selected).
  ii. [**Crossover**] With a crossover probability p_c, cross over the parents to form 2 new offspring (children). If no crossover was performed, the offspring is an exact copy of parents.
  iii. [**Mutation**] With a mutation probability p_m, mutate new offspring at each locus (position in chromosome).
  iv. [**Replace**] Place new offspring in the new population.
S4: [**Fitness**] Evaluate the fitness $g(x')$ of each chromosome x' in the new population.
S5: [**Test**] If the end condition is satisfied, **STOP**, and return the fittest solution found; otherwise, go to **S3**.

2.5.5 Other Local Search Methods

Scatter Search

The Scatter Search (SS) was first introduced in 1977 by Fred Glover [115] as a heuristic for integer programming. SS is a population based algorithm that stores solutions in a set, called the *reference set* and constructs new solutions by combining existing ones.

Initially, a set of diverse solutions P is constructed. The reference set *RefSet*, is constructed by extracting adequate solutions from P with quality and diversity in mind. With quality, the initial solutions are usually generated from a heuristic

procedure. With diversity, the solutions may be generated from different heuristics to explore different regions of the solution space. Then, a number of subsets of solutions is generated systematically. The solutions of these subsets will be combined to generate new solutions that may replace others in *RefSet*. In other words, new solutions are improved with a local search method before considering their inclusion in *RefSet*. If a new solution has been added to *RefSet*, new subsets are generated and the process is repeated, otherwise, the algorithm is terminated.

More formally, the basic SS algorithm comprises of the following five interacting methods which lead to the well-known template published in 1998 by Glover [119]:

1. Diversification Generation Method.
2. Improvement Method.
3. Reference Set Update Method.
4. Subset Generation Method.
5. Solution Combination Method.

This template has served as the main reference for most of the SS implementations to date. An excellent introduction to the principles of SS is given in Martí *et al.* [204]. The book by Laguna and Martí [174] covers standard implementations of both basic and advanced SS designs.

Greedy Randomised Adaptive Search Procedures – GRASP

GRASP is a multistart or iterative procedure where each GRASP iteration consists of two phases: a construction phase and a local search phase. It was first proposed by Feo and Resende [93] in 1989. The construction phase is essentially a randomised greedy algorithm where a feasible solution is iteratively constructed, one element at a time. The construction phase is therefore capable of producing a diverse set of starting solutions for the local search. The local search phase in the basic GRASP is a simple descent algorithm that finds local optima. The underlying principle is to investigate many good starting solutions through the greedy procedure and thereby increase the possibility of finding a good local optimum on at least one starting solution.

The basic GRASP construction phase is similar to the semi-greedy heuristic introduced by Hart and Shogan [140]. At each construction iteration, a *candidate list* is formed, which lists all of the candidate elements which can be added to the current partial solution in order of their myopic benefit with respect to a *greedy* function. One element is chosen randomly from the candidate list to be added to the partial solution. The heuristic is adaptive because the benefits associated with every element are updated at each iteration of the construction phase to reflect the changes due to the selection of the previous element. Comprehensive reviews on the extensions and applications of GRASP can be found in Feo and Resende [94] and Resende [243].

Ant Colony Optimisation

The Ant Colony Optimisation (ACO) algorithm was first introduced by Marco Dorigo [73] in his thesis in 1992. It was inspired by the behavior of ants in finding paths from the nest to food. We refer to Dorigo *et al.* [74] for an overview of the recent work on ACO.

In the real world, ants will first wander around randomly. The ants communicate information about a path using *pheromone* trails. Ants deposit the pheromone on the ground while walking from food source to the nest and vice versa. If other ants find such a path, they are unlikely to keep wandering around at random, but will instead follow the trail marked by the strong pheromone concentration; returning and reinforcing the trail with its own pheromone if they eventually find food. The pheromone trail allows the ants to find their way back to the food source (or to the nest).

However, the pheromone trail evaporates over time, thus reducing its 'attractive' strength. The longer it takes for an ant to travel down the path and return again, the more time the pheromones evaporate. On the other hand, a shorter path gets marched over faster. Thus, the pheromone density remains high as it is laid on the path as fast as it evaporates.

As a result, when one ant finds a good (i.e. short) path from the nest to a food source, other ants are more than likely to follow that path, and positive feedback eventually causes all the ants to follow a single path. The idea of the basic ACO is to mimic this behavior with '*artificial ants*' walking around the graph which represents the problem to be solved.

Variable Neighborhood Search

Variable Neighborhood Search (VNS) was first proposed by Mladenović and Hansen [213] in 1997. They examine the idea of systematically changing the neighbourhoods within a local search algorithm. VNS explores increasingly distant neighbourhoods of the current solution, and moves from there to a new one as long as improvements are found. This algorithm is simple yet effective and can be implemented easily using any local search method as the inner heuristic.

Initially, a set of neighbourhood structures $N_k, k = 1, \ldots, k_{\max}$, is selected randomly. Then, starting from the first neighbourhood ($k := 1$), an initial solution (current solution) is randomly generated and a local optimum is obtained by applying some local search method (e.g. descent method). The initial solution is generated at random in order to avoid cycling. The current solution is updated if a better solution is obtained and the search continues at $k := 1$; otherwise, proceed to $k := k + 1$. The process is repeated until a stopping criterion is reached.

Various extensions have been derived over the years. A series of comprehensive reviews on the principles and applications of the VNS can be found in Hansen and Mladenović [134, 135, 136, 137, 138].

Iterated Local Search

Iterated Local Search (ILS) is a simple but effective procedure to explore multiple local optima, which can be implemented in any type of local search algorithm. A detailed description of the ILS and its applications can be found in Lourenço *et al.* [199].

Under the basic form of ILS, the next starting solution is obtained from the current local optimum by applying a pre-specified type of random move to it. We refer to such a move as a *perturbation*. Starting from an initial current solution, a local search algorithm is applied to find a new current solution which is a local optimum. Having decided on a current solution, a perturbation is applied. If the solution leading from the perturbation fails the acceptance test, it will be rejected. In this case, another perturbation is executed, and the process is repeated until the perturbation is accepted. Then, when a local optimum is found, the entire procedure is repeated until a stopping criterion is reached.

Chapter 3

Combinatorial Optimisation Problems Studied

3.1 Introduction

In this chapter, we study the three combinatorial optimisation problems mentioned in Chapter 1 in more detail. In Section 3.2, we introduce the basic concepts of bicriteria objective function. In Section 3.3, we give a general introduction to the machine scheduling problem and discuss a typology to the problem. We also present a lower bounding scheme for a single machine family scheduling problem with setup times in Section 3.3.2 which subsequently becomes the lower bound we employ for the computational experiments in Chapter 5, Section 5.6.

Section 3.4 begins with a general overview of the cutting and packing problem, and follows with the typologies of the problem. We discuss some of the well-known approaches (placement routines, exact and local search methods) that can be used to solve the two-dimensional rectangular single bin size bin packing problem. Lower bounds for the two-dimensional rectangular single bin size bin packing problem in both oriented and non-oriented cases are discussed in Section 3.4.3. The lower bounds for the non-oriented cases are used in the computational experiments in Section 6.7. Section 3.5 concentrates on reviewing some of the

well-known approaches, both heuristic and exact, that can be used to solve the symmetric and asymmetric versions of travelling salesman problems.

3.2 Basic Concepts of Bicriteria Objective Function

In this section, we introduce the basic concepts of bicriteria objective function for solving combinatorial optimisation problems. These concepts provide some guidelines on solving the bicriteria problems in Chapter 6 and 7. These concepts, as well as most of the definitions presented are extracted from Hoogeveen [149].

Suppose that for a given combinatorial optimisation problem, there are two performance criteria, say f and g, that need to be considered. Without loss of generality, assume that these criteria are to be minimised. Unless we are extremely lucky, there will be no solution that achieves the minimum value for both performance criteria simultaneously. Depending on the relationship between the performance criteria, two distinct approaches can be distinguished: *hierarchical optimisation* and *simultaneous optimisation*.

Hierarchical Optimisation

Hierarchical Optimisation is used when one of the performance criterion is far more important than the other one. Suppose that criterion f is more important than criterion g. In the first stage, the optimum value, say f^*, with respect to criterion f is obtained. In the second stage, the second criterion g is optimised subject to the additional constraint that $f \leq f^*$. This approach also can be referred to as a *lexicographical optimisation* approach.

Simultaneous Optimisation

Simultaneous Optimisation is used when no criterion is dominant. Thus, the performance of the second criterion can be greatly improved while losing only a little performance on the first criterion. Evans [86] and Fry *et al.* [107] distinguish three

different approaches in simultaneous optimisation: *priori optimisation*, *interactive optimisation*, and *posteriori optimisation*.

In *priori optimisation*, both criteria are aggregated into one *composite objective function* $F(f, g)$ for some given function F, after which an optimum solution is determined for this one problem as a whole. F can be a linear function such as $\alpha f + g$, where α is a given constant that indicates the relative importance of criterion f with respect to criterion g, but it may just as well be a quadratic or even more complex function.

An *interactive optimisation* is used when active involvement of a decision maker is required during the solution process. Given one or more already obtained, relevant solutions, the decision maker must indicate which one is preferable, and if not satisfied yet, in which direction the search should continue.

A *posteriori optimisation* is employed when it is computationally inaccessible in optimising the function $F(f, g)$ in a direct manner, especially when the function F is nonlinear. This problem can be solved in two ways. We first select from the set of solutions a subset that contains an optimum solution. If the function F is known, then we compute the optimum solution in this set. If F is not known, then we present this set to the decision maker and let him/her choose the solution.

By applying simultaneous optimisation on a function $F(f, g)$, where both f and g are to be minimised, there exists a *Pareto optimal* solution by which the optimum is attained. Usually, the number of Pareto optimal points is finite. However, the number of Pareto optimal points can become infinite subject to the constraints and assumptions of the problem to be solved. A *trade-off* curve is defined as the curve that contains all Pareto optimal points. Moreover, an *efficient frontier* can be defined as a piecewise-linear convex function, where each endpoint corresponds to the solution of one of the *lexicographical optimisation* problems, where each breakpoint is Pareto optimal, and each Pareto optimal point is located either on or above this function (cite in Hoogeveen [149]). Note that the trade-off curve and the efficient frontier are equal only if the trade-off curve is convex.

3.3 Machine Scheduling Problems

Machine Scheduling Problems (MSPs) exist in many diverse areas, such as flexible manufacturing systems, production planning, airline industry, hospital, etc. For some latest surveys on the real world applications of the machine scheduling problems, see the collection of Leung [185]. The main focus is on the efficient allocation of one or more resources to activities over time. Due to the complexity studies conducted during the last three decades, it is now widely understood that most machine scheduling problems are NP-hard (see Lenstra *et al.* [184] for more details). Some excellent and comprehensive reviews of the MSPs can be found in Conway *et al.* [58], Baker [20], Lawler *et al.* [181], Anderson *et al.* [8], Chrétienne *et al.* [51] and Pinedo [229]. The recent textbook by Leung [185] provides excellent coverage of the most recent and advanced topics on scheduling problems. Moreover, an annotated bibliography is given in Hoogeveen *et al.* [150].

MSPs can be briefly described as follows (as in Anderson *et al.* [8]):

"There are m *machines*, which are used to process n *jobs*. A *schedule* specifies, for each machine i ($i = 1, 2, \ldots, m$) and each job j ($j = 1, 2, \ldots, n$), one or more time intervals throughout which processing is performed on j by i."
A schedule is *feasible* if there is:

- no overlapping of time intervals corresponding to the same job (so that a job cannot be processed by more than one machine at one time),
- no overlapping of time intervals corresponding to the same machine (so that a machine cannot process more than one job at one time), and
- satisfies various requirements relating to the specific problem type (machine environment, job characteristics, and optimality criterion) which will be discussed in detail in Section 3.3.1.

In the study of the MSPs, we focus specifically on Single Machine Family Scheduling Problem (SMFSP) where jobs are partitioned into families and set up is required between these families. The objective is to find a schedule which

minimises the maximum lateness of the jobs in the presence of the sequence independent family setup times. Details of the research are given in Chapter 5.

A single machine scheduling problem is one where there are n jobs to be scheduled on a single machine. The assumption is that all jobs and the machine are available at time zero and preemption of jobs are generally not allowed. Each of the jobs j ($j = 1, \ldots, n$), is characterised by its processing time p_j, and associated due date d_j. Other parameters of job j that occur in some problems include a release date r_j, a deadline $\bar{d}_j$, and a weight w_j. An early survey of this problem is given by Gupta and Kyparisis [133].

Much of the early work on scheduling was concerned with the analysis of single machine scheduling systems. These include Jackson's derivation of the Earliest Due Date (EDD) rule in 1955 where jobs are sequenced in non-decreasing order of their due dates (see Jackson [156]), and Smith's derivation of the Shortest Weighted Processing Time (SWPT) rule in 1956 where jobs are sequenced in non-decreasing order of their processing time to weight ratios (see Smith [256]). These orderings are used as priority rules for scheduling more complex systems.

The study of the single machine scheduling problem is still very important for several reasons, but most relevant is that a good understanding of this problem provides a support to model the behaviour of a complex system. It is important to understand the working of the system components, and quite often the single machine problem appears as an elementary component in a large scheduling problem (cite in Baker [20]).

3.3.1 Typology of Machine Scheduling Problems

A typology is a systematic classification of objects into homogenous categories based on a given set of criteria. It helps to unify definitions and notations. In this subsection, we will discuss in detail the specific problem type for the MSPs. Most of the definitions presented in this subsection are extracted from Anderson *et al.* [8].

Machine Environment

Different configurations of machines create different production systems for the problem. However, in each case, all machines become available to process jobs at time zero. These production systems can be classified as follows:

- *single stage*: one operation for each job involving either a single machine or m machines operating in parallel;
- *multi stage*: jobs require operations on different machines involving either flow shop, job shop or open shop scheduling.

In the case of parallel machine scheduling, each machine has the same function and each job j has to spend a given time on any of the m machines. This system can be decomposed further as follows:

- *identical parallel machines*: processing time of an operation is independent of the machine assignment;
- *uniform parallel machines*: machines operate at different speeds but are otherwise identical; and
- *unrelated parallel machines*: processing time of an operation depends on the machine assignment.

A comprehensive review of parallel machine scheduling can be found in Cheng and Sin [49].

In multi-stage machine scheduling, there are m machines, each having a different function. Each job j consists of several operations, each of which has to be executed on a designated machine, and no job can undergo more than one operation at a time. The details of the multi stage system are as follows:

- *flow shop*: a job is processed once on each machine in the routing of $1, 2, \ldots, m$. All jobs follow the same routing of machines.
- *open shop*: each job is also processed once on each machine, but the machine routing can differ between jobs and forms part of the decision process.
- *job shop*: each job has a prescribed routing through the machines, and the routing may differ between jobs.

Vaessens *et al.* [268], Jones and Rabelo [162], and Jain and Meeran [157] provide surveys on the job shop scheduling problems while surveys on flow shop scheduling problems are given by Dudek *et al.* [79] and Cheng *et al.* [48]. A detailed survey on complexity results for open shop scheduling problems is given by Kubiak *et al.* [173].

Job Characteristics

In the case of single machine and identical parallel machines, we denote the *processing time* for job j as p_j. For uniform parallel machines, the processing time on machine i may be expressed as p_j/u_i, where u_i is the speed of machine i. We denote p_{ij} as the processing time on machine i for the case of unrelated parallel machine, flow shop and open shop scheduling problem. In a job shop, p_{ij} denotes the processing time of the ith operation of job j. Job availability may be restricted by imposing a *release date* r_j, which is the time when job j is available for processing, or a *deadline* $\overline{d}_j$, which specifies the time by which it should ideally be completed.

Suppose that jobs within a schedule can be partitioned into F families according to the similarity of their production requirements. As a result of this similarity, no set up on a machine is required when following another job from the same family. However, a *family setup time* on machine i is required when a job of family g is immediately preceded by a job of a different family f. We denote the *family setup time* as s_{ifg}, or s_{i0g} if there is no preceding job. If for each g, we can write $s_{ifg} = s_{i0g} = s_{ig}$ for all $f \neq g$, then the setup times on machine i are *sequence independent*; otherwise, they are *sequence dependent*. If for each machine i, $s_{ifg} = s_{fg}$ for all families f and g including the case $f = 0$, then the setup times are *machine independent*; otherwise, they are *machine dependent*. Hence, by definition, the setup times for a single machine are machine independent.

The job characteristics also include the possibility of allowing *preemption* and of specifying *precedence constraints*. If *preemption* is allowed, then an operation may be interrupted and resumed at a later time; otherwise, an operation, once started, must be processed until completion without interruption. A *precedence constraint* stipulates that a certain job cannot start before another one has been completed.

Optimality Criteria

The optimality criterion is usually a function of the job completion times $C_1, C_2, \ldots, C_n$. Common criteria are *maximum completion time* $C_{\max}=\max_j C_j$, and *total completion time* $\sum_j C_j$. For a given schedule, if a due date d_j is specified for each job j, we can compute for job j its:

- *lateness*, $L_j = C_j - d_j$,
- *tardiness*, $T_j = \max_j\{0, C_j - d_j\}$,
- *earliness*, $E_j = \max_j\{0, d_j - C_j\}$, and
- *unit tardiness*, $U_j = 1$ if $C_j > d_j$, $U_j = 0$ otherwise.

Important criteria involving job due dates are the *maximum lateness* $L_{\max}=\max_j L_j$, *total tardiness* $\sum_j T_j$, *total earliness* $\sum_j E_j$, and *number of late jobs* $\sum_j U_j$. If each job j has a positive *weight* w_j, then we can also have weighted versions of these criteria.

Throughout the study, we will adopt the representation scheme of Graham *et al.* [128] for the machine scheduling problem. This is a three-field representation $\alpha|\beta|\gamma$ which indicates the specific problem type:

α: machine environment,

β: job characteristics,

γ: optimality criterion which involves the minimisation.

These characteristics are summarised in Table 3.1.

Table 3.1: Graham *et al.*'s Typology of Machine Scheduling Problems

Characteristic	Symbol	Description
Machine Environment α	$\alpha_1 = \circ$	a single machine
	$\alpha_1 = P$	identical parallel machines
	$\alpha_1 = Q$	uniform parallel machines
	$\alpha_1 = R$	unrelated parallel machines
	$\alpha_1 = F$	a flow shop
	$\alpha_1 = O$	an open shop
	$\alpha_1 = J$	a job shop
	$\alpha_2 = \circ$	the number of machines is arbitrary
	$\alpha_2 = m$	there are a fixed number of machines m
Job Characteristics β	$\beta_1 = \circ$	no release dates are specified
	$\beta_1 = r_j$	jobs have release dates
	$\beta_2 = \circ$	no deadlines are specified
	$\beta_2 = \tilde{d}_j$	jobs have deadlines
	$\beta_3 = \circ$	there are no setup times
	$\beta_3 = s_{ifg}$	there are general family setup times
	$\beta_3 = s_{fg}$	there are machine independent family setup times
	$\beta_3 = s_{if}$	there are sequence independent family setup times
	$\beta_3 = s_f$	there are machine and sequence independent family setup times
	$\beta_4 = \circ$	no precedence constraints are specified
	$\beta_4 = prec$	jobs have precedence constraints
	$\beta_4 = pmtn$	preemption of jobs is allowed
Optimality Criterion γ (involves the minimisation of)	$C_{\max}$	maximum completion time
	$L_{\max}$	maximum lateness
	$\sum_j (w_j)C_j$	total (weighted) completion time
	$\sum_j (w_j)T_j$	total (weighted) tardiness
	$\sum_j (w_j)U_j$	total (weighted) number of late jobs
	$\sum_j (w_j)E_j$	total (weighted) earliness

Let $\circ$ denote the empty symbol. The first field takes the form $\alpha = \alpha_1\alpha_2$, where α_1 and α_2 are interpreted as in Table 3.1. Note that for a single machine problem, $\alpha_1 = \circ$ and $\alpha_2 = 1$, whereas $\alpha_1 \neq \circ$ and $\alpha_2 \neq 1$ for other types of problems. An example of the representation scheme is $1|\ |\sum w_j C_j$, which denotes the minimisation of total weighted completion time in a single machine scheduling problem. Another example is problem $Pm|r_j|L_{\max}$, which denotes the minimisation of maximum lateness on a fixed number m of identical parallel machines with jobs release dates.

3.3.2 Lower Bounds for $1|s_f|L_{\max}$

In this subsection, we present a lower bounding scheme for a SMFSP with family setup times. The objective of the problem is to find a schedule which minimises the maximum lateness $L_{\max}$ of the jobs in the presence of the sequence independent family setup times s_f. The lower bounds are used to measure the performance of the heuristic solution found when the exact solution to the problem is unknown. The lower bounds are obtained from the bounds proposed by Hariri and Potts [139]. The complete explanation and derivation of the lower bounds can be found in Hariri and Potts [139] where the lower bounds were used in their branch and bound algorithm.

We first relax all setup times except the first job of each family and solve the resulting problem by Jackson's EDD rule. We use the subscript pair (f, i) to identify the ith job from family f. Let S be an arbitrary subset of jobs, and let

- d_{fi} = due date of job (f, i),
- p_{fi} = processing time of job (f, i),
- $C(S)$ = lower bound on the completion time of the job sequenced last amongst jobs of S,
- $\bar{S}$ = subset of S which jobs may be sequenced last amongst job of S in a feasible schedule.

Thus, a valid lower bound on the maximum lateness is

$$LB(S) = C(S) - \max_{(f,i)\in\bar{S}}\{d_{fi}\}. \tag{3.1}$$

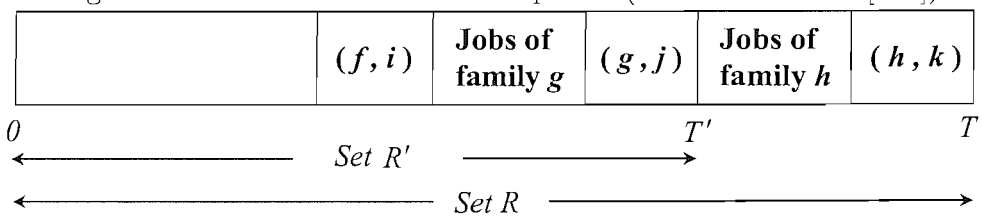
In a feasible schedule, there is a setup time, s_f before the first job in each family, f $(f = 1, \ldots, F)$. Before applying the EDD rule to the feasible schedule, we can relax the setup time by resetting the processing time of the job in each family f using $p_{f1} = p_{f1} + s_f$. Suppose that job (h, k) has maximum lateness, we have the lower bound

$$LB_0 = T - d_{hk} \tag{3.2}$$

where T is the completion time of job (h, k) in the EDD sequence. The lower bounding scheme is terminated with LB_0 as the lower bound if all jobs before (h, k) in the EDD sequence are from family h. If there is a job in the EDD sequence that is scheduled before (h, k) and which does not belong to family h, the search for improving the lower bound continues.

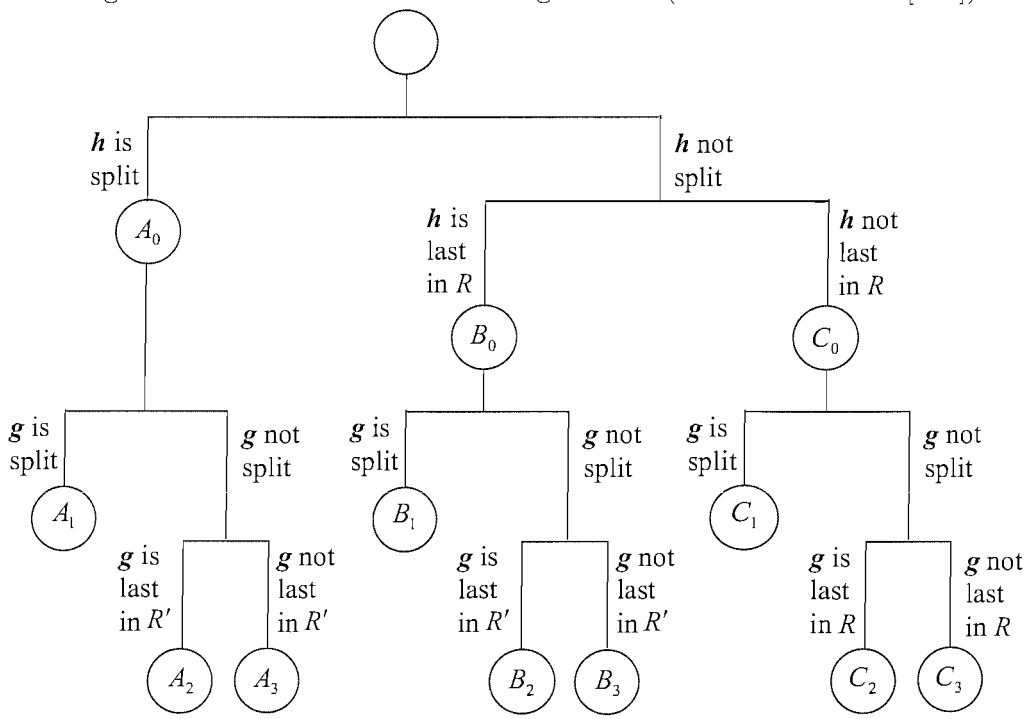
Let (g, j) be the last job in the family g that appears before job (h, k) in the EDD sequence, and T' denote the completion time of job (g, j) in this sequence. Also, let R' denote the set of jobs in the EDD sequence up to and including job (g, j) and let denote R the set of jobs in the EDD sequence up to and including job (h, k). Figure 3.1 shows the structure of the EDD sequence, and the sets R' and R.

Figure 3.1: Structure of an EDD sequence (Hariri and Potts [139])



In the subsequent analysis, we discard all jobs except those of the set R. The various cases of analysis are depicted in Figure 3.2. The bounds at the various nodes of interest are derived below.

Figure 3.2: Cases in lower bounding scheme (Hariri and Potts [139])



Node A_0: jobs of family h do not form a single batch.

Due to the additional family setup time for family h, we get completion time, $C(R) = T + s_h$. From equation (3.2), we obtain

$$LB_{A_0} = T + s_h - d_{hk}. \tag{3.3}$$

Node A_1: jobs of family g and of family h do not form a single batch.

Similar to node A_0, $C(R) = T + s_g + s_h$ because of the additional setup times for families g and h, yielding

$$LB_{A_1} = T + s_g + s_h - d_{hk}. \tag{3.4}$$

Node A_2: all jobs of family g are scheduled after the other jobs of R'.
Using the information that job (g, j) is completed no earlier than time T', the lower bound, $LB(R' - \{(g, j'+1), \ldots, (g, j)\})$ for $j' = 1, \ldots, j$, where $C(R' - \{(g, j'+1), \ldots, (g, j)\}) = T' - \sum_{i=j'+1}^{j} p_{gi}$. The best of these lower bounds is

$$LB_{A_2} = \max_{j'=1,\ldots,j} \left\{ T' - \sum_{i=j'+1}^{j} p_{gi} - d_{gj'} \right\}. \tag{3.5}$$

Node A_3: job (g, j) is not scheduled last amongst jobs of R'.
Excluding the jobs of family g, let job (f, i) have the largest due date amongst jobs of R', yields

$$LB_{A_3} = T' - d_{fi}. \tag{3.6}$$

Thus, the overall lower bound for nodes labelled A is

$$LB_A = \max\left\{LB_{A_0}, \min\left\{LB_{A_1}, LB_{A_2}, LB_{A_3}\right\}\right\}. \tag{3.7}$$

Nodes B_0, B_1, B_2, and B_3:
Since the analysis of nodes labelled B is similar to that for the A nodes, the lower bounds are stated without derivation.

$$LB_{B_0} = \max_{k'=1,\ldots,k} \left\{ T - \sum_{i=k'+1}^{k} p_{hi} - d_{hk'} \right\}, \tag{3.8}$$

$$LB_{B_1} = T' + s_g - d_{gj}, \tag{3.9}$$

$$LB_{B_2} = \max_{j'=1,\ldots,j} \left\{ T' - \sum_{i=j'+1}^{j} p_{gi} - d_{gj'} \right\}, \tag{3.10}$$

$$LB_{B_3} = T' - d_{fi}. \tag{3.11}$$

Thus, the overall lower bound for nodes labelled B is

$$LB_B = \max\left\{LB_{B_0}, \min\left\{LB_{B_1}, LB_{B_2}, LB_{B_3}\right\}\right\}. \tag{3.12}$$

Nodes C_0, C_1, C_2, and C_3:

As for the nodes labelled B, we state the lower bounds for the C nodes without derivation. Note that there is no explicit expression for LB_{C_0}.

$$LB_{C_1} = T + s_g - d_{gj}, \tag{3.13}$$

$$LB_{C_2} = \max_{j'=1,\ldots,j} \left\{ T - \sum_{i=j'+1}^{j} p_{gi} - d_{gj'} \right\}, \tag{3.14}$$

$$LB_{C_3} = T - d_{fi}. \tag{3.15}$$

Therefore, the overall lower bound for nodes labelled C is

$$LB_C = \min \{LB_{C_1}, LB_{C_2}, LB_{C_3}\}. \tag{3.16}$$

The overall lower bound for the problem is the minimum of the individual bounds from nodes labelled A, B and C: it can be expressed as

$$LB_1 = \min \{LB_A, LB_B, LB_C\}. \tag{3.17}$$

Recall that if R contains only jobs of family h, then the lower bound is $LB = LB_0$; otherwise, it is $LB = LB_1$. The computation of LB_0 and LB_1 requires $O(n \log n)$ time.

3.4 Cutting and Packing Problems

Cutting and Packing (C&P) problems are optimisation problems that are concerned with finding a good arrangement of multiple small items in one or more larger objects. This type of problem is encountered in many areas of business and in industry (e.g. wood, glass, and textile industries, newspaper paging, cargo loading, etc) and forms part of the combinatorial optimisation problems found in operational research. The usual objective of the allocation process is aimed at maximising the utilisation of the large objects, or maximising the value of the small items packed.

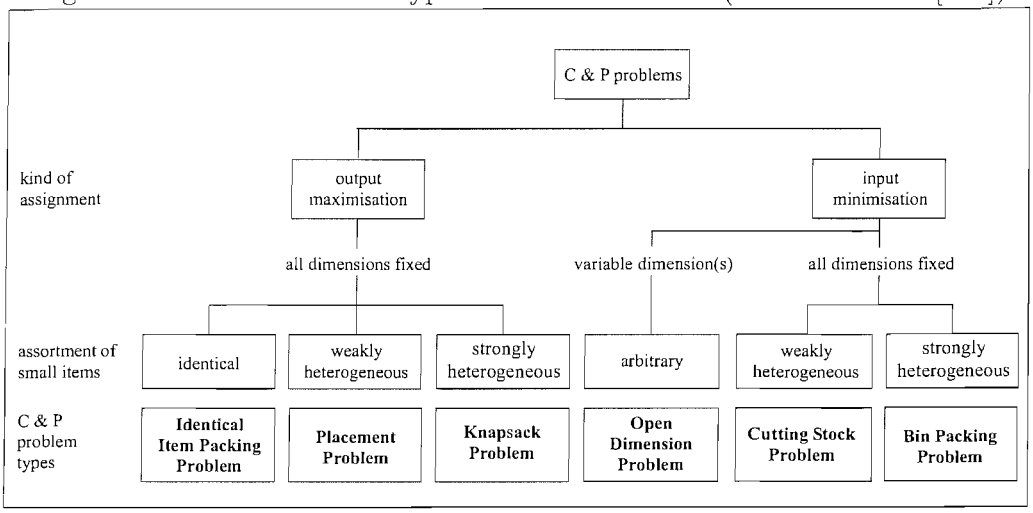
The reduction of production costs is one of the major issues in manufacturing industries. High material utilisation is of particular interest to industries involved with mass-production, since small improvements of the layout can result in large savings of material and considerably reduce production costs. The complexity of the problem and the solution approach depend on the geometry of the items to be placed and the constraints imposed.

Although C&P problems have been studied since the mid-fifties, Gilmore and Gomory's articles in the 1960s [112, 113, 114] on linear programming approaches to one, two and more dimensional cutting stock problems are the first to present techniques which could be practically applied to medium size real-world problems.

In the study of the C&P problems, we focus on the non-oriented case of the two-dimensional rectangular single bin size bin packing problem where a set of small rectangles has to be allocated to one or more bins. The rectangles to be packed may be rotated by 90°. The objective is to pack without overlaps, all the rectangles into the minimum number of bins. Details of the research are discussed in Chapter 6.

C&P problems occur in various application areas involving different constraints and objectives. In the following, we briefly define the basic problem types (see Figure 3.3) of C&P problems based on the improved typology given by Wäscher *et al.* [274].

Figure 3.3: Basic Problem Types of C&P Problems (Wäscher *et al.* [274])



The C&P problem can be broadly classified into two sub-problems: output (value) maximisation problem and input (value) minimisation problem. In both cases, a set of small items has to be assigned to a given set of large objects. With respect to the assortment of the small items, Wäscher *et al.* [274] distinguish three cases, namely identical, weakly heterogeneous and strongly heterogeneous.

For the identical small items, all items are of the same shape and size. The small items with weakly heterogeneous assortment can be grouped into relatively few classes (in relation to the total number of items), for which the items are identical with respect to shape and size. The set of strongly heterogeneous assortment of small items is characterised by the fact that only very few items are of identical shape and size.

Output Maximisation:

In this case, the set of large objects is not sufficient to accommodate all the small items. Thus, all large objects are to be used to which a selection of the small items of maximal value has to be assigned. Based on the assortment of the small items, a problem in this category can be classified as an identical item packing problem, a placement problem or a knapsack problem.

- **Identical Item Packing Problem**

  This problem consists of the assignment of the maximum number of identical small items to a limited set of large objects. Dowsland and Dowsland [78] define this problem as a manufacturer's pallet loading problem where identical items have to be loaded onto the pallet.

- **Placement Problem**

  In this problem, a weakly heterogeneous assortment of small items has to be assigned to a limited set of large objects. Dowsland and Dowsland [78] refer to this problem as a distributor's pallet loading problem where the pallet has to be packed with non-identical items.

- **Knapsack Problem**

  In this problem, a strongly heterogeneous assortment of small items which has to be allocated to a limited set of large objects while observing the capacity constraint of the large objects such that the total value of the items packed is maximised.

Input Minimisation:

In this case, the set of large objects is sufficient to accommodate all small items. All small items are to be assigned to a selection of the large object(s) of minimal value. Based on the assortment of the small items, a problem in this category can be classified as an open dimension problem, a cutting stock problem or a bin packing problem

- **Open Dimension Problem**

  This problem involves a set of small items which has to be assigned completely to one or more large objects. The large objects are given but their extension in at least one dimension can be considered as a variable. Dowsland and Dowsland [78] define this problem as a strip packing problem where the large objects are strips with fixed width but unlimited height. The objective is to minimise the total height needed to pack all the small items.

- **Cutting Stock Problem**

  In this problem, a weakly heterogeneous assortment of small items is completely assigned to a selection of large objects of minimal value.

- **Bin Packing Problem**

  This problem is concerned with packing a set of strongly heterogeneous small items into the minimum number of large objects.

3.4.1 Typologies of Cutting and Packing Problem

Due to the diversity of the problem and application areas, similar packing problems appear under different names in the literature. In order to facilitate the information exchange, Dyckhoff [82] introduces four characteristics according to which C&P problems are categorised. However, his typology was not widely accepted. As a result, Wäscher *et al.* [274] develop a revised classification of the typology. The details of the typologies are given below:

Dyckhoff's Typology:

Dyckhoff [82] seeks to identify common characteristics and properties to discriminate between problem types. As a result, he systematically classified packing problems into a 4-field representation of $\alpha|\beta|\gamma|\delta$ where,

α: Dimensionality.

β: Kind of Assignment.

γ: Assortment of Large Objects.

δ: Assortment of Small Items.

These characteristics and the values they can take on are summarised in Table 3.2.

Table 3.2: Dyckhoff's Typology of Cutting and Packing Problems (Dyckhoff [82])

Characteristic	Symbol	Description
Dimensionality	1	one dimensional
	2	two dimensional
	3	three dimensional
	N	N dimensional with $N > 3$
Kind of Assignment	B	all objects and a selection of items
	V	a selection of objects and all items
Assortment of Large Objects	O	one object
	I	identical figures
	D	different figures
Assortment of Small Objects	F	few items (of different figures)
	M	many items of many different figures
	R	many items of relatively few different (non-congruent) figures
	C	congruent figures

Hence, a classical Two-Dimensional Bin Packing Problem (2DBPP) can be classified as 2|V|I|M, where,

2 : two dimensional.
V : a selection of objects and all items.
I : identical figure for large objects.
M: many items of many different figures for small items.

Wäscher *et al.*'s Typology:

After almost 15 years since Dyckhoff's initial publication, it became obvious that Dyckhoff's typology was insufficient with respect to recent developments. In 2005, Wäscher *et al.* [274] propose an improved typology with the aim of allowing for a complete categorisation of all known C&P problems and its corresponding literature. They highlight the following three drawbacks faced by the Dyckhoff's typology in the recent development in the field of C&P:

- not necessarily all C&P problems (in the narrow sense) can be assigned uniquely to problem types;
- Dyckhoff's typology is partially inconsistent; its application might have confusing results;
- application of Dyckhoff's typology does not necessarily result in homogeneous problem categories.

These drawbacks are discussed further with help from examples in Wäscher *et al.* [274]. The details of the typology are summarised in Table 3.3.

Table 3.3: Wäscher *et al.*'s Typology of Cutting and Packing Problems (Wäscher *et al.* [274])

Characteristic	Description	Further Description
Dimensionality	$1, 2, 3, N(N > 3)$	
Kind of Assignment	Output (value) maximisation Input (value) minimisation	e.g. knapsack problem e.g. bin packing problem
Assortment of Small Items	Identical small items Weakly heterogeneous assortment Strongly heterogeneous assortment	
Assortment of Large Objects	One large object	All dimensions fixed One or more variable dimensions
	Several large objects (all dimensions fixed)	Identical large objects Weakly heterogeneous assortment Strongly heterogeneous assortment
Shape of Small Items	Regular Irregular (or non-regular)	e.g. rectangles, circles, cylinders, etc. e.g. shirts, shoes, swimsuit, etc.

In order to further define the typology, the characteristics for the *kind of assignment* are structured further into Intermediate Problem Types (IPT). This is achieved by taking into consideration the assortment of the large objects as well as the small items as an additional differentiating criterion. Wäscher *et al.* [274] summarise the system of the IPT as in Table 3.4 and 3.5. In the final stage, they systematically classify the C&P problems according to the following system:

$$\{1, 2, 3, n\} - \text{dimensional}\{\emptyset, \text{rectangular}, \text{circular}, \ldots, \text{irregular}\}\{\text{IPT}\}.$$

As mentioned earlier, we concentrate our study on the non-oriented two-dimensional rectangular single bin size bin packing problem where the rectangles may rotate 90°. This problem is classified by Wäscher *et al.* [274] as: non-oriented 2DRSB-SBPP. Thus, for the rest of the thesis, we refer without loss of generality, to the problem in our study as non-oriented 2DRSBSBPP.

Table 3.4: Landscape of IPT: Output Maximisation (Wäscher *et al.* [274])

characteristics of the large object \ assortment of the small items		identical	weakly heterogeneous	strongly heterogeneous
all dimensions fixed	one large object	Identical Item Packing Problem **IIPP**	Single Large Object Placement Problem **SLOPP**	Single Knapsack Problem **SKP**
	identical		Multiple Identical Large Object Placement Problem **MILOPP**	Multiple Identical Knapsack Problem **MIKP**
	heterogeneous		Multiple Heterogeneous Large Object Placement Problem **MHLOPP**	Multiple Heterogeneous Knapsack Peoblem **MHKP**

Table 3.5: Landscape of IPT: Input Maximisation (Wäscher *et al.* [274])

characteristics of the large object \ assortment of the small items		weakly heterogeneous	strongly heterogeneous
all dimensions fixed	identical	Single Stock Size Cutting Stock Problem **SSSCSP**	Single Bin Size Bin Packing Problem **SBSBPP**
	weakly heterogeneous	Multiple Stock Size Cutting Stock Problem **MSSCSP**	Multiple Bin Size Bin Packing Problem **MBSBPP**
	strongly heterogeneous	Residual Cutting Stock Problem **RCSP**	Residual Bin Packing Problem **RBPP**
one large object variable dimension(s)		Open Dimension Problem **ODP**	

3.4.2 Approaches to 2DRSBSBPP

Before the start of the survey, we define the 2DRSBSBPP as follows:
"Given a set of n rectangular items $j \in J = \{1, 2, \ldots, n\}$, each defined by a height h_j, and a width w_j, and an unlimited number of identical rectangular bins, each having a height H, and a width W. The objective is to allocate without overlaps, all the rectangles into the minimum numbers of bins." The 2DRSBSBPP is classified as a class of NP-hard problem by Garey and Johnson [108].

A considerable amount of research has been carried out and various approaches have been proposed to solve the 2DRSBSBPP. In the following subsection, we concentrate on the review of the literature for the problem. Some excellent and comprehensive reviews of the approaches to the problem can be found in Dowsland and Dowsland [78], Dyckhoff and Finke [83], Lodi *et al.* [195, 196, 197], and Hopper and Turton [152]. Moreover, an annotated bibliography is given in Dyckhoff *et al.* [84]. The approaches can be broadly classified into three methods: heuristic placement routines, exact approaches and lower bounds, and local search methods.

3.4.2.1 Heuristic Placement Routines

Most of the heuristic placement routines from the literature can be classified in two families (see Lodi *et al.* [196]):
– *One-phase algorithms*: directly pack the rectangles into the finite bins.
– *Two-phase algorithms*: start by packing the rectangles into a single strip, (i.e. a bin having width W, and infinite height). In the second phase, the strip solution is used to construct a packing into finite bins.

The majority of the approaches are level algorithms, i.e. the bin packing is obtained by placing the rectangles, from left to right, in rows forming levels. The first level is the bottom of the bin, and subsequent levels are produced by the horizontal line coinciding with the top of the tallest rectangle packed on the level below. Three classical strategies for level packing are suggested by Coffman *et al.* [56]. Note that j = current rectangle.

1. **Next-Fit (NF)**: rectangle j is packed left justified on a level if it fits. Otherwise, the level is closed and a new level is created to pack the rectangle left justified.

2. **First-Fit (FF)**: rectangle j is packed left justified on the first level where it fits. If no level can accommodate j, a new level is initialised as in NF.

3. **Best-Fit (BF)**: rectangle j is packed left justified on that level, among those where it fits, for which the resulting packing has the minimum remaining horizontal space. If no level can accommodate j, a new level is initialised as in NF.

In addition, sorting the rectangles in decreasing width, height, or area in combination with NF, FF, and BF routine, can improve the average performance of the simple placement routines. These routines are referred to as NFD, FFD, and BFD respectively (D = Decreasing), and can be implemented to run in $O(n \log n)$ time.

For the remainder of this subsection, we denote a *current bin list* as a list of all possible bins in which the next rectangle can be packed. We discuss each of the heuristic placement routine by classifying them as **H,A,(R),T**, where:

H : name of the **H**euristic placement routine.
A : **A**bbreviation of the routine.
(R): name of the **R**esearcher(s) who popularised the routine.
T : **T**ime complexity of the routine.

One-Phase Algorithms

Finite Next-Fit, FNF, (Berkey and Wang [32]), $O(n \log n)$:
Only one bin is held in the current bin list. Rectangles are packed into finite bins using the NF routine. When the next rectangle to be packed cannot fit into the current bin, the bin is removed and a new empty bin is added.

Finite First-Fit, FFF, (Berkey and Wang [32]), $O(n \log n)$:
All the bins that have been created are maintained in the current bin list. Each rectangle is packed on the lowest level of the first bin where it fits. If no level in the bins can accommodate it, a new level is created either in the first suitable bin, or by initialising a new bin.

Bottom-Left, BL, (Baker *et al.* [16], Jakobs [158]), $O(n^2)$:
This is a different classical approach which does not pack the rectangles by level packing heuristic. The rectangle is packed as near to the bottom of the bin as it will fit and then as far to the left as it can be placed at that bottom-most level. Starting from the top right corner of the bin, each rectangle makes successive moves of sliding as far as possible to the bottom of the bin and then as far as possible to the left of the bin until the rectangle is placed in a stable position. Figure 3.5 shows the placement of a sequence of rectangles described in Figure 3.4. The major disadvantage of this routine is the creation of empty areas in the layout, when larger rectangles block the movement of successive ones.

Figure 3.4: Bin and Item Dimensions

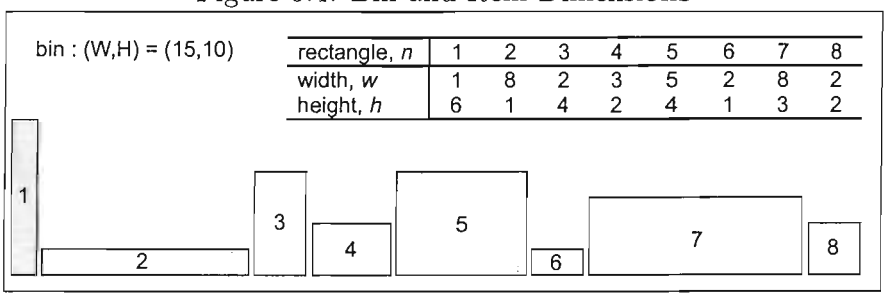
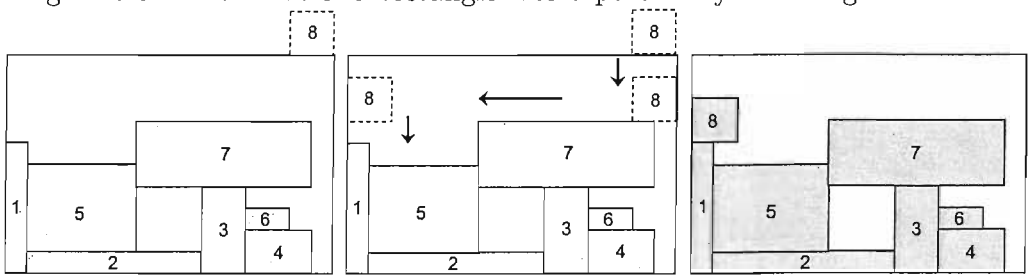


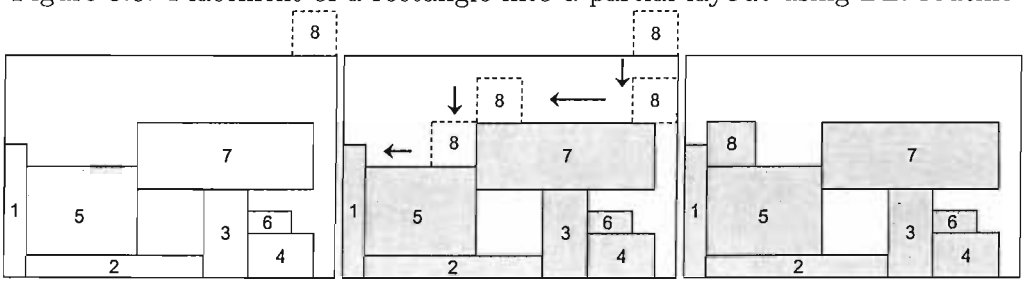
Figure 3.5: Placement of a rectangle into a partial layout using BL routine



Finite Bottom-Left, FBL, (Berkey and Wang [32]), $O(n^2)$:
This routine is a variation of BL for the finite bin case. The routine initially sorts the rectangles by non-increasing width. The rectangle is then packed in the lowest position of any initialised bin, left justified. If no bin can allocate it, a new one is initialised.

Improved Bottom-Left, BLi, (Liu and Teng [192]), $O(n^2)$:
Like the BL routine, it starts by placing the rectangle on the top right corner of the bin. It is then moved as far as possible to the bottom. Instead of moving it the complete distance to the left in the next step until it collides as in the BL routine, the BLi routine moves the rectangle along the partial layout by giving downward movement priority so that rectangles only slide leftwards if no downward movement is possible. In Figure 3.6, the allocation of the same sequence of rectangle used in Figure 3.4 is shown. Liu and Teng [192] give two numerical examples to compare the performance of the BL and BLi. Comptutational experiments show that BLi constantly outperformed the BL.

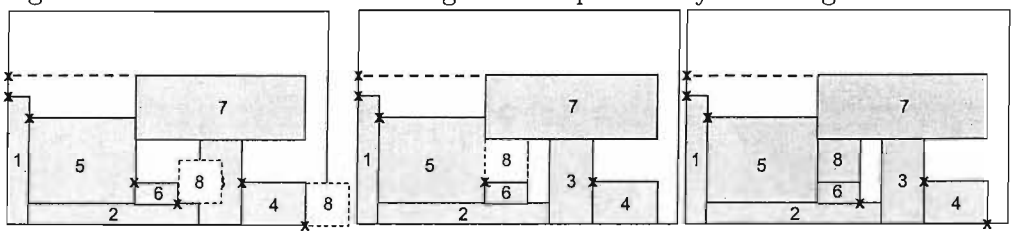
Figure 3.6: Placement of a rectangle into a partial layout using BLi routine



Bottom-Left Fill, BLF, (Chazelle [46]), $O(n^2)$:
BLF is a modified version of the BL placement routine. BLF places rectangles by searching a list of location points that indicate potential positions where rectangles may be placed. These points are maintained in a bottom left ordering sequence. The algorithm starts with the lowest and leftmost point, where the rectangle is placed and left justified. Then, the rectangle is checked for overlap with any other

rectangles that form the partial layout in the bin. If it does not overlap, the rectangle is placed and the point list is updated to indicate any new points. If the rectangle overlaps, the next point list is selected until the rectangle can be placed without overlap occurring or a new bin is initialised if no bin can accommodate it. Figure 3.7 demonstrates the placement policy using the same ordered list of rectangles as in Figure 3.4. Since the generation of the layout is based on the allocation of the lowest sufficiently large area in the partial layout rather than on a series of bottom left moves, it is capable of filling existing gaps in the packing pattern. Compared to the BL and BLi routine, this method results in a denser packing pattern.

Figure 3.7: Placement of a rectangle into a partial layout using BLF routine



Hopper and Turton [153] performed a series of computational experiments between the BL and BLF and found that BLF outperformed BL by up to 25%. Moreover, preordering the rectangles in non-increasing width or height for both placement routines increased the packing quality by up to 10% compared to random sequence.

Alternate Direction, AD, (Lodi *et al.* [194]), $O(n^3)$:

The routine starts by sorting the rectangles according to non-increasing height, and by computing a lower bound, L on the optimal solution value. Then, L bins are initialised by packing on their bottom a subset of the rectangles, following a BFD routine. As an example, consider the 12 rectangles shown in Figure 3.8 with $L = 2$. Rectangles $1, 2, 3, 7,$ and, 9 are packed into the initialised bins with a BFD routine (Figure 3.9). The remaining rectangles are packed, one bin at a time, into

bands according to the current *direction* associated with the bin. In this case, the current direction is "from right to left". If the direction is "*from left to right*" ("*from right to left*"):

- the first rectangle of the band is packed with the left (*right*) edge touching the left (*right*) edge of the bin, in the lowest possible position;
- each subsequent rectangle is packed with its left (*right*) edge touching the right (*left*) edge of the previous rectangle in the band, in the lowest position.

Once no rectangle can be packed in either direction in the current bin, the next initialised bin becomes the current one. If no rectangle can be packed into any of the initialised bin, a new bin is opened. Figure 3.9 shows the solution found by AD where the rectangles used are as described in Figure 3.8.

Figure 3.8: Bin and Item Dimensions

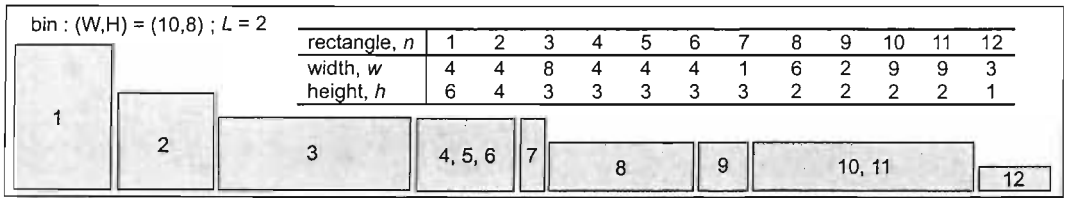
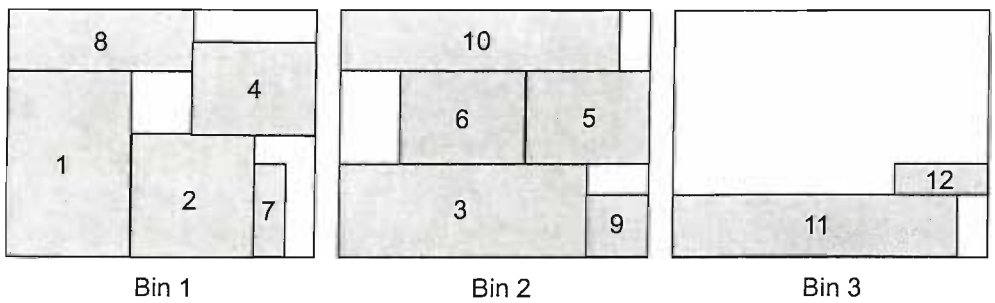


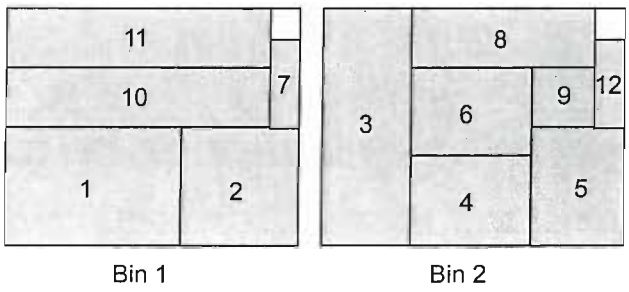
Figure 3.9: Solution found by AD routine



Touching Perimeter, TP, (Lodi *et al.* [194]), $O(n^3)$:

This routine is designed for the non-oriented case where the rectangles to be packed may be rotated by 90°. It starts by sorting the rectangles in non-increasing area and by horizontally orienting them. It then initialises L bins, where L is the lower bound, and packs one rectangle at a time, either in the existing bin, or by initialising a new one. The choice of the bin and of the packing position is done by evaluating a score (percentage of the rectangle perimeter which touches the bin and the other items that are already packed). For each candidate packing position, the score is evaluated twice, for the two rectangle orientations (if both are feasible). The position with the highest score is selected and ties are broken by choosing the bin with the maximum packed area. Figure 3.10 shows the solution found by TP using the same example in Figure 3.8. The rectangles are sorted by TP as $(1, 3, 10, 11, 2, 4, 5, 6, 8, 9, 7, 12)$, with rectangle 1 and 7 rotated 90° before the packing commences.

Figure 3.10: Solution found by TP routine



Two-Phase Algorithms

Hybrid First-Fit, HFF, (Chung *et al.* [54]), $O(n \log n)$:

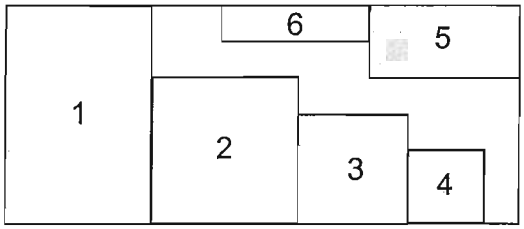
In the first phase, a strip packing is obtained through the First-Fit Decreasing Height (FFDH). Let $H_1, H_2, \ldots$ be the height of the resulting levels in a single strip, and observe that $H_1 \geq H_2 \geq \ldots$. A finite bin packing solution is then obtained by solving a one-dimensional bin packing problem (with rectangle sizes H_i and bin capacity H) through the FFD algorithm: initialise bin 1 to pack level 1, and, for $i = 2, 3, \ldots$, pack the current level i into the lowest indexed bin where it fits, if any. If no bin can accommodate i, initialise a new bin.

Finite Best-Strip, FBS, (Berkey and Wang [32]), $O(n \log n)$:
The first phase is performed by first packing the rectangles into levels in an open ended strip using the BF routine to select the level for packing the next rectangle. In the second phase, a one-dimensional bin packing problem is solved through the BFD routine: a level is packed in that bin, among those where it fits, if any, for which the unused vertical space is a minimum, or by initialising a new bin.

Hybrid Next-Fit, HNF, (Frenk and Galambos [106]), $O(n \log n)$:
NFD is adopted in the first phase to pack the rectangles into levels in an open ended strip. In the second phase, a one-dimensional bin packing problem is solved through the NFD algorithm: a level is packed in the current bin if it fits, or otherwise, on a new level, created either in the current bin (if possible), or in a new one.

Floor-Ceiling, FC, (Lodi *et al.* [194]), $O(n^3)$:
This routine can be applied to both oriented and non-oriented cases. The rectangles are initially sorted in non-increasing order of their shortest edge, and horizontal oriented (for non-oriented case only). We first denote the horizontal line defined by the top/bottom edge of the tallest rectangle packed on a level as the ceiling/floor of the level. In the first phase, the current rectangle is packed, in order of preference:
– on a floor, according to a Best-Fit strategy, or
– on a ceiling (if the rectangle cannot be packed on the floor below),
– on the floor of a new level.
In the second phase, the levels are packed into finite bins, either through the BFD algorithm or by using an exact algorithm for the one-dimensional bin packing problem. A possible FC packing pattern is shown in Figure 3.11.

Figure 3.11: Floor Ceiling



Knapsack Problem, KP, (Lodi *et al.* [194]), $O(n^3)$:
Start by sorting the rectangles in non-increasing height such that $h_j \geq h_{j+1}$ for $j = 1, 2, \ldots, n-1$. At each iteration of the strip packing phase, a new level is initialised with the tallest unpacked rectangle, say j^*. The level packing is then completed by solving an instance of knapsack problem having an element for each unpacked rectangle j, with profit $p_j = w_j h_j$, cost $c_j = w_j$ and capacity $q = W - w_{j^*}$. The problem is to select a subset of rectangles in which the total cost does not exceed q, and the total profit is a maximum. In the second phase, the levels are packed into finite bins through a one-dimensional bin packing problem.

Lodi *et al.* [194] compare the AD, TP, FC and KP placement routines with the FFF and FBS placement routines proposed by Berkey and Wang [32] on a series of computational experiments using benchmark problem instances that include up to 100 rectangles. The placement routines are compared in different combinations of requirements based on the rectangles' orientation and guillotine cuts constraint. Computational results show that their proposed placement routines outperformed both FFF and FBS routines with the TP performs the best.

3.4.2.2 Exact Approaches and Lower Bounds

Much of the work on the exact approaches for SBSBPP concentrate on the one-dimensional case. For example, linear programming (by Valério de Carvalho [68] and Applegate *et al.* [11]), column generation algorithm and branch and bound approaches (by Valério de Carvalho [67] and Vanderbeck [269]). These approaches are too wide to cover in one section and are beyond the scope of this study.

Therefore, as mentioned earlier, we limit the survey to the exact methods for the two-dimensional cases.

Gilmore and Gomory [114] made a first attempt at modelling the 2DRSBS-BPP by extending their model for the one-dimensional stock cutting problem (in [112, 113]). Their approach is based on the concept of *pattern*. Given the set of rectangles and the bins, they define a pattern as a subset of the rectangles that can be loaded into the bin without causing an overlapping of the rectangles themselves. Considering that, in the optimal solution, one pattern is used for each bin, the objective of minimising the number of patterns used is equal to minimising the number of bins used in the solution. In this model, the authors develop a column generation algorithm to help solve the problem.

In 1998, Martello and Vigo [203] improve the lower bounds proposed by Martello and Toth [202] for the oriented 2DRSBSBPP which are used within a branch and bound (B&B) approach. They use both FFF and FBF heuristic placement routines to initialise a feasible solution for the B&B approach. Their B&B approach is based on a two level branching scheme:
– *outer branch-decision tree*: at each iteration node, a rectangle is assigned to a bin without specifying its actual position; and
– *inner branch-decision tree*: a feasible packing (if any) for the rectangles currently assigned to a bin is determined, possibly through enumeration of all the possible patterns.
They show that the worst-case performance ratio of the lower bounds is $\frac{2}{3}OPT$ (OPT = optimal solution). The derivation of the lower bounds are discussed in detail in Section 3.4.3.1. In the paper, the well known continuous lower bound for the problem is analysed and the worst-case performance ratio is calculated as $\frac{1}{4}OPT$. Computational results for the benchmark problem instances show that the B&B is capable of finding the optimal solution for problems involving up to 120 rectangles.

Fekete and Schepers [91, 92] propose a generic approach for obtaining fast lower bounds for the oriented case, based on dual feasible functions. Worst-case analysis shows that the asymptotic worst-case performance ratio of the lower bounds is $\frac{3}{4}OPT$ and can be implemented in linear time (i.e. $O(n)$) if the given n rectangles are sorted by size. Computational results illustrate that the lower bounds outperform the lower bounds proposed by Martello and Vigo [203].

Dell'Amico *et al.* [69] present a lower bound for the non-oriented case of the problem which is then used within an B&B approach developed by Martello and Vigo [203]. Before the bound is calculated, each rectangle is replaced by a number of square items by cutting it with a CUTSQ procedure (see Figure 3.12). Unit squares of size one are not produced, as they are not use in the subsequent lower bound computations. The derivation of the lower bound is discussed in detail in Section 3.4.3.2 which subsequently becomes the lower bound we employ for the computational experiments in Section 6.7. The computational results for instances up to 100 rectangles show that the proposed bound is considerably better than the continuous lower bound.

Boschetti and Mingozzi [34] improve the lower bounds proposed by Martello and Vigo [203] for the oriented case of 2DRSBSBPP. They show that the lower bounds also dominate the lower bounds proposed by Fekete and Schepers [91, 92]. However, the main disadvantage of their lower bounds lie in the computational complexity of the bounds. In the same year, Boschetti and Mingozzi [35] also devise tighter lower bounds for the non-oriented case. Computational results show the effectiveness of the lower bounds which dominate the bounds proposed by Dell'Amico *et al.* [69].

In 2003, Pisinger and Sigurd [230] propose a hybrid branch and price/constraint programming algorithm for solving the oriented 2DRSBSBPP. They use the column generation principle of Gilmore and Gomory and solve the specific pricing problem by means of constraint programming. They also propose new lower bounds using the delayed column generation. The computational results show that the lower bounds obtained through delayed column generation are tighter

than bounds proposed by Martello and Vigo [203] and Fekete and Schepers [91, 92].

In 2004, Puchinger and Raidl [236, 237] present an integer linear programming formulations solved by CPLEX for both restricted and unrestricted versions of the 2DRSBSBPP. Furthermore, a branch and price approach is proposed by formulating the original problem as a set covering problem. Fast column generation is performed by applying a hierarchy of four methods, namely a greedy heuristic, an evolutionary algorithm, and both a restricted and unrestricted integer linear programming for the pricing problem. Extensive computational experiments are performed and the results show that the lower bounds obtained by column generation are strong.

3.4.2.3 Local Search Methods

Since the 2DRSBSBPP belongs to the class of NP-hard problems, exact approaches are bound to work well for small to medium sized problem instances only. Real world applications which include up to thousands of rectangles have to be solved heuristically or by local search methods. Much of the research on the local search methods for SBSBPP focus on the one-dimensional case. These include Ant Colony Optimisation (by Brugger *et al.* [37] and Levine and Ducatelle [186]), Genetic Algorithms (by Falkenauer [87, 88], Runarsson *et al.* [245], and Iima and Yakawa [155]), and Simulated Annealing (by Kämpke [165]). Since we concentrate on the two-dimensional case, these approaches are beyond the scope of this study. For the remainder of this subsection, we limit the survey to the use of genetic algorithms and tabu search for the two-dimensional cases.

Genetic Algorithms

A common feature found in most Genetic Algorithms (GAs) developed for SBSBPP is their two-stage approach, where a GA is combined with a heuristic placement routine. In this two-stage approach, a GA manipulates the encoded solutions, which are then evaluated by a decoding algorithm transforming the packing sequence into the corresponding physical layout. Since domain knowledge is built

into the decoding procedure, the size of the search space can be reduced. The search space is further restricted to feasible solutions only. As a result, the packing strategy generates only non-overlapping layouts.

The first researcher to implement GAs in the domain of packing was Smith [255] in 1985. He applies a GA to a two-dimensional rectangular packing problem with fixed orientation. The objective of his GA is to put as many blocks into a single rectangular region as possible. He uses permutations of rectangles to encode the instances. Thus, the original problem becomes a sequencing problem and heuristics are used to transform those permutation into packing schemes. Experimental results have shown that his GA can produce the same packing density 300 times faster than a dynamic programme.

In 1994, Hwang *et al.* [154] design a GA for the 2DRSBSBPP where the rectangles are represented by a permutation and packed into the bins by a two-stage heuristic. In the first stage, the level-oriented FF placement routine places rectangles onto an open ended strip of unlimited height, constructing the layout as a sequence of levels. Each level forms a rectangular block containing one or more rectangles. In the next stage, the packed strip is decomposed at each level forming a block of rectangles of fixed width and with the height equal to the height of the level. The blocks are then packed using the FFD or the BFD routine into a fixed size bin reducing the problem to a one-dimensional problem. The authors implement two GAs using the FFD and the BFD routines in the decoding stage. Comparisons are made with the HFF routine, which is a combination of the FFDH and FFD routines. The GAs consistently outperformed the heuristic one (HFF), whereby the one using the BFD routine performed best.

A year later, Kröger [172] develops a sequential and a parallel GA to solve a constrained 2DRSBSBPP, which demands a guillotine restriction for the valid packing layout. He uses a problem specific encoding which represents the essential structure of a packing scheme by a binary tree. The major motivation for the crossover operator is to combine as many partial solutions (subtrees) from both parents as possible, thus preserving the main characteristics and providing

a systematic continuation of the search. The author compares the sequential GA with the FF placement routine, simulated annealing, and a random search strategy which randomly generates valid preorder strings. The proposed algorithm produces high quality solutions compared with other heuristic methods, even in its sequential version. The concept of a *meta-rectangle* is also proposed in the paper. Due to the guillotine constraint of the packing schemes, each group of neighbouring rectangles forms a partial arrangement with still a rectangular shape (meta-rectangle). Then, each meta-rectangle temporarily *freezes* a hyperplane of an existent solution. Thus, the complexity of a problem is reduced and the algorithm's search can be guided into the most promising parts of the solution space.

Hopper and Turton [151] propose two GAs for the rectangle packing problem in 1999. The GAs as well as the BL and BLF placement routines have been tested on a number of packing problems. The GA combined with the BLF routine outperforms the GA using the BL routine as well as the heuristic placement routines. They conclude that, the performance difference between the two GAs implementation is due to the improved placement routine.

Two years later, Hopper and Turton [153] compare several local search algorithms including GA, simulated annealing (SA), naïve evolution (NE), hill climbing and random search. The authors show that the combination between the GA, SA, and NE with the BLF routine all gave similar results but they are better than the combinations with the BL routine as well as the heuristic routine with height or width sorted input sequence.

Tabu Search

Lodi *et al.* [193, 194] develop an Unified Tabu Search (UTS) code for multi dimensional rectangular SBSBPP in 1999. The main characteristic of the unified framework in tabu search explained in [193, 194] is an adoption of a search scheme and a neighbourhood which are independent of the specific packing problem to be solved. The UTS is based on two possible neighbourhood moves. Both neighbourhoods consist of moves involving the rectangles of a particular bin, which is called

target bin and is defined in Section 6.6.1. At each iteration, the algorithm considers a rectangle j currently packed in the target bin b and tries to remove j from b. The first neighbourhood move attempts to directly pack j into a different bin. In the second neighbourhood move, the algorithm tries to recombine the rectangles of two different bins so that one of them can accommodate j. The approach is discussed in more details in Section 6.6.1.

Lodi *et al.* [193] consider an oriented 2DRSBSBPP with guillotine cuts constraint. They propose a simple deterministic algorithm which is used in the initialisation of the UTS approach. The proposed algorithm runs in $O(n \log n)$ time with a worst-case performance ratio of 4. The algorithm is based on a technique developed by Martello and Vigo [203] in proving the worst-case performance of the continuous lower bound for 2DRSBSBPP. The UTS approach is developed by applying two simple heuristic placement routines (i.e. FFF and FBS) in the neighbourhood search. Their UTS algorithm outperformed both heuristics for problem instances up to 120 rectangles. Also, the comparison with a B&B approach is comparable for instances that include up to 100 rectangles.

A year later, Lodi *et al.* [194] introduce four new heuristic placement routines: FC, KP, AD, and TP, as described earlier which are developed according to different combinations of requirements based on the rectangle's orientation and guillotine cuts constraint. The proposed heuristic placement routines are then used in the UTS approach to generate an initial layout. Computational results on the benchmark problem instances that include up to 100 rectangles show that the proposed heuristic placement routines outperformed FFF and FBS routines (by Berkey and Wang [32]). They conclude that the UTS, in general, has improved the initial deterministic solution produced by the heuristic placement routine and is comparable to a B&B approach. Further investigation shows that both papers give similar results.

3.4.3 Lower Bounds for 2DRSBSBPP

In this subsection, we present lower bounds for 2DRSBSBPP in both oriented and non-oriented cases. The lower bounds are obtained from the bounds proposed by Martello and Vigo [203] (oriented) and Dell'Amico *et al.* [69] (non-oriented) that are used in their branch and bound algorithm. These derivation of the lower bounds are extracted from their papers.

3.4.3.1 Oriented Rectangular

We first define the bin with dimensions (W,H) and rectangles with width w_j, and height h_j. Let, $j \in J = \{1,\ldots,n\}$, where n is the total rectangles to be placed. The simplest bound for 2DRSBSBPP is the *Continuous Lower Bound* L_0, which can be computed in $O(n)$ time and has a worst-case performance ratio of $\frac{1}{4}$ (Martello and Vigo [203]):

$$L_0 = \left\lceil \frac{\sum_{j=1}^{n} w_j h_j}{WH} \right\rceil . \tag{3.18}$$

The idea is to calculate the total area of the rectangles and divide it by the area of a bin. The rounded up value obtained is a valid lower bound. This bound does not take into account the fact that many rectangles cannot be packed together in a bin. More accurate lower bounds which explicitly take into consideration both dimensions of the rectangles are introduced by Martello and Vigo [203].

We first present a lower bound that can be computed in linear time. Let $J^W = \{j \in J : w_j > \frac{1}{2}W\}$ and observe that no two rectangles of J^W may be packed side by side into a bin. Given any integer p, with $1 \leq p \leq \frac{1}{2}H$, let

$$\begin{aligned} J_1 &= \{j \in J^W : h_j > H - p\}, \\ J_2 &= \{j \in J^W : H - p \geq h_j > \frac{1}{2}H\}, \\ J_3 &= \{j \in J^W : \frac{1}{2}H \geq h_j \geq p\}. \end{aligned} \tag{3.19}$$

Note that no two rectangles of $J_1 \bigcup J_2$ may be packed into the same bin, so $|J_1 \bigcup J_2|$ is a valid lower bound on the optimal solution. The lower bound can be strengthen

by observing that no rectangle in J_3 will fit into a bin used for a rectangle in J_1. Hence, for any given integer p, with $1 \leq p \leq \frac{1}{2}H$, a valid lower bound on the optimal solution is

$$L_1^W(p) = \max\{L_\alpha^W(p), L_\beta^W(p)\}, \text{ where} \tag{3.20}$$

$$L_\alpha^W(p) = |J_1 \cup J_2| + \max\left\{0, \left\lceil \frac{\sum_{j\in J_3} h_j - (|J_2|H - \sum_{j\in J_2} h_j)}{H} \right\rceil\right\}, \tag{3.21}$$

$$L_\beta^W(p) = |J_1 \cup J_2| + \max\left\{0, \left\lceil \frac{|J_3| - \sum_{j\in J_2} \left\lfloor \frac{H-h_j}{p} \right\rfloor}{\left\lfloor \frac{H}{p} \right\rfloor} \right\rceil\right\}. \tag{3.22}$$

Both $L_\alpha^W(p)$ and $L_\beta^W(p)$ are obtained by adding to $|J_1 \bigcup J_2|$ the minimum number of additional bins needed for the rectangles of J_3. Thus, a valid lower bound on the optimal solution is

$$L_1^W = \max_{1\leq p\leq (1/2)H}\{L_1^W(p)\}. \tag{3.23}$$

The overall computation of L_1^W can be performed in $O(n^2)$ time, since $L_\alpha^W(p)$ and $L_\beta^W(p)$ can be determined in $O(n)$ time.

Now let $J^H = \{j \in J : h_j > \frac{1}{2}H\}$, $1 \leq p \leq \frac{1}{2}W$ and

$$\begin{aligned} J_1 &= \{j \in J^H : w_j > W - p\}, \\ J_2 &= \{j \in J^H : W - p \geq w_j > \frac{1}{2}W\}, \\ J_3 &= \{j \in J^H : \frac{1}{2}W \geq w_j \geq p\}. \end{aligned} \tag{3.24}$$

It is clear that from the above results, a valid lower bound on the optimal solution is

$$L_1^H = \max_{1\leq p\leq (1/2)W}\{L_1^H(p)\}, \text{ where} \tag{3.25}$$

$$L_1^H(p) = \max\{L_\alpha^H(p), L_\beta^H(p)\}, \text{ where} \tag{3.26}$$

$$L_\alpha^H(p) = |J_1 \cup J_2| + \max\left\{0, \left\lceil \frac{\sum_{j\in J_3} w_j - (|J_2|W - \sum_{j\in J_2} w_j)}{W} \right\rceil\right\}, \tag{3.27}$$

$$L_\beta^H(p) = |J_1 \cup J_2| + \max\left\{0, \left\lceil \frac{|J_3| - \sum_{j\in J_2} \left\lfloor \frac{W-w_j}{p} \right\rfloor}{\left\lfloor \frac{W}{p} \right\rfloor} \right\rceil\right\}. \tag{3.28}$$

Thus, an overall lower bound on the optimal solution can be computed in $O(n^2)$ time as

$$L_1 = \max\{L_1^W, L_1^H\}. \tag{3.29}$$

In the remainder of this subsection, we present lower bounds which explicitly take into account both dimensions of the items. Given an integer value q, $1 \leq q \leq \frac{1}{2}W$, let

$$\begin{aligned} K_1 &= \{j \in J : w_j > W - q\}, \\ K_2 &= \{j \in J : W - q \geq w_j > \frac{1}{2}W\}, \\ K_3 &= \{j \in J : \frac{1}{2}W \geq w_j \geq q\}. \end{aligned} \tag{3.30}$$

First observe that $K_1 \bigcup K_2 \equiv J^W$ and is independent of q. Hence a valid lower bound on the number of bins is needed for the rectangles in $K_1 \bigcup K_2$ is given by L_1^W. We can tighten this value by considering the rectangles in K_3 and observing that none of them can be packed beside an rectangle of K_1. Then a valid lower bound on the optimal solution is

$$L_2^W = \max_{1 \leq q \leq (1/2)W} \{L_2^W(q)\}, \text{ where} \tag{3.31}$$

$$L_2^W(q) = L_1^W + \max\left\{0, \left\lceil \frac{\sum_{j \in K_2 \cup K_3} h_j w_j - (HL_1^W - \sum_{j \in K_1} h_j)W}{HW} \right\rceil\right\}, \tag{3.32}$$

and can be computed in $O(n^2)$ time.

Similarly, the L_2^H can also be obtained as follows. Let, $1 \leq q \leq \frac{1}{2}H$ and

$$\begin{aligned} K_1 &= \{j \in J : h_j > H - q\}, \\ K_2 &= \{j \in J : H - q \geq h_j > \frac{1}{2}H\}, \\ K_3 &= \{j \in J : \frac{1}{2}H \geq h_j \geq q\}. \end{aligned} \tag{3.33}$$

A valid lower bound on the optimal solution is

$$L_2^H = \max_{1 \leq q \leq (1/2)H} \{L_2^H(q)\}, \text{ where} \tag{3.34}$$

$$L_2^H(q) = L_1^H + \max\left\{0, \left\lceil \frac{\sum_{j \in K_2 \cup K_3} h_j w_j - (WL_1^H - \sum_{j \in K_1} h_j)H}{HW} \right\rceil\right\}. \tag{3.35}$$

Thus, the overall lower bound,

$$L_2 = \max\{L_2^W, L_2^H\}. \tag{3.36}$$

It is worth mentioning that lower bounds L_1 and L_2 have the same worst-case time complexity, but the computation of L_2^W (L_2^H) requires the value of L_1^W (L_1^H). Thus, the average computing time required for L_2 is approximately twice the time required for L_1.

Martello and Vigo [203] further describe a lower bound which is computationally more expensive, but can in some cases improve the previous one. Given any pair of integers (p, q), with $1 \leq p \leq \frac{1}{2}H$ and $1 \leq q \leq \frac{1}{2}W$. Let,

$$\begin{aligned} I_1 &= \{j \in J : h_j > H - p \quad \text{and} \quad w_j > W - q\}, \\ I_2 &= \{j \in J \backslash I_1 : h_j > \frac{1}{2}H \quad \text{and} \quad w_j > \frac{1}{2}W\}, \\ I_3 &= \{j \in J : \frac{1}{2}H \geq h_j \geq p \quad \text{and} \quad \frac{1}{2}W \geq w_j \geq q\}. \end{aligned} \tag{3.37}$$

Observe that $I_1 \bigcup I_2$ is independent of (p, q), that no two items of $I_1 \bigcup I_2$ may be packed into the same bin, and that no rectangle in I_3 will fit into a bin containing a rectangle in I_1. Given a bin $W \times H$ containing a rectangle of size $w_j \times h_j$, the maximum number of $p \times q$ items that can be packed into the bins is

$$m(j, p, q) = \left\lfloor \frac{H}{p} \right\rfloor \left\lfloor \frac{W - w_j}{q} \right\rfloor + \left\lfloor \frac{W}{q} \right\rfloor \left\lfloor \frac{H - h_j}{p} \right\rfloor - \left\lfloor \frac{H - h_j}{p} \right\rfloor \left\lfloor \frac{W - w_j}{q} \right\rfloor. \tag{3.38}$$

Hence, a valid lower bound on the optimal solution that can be computed in $O(n^3)$ time is given by

$$L_3 = \max_{1 \leq p \leq (1/2)H, 1 \leq q \leq (1/2)W} \{L_3(p, q)\}, \text{ where} \tag{3.39}$$

$$L_3(p, q) = |I_1 \cup I_2| + \max\left\{0, \left\lceil \frac{|I_3| - \sum_{j \in I_2} m(j, p, q)}{\left\lfloor \frac{H}{p} \right\rfloor \left\lfloor \frac{W}{q} \right\rfloor} \right\rceil\right\}. \tag{3.40}$$

Thus, the overall lower bound for oriented 2DRSBSBPP is

$$L_4 = \max\{L_2, L_3\}. \tag{3.41}$$

3.4.3.2 Non-Oriented Rectangular

In the following subsection, we present a lower bound for non-oriented 2DRSBS-BPP proposed by Dell'Amico *et al.* [69]. Without loss of generality, we assume that all input data are positive integers and bins and items are given in '*horizontal*' orientation, i.e. that $W \geq H$ and $w_j \geq h_j$ for $j = 1, \ldots, n$. In order to ensure feasibility, we assume that $w_j \leq W$ and $h_j \leq H$ for $j = 1, \ldots, n$.

A valid lower bound comes from the following relaxation. Given an instance of the problem, we replace each rectangle by a number of square items obtained by appropriately cutting it using procedure CUTSQ as described in Figure 3.12. For the resulting instance, there is no difference between allowing 90° rotation or not. Note that in the case of a rectangle j where $w_j = h_j$, there is no need to apply the CUTSQ procedure. Note that squares of size one are not produced, as they are of no use in the subsequent lower bound computations.

Figure 3.12: Procedure CUTSQ (Dell'Amico *et al.* [69])

```
procedure CUTSQ:
    J_SQ := ∅;
    for j := 1 to n do
        S := ∅;
        while h_j > 1 do
            k := ⌊w_j/h_j⌋;
            add k squares of size h_j to S;
            w_j := w_j − k h_j;
            swap w_j and h_j
        end while;
        J_SQ := J_SQ ∪ S
    end for
end.
```

Let $M = \{1, \ldots, m\}$ where $m = |J_{SQ}|$ is the number of resulting squares, and let l_j ($j \in M$) be the resulting edge sizes. Given an integer value q, $0 \leq q \leq \frac{1}{2}H$, let

$$\begin{aligned} S_1 &= \{j \in M : l_j > W - q\}, \\ S_2 &= \{j \in M : W - q \geq l_j > \frac{1}{2}W\}, \\ S_3 &= \{j \in M : \frac{1}{2}W \geq l_j > \frac{1}{2}H\}, \\ S_4 &= \{j \in M : \frac{1}{2}H \geq l_j \geq q\}. \end{aligned} \tag{3.42}$$

Recall that $W \geq H$, and observe that, by definition:

- each square of $S_1 \bigcup S_2$ requires a separate bin;
- no square of S_3 can be packed into a bin containing a square of S_1;
- no square of S_3 can be packed over a square of S_2;
- at most one square of S_3 can be packed beside a square of S_2.

Let,

$$\tilde{L} = |S_2| + \max\left\{ \left\lceil \frac{\sum_{j \in S_3 \backslash \bar{S}_3} l_j}{W} \right\rceil, \left\lceil \frac{|S_3 \backslash \bar{S}_3|}{\left\lfloor \frac{W}{\lfloor H/2+1 \rfloor} \right\rfloor} \right\rceil \right\}, \tag{3.43}$$

where

$\bar{S}_3$ is the set of the largest squares of S_3 that can be packed into the bins that pack the squares of S_2; and

$S_3 \backslash \bar{S}_3$ is the set of the squares in S_3 that do not belongs to $\bar{S}_3$.

A valid lower bound on the optimal solution value is

$$L_5 = \max_{0 \le q \le (1/2)H} \{L(q)\}, \text{ where} \tag{3.44}$$

$$L(q) = |S_1| + \tilde{L} + \max\left\{0, \left\lceil \frac{\sum_{j \in S_2 \cup S_3 \cup S_4} l_j^2 - (WH\tilde{L} - \sum_{j \in S_{23}} l_j(H - l_j))}{WH} \right\rceil\right\}, \tag{3.45}$$

where $S_{23} = \{j \in S_2 \cup S_3 : l_j > H - q\}$ and can be computed in $O(m)$ time.

If $W = H$, Equation 3.45 can be simplifies to

$$L'(q) = |S_1 \cup S_2| + \max\left\{0, \left\lceil \frac{\sum_{j \in S_2 \cup S_4} l_j^2}{W^2} - |S_2| \right\rceil\right\}. \tag{3.46}$$

Dell'Amico *et al.* [69] also mention that for instances where some rectangles that cannot be rotated (i.e. $w_j > H$ for some j), an alternative bound can be obtained as follows. Let $T = \{j : w_j > H\}$, apply CUTSQ only to the items of $\{1, \ldots, n\} \backslash T$ and compute, for the instance defined by T plus the resulting squares, any lower bound for the oriented case. By using lower bound L_4 and improve L_5 by setting $L_5 = \max\{L_5, L_4\}$, they conclude that the overall lower bound for non-oriented 2DRSBSBPP is given by

$$LB_o = \max\{L_0, L_5\}. \tag{3.47}$$

3.5 Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is a classical combinatorial optimisation problem. It was first documented as early as 1759 by Euler, whose interest was solving the knight's tour problem (as cited by Hoffman and Wolfe [146]). A correct solution would have a knight visit each of the 64 squares on a chessboard exactly once on its tour. The term '*travelling salesman*' was first used in 1832, in a German book written by a veteran travelling salesman. Mathematical problems related to the TSP were treated in the 1800s by Sir William Rowan Hamilton on solving the Hamiltonian cycle in Graph Theory. The general form of the TSP was first stated by Karl Menger in 1930s, but it is not until 1954 when the first mathematical formulation for the TSP appears courtesy of Dantzig *et al.* [63]. Since then, a huge amount of research has been done on this problem over the years as summarised in Table 3.6. This table represents the latest problem instances for the TSP that are solved to optimality. The most widely used collection of TSP instances in recent computational studies is Gerhard Reinelt's TSPLIB [266] test sets. The TSPLIB is made up of over 100 instances arising from industrial, geographic, and academic sources. To supplement this collection, further instances are available in the National TSP [216], World TSP [283] and VLSI TSP [270] collections. The long history of the TSP can be found in Hoffman and Wolfe [146].

The TSP is one of the most studied combinatorial optimisation problems of our time and is simple to state but very difficult to solve. The problem has been formulated in several different ways (see Langevin *et al.* [176]). We use the following formulation as stated by Johnson and McGeoch [161]:
Given a set $\{c_1, c_2, \ldots, c_n\}$ of *cities* and for each pair $\{c_i, c_j\}$ of distinct cities, there exist a *distance* $d(c_i, c_j)$. The objective is to find an ordering π of the cities that minimises the *tour length*, i.e. the quantity

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}). \tag{3.48}$$

The TSP is classified as a class of NP-Complete problems by Garey and Johnson [108]. The problem is *symmetric* (STSP) if, and only if, $d(c_i, c_j) = d(c_j, c_i)$ for $1 \leq i, j \leq n$; otherwise it is *asymmetric* (ATSP). A special case of STSP is Euclidean TSP where the distance between the cities satisfy the '*triangle inequality*': $d(c_i, c_j) + d(c_j, c_k) \geq d(c_i, c_k)$ for all $i, j, k \in n$. The cities are given as points with integer coordinates in the two-dimensional plane, and the distance are computed according to the Euclidean metric.

Table 3.6: Milestones in the solution of TSP instances solved to optimality (extracted from [72])

Year	Research Team	Size of Instance	Name
1954	G. Dantzig, R. Fulkerson, and S. Johnson	49 cities[a]	dantzig42
1971	M. Held and R.M. Karp	64 cities	64 random points
1975	P.M. Camerini, L. Fratta, and F. Maffioli	67 cities	67 random points
1977	M. Grötschel	120 cities	gr120
1980	H. Crowder and M.W. Padberg	318 cities	lin318
1987	M. Padberg and G. Rinaldi	532 cities	att532
1987	M. Grötschel and O. Holland	666 cities	gr666
1987	M. Padberg and G. Rinaldi	2,392 cities	pr2392
1994	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	7,397 cities	pla7397
1998	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	13,509 cities	usa13509
2001	D. Applegate, R. Bixby, V. Chvátal, and W. Cook	15,112 cities	d15112
2004	D. Applegate, R. Bixby, V. Chvátal, W. Cook, and K. Helsgaun	24,978 cities	sw24978

[a] optimal tour through the 42 cities uses roads that pass through the 7 cities that are excluded.

TSP often comes up as a subproblem in more complex combinatorial problems, the best known and important one of which is the Vehicle Routing Problem (VRP), that is, the problem of determining for a fleet of vehicles which customers should be served by each vehicle and in what order each vehicle should visit the customers assigned to it. Although transportation applications are the most natural setting for the TSP, the simplicity of the model has led to many interesting applications in other areas. For example,

- Computer wiring (Lenstra and Rinnooy Kan [183]): some computer systems can be described as modules with pins attached to them. It is often desired to link these pins by means of wires, so that exactly two wires are attached to each pin and total wire length is minimised.
- X-ray Crystallography (Bland and Shallcross [33]): some experiments in crystallography consist of taking a large number of X-ray intensity measurements on crystals by means of a detector. Each measurement requires that a sample of the crystal be mounted on an apparatus and that the detector be positioned appropriately. The order in which the various measurements on a given crystal are made can be seen as the solution of a TSP.
- Hole drilling (Reinelt [241]): the holes to be drilled on boards or metallic sheets are the cities, and the tour is the distance it takes to move the drill head from one hole to the next.

Some excellent surveys of published research on the TSP can be found in Lawler *et al.* [180], Reinelt [242], Jünger *et al.* [164], and Johnson and McGeoch [161].

3.5.1 Heuristic Methods for TSP

In this subsection, we address some of the well known heuristic methods for solving the TSP. The objective of this subsection is to give a general overview of the approaches used to solve the TSP rather that comparing the effectiveness among the approaches used in solving the problem. Generally speaking, TSP heuristics can be classified as *tour construction*, *tour improvement*, and *composite* heuristics.

3.5.1.1 Tour Construction Heuristics

In brief, tour construction procedures gradually build a tour by selecting each vertex in turn and by inserting them one by one into the current tour. Various techniques are used for selecting the next vertex and for identifying the best insertion place. While the procedures are very fast, the solution quality is usually rather poor.

Nearest Neighbour:

Bellmore and Nemhauser [30] propose this procedure where a feasible tour is constructed by including the most advantageous city at each step. This heuristic requires $O(n^2)$ time and the steps are as follow:

S 1: Consider an arbitrary vertex as a starting point.

S 2: Determine the closest vertex to the last vertex considered and include it in the tour. If any vertex has not yet been considered, repeat **S2**.

S 3: Link the last vertex of the tour to the first one.

A possible modification is to consider in turn all n vertices as a starting point. The overall time complexity is then $O(n^3)$ and the resulting tour is generally better.

Insertion:

Rosenkrantz *et al.* [244] consider a class of insertion procedures that use various criteria. The steps are briefly summarised as follow:

S 1: Construct a first tour consisting of two vertices.

S 2: Consider in turn all vertices not yet in the tour. Insert in the tour a vertex chosen with respect to a given criterion, for example:

- **(a)** the vertex yielding the least distance increment,
- **(b)** the vertex closest to the current tour,
- **(c)** the vertex furthest away from the tour,
- **(d)** the vertex forming the largest angle with two consecutive vertices of the tour, etc.

If any vertex has not yet been considered, repeat **S2**.

S 3: Link the last vertex of the tour to the first one.

Depending on the criterion that is used, the complexity of this heuristic varies between $O(n \log n)$ and $O(n^2)$.

Clarke-Wright:

This procedure is derived from a more general VRP proposed by Clarke and Wright [55]. In terms of the TSP, the steps are given as follow:

S 1: Consider an arbitrary vertex as a depot (e.g. city 1) where the tour returns to the depot after each visit to another vertex.

S 2: Calculate the *saving* $s_{ij} = c_{1i} - c_{ij} + c_{j1}$ for all pairs of vertices i and j. Note that the saving is the amount by which the tour would be shortened if the tour went directly from one vertex to the other, bypassing the depot. Order the savings in non-increasing order.

S 3: Starting from the top of the order, perform the bypass so long as it does not create a cycle of non-depot vertices or cause a non-depot vertex to become adjacent to more than two other non-depot vertices.

S 4: Repeat **S3** until there are only two non-depot vertices remaining connected to the depot.

This heuristic can be implemented to run in $O(n^2 \log n)$ time.

Christofides:

Christofides [52] proposes a heuristic that is run in $O(n^3)$ time. The heuristic proceeds as follows:

S 1: Construct a minimum spanning tree T for the set of cities.

S 2: Construct a minimum matching M for the set of all odd degree vertices in T.

S 3: Find an Eulerian tour for the Eulerian graph (i.e. a cycle that passes through each edge exactly once) that is the union of T and M.

A travelling salesman tour can then be constructed by traversing this cycle while taking shortcuts to avoid multiple visited vertices.

3.5.1.2 Tour Improvement Heuristics

These heuristics start with an arbitrary initial tour and then search for improvement by edge exchange method. The method locally modifies the current solution by deleting k edges from the current tour and reconnecting the resulting paths using k new edges so as to generate a new improved solution. Typically, these heuristics are applied iteratively until a local optimum is found. The major drawback of these heuristics is the possibility of becoming trapped at a local optimum.

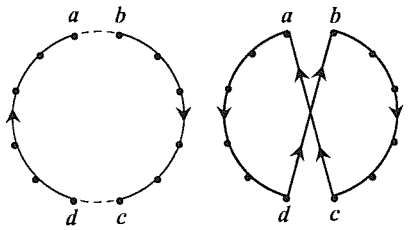
k-Opt:

S 1: Consider an initial tour.

S 2: Remove k edges, thus breaking the tour into k paths. Reconnect the k paths in all possible ways. If any reconnection yields a shorter tour, consider this tour as a new solution and repeat **S2**. **STOP** when no improvement can be obtained.

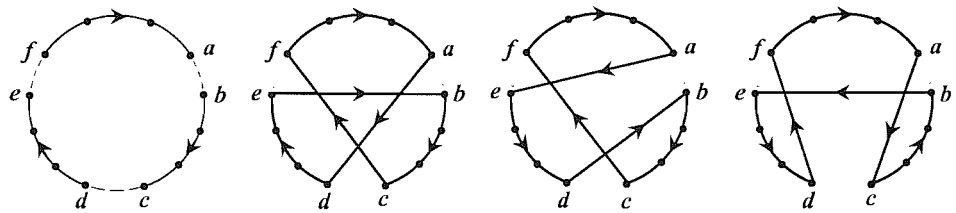
In the case of 2-Opt, a neighbouring solution is obtained from the current solution by deleting two edges, reversing one of the resulting paths and reconnecting the tour (see Figure 3.13). The 2-Opt heuristic is proposed by Croes [62] and requires $O(n^2)$ time.

Figure 3.13: A 2-Opt move: original tour (left) and resulting tour (right)



Lin [189] proposes a 3-Opt move, where three edges are deleted. The three resulting paths are then put together in a new way, possibly reversing one of them (see Figure 3.14). The computational results show that the 3-Opt moves is more effective than 2-Opt moves, though the size of the neighbourhood is larger and hence more time consuming to search. The time complexity for searching the neighbourhood defined by 3-Opt is $O(n^3)$.

Figure 3.14: 3-Opt moves: original tour (far left) and possible resulting tours (right)

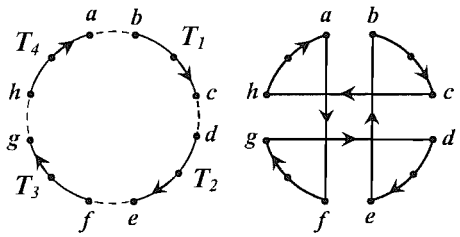


Bentley [31] derives a 2.5-Opt heuristic for geometric TSP, which expands the 2-Opt neighbourhood to include a simple form of 3-Opt move that can be found with little extra effort. In a 2.5-Opt move, one relocates a single city from its current location to a position between two current tour neighbours elsewhere in the tour. This corresponds to the situation where b and c are the same city in Figure 3.14. The 2.5-Opt heuristic achieved a better average tour over 2-Opt but

with a much longer computation time. However, it is still worse than the 3-Opt heuristic.

Lin and Kernighan [190] suggest a 4-Opt move which is also known as the *double-bridge* move. A double-bridge move can be viewed as a combination of two 2-Opt moves. The heuristic starts by breaking four edges in the tour thus forming four paths. Suppose the resulting four paths are $T_1T_2T_3T_4$ in order (see Figure 3.15). The move rearrange these into the new ordering $T_1T_4T_3T_2$ (without reversing any of the paths) and this yields the new tour. Computational experiments show that the algorithms based solely on 4-Opt move did not find noticeable better solutions than the 3-Opt heuristic.

Figure 3.15: A 4-Opt move: original tour (left) and resulting tour (right)



Or-Opt:

Or [221] proposes a simplified exchange procedure requiring only $O(n^2)$ operations at each step, but producing tours nearly as good on average as those obtained with a 3-Opt heuristic. The steps are as follow:

S 1: Consider an initial tour and set $t := 1$ and $s := 3$.

S 2: Remove a chain of s consecutive vertices from the tour, starting with the vertex in position t, and temporarily insert it between all remaining pairs of consecutive vertices on the tour.

- If the temporary insertion yields a shorter tour, implement it immediately, thus defining a new tour. Set $t := 1$ and repeat **S2**.
- If no temporary insertion yields a shorter tour, set $t := t+1$. If $t = n+1$, then proceed to **S3**, otherwise repeat **S2**.

S 3: Set $t := 1$ and $s := s - 1$. If $s > 0$, go to **S2**, otherwise **STOP**.

Lin-Kernighan (LK):

Lin and Kernighan [190] develop a sophisticated edge exchange procedure where the number of edges k, to be exchanged in each step is regarded as a variable. LK uses a very complex neighbourhood search structure which requires $O(n^5)$ time. The LK algorithm was considered for many years to be the 'uncontested champion' of heuristic methods for the TSP (cite in Johnson and McGeoch [161]). The best description of the full details of the LK algorithm is the original paper by Lin Kernighan in 1973.

3.5.1.3 Composite Heuristics

The procedures combine tour construction and tour improvement heuristics. The idea behind a composite heuristic is to obtain a good initial solution quickly and then apply a computationally more expensive improvement heuristic to get to a near optimal solution.

CCAO:

Golden and Stewart [127] design a composite heuristic for symmetrical Euclidean TSP. It exploits a well known property of such problems, namely that in any optimal solution, vertices located on the convex hull of all vertices are visited in the order in which they appear on the convex hull boundary. This heuristic constructs an initial tour consisting of the convex hull of vertices. Vertices not yet on the tour are gradually included by first considering all possible insertions, and then selecting the best move according to a largest angle criterion. The Hamiltonian tour is then improved by the Or-Opt as the post-optimisation procedure. The heuristic can be summarised as follows:

S 1: (**C**: **C**onvex Hull) Define an initial (partial) tour by forming the convex hull of vertices.

S 2: (**C**: **C**heapest Insertion) For each vertex k not yet contained in the tour, identify the two adjacent vertices i_k and j_k on the tour such that $c_{i_k k} + c_{k j_k} - c_{i_k j_k}$ is minimised (c_{ij} = cost or distance from city i to city j).

S 3: (**A**: Largest **A**ngle) Select the vertex k^* that minimises the angle between edges (i_k, k) and (k, j_k) on the tour, and insert it between i_{k^*} and j_{k^*}.

S 4: Repeat **S2** and **S3** until a Hamiltonian tour of all vertices is obtained.

S 5: (**O**: **O**r-Opt) Apply the Or-Opt procedure to the tour and **STOP**.

GENIUS:

Gendreau *et al.* [109] develop an efficient heuristic algorithm called GENIUS, which is a combination of a Generalised Insertion (GENI) procedure and an Unstringing and Stringing (US) post-optimisation routine. This routine consists of removing a vertex from a feasible tour and of inserting it back at different position. There are two insertion and unstringing types in GENI and US procedures respectively. Detailed descriptions of the two insertion and unstringing types can be found in Gendreau *et al.* [109]. The procedures are summarised as follow (extracted from Gendreau *et al.* [109]):

GENI procedure:

S 1: Create an initial tour by selecting an arbitrary subset of three vertices. Initialise the p-neighbourhoods of all vertices (p-neighbourhood = set of the p vertices on the tour closest to v, where v is an arbitrary vertex not yet on the tour).

S 2: Randomly select a vertex v not yet on the tour. Implement the cheapest insertion of v considering the two possible orientations of the tour and the two insertion types. Update the p-neighbourhoods of the all vertices after vertex v is inserted into the tour.

S 3: If all vertices are now part of the tour, **STOP**, otherwise go to **S2**.

US routine:

S 1: Consider an initial tour τ of cost z. Set $\tau^* := \tau$, $z^* := z$ and $t := 1$.

S 2: Starting from tour τ, apply the unstringing and stringing procedures with vertex v_t, considering in each case the two possible types of operations and the two possible orientations of the tour. Let τ' be the tour obtained and let z' be its cost. Set $\tau := \tau'$ and $z := z'$.

- If $z < z^*$, set $\tau^* := \tau$, $z^* := z$ and $t := 1$; repeat **S2**.
- If $z \geq z^*$, set $t := t + 1$.
- If and $t = n + 1$, **STOP**: the best available tour is τ^* and its cost is equal to z^*, otherwise repeat **S2**.

3.5.2 Exact and Local Search Approaches for TSP

A large number of exact algorithms such as Linear Programming (LP), Dynamic Programming (DP) and B&B approach have been proposed for the TSP.

One of the earliest exact algorithm formulations is due to Dantzig [63] in 1954. He formulates the problem as an Integer LP problem involving zero-one variables and using a cutting plane approach to prove the optimality of a heuristic solution to a 49-city problem (an impressive size at that time).

The recursive technique of DP for the TSP is suggested by Bellman [29] and independently by Held and Karp [141] in 1962. At that time, DP could only solve relatively small problem instances (up to 17 cities), due to its enormous storage requirements. With the advances of the modern computer, we would expect to solve larger problem instances today.

The B&B approach also plays an important role in the development of the TSP. The first ever B&B approach for TSP was developed in 1958 by Eastman [85] and it has became a general tool for hard problems in combinatorial optimisation problems. The success of the algorithm also led to the derivation of the well known lower bound for STSP by Held and Karp [142, 143].

Over the past fifty years, the record for the largest non-trivial TSP instance solved to optimality has increased substantially from 49 cities problem in 1954 up to 24 978 Swedish cities problem in May 2004. Although the advances seen can be partly attributed to the increase in computing power, much of the improvement is due to the major developments in the use of B&B approaches. Applegate *et al.* [9] develop a very successful Branch and Cut algorithm which is derived from the B&B approach by employing the cutting plane method to strengthen the relaxations used for branching. Laporte [177] gives an excellent overview on the exact and approximate algorithms for TSP.

In recent years, local search methods like the Iterated Lin-Kernighan (by Johnson and McGeoch [161]) and Chained Lin-Kernighan (by Applegate *et al.* [10, 12]), based on the Martin-Otto-Felten approach described by Martin *et al.* [205, 206] are

widely believed to be the most cost-effective way to improve on the LK algorithm. The successful implementation of the Chained Lin-Kernighan on large scale STSP has led to the development of the state-of-the-art *Concorde* software for solving the large STSP. Concorde uses the Chained Lin-Kernighan as part of its exact solution procedure based on the Branch and Cut algorithm.

Other promising methods for solving TSP are Guided Local Search (GLS) as proposed by Voudouris and Tsang [272] and Guided Variable Neighbourhood Search (GVNS) as proposed by Burke *et al.* [39].

GLS augments the cost function of the problem to include a set of penalty terms for the edges and passes this problem, instead of the original one, for minimisation by the local search procedure. Each time a local search gets caught in a local optimum, the penalties are modified and local search is called again to minimise the modified cost function. Local search is confined by the penalty terms and focuses attention on promising regions of the search space. As the penalties build up for edges frequently appearing in local optima, the algorithm starts exploring new regions in the search space by including edges not previously use and therefore not penalised. Voudouris and Tsang [272] combine the GLS with a neighbourhood reduction scheme, called Fast Local Search (FLS) which significantly speeds up the operations of the algorithm. The GLS implementation that uses the FLS and 2-Opt move within the local search procedure easily outperform some general local search methods such as simulated annealing and tabu search. Furthermore, they demonstrate that GLS with FLS-2Opt is highly competitive, if not better, than some of the best specialised algorithms for the STSP such as Iterated LK and genetic local search.

GVNS proposed by Burke *et al.* [39] uses the notion of *guided shakes* within Variable Neighbourhood Search as a method to restart the search when it becomes trapped in a local optimum. This is shown to improve on the performance of random shaking strategies suggested in the original work by Hansen and Mladenović [134]. GVNS uses a hybrid approach of the HyperOpt/3-Opt as the local search heuristic. This approach yields very good results for the test problems in ATSP.

The first researcher who tackle the TSP with GAs appears to be Brady [36] in 1985. His example is soon followed by Grefenstette *et al.* [131], Goldberg and Lingle [126], Oliver *et al.* [220] and many others. Comprehensive surveys can be found in Potvin [235], Schmitt and Amini [248], and Larrañaga *et al.* [178]. Moreover, an annotated bibliography is given in Alander [5]. Despite the leap in the performance of GAs since the work of Mühlenbein *et al.* [215] in 1988, it is only recently that they have appeared competitive. Recent papers by Freisleben and Merz [105], Jayalakshmi *et al.* [159], and Choi *et al.* [50] yield encouraging results supporting the competitiveness of GAs.

Freisleben and Merz [105] propose a genetic local search algorithm for solving both STSP and ATSP. Their approach is based on the combination of GA and local search methods that employ heuristics such as LK for STSP and Nearest Neighbour for ATSP. Local search techniques are used to efficiently find the local optima in the TSP search space, and GA is used to broaden the search in order to find improved local optima. A new crossover operator, Distance Preserving Crossover (DPX) as explained in Section 4.6.2 has been developed to enable the GA to perform a particular '*jump*' within the search space of local optima. Furthermore, there is a mutation operator which performs random jumps within the neighbourhood of the local optima, and a new replacement strategy which maintains a sufficient degree of diversity within the population. The computational results presented for several symmetric and asymmetric TSP instances have shown that the approach is able to produce high quality solutions in reasonable time.

A Hybrid GA (HGA) is designed by Jayalakshmi *et al.* [159]. They develop three heuristics for the Euclidean TSP. One of the heuristics, called *Initialisation Heuristic* (IH), is applicable only to the Euclidean TSP and is for generating the initial population. The other two heuristics: *RemoveSharp* and *LocalOpt*, can be applied to all forms of symmetric and asymmetric TSPs. Both heuristics are greedy in nature. Results obtained by HGA outperform the results obtained by existing GA implementations for certain problems, where the convergence rate is found to be high and the optimal solution is obtained in a fewer number of generations.

Choi *et al.* [50] present a GA to solve the ATSP. The GA proposed extends the search space by purposefully generating and including infeasible solutions in the population. Instead of trying to maintain the feasibility with crossover operations, it searches through both feasible and infeasible regions for good quality solutions. The Karp's patching algorithm (see Karp [166]) is used as a repair algorithm to convert infeasible solutions to feasible ones from time to time. A comparative computational study using benchmark problems shows that the proposed GA is a viable option for ATSP.

Glover [116] appears to be the first researcher who developed the tabu search algorithm for the STSP. Limited results were reported by Glover [117], Knox and Glover [171], Knox [169, 170], and Malek *et al.* [201]. All these algorithms use 2-Opt as their basic moves, but they differ with respect to the size of the tabu list used and the implementation of the aspiration criterion. Tsubakitani and Evans [267] study the problem of optimising the size of the tabu list when applying tabu search with a short term memory function to the STSP. Their study revealed that good tabu list sizes are smaller than generally believed. Computational results show that tabu search generates better solution quality when constructed to a fixed computation time compared to 2-Opt and 3-Opt moves, for a variety of small problem instances. However, Johnson and McGeoch [161] conclude from the literature that tabu search appears to be inferior to the Lin-Kernighan method in terms of the solution quality obtainable within a fixed computation time.

The first simulated annealing applied to TSP was due to Kirkpatrick *et al.* [168] and independently by Černý [45]. Since then, the TSP has continued to be a prime testbed for the approach and its variants. Generally speaking, simulated annealing is unable to compete with a single run of Lin Kernighan in terms of the solution quality obtainable within a fixed computation time. However, over longer time periods, simulated annealing can outperform multi-start Lin Kernighan on some instances (Johnson and McGeoch [161]).

Chapter 4

Genetic Algorithms

4.1 Introduction

The aims of this chapter are to give more detailed descriptions of the main components in a standard Genetic Algorithm (GA) and our proposed MultiCrossover Genetic Algorithms (MXGAs). A brief overview of a GA is presented in Section 2.5.4. Each main component of the GAs is described and a brief summary of the variety of approaches often used in the components is provided. Note that the main objective of this chapter is to give an overview of the approaches used in the GAs rather than comparing the effectiveness and efficiency of the approaches in solving combinatorial optimisation problems. The efficiency of an approach depends on the representation used and differs from one problem domain to another. In the remainder of this chapter, without loss of generality, we refer to the chromosome as an individual, and genes in the chromosome as the elements in the individual.

In Sections 4.2 – 4.8, the approaches used in each component of GAs are briefly explained in chronological order. The general framework of the proposed MXGA is addressed in Section 4.9. We end this chapter by giving a summary of the GAs in Section 4.10.

4.2 Representation

Each individual represents a legal solution to the problem and is composed of a string of elements with length L. The binary alphabet $\{0,1\}$ is often used to represent these elements but depending on the application, integers or real numbers are used. In fact, almost any representation can be used that enables a solution to be encoded as a finite length string. Figure 4.1 shows some examples of commonly used representation for individuals.

Figure 4.1: Examples of Individuals

Binary Representation

Individual : 1 0 0 0 1 0 1 1 0 1

Example of problem: Knapsack Problem

Encoding: '1'= item in the knapsack, '0' otherwise.

Permutation Representation

Individual : 3 7 2 8 10 1 9 6 4 5

Example of problem: Cutting and Packing Problem

Encoding: sequence of the items to be placed.

Matrix Representation

Individual :

	j_1	j_2	j_3	j_4
i_1	0	0	0	1
i_2	0	0	1	0
i_3	1	0	0	0
i_4	0	1	0	0

Example of problem: Travelling Salesman Problem

Encoding: '1'= city j is visited immediately after city i, '0' otherwise.

Real Value Representation

Individual : 1.23 5.78 3.56 0.89 1.02

Example of problem: Finding Weights for Neural Network

Encoding: values represent weights for inputs.

Since the Travelling Salesman Problem (TSP) is one of the most studied combinatorial optimisation problems, there is no surprise that there have been many different representations used to solve the TSP using GAs. As a result, the TSP is a good example problem to use as a basis for describing the different types of representation. In the remainder of this section, we explain briefly the representations used in GAs to solve the TSP. Excellent reviews are given in Potvin [235] and Larrañaga *et al.* [178].

Binary Representation

In this representation, each city in a n-cities TSP is represented as an element with a string of $\lceil \log_2 n \rceil$ bits, and an individual is a string of $n\lceil \log_2 n \rceil$ bits. For example, in a TSP with six cities, the cities are represented by 3-bits strings as given in Table 4.1. Thus, the tour of cities

$$T: \ 1-2-3-4-5-6$$

is represented by

$$C = (000 \ \ 001 \ \ 010 \ \ 011 \ \ 100 \ \ 101).$$

Note that there exist 3-bit strings which do not corresponds to any city: 110 and 111.

Table 4.1: Binary Representation of a 6-cities TSP (Larrañaga *et al.* [178])

i	City i	i	City i
1	000	4	011
2	001	5	100
3	010	6	101

Although the binary strings constitute the most natural way of representation in GAs, it is considered to be not very appropriate for the TSP as commented by Whitley *et al.* [278]:

> "*Unfortunately, there is no practical way to encode a TSP as a binary string that does not have ordering dependencies or to which operators can be applied in a meaningful fashion. Simply crossing strings of cities produces duplicates and omissions. Thus, to solve this problem some variation on standard genetic crossover must be used. The ideal recombination operator should recombine critical information from the parent structures in a non-destructive, meaningful manner.*"

Path Representation

This is considered to be the most natural way to encode TSP tour. In this representation, the n cities to be visited are sequenced in order according to a list of n elements, so that if city i is the jth element of the list, city i is the jth city to be visited. Hence, the tour of cities

$$T: \ 2-8-5-3-7-9-1-10-6-4$$

is simply represented by

$$C = (2 \ \ 8 \ \ 5 \ \ 3 \ \ 7 \ \ 9 \ \ 1 \ \ 10 \ \ 6 \ \ 4).$$

This representation has encouraged a great number of crossover and mutation operators to be developed. These operators are discussed in detail in Section 4.6 and 4.7.

Adjacency Representation

This representation is developed by Grefenstette *et al.* [131] and is designed to facilitate the manipulation of edges between cities in the tour. The crossover operator developed based on this representation produces offspring that inherit most of the edges from their parents. A tour is represented as a list of n cities. City j is listed in position i in the individual if, and only if, the tour leads from city i to city j (i.e. there is an edge from city i to city j in the tour). Hence, the tour T mentioned above can be encoded as

$$C = (10 \ \ 8 \ \ 7 \ \ 2 \ \ 3 \ \ 4 \ \ 9 \ \ 5 \ \ 1 \ \ 6).$$

Ordinal Representation

This representation is also developed by Grefenstette *et al.* [131]. The encoding is based on a '*reference tour*'. Assume, for example, that the reference tour is given by

$$R = (1 \ \ 2 \ \ 3 \ \ 4 \ \ 5 \ \ 6 \ \ 7 \ \ 8 \ \ 9 \ \ 10).$$

Now the tour T mentioned earlier is represented as

$$C = (2\ \ 7\ \ 4\ \ 2\ \ 4\ \ 4\ \ 1\ \ 3\ \ 2\ \ 1).$$

This approach is interpreted as follows. The first number of C is a '2'. This means that the first city of the tour is the second element of list R. The second element is then removed from R and the partial tour is: $2-$. The second element of C is a '7'. Therefore, the second city of the tour is the 7th element of list R, which is city '8'. The 7th element is then removed from R and the partial tour is: $2-8-$. The process is repeated until all the elements of R have been removed and the final tour is

$$T:\ 2-8-5-3-7-9-1-10-6-4.$$

Matrix Representation

Fox and McMahon [104] represent a tour as a matrix in which the element in row i and column j is a '1', if and only if, in the tour city i is visited before city j. For example, the tour $(2-3-1-4)$ is represented by the matrix

$$C = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Seniw [251] and Homaifar *et al.* [148] have an alternative approach. They defined the matrix element in the ith row and the jth column to be '1' if, and only if, the tour city j is visited immediately after city i. This implies that a legal tour is represented by a matrix of which each row and each column contains precisely one '1'. For example, the tour $(2-3-1-4)$ mentioned earlier can be represented by matrix

$$C = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

4.3 Initial Population

Once a suitable representation has been decided, an initial population of size P_{pop} is created to serve as the starting point for the GA. The initial population can be created using two methods:
(1) randomly or
(2) by using specialised problem specific information.

A small population size may not be able to cover the solution space adequately, whereas a large population size may incur a heavy computational burden without making acceptable progress toward a high quality solution in a reasonable amount of time. Many report implementations where a population size between 30 and 200 is usually recommended (see Grefenstette [129], Goldberg [122] and Alander [3]).

4.4 Fitness Evaluation

This involves defining an objective or fitness function against which each individual is tested for suitability to be introduced to the population under consideration. As the algorithm proceeds, one would expect the individual fitness of the '*best*' individual to increase as well as the fitness of the population as a whole.

If the GA has been correctly implemented, the population will evolve over successive generations so that the fitness of the individuals in each generation will increase towards better local optima. *Convergence* is the progression towards increasing uniformity. De Jong [163] gives the following definition in his thesis in 1975:

> "*A gene is said to have converged when 95% of the population share the same value. ... The population is said to have converged when all of the genes have converged.*"

However, if the population converges too quickly, it often leads to the problem of *premature convergence*. Premature convergence is a commonly cited problem with GAs when a few comparatively highly fit (but not optimal) individuals dominate the population, causing it to converge on a local optimum. Once the population has converged (every individual in the population is identical), the ability of the GAs to continue to search for better solutions is effectively eliminated. Crossover of almost identical individuals produce little that is new. New areas of the solution space can only be explored by mutation, which simply performs a slow random search.

4.5 Selection Mechanism

Individuals are selected from the population to be the parents for crossover with a given *selection probability* p_s. According to Darwin's evolution theory, the best ones should survive and create new offspring. In the remainder of this section, we define the selection probability p_s, of an individual i as:

$$p_s(i) = \frac{f_i}{\overline{f}} \text{ where} \tag{4.1}$$

f_i is the fitness value associated with individual i; and
$\overline{f}$ is the mean fitness of the current population.
We refer to the *selection pressure* as the degree to which selection favours fitness. The selection pressure is characterised by the take over time Γ, the number of generations taken for the best individual in the initial generation to completely dominate the population (mutation and crossover are switched off). When the selection pressure is low, the selection procedure allows less fit individuals to reproduce at close to the rate of fitter individuals while maintaining the diversity and variation in the population. When the selection pressure is high, the selection procedure strongly emphasises highly fit individuals, assuming that the early diversity with the slow selection has allowed the population to find the right part of the search space.

A comparative analysis of selection mechanism used in GAs is given by Goldberg and Deb [124]. Note that the name of the author/researcher who proposed the approach is stated next to the name of the selection mechanism.

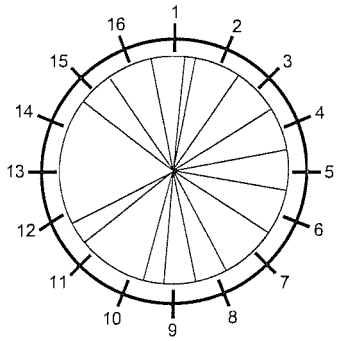
Roulette Wheel (Holland [147], 1975):
Each individual is represented by a space in a roulette wheel that proportionally corresponds to its selection probability $p_s(i)$. By representing the spinning of the roulette wheel, parents are chosen using '*stochastic sampling with replacement*' strategy. This strategy is very sensitive to fitness function design. For example, let four individuals have fitness of 0.004, 0.002, 0.003, and 0.500. The '0.500' individual (*super individual*) will take up almost the entire wheel, and so will be likely to be alone in the next generation (i.e. the selection pressure very high). While on another example where individuals with fitness 998, 997, 999, and 1000 have virtually no selection pressure at all. There are also problems when dealing with zero and negative fitness.

Rank (Baker [18], 1985):
Rank selection first ranks the population from 1 to P_{pop} (population size) according to their individual selection probability $p_s(i)$. The worst will have rank 1, second worst rank 2, etc. and the best will have rank P_{pop}. The parents are selected through roulette wheel selection, but the segments of the wheel are proportional to the individual's rank, rather than its selection probability. This strategy reduces the dominating effects of super individuals, and thus reduces the selection pressure when the fitness variance is high. But, this can lead to slower convergence, because the best individuals do not differ so much from the others.

Stochastic Universal Sampling (SUS) (Baker [19], 1987):
Assume that the population is laid out in random order as in a pie graph, where each individual is assigned space on the pie graph in proportion to its selection probability $p_s(i)$. Next, an outer roulette wheel is placed around the pie with P_{pop} (population size) equally spaced pointers. A single spin of the roulette wheel will

now simultaneously pick all P_{pop} parents. An example with a $P_{pop} = 16$ is given in Figure 4.2. A single spin of the outer roulette wheel will simultaneously select all 16 parents.

Figure 4.2: An example of a Stochastic Universal Sampling



Sigma Scaling (Tanese [263], 1989):
This is achieved by mapping 'raw' fitness values of an individual i to its 'Expected Value' $(ExpVal)_i$ so as to make the GA less susceptible to premature convergence. $(ExpVal)_i$ of an individual i is the expected number of times an individual will be selected to reproduce. This strategy also helps to keep the selection pressure relatively constant over the course of the run rather than depending on the fitness variance in the population. The expected number of times of an individual i to be selected to reproduce at time t is given as follows:

$$ExpVal(i,t) = \begin{cases} 1 + \frac{f(i) - \bar{f}(t)}{2\sigma(t)} & \text{if } \sigma(t) \neq 0 \\ 1.0 & \text{if } \sigma(t) = 0 \end{cases} \quad \text{where} \tag{4.2}$$

$f(i)$ is the fitness of i;
$\bar{f}(t)$ is the mean fitness of the population at time t;
$\sigma(t)$ is the standard deviation of the population fitness at time t.

In early stage, when $\sigma(t)$ is typically high, the fitter individuals will not be many standard deviations above the mean, and so they will not dominate the offspring. But, in the later stage, when the population is closer to convergence and the $\sigma(t)$ is typically lower, the fitter individuals will stand out more, allowing evolution to continue.

Tournament (Goldberg *et al.* [125], 1989):
There are several variants and the idea is simple. In the binary tournament selection, a pair of individuals are selected randomly from the population. The fitter of the two is selected to be the parent. The two are then returned to the original population and can be selected again. This is repeated until P_{pop} individuals have been selected. Larger tournaments may also be used, where the fittest of K ($K \leq P_{pop}$) randomly chosen individuals are selected.

Using larger tournaments has the effect of increasing the selection pressure, since less fit individuals are less likely to be selected, while fitter individuals have an increased likelihood.

In order to control the selection pressure, a probabilistic binary tournament selection can be employed. The fitter individual is selected to be the parent with a probability p, where $0.5 < p < 1$. The selection pressure can be lowered by using lower values of p, since less fit individuals are comparatively more likely to be selected, while fitter individuals are less.

Boltzmann Tournament(Goldberg [123], 1990 and Mahfoud [200], 1991):
This is an approach which thermodynamically control the selection pressure of a GA, using principles from Simulated Annealing (SA). In order to do so, a mixture of SA acceptance probabilities and three-way tournament selection is proposed.

First of all, three individuals (a_1, a_2, a_3) are randomly selected from the population. Individual a_2 must differ from a_1 by a fitness amount of φ. Individual a_3 must also differ from a_1 and a_2 by at least φ. Then, a_2 and a_3 will compete using a winning probability according to a logistic probability function of fitnesses and temperature. The winner will compete against a_1 using the similar winning probability, and the best individual will be selected as the parent for crossover.

Using the fitness values f_1, f_2, and f_3 of individuals a_1, a_2 and a_3 respectively, the winning probabilities for a_2 over a_3 (p'), a_1 over a_2 (p''), and a_1 over a_3 (p''')

are given by (extracted from Goldberg [123]):

$$\begin{aligned} p' &= \frac{1}{1+e^{(f_2-f_3/T)}}, \\ p'' &= \frac{1}{1+e^{(f_1-f_2/T)}}, \text{ and} \\ p''' &= \frac{1}{1+e^{(f_1-f_3/T)}}. \end{aligned} \tag{4.3}$$

Then, the overall winning probabilities p_1, p_2 and p_3 for a_1, a_2 and a_3, are given by (extracted from Mahfoud [200]):

$$\begin{aligned} p_1 &= p'(1-p'') + (1-p')(1-p'''), \\ p_2 &= p'p'', \text{ and} \\ p_3 &= (1-p')p'''. \end{aligned} \tag{4.4}$$

At the early stage, the temperature starts out high (i.e. the initial temperature, T_0), which means that the selection pressure is low. The temperature is gradually lowered in the later stage according to the rule

$$T_{t+1} = \alpha \cdot T_t \ , \tag{4.5}$$

where α is the cooling coefficient. By doing this, we gradually increase the selection pressure, thereby allowing the GA to narrow in ever more closely to the best part of the search space while maintaining the 'appropriate' degree of diversity. At each temperature T_t, a number of function evaluations N_f, are performed (i.e. the time to reach equilibrium at a given temperature).

According to Goldberg [123], the value of φ is initially set to 0.5 and changed after each selection of an individual according to the rule

$$\varphi_i = \begin{cases} -T \cdot \ln\left(\frac{2}{|p_2-p_1|+1} - 1\right), & \frac{1}{2}\left(|p_2-p_1|+1\right) < 1 \\ \varphi_{\max} & , \frac{1}{2}\left(|p_2-p_1|+1\right) \geq 1 \end{cases} \tag{4.6}$$

(with $\varphi_{\max}$ being a value large enough to guarantee acceptance).

4.6 Crossover Operator

Crossover is a strategy of producing new offspring by replacing some of the elements in one parent with the corresponding elements of the other parent. Crossover is used with the hope that the new offspring will inherit good characteristics of both parents. There are many different variants of crossover operators that are specially designed to suit the different type of representation and problems. Not all the crossover operators discussed below are suitable for all problems. For instance, the sorted match crossover operator is specially designed for TSP but is not suitable for other problems such as machine scheduling problem.

Crossover is not usually applied to all pairs of selected parents. Instead, a random choice is made based on a *crossover probability* p_c. Empirical studies have shown that better results are achieved by a crossover probability of between 0.60 and 0.95 (see Grefenstette [129] and Schaffer *et al.* [247]). If crossover is not applied to the selected parents, two offspring are produced simply by duplicating the selected parents via the *reproduction* strategy. This gives each individual a chance of passing on its elements without the disruption of crossover.

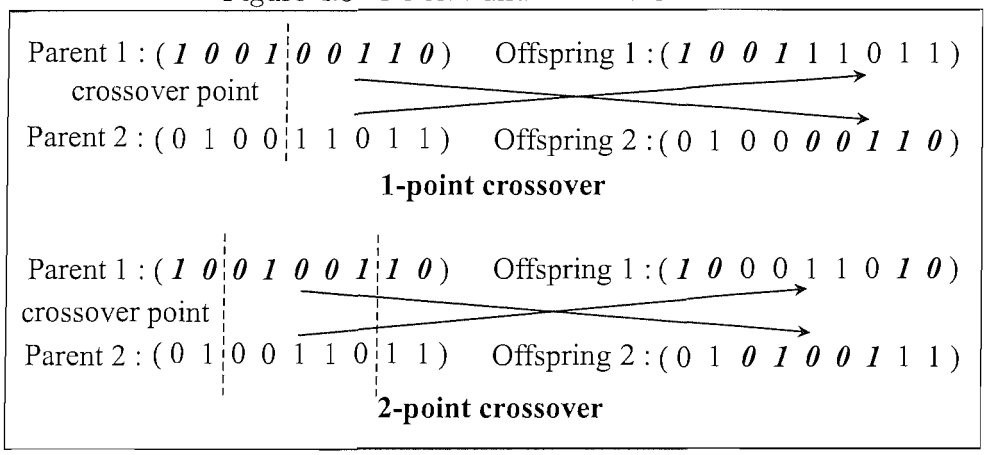
In the remainder of this section, we describe briefly some of the crossover operators used in literature by classifying them based on the representation framework used. The author/researcher who proposed the operator is listed next to the name of the crossover operator.

4.6.1 Binary Representation

1- and 2-Point Crossover (Holland [147], 1975):
1-point crossover involves taking the two selected parents and crossing them at a randomly chosen point. The parents exchange '*tails*' to generate two offspring. In 2-point crossover, two randomly chosen points are selected. Substrings between the two crossover points swap their positions between the two parents, rendering two offspring. Figure 4.3 shows examples of 1-point and 2-point crossover of two

parents using binary representation. In fact, we can choose more crossover points so that the search can diversify into other '*interesting*' regions of the solution space.

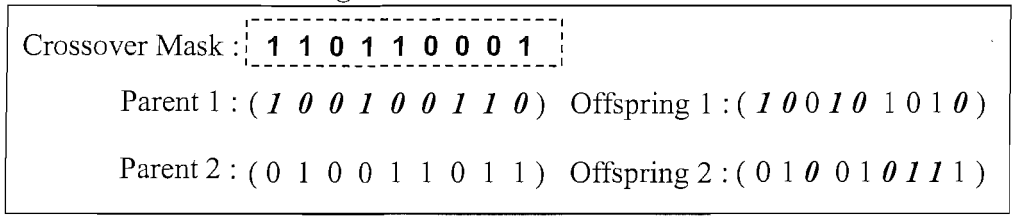
Figure 4.3: 1-Point and 2-Point Crossover



Uniform Crossover, UX (Syswerda [261], 1989):

This crossover performs at each point a random decision to produce the offspring. Each element in the offspring is created by copying the corresponding element from one of the parents. The element is chosen according to a randomly generated *crossover mask* using a binary representation. A '1' in the crossover mask means the element is copied from the first parent, and '0' means the element is copied from the second parent, as shown in Figure 4.4. The offspring therefore contains a mixture of elements from each parent. The process is repeated to produce the second offspring. A new crossover mask is randomly generated for each pair of parents.

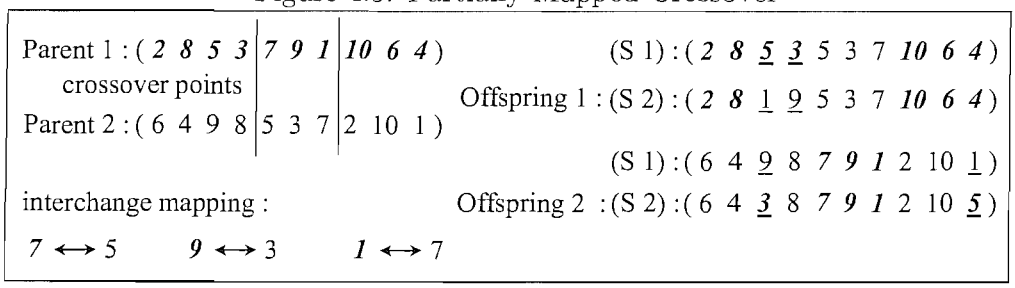
Figure 4.4: Uniform Crossover



4.6.2 Path Representation

Partially Mapped Crossover, PMX (Goldberg and Lingle [126], 1985):
For PMX, it is not the values of the element which are crossed, but the order in which they appear. Offspring inherit elements with ordering information from each parent. PMX first randomly selects two crossover points on both parents (see Figure 4.5). In order to create an offspring, the substring between the two crossover points in the first parent replaces the corresponding substring in the second parent (see S1). Then, the interchange mapping is applied outside of the crossover points as many time as necessary, in order to eliminate duplicates and recover all elements (see S2).

Figure 4.5: Partially Mapped Crossover

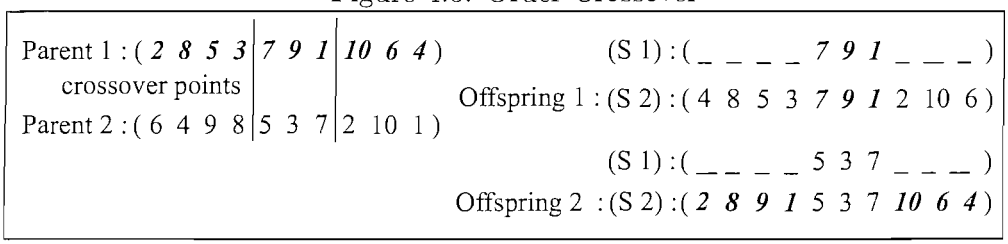


Note that the absolute positions of some elements of both parents are preserved. In fact, the number of elements that do not inherit their positions from one of the two parents is at most equal to the length of the substring.

Order Crossover, OX (Davis [65], 1985):
Two crossover points are selected randomly on the selected parents (see Figure 4.6). The substring between the crossover points in the first parent is copied to the first offspring (see S1). Then the remaining positions in the first offspring are filled by considering the sequence of elements in the second parent, starting after the second crossover point (when the end of the individual is reached, the sequence continues at position 1) (see S2). Note that the duplicates are not considered. Similarly, the second offspring is formed by taking the substring from the second

parent and by considering the sequence of elements in the first parent to fill up the empty spaces in the offspring. This operator tries to preserve the relative order of the elements in both parents.

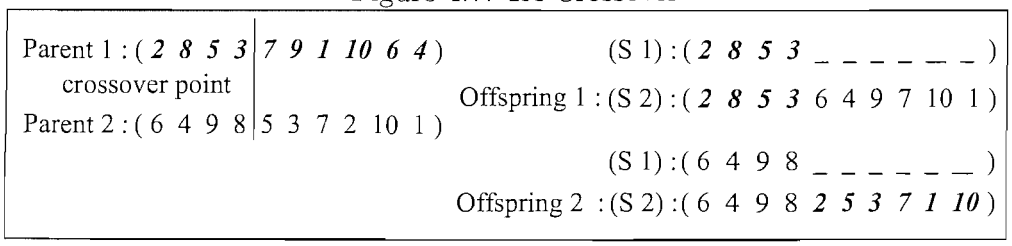
Figure 4.6: Order Crossover



1X (Davis [65], 1985):

This operator is the simplification of OX where a 1-point crossover is used instead of a 2-point crossover (see Figure 4.7). The substring before the crossover point in the first parent is copied to the first offspring (see S1). The remaining positions in the first offspring are filled in the order of the second parent (see S2).

Figure 4.7: 1X Crossover



Sorted Match Crossover, SMX (Brady [36], 1985):

This operator searches for substrings in both parents which have the same length, start with the same element, end with the same element and contain the same set of elements. If such substrings are found, the fitness of these substrings are determined. The substring with the lower fitness value in a parent is replaced with the substring with the higher fitness value to form a new offspring. In Figure 4.8, parent 1 contains the substring (8 5 3 7 9) and the parent 2 contains substring (8 7 5 3 9). These substrings have the same length, both begin with element '8', end with element '9', and both contain the same elements. Suppose that the fitness

value of substring (8 7 5 3 9) is higher than the fitness value of the substring (8 5 3 7 9). Then, the new offspring is created by replacing the substring (8 5 3 7 9) in parent 1 with the substring (8 7 5 3 9).

Figure 4.8: Sorted Match Crossover

Parent 1: (2 ***8 5 3 7 9*** 1 10 6 4)
Parent 2: (6 4 ***8 7 5 3 9*** 2 10 1)
Offspring: (2 ***8 7 5 3 9*** 1 10 6 4)

Mühlenbein *et al.* [215] concluded that this operator was useful in reducing the computation time, but it is a weak scheme for crossover. If no substring can be found in both parents which fulfill the basic requirements of the operator, or both substrings found contained the same sequence of elements and have the same fitness value, then this operator has failed to produce a new offspring.

Cycle Crossover, CX (Oliver *et al.* [220], 1987)**:**
This crossover focuses on subsets of elements that occupy the same subset of position in both parents. It tries to inherit the position of each element from one of the two parents. An example is given in Figure 4.9 where the underlined elements in both parents are the subset of elements $\{2, 6, 10\}$ that occupied the same subset of positions $\{1, 8, 9\}$ in both parents. These elements are copied from one parent to the offspring (at the same position), and the remaining positions are filled with the elements of the other parent in the same order in which they appeared in the parent.

Figure 4.9: Cycle Crossover

Parent 1 : (***2*** ***8 5 3 7 9 1*** ***10*** ***6*** ***4***)	Offspring 1 : (***2*** 4 9 8 5 3 7 ***10*** ***6*** 1)
Parent 2 : (6 4 9 8 5 3 7 2 10 1)	Offspring 2 : (6 ***8 5 3 7 9 1*** 2 10 ***4***)

Maximal Preservative Crossover, MPX (Mühlenbein *et al.* [215], 1988)**:**
This operator works in a similar way to the PMX operator. An example is given in Figure 4.10. A substring of the first parent is randomly selected, whose length

l, is within the range of $10 \leq l \leq L/2$, where L is the length of the individual. If $L < 20$, then length $l < 10$. All the elements of the chosen substring are removed from the second parent. An offspring is constructed by first copying the substring from the first parent into the first part of the offspring (see S1). Then, the remaining part of the offspring is filled up with elements in the same order as they appear in the second parent (see S2).

Figure 4.10: Maximal Preservative Crossover

Parent 1 : (*2 8 5* <u>*3 7 9*</u> *1 10 6 4*)	(S 1) : (*3 7 9* _ _ _ _ _ _ _)
substring : (3 7 9)	Offspring 1 : (S 2) : (*3 7 9* 6 4 8 5 2 10 1)
Parent 2 : (6 <u>4 9 8 5</u> 3 7 2 10 1)	(S 1) : (4 9 8 5 _ _ _ _ _ _ _)
substring : (4 9 8 5)	Offspring 2 : (S 2) : (4 9 8 5 *2 3 7 1 10 6*)

This operator will only destroy a limited number of edges between the elements. In fact, the maximum number of edges which may be destroyed is equal to the length of the chosen substring.

Edge Recombination Crossover, ERX (Whitley *et al.* [278], 1989)**:**
This operator is designed for the symmetric TSP. It tries to use the edges which are contained in both parents as much as possible. The steps to generate one offspring is given below and an example lies in Figure 4.11.

1. Designing of the Edge table: assign a list of neighbours in parent 1 and parent 2 to each city (the sign '-' means that the corresponding city is a neighbour in both parents). The first and the last cities are considered as neighbours for the TSP.
2. An arbitrary first city is chosen from the table with the smallest list of neighbours and is called the current city.
3. The following iterative procedure is used:
   (a) Select a city which is a neighbour of the current one and which has the fewest remaining neighbours (breaking ties randomly), or select an arbitrary remaining cities if the current city has no remaining neighbour.
   (b) The city is added to the tour and becomes the new current city.
   (c) If all n cities are not selected, go to (a).

Figure 4.11: Edge Recombination Crossover

Parent 1 : (2 8 5 3 7 9 1 10 6 4)

Parent 2 : (6 4 9 8 5 3 7 2 10 1)

Offspring 1 : (**3** 5 8 **2** 7 9 4 6 **1** 10)

Offspring 2 : (**5** 3 7 **9** 8 2 4 6 **10** 1)

where **x** = random breaks ties

Edge table :

operation	neighbour
1	6 9 -10
2	4 7 8 10
3	-5 -7
4	2 -6 9
5	-3 -8
6	1 -4 10
7	2 -3 9
8	2 -5 9
9	1 4 7 8
10	-1 2 6

Linear Order Crossover, LOX (Falkenauer and Bouffouix [89], 1991):
This operator is a modified version of OX, proposed to solve job-shop scheduling problems. The LOX operator differs from the OX in that the relative positions of two elements are important and will be preserved. An example is given in Figure 4.12 by using the same parents as in OX. Two crossover points are selected randomly. The elements in the second parent are copied to the first offspring. The elements in the substring (7 9 1) are removed from the first offspring, leaving three empty spaces to be filled (see S1). The elements are first slid to the left up to the point when no empty space remains on the left of the cross site. Then the elements are slid to the right, leaving only empty spaces between the crossover points. Finally, the empty spaces are filled with substring (7 9 1) (see S2). The second offspring is generated analogously using the substring (5 3 7) and the first parent.

Figure 4.12: Linear Order Crossover

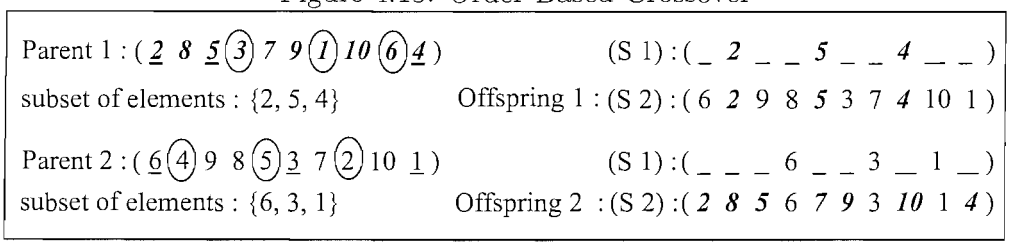
Parent 1 : (***2 8 5 3*** | ***7 9 1*** | ***10 6 4***)
crossover points
Parent 2 : (6 4 9 8 | 5 3 7 | 2 10 1)

(S 1) : (6 4 _ 8 | 5 3 _ | 2 10 _)
Offspring 1 : (S 2) : (6 4 8 5 | ***7 9 1*** | 3 2 10)

(S 1) : (***2 8*** _ _ | _ ***9 1*** | ***10 6 4***)
Offspring 2 : (S 2) : (***2 8 9 1*** | 5 3 7 | ***10 6 4***)

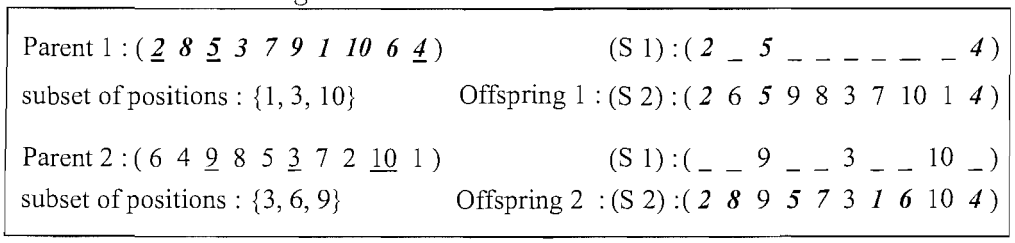
Order-Based Crossover, OBX (Syswerda [262], 1991)**:**
This crossover focuses on the relative order of the elements on the parents. An example is given in Figure 4.13. At first, a subset of elements (underlined) are randomly selected from the first parent. In the first offspring, these elements appear in the same order as in the first parent, but at positions (circled) taken from the second parent (see S1). Then, the remaining positions are filled with the elements of the second parent (see S2). Similarly, the second offspring is constructed by placing the subset of elements (underlined) selected from the second parent at the positions (circled) taken from the first parent. Then, the remaining empty spaces are filled with the elements of the first parent.

Figure 4.13: Order-Based Crossover



Position-Based Crossover, PBX (Syswerda [262], 1991)**:**
A subset of positions are randomly selected in the first parent (Figure 4.14). Then, the elements found at these positions are copied to the first offspring (at the same positions) (see S1). The other positions are filled with the remaining elements, in the same order as in the second parent without duplication (see S2).

Figure 4.14: Position-Based Crossover



Subtour Exchange Crossover, SXX (Yamamura *et al.* [284], 1992):
A pool of offspring are generated by enumerating all the substrings of the parents consisting of the same set of elements. Then, the best offspring is selected from the pool of offspring. An example is given in Figure 4.15. Suppose that two substrings consist of the same set of elements (e.g. $\{3,5,7,8\}$ and $\{4,6\}$ were found in both parents). Two possible offspring are generated by exchanging the common substrings from the parents.

Figure 4.15: Subtour Exchange Crossover

Parent 1 : (*2* ***8 5 3 7*** *9 1 10* ***6 4***)	Offspring 1 : (*2* 7 3 8 5 *9 1 10* 4 6)
Parent 2 : (4 6 1 7 3 8 5 10 2 9)	Offspring 2 : (*6 4* 1 *8 5 3 7* 10 2 9)

Distance Preserving Crossover, DPX (Freisleben and Merz [105], 1996):
The contents of the first parent is copied to the offspring and all edges that are not in common with the other parent are deleted. The resulting fragments of the broken substring are reconnected using different edges to those contained in either of the parents. To do this, a greedy reconnection procedure is used. Suppose that the edge (i,j) has been broken, the nearest available neighbour k of i is taken and the edge (i,k) is added to the substring, provided that (i,k) is not contained in any of the parents. This process continues until all fragments have been reconnected. For example, consider the parents in Figure 4.16. By copying parent 1 to the offspring and deleting the edges not contained in both parents leads to the substring fragments: 2, 8–5–3–7, 9, 1–10, and 6–4. Offspring 1 is an example of reconnection that does not use edge connection from either parent.

Figure 4.16: Distance Preserving Crossover

Parent 1 : (*2 8 5 3 7 9 1 10 6 4*)
Parent 2 : (6 4 9 8 5 3 7 2 10 1)
Fragments : | *2* | *8 5 3 7* | *9* | *1 10* | *6 4* |
Offspring 1 : (*7 3 5 8 1 10 9 2 6 4*)

Alternating-Position Crossover, APX (Larrañaga *et al.* [179], 1997):
The offspring is constructed by selecting the next element alternately from both parents, omitting the elements already present in the offspring. An example is given in Figure 4.17. By doing so, we believed that this operator will destroys too many edges between the elements.

Figure 4.17: Alternating-Position Crossover

Parent 1 : (***2 8 5 3 7 9 1 10 6 4***)	Offspring 1 : (***2*** 6 ***8*** 4 5 ***9 3 7 1 10***)
Parent 2 : (6 4 9 8 5 3 7 2 10 1)	Offspring 2 : (6 ***2*** 4 ***8*** 9 ***5 3 7 1 10***)

Complete Subtour Exchange Crossover, CSEX (Katayama *et al.* [167], 1998):
This operator is a modification of SXX, where the common substring used in CSEX is either identical or symmetrical in the sequence of elements. By using the same parents in Figure 4.15, the common substrings are (8 5), (3 7) and (6 4) in parent 1, and (8 5), (7 3) and (4 6) in parent 2. A pool of offspring is generated by enumerating all the common substrings in both parents. The best offspring is then selected from the pool of offspring. If K common substrings are included within the parents, a maximum of $2 \times 2^K - 2$ offspring are generated. Figure 4.18 gives two possible offspring from the parents using CSEX.

Figure 4.18: Complete Subtour Exchange Crossover

Parent 1 : (***2*** ***8 5*** ***3 7*** ***9 1 10*** ***6 4***)	Offspring 1 : (***2 8 5*** 7 3 ***9 1 10*** 4 6)
Parent 2 : (4 6 1 7 3 8 5 10 2 9)	Offspring 2 : (***6 4*** 1 ***3 7*** 8 5 10 2 9)

Subtour Preservation Crossover, SPX (Soak and Ahn [257], 2004):
This operator is designed for TSP where a similar subtour enumeration technique to SXX, CSEX and DPX is used. This operator generates offspring using common subtour which parents share and edges included in each parents. The procedure of SPX is divided into two steps. The first step is to enumerate all common subtour,

and next is to reconnect each subtour and any isolated cities. Although SPX finds the same subtours from two identical parents, it can generates different offspring by selecting different starting point and each process of reconnecting subtours and isolated cities. When a city is selected as a starting point, shorter edge between two possible edges is selected. If a shorter edge has already been selected, another edge is selected. And if both edges are already selected, random selection is performed among the endpoints not yet selected. This process continues until a tour is formed.

4.6.3 Adjacency Representation

Alternate Edges Crossover, AEX (Grefenstette *et al.* [131], 1985):
This operator is used in TSP where a starting edge (i, j) in the offspring is selected randomly in one parent. Then, the tour in the offspring is extended by selecting the edge (j, k) in the other parent. The offspring is progressively extended in this way by alternately selecting edges from each parent. When an edge introduces a cycle, the next edge is selected at random (and is not inherited from the parents). An example is given in Figure 4.19. Note that edge $(9, 3)$ in offspring 1 and edge $(7, 9)$ in offspring 2 are not inherited from any of their parents.

Quite often, the AEX introduces too many random edges between the elements in the offspring and good substrings are often disrupted by the crossover operator. Since the offspring must inherit as many edges as possible from the parents, the introduction of random edges should be minimised. As reported in Grefenstette *et al.* [131], the results with this operator have been uniformly discouraging.

Figure 4.19: Alternate Edges Crossover

Adjacency Representation		Actual Tour
Parent 1 : (10 3 8 5 2 1 4 6 7 9)	⇨	9 – 7 – 4 – 5 – 2 – 3 – 8 – 6 – 1 – 10
Parent 2 : (4 9 7 6 10 5 2 1 8 3)		3 – 7 – 2 – 9 – 8 – 1 – 4 – 6 – 5 – 10
Offspring 1 : (10 9 8 6 2 5 4 1 3 7)	⇨	7 – 4 – 6 – 5 – 2 – 9 – 3 – 8 – 1 – 10
Offspring 2 : (4 3 7 5 10 1 9 6 8 2)		2 – 3 – 7 – 9 – 8 – 6 – 1 – 4 – 5 – 10

Subtour Chunk Crossover, SCX (Grefenstette *et al.* [131], 1985):
This operator is developed for TSP where an offspring is constructed by first copying a random length subtour of the first parent. Then, the partial tour is extended by choosing a random length subtour from the second parent. The offspring is progressively extended this way by alternately selecting a subtour from two parents. A subtour is not added if it produces an illegal tour. In this case, an edge is chosen at random from the edges that do not produce a cycle and added into the partial tour. This operator should performs better than the AEX. However, its performance is still not very encouraging as it does not take into account the available information about the edges.

Heuristic Crossover, HX (Grefenstette *et al.* [131], 1985):
This operator is designed for TSP where the procedure of generating an offspring is as follows:

S 1: Select randomly a starting city from one of the two parents.
S 2: Compare the edges leaving the current city in both parents and select the shorter edge.
S 3: If the added edge creates a cycle in the partial tour, try the other edge. If it also introduces a cycle, extend the tour with a random edge that does not introduce a cycle.
S 4: Repeat **S2** and **S3** until all cities are included in the tour.

Jog *et al.* [160] suggest to replace the random edge selection by the selection of the shortest edge in a pool of q random edges, where q is a parameter.

4.6.4 Matrix Representation

In the next two crossover operators, insertion crossover and union crossover, we use the matrix representation where the element in row i and column j is a '1' if, and only if, in the tour city i is visited before city j.

Insertion Crossover, MIX (Fox and McMahon [104], 1987):
An offspring O, is constructed from two parents (P1 and P2) in the following way:

S 1: For all $i, j \in \{1, 2, 3, \ldots, n\}$ define

$$O_{ij} := \begin{cases} 1 \text{ if } \mathrm{P1}_{ij} = \mathrm{P2}_{ij} = 1; \\ 0 \text{ otherwise.} \end{cases}$$

S 2: Some 1's which are unique for one of the parents are "added" to O.
The matrix is completed using the analysis of the sum of rows and columns, in such a way that the result is a legal tour. This operator preserves all precedence relationship which are common to both parents. For example (extracted from Larrañaga *et al.* [178]), Fox and McMahon [104] represent the parent tours $(2-3-1-4)$ and $(2-4-1-3)$ as

$$\mathrm{P1} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \text{ and } \mathrm{P2} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}.$$

After **S1**, we have

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

This matrix can be completed in six different ways, since the only restriction on the offspring tour is that it starts in city 2. One possible offspring is the tour $(2-1-4-3)$ which is represented by:

$$O = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Union Crossover, MUX (Fox and McMahon [104], 1987):
This operator tries to combine some precedence relationships taken from each parent. First, it divides the set of cities into two disjoint groups. For example, by using the two parents above, we could divide the set of cities into $\{1, 2\}$ and $\{3, 4\}$. The matrix elements of an offspring is constructed by placing the first group from the first parent and the second group from the second parent. Hence, we have

$$\left(\begin{array}{cccc} 0 & 0 & ? & ? \\ 1 & 0 & ? & ? \\ \hline ? & ? & 0 & 0 \\ ? & ? & 1 & 0 \end{array}\right).$$

The resulting matrix is completed by an analysis of the sum of the rows and columns. One possible offspring is the tour $(4-3-2-1)$ which is represented by:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix}.$$

Matrix Crossover, MMX (Homaifar *et al.* [148], 1991):
This crossover operator uses a matrix representation where the element in the ith row and the jth column is '1', if and only if, city j is visited immediately after city i in the tour. This operator deals with column positions rather than element positions. First, a crossover point is selected at random. Then, MMX exchanges all the entries of the two parents determined by the crossover point(s). With the example given above, Homaifar *et al.* [148] represent the parent tours (i.e. $(2-3-1-4)$ and $(2-4-1-3)$) as:

$$\text{P1} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad \text{and} \quad \text{P2} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}.$$

Suppose that the crossover point is chosen between the first and the second column, which yields

$$\left(\begin{array}{c|ccc} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}\right) \quad \text{and} \quad \left(\begin{array}{c|ccc} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{array}\right).$$

One of resulting offspring after the crossover is

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Note that this matrix does not represent a legal tour. MMX may result in infeasibility in the form of duplications or cycles. These two problems are treated in two steps:

S 1: Remove the duplication by moving a '1' from each row with duplicate '1's into another row that has no '1' entries.

S 2: Cut and connect cycles to produce a legal tour while preserving as many of the existing edges from the parent as possible.

By applying **S1** to the offspring, we have a new offspring which represents the legal tour $(1-3-2-4)$

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}.$$

4.7 Mutation Operator

After a crossover or a reproduction is performed, mutation takes place. This is to prevent too many identical individuals from being in the same generation which leads to premature convergence and the danger of becoming trapped in a local optimum.

Although mutation is intended to prevent the GAs from falling into a local optimum, if the mutation rate is too high then the GAs will in fact change to a random search. Any efficient optimisation algorithm must use two techniques to find a global optimum:
–*exploration*: to investigate new and unknown areas in the search space, and
–*exploitation*: to make use of knowledge found at points previously visited to help find better points.

These two requirements are contradictory, but a good search algorithm provides a balance between the two. A purely random search is good for *exploration*, but does no *exploitation*, while a purely descent method is good at *exploitation*, but does little *exploration*. Combinations of these two strategies can be quite ideal, but it is difficult to find that right balance. GAs use both crossover and mutation operators to *explore* and *exploit* the search space in the hope of finding good optimal solutions (see Goldberg [121]).

In the binary representation for instance, if we only use the crossover operator to produce offspring, one potential problem that may arise is that if all the individuals in the initial population have the same value at a particular element, then all future offspring will have this same value at that element. For example, if all the individuals in the population have a '0' in fifth element, then all future offspring will have a '0' at fifth element after the crossover takes place.

Binary mutation is applied randomly to each offspring individually that alters each element from '1' to '0' or vice versa with a given probability, p_m. This value is called the *mutation rate*. Various optimal mutation rates have been reported. The most common approaches are either to use a small mutation probability (e.g. $p_m = 0.001$), or to use a value $p_m = 1/L$, where L is the length of the individual (see Grefenstette [129], Fogarty [99], Hesser and Männer [145], and Bäck [15]).

Various mutation operators have been designed for the integer representation (e.g permutation, path, adjacency and ordinal). As opposed to the binary mutation operator, which introduces small changes into the individual, the mutation

operator for integer representation often greatly modifies the offspring. These operators are briefly summarised below. In the remainder of this section, without loss of generality, we refer to the following offspring for mutation:

Offspring : (2 8 5 3 7 9 1 10 6 4).

Simple Inversion Mutation (SIM) (Holland [147], 1975):
This operator was first introduced by Holland and then popularised by Grefenstette [130] in 1987. This operator first selects randomly two cut points in the offspring, and then reverse the substring between these two cut points to form a new offspring. Consider the offspring and suppose that the first cut point is chosen between the second and third element, and the second cut point between the 6th and 7th element. This results in:

New Offspring : (2 8 ***9 7 3 5*** 1 10 6 4).

Insertion Mutation (ISM) (Fogel [100], 1988):
This operator starts by randomly choosing an element and removing it from the offspring. The element is then inserted in a randomly selected place. For example, consider again the offspring mentioned earlier, and suppose that the operator selects the third element, removes it, and randomly inserts it after 8th element. Hence, the resulting new offspring is:

New Offspring : (2 8 3 7 9 1 10 6 ***5*** 4).

This operator is also called the Position Based mutation by Syswerda [262].

Exchange Mutation (EM) (Banzhaf [25], 1990):
EM is achieved by first randomly selecting two elements in the offspring, and then exchanging their position. Consider the offspring mentioned above, and suppose that the third and 9th element are randomly selected. This results in a new offspring:

New Offspring : (2 8 ***6*** 3 7 9 1 10 ***5*** 4).

This operator is also referred to as the Swap mutation (by Oliver *et al.* [220]), the Point mutation (by Ambati *et al.* [7]), the Reciprocal Exchange mutation (by Michalewicz [210]), and the Order Based mutation (by Syswerda [262]).

Scramble Mutation (SM) (Syswerda [262], 1991):
This is achieved by first selecting a random substring from the offspring and then scrambling the elements in the substring. For example, consider the offspring used earlier, and suppose that the substring (5 3 7 9) is chosen. One possible result is:

New Offspring : (2 8 ***9 5 3 7*** 1 10 6 4).

Displacement Mutation (DM) (Michalewicz [210], 1992):
This operator first selects a substring at random from the offspring. Then, the substring is removed from the offspring and reinserted in a randomly selected place. Suppose that the substring (5 3 7 9) is selected from the offspring. Hence, after the removal of the substring, we have (2 8 1 10 6 4). Suppose that we randomly select the 4th element to be the element after which the substring is inserted. This results in a new offspring:

New Offspring : (2 8 1 10 ***5 3 7 9*** 6 4).

This mutation also called Cut mutation by Banzhaf [25].

Inversion Mutation (IVM) (Fogel [101], 1990 and [102], 1993):
This operator is similar to the DM. The main difference is the substring selected is inserted into the offspring in the reversed order. Based on the offspring mentioned earlier, suppose that the substring (5 3 7 9) is chosen, and that this substring is inserted in reversed order immediately after 4th element. This gives:

New Offspring : (2 8 1 10 ***9 7 3 5*** 6 4).

Banzhaf [25] referred to the IVM as the Cut-Inverse mutation.

4.8 Replacement Strategy

At the end of each generation, the parent population will be replaced by the offspring population. The proportion of individuals in the parent population which are replaced in each generation is defined as the '*generation gap*'.

Holland's GA assume replacement of the whole population *en bloc* at each generation without considering the quality (fitness) of the parent population. He has used a generation gap of 1. We refer to his method as the generational evolution. From the optimisation point of view, this seems a bad decision to make. We may have spent considerable effort in obtaining a good solution, only to run the risk of throwing it away and thus preventing it from taking part in future generations. One of the problems with generational evolution is that an entire generation must be built before we can begin to test the quality of the new individuals generated.

For this reason, De Jong [163] introduced the concepts of *elitism* and *steady-state*. It is inevitable that the fittest individual in each generation can be lost if it is not selected to reproduce or if it is destroyed by crossover or mutation. In the elitism replacement scheme, the fittest individual so far will survive for the next generation by only replacing the remaining $(P_{pop} - 1)$ individuals of the population with the offspring.

Steady-state replacement scheme take this a stage further by replacing only a few (typically two) of the least fit individuals in each generation. This method has advantages as it is implicitly elitist for all high potential individuals and the new individuals added to the population immediately contribute to the quality of the population.

Syswerda [262] shows that, when the individuals to be replaced in the steady-state are randomly selected, the performance of the generational and the steady-state GAs are approximately the same. Most users of steady-state replacement schemes utilise an exponential ranking to select the individuals to be replaced or the worst fit individuals for replacement (see Syswerda [261] and Whitley *et al.* [278]).

4.9 MultiCrossover Genetic Algorithms

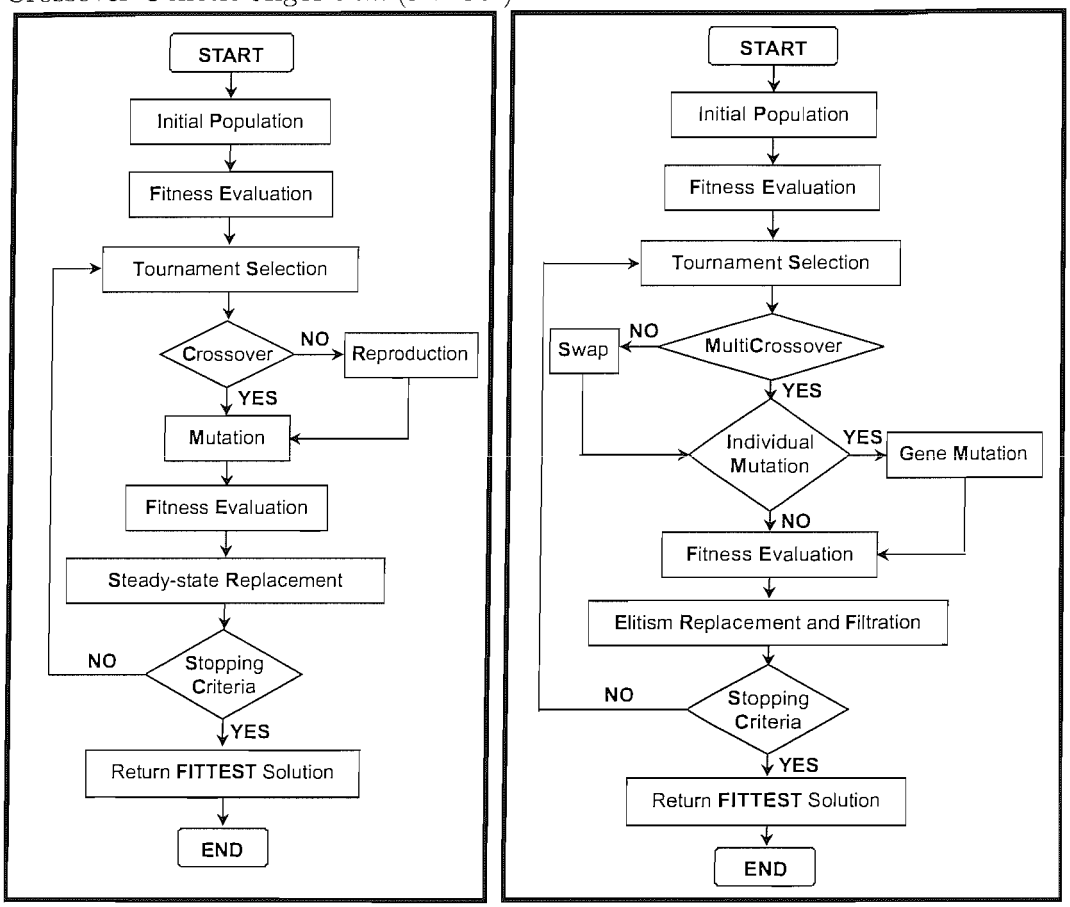
In this section, we concentrate on the discussion of our proposed MultiCrossover Genetic Algorithms (MXGAs) for solving combinatorial optimisation problems. The usual strategy within a GA is to generate a pair of offspring during crossover. We hypothesise that generating multiple offspring during the crossover can improve the performance of a GA.

Since the first crossover operator developed by John Holland [147] in 1975, substantial amount of efforts have been put into the developments of new crossover operators by the GAs community. New techniques are developed to further improve the performance of the crossover operators and the GAs as a whole. The new designs of the crossover operators are also due to the introduction of new gene representations. It is usually the case that a new crossover operator is more complex and complicated than the previous one. Quite often some of the new crossover operators are also quite computationally time consuming in generating offspring.

Our proposed MXGA utilises a multicrossover operator in an effort to achieve better solution quality when compared with a Standard GA (SGA) and other local search algorithms such as Tabu Search and Steepest Descent Method. In the MXGA, offspring for the population are selected from a candidate list of temporary offspring generated via some simple but yet effective classical crossover operators. Various techniques are also introduced into the proposed MXGA to further enhance the solution quality.

In the remainder of this section, we explain the general framework of the proposed MXGA and we also highlight the differences with the SGA in some of the remaining subsections. Since different problem domains require different representation schemes and constraints, the detailed descriptions of the architecture of the MXGA will be given in the later chapters where we solve different combinatorial optimisation problems using MXGAs. A comparison of the general framework of the SGA and the MXGA is given in Figure 4.20.

Figure 4.20: The Framework of a Standard Genetic Algorithm (SGA) vs. Multi-Crossover Genetic Algorithm (MXGA)



Standard Genetic Algorithm (SGA) MultiCrossover Genetic Algorithm (MXGA)

4.9.1 Initial Population

Before an initial population is generated as the starting point for the MXGA, a suitable representation is chosen based on the type of combinatorial optimisation problem to be solved. Once the representation has been decided, a initial population P_{pop}, which is of size 100 in our implementation, is uniformly randomly generated using a random number generator. We have assumed that the size of P_{pop} is kept constant throughout the process.

4.9.2 Selection Mechanism

We use a probabilistic binary tournament selection scheme as the selection mechanism of each parent in the MXGA. As the name suggests, two individuals are chosen at random from the population. A random number r, is then chosen from an uniform distribution defined on [0,1]. If $r < k$ (where k is a parameter), the fitter of the two individuals is selected to be the parent; otherwise the less fit individual is selected. The two are then returned to the original population and can be selected again.

In our study, we set the value of k as 0.75. In other words, we give a 75% chance for the fitter individual to be selected as the parent compared to the less fit individual which only has a 25% chance to be selected. The probabilistic binary tournament selection is a preferred choice over other selection mechanisms such as the Roulette Wheel because the latter method is very sensitive to the fitness function design, making the selection pressure too high or negligible, rendering the selection mechanism unreliable.

4.9.3 Multicrossover Operator

Recall that in all the crossover operators (except SXX and CSEX) applied in a SGA, exactly two offspring are generated from a pair of selected parents each time a crossover operator is executed. The design of the crossover operator needs to be carefully considered based on the representation scheme used and the problem type to be solved. This ensures that only valid offspring can be generated without violating any constraints of the representation scheme and the problem itself. Some crossover operators (e.g. MMX) even need a repair mechanism to generate valid offspring from the invalid ones.

Unlike others, the multicrossover operator in the MXGA uses some of the simple yet effective classical crossover operators such as 1-point and 2-point crossover as the crossover strategy in generating offspring. The main feature of the mul-

ticrossover operator is that it first generates a candidate list of valid temporary offspring from a pair of selected parents through repeated applications of the proposed crossover strategy. Two valid temporary offspring are generated each time by a sequence of steps defined by the crossover strategy. By repeating the strategy for t times, where t is a parameter, a candidate list of $2t$ valid temporary offspring is then generated. Finally, the best and a selected temporary offspring (using the probabilistic binary tournament selection mechanism) will be chosen to be the offspring for the current generation. Initial computational experiments on various combinations of selection methods (e.g. best, worst, random, roulette wheel, ranking and tournament) support the above decision of selecting the offspring from the candidate list of temporary offspring.

The number of repeated applications t needed to generate a candidate list of temporary offspring remains as a parameter. On one hand, generating a large candidate list of temporary offspring can be very time consuming and will results in less time spent on exploring the other regions of the solution space, especially when there is a fixed computation time. On the other hand, a small candidate list of temporary offspring may fail to exploit the potential of the multicrossover operator. For instance, by taking $t = 1$, this in fact can be regarded as a standard crossover operator in a SGA, where both temporary offspring are selected as the offspring for the current generation.

In order to overcome the above problem, the design of the crossover strategy in generating two valid temporary offspring has to be simple, fast and yet effective. With less time spent on repeating the strategy in generating two valid temporary offspring, this means we have more time to spare on exploring the solution space. With this in mind, we develop multicrossover operators which are simple to implement and effective in finding good solutions. The design of the multicrossover operator is based on the representation framework used and the problem type to be solved. Note that the multicrossover operator will only be applied to the selected parents with a given *crossover* probability p_c.

4.9.4 Swap Operator

Depending on the outcome of the crossover probability p_c, the multicrossover may not be applied to the selected parents. Instead of the exact duplicate of the parents as in a SGA via the *reproduction*, a new operator called '*swap*' is used in the MXGA to produce two offspring that are different from their parents. By doing this, we introduce more diversity to the search space.

The basic step of this operator is to randomly select a swap point in a parent and then swap the substrings separated by the swap point to form a new offspring. More swap points can be selected within a parent, but depending on the problem, this might destroy some of the good features from the parent. But on the contrary, this may lead to a more *interesting* region in the search space. In some sense, this swap operator could be regarded as a '*giant*' mutation where the elements in the parent are randomly reassigned.

4.9.5 Mutation Operator

An ideal GA should maintain a high degree of diversity within the population as it evolves from one generation to the next. Otherwise, the population may converge prematurely before the desired solution is found.

We apply the mutation operator in the MXGA in two stages. At first, a subset of individuals is selected from the new offspring population with a given *individual mutation* probability p_M. An offspring is selected if a randomly number q from an uniform distribution defined on [0,1], assigned to the offspring is less or equal to the individual mutation probability (i.e. $q \leq p_M$). Then, the selected offspring will go through the second stage of the mutation process, where each element in the offspring is visited and altered with a given *gene mutation* probability p_m.

There are some concerns in applying the mutation to the offspring. Too low a mutation rate implies too little exploration. With a very small mutation rate (e.g. $p_m = 0.0001$), further exploration of a population ceases once the population

has converged to a local optimum. On the other hand, a very high mutation rate will change every element in the individual randomly. Then the whole evolutionary process is simply a random search with no exploitation of the information acquired from the previous generation.

In this study, we use the value of 0.25 and $1/L$ for p_M and p_m respectively, where L is the length of an individual. In other words, we allow on average, 25% of the offspring population to undergo the gene mutation stage. By doing this, we not only keep the mutation to the offspring on a moderate scale, we also help speed up the MXGA by keeping some of the good offspring generated by the multicrossover operator to the next generation without any disruption by the mutation. The value of $1/L$ is used to allow the gene mutation to introduce small changes in the selected offspring.

4.9.6 Replacement and Filtration Strategies

If a generational evolution is used in a GA, there is no competition between the parents and the offspring so the offspring replaces the parents irrespective of their fitness values. All individuals have the lifetime of exactly one generation and parents are always thrown away, so there is a distinct possibility that valuable information is lost before the next generation.

Our proposed MXGA uses Elitism Replacement scheme where the offspring have to compete with their parents to gain admission to the new population. One advantage of the elitist scheme is that good solutions once found are never lost unless even better solutions are created. During the elitism replacement stage, both parent and offspring population are combined into a single population of size $2P_{pop}$. Then, the individuals of the combined population are sorted in a non-increasing order of their associated fitness value f_i, so that $f_1 \geq f_2 \geq \ldots \geq f_{2P_{pop}}$. The individuals of the new population for the next generation of size P_{pop} is thus the first half of the combined population. The rule is to always select the fittest individuals from the combined population before proceeding to the next generation.

After P_{pop} (population size) individuals have been selected, a process called '*filtration*' is used to identify the identical individuals from the new population. Two individuals are said to be identical if, and only if, the sequence of the elements in both individuals (i.e. *genotype*) are identical. The identical individuals will be removed and replaced by uniformly randomly generated new individuals to avoid '*premature convergence*' and to add diversity to the new population.

As the filtration procedure involves the process of "*identify*", "*re-generate*" and "*re-evaluate*" of the new individuals, which requires a certain amount of computational time, it is sensible to just invoke the procedure every R generations (where R is a parameter, e.g. 50).

4.10 Summary

GAs have been theoretically and empirically proven to provide robust search mechanisms in complex spaces. However, traditional GAs, although robust, are generally not the most successful optimisation algorithm on any particular domain.

The advantage of GAs comes from the fact that the technique can deal with a wide range of problem areas. GAs do not guarantee the global optimum solution to a problem, but they are generally good at finding 'acceptably good' solutions to problems in a reasonable time frame.

Beasley *et al.* [26] give the following remarks:

> "*Where specialised techniques exist for solving particular problems, they are likely to outperform GAs in both speed and accuracy of the final result. The main ground for GAs then, is in difficult areas where no such techniques exist. Even where existing techniques work well, improvements have been made by hybridising them with a GA.*"

Hybridising GAs with the most successful optimisation methods for particular problems is the best of both worlds. When correctly implemented, these algorithms

should do no worse than the method with which the hybridising is done. However, they are still rooms for improvements on a standard GA. See Whitley [277] for more details. Some principle attractions of GAs are given in Reeves [239]:

1. **Generality**: the algorithms work on a coding of a problem, so it is easy to write one general computer program for solving many different optimisation problems. However, a specific coding may have a significant impact on the GA's ability to find good solutions, unless the operator used to search the space is carefully selected with respect to the coding.

2. **Nonlinearity**: many conventional optimisation techniques rely on unrealistic assumptions of linearity, convexity, differentiability, etc. This is not needed by GAs; the only requirement is the ability to calculate some measure of performance, which may be highly complicated and non-linear.

3. **Robustness**: empirical evidence is strong that although it is possible to fine-tune a GA to work better on a given problem, it is nonetheless true that a wide range of parameter settings (population size, crossover and mutation rate, etc.) will give very acceptable results.

4. **Ease of modification**: even relatively minor modification to a particular problem may cause severe difficulties to many heuristics. By contrast, it is easy to change a GA to model variations of the original problem.

5. **Parallel nature**: quite apart from the property of intrinsic parallelism which GAs have been shown to possess, there is great potential for implementing GAs in parallel.

In this chapter, a general framework of the proposed MXGA is introduced. The novelty of the proposed MXGA is the development of the multicrossover operator in generating offspring. The proposed multicrossover operator uses a simple but yet effective classical crossover strategy to generate a candidate list of temporary offspring. The best and a selected temporary offspring are then chosen to be the

offspring of the current generation. Various techniques such as swap procedure, 2-stage mutation operator, elitism replacement and filtration strategy are also introduced into the proposed MXGA to further enhance the solution quality.

Chapter 5

Single Machine Family Scheduling Problem

5.1 Introduction

In this chapter, we address a Single Machine Family Scheduling Problem (SMFSP) where jobs are partitioned into families and setup is required between these families. The objective is to find a schedule which minimises the maximum lateness $L_{\max}$ of the jobs in the presence of the sequence independent family setup times s_f. This SMFSP can be represented as $1|s_f|L_{\max}$ based on the standard classification of Graham *et al.* [128] (as described in Section 3.3.1). We restrict our study to *offline machine scheduling* where it is assumed that the data for the problem instances are known with certainty in advance. According to Hariri and Potts [139], the problem can be defined as follows:

"Given are N jobs, each characterised by a processing time p_j, on a single machine, and a due date d_j, for $j = 1, 2, \ldots, N$, and a partition into F families. For each family f, for $f = 1, 2, \ldots, F$, jobs are split into batches, where a batch is defined as a maximal set of contiguously scheduled jobs from the same family which share the same set up. A sequence independent family setup time s_f, is required at the start of the schedule and also when there is a switch of jobs from another family.

The objective is to find a schedule which minimises the maximum lateness $L_{\max}$, of the jobs in the presence of the family setup times."

The SMFSP for arbitrary F is an NP-hard problem as shown by Bruno and Downey [38]. The general assumptions of the SMFSP in this study are:

1. all input data are positive integers;
2. each job becomes available for processing at time zero;
3. the machine becomes available for processing at time zero;
4. the machine can only process at most one job at a time;
5. once processing begins on a job, it is processed to completion without interruption;
6. the machine cannot perform any processing while undergoing a setup.

Set ups include adjusting tools, positioning work in process material, paint drying, cleanup, chemical reaction to be completed, etc. The majority of scheduling research assumes setup times as either negligible and hence ignored or considered as part of the processing time. While this assumption simplifies the problem, it adversely affects the solution quality for many applications which require explicit treatment of setup (cite in Allahverdi *et al.* [6]). For example, imagine jobs that each belong to a particular family, where jobs in a family tend to be similar in some way, such as their required tooling or their container size. As a result of this similarity, a job does not require a set up when following another job from the same family, but a known *family setup time* is required when a job follows a member of some other family. We call this a *family scheduling model.* Typically, there is a large number of jobs, but a relatively small number of families.

The main motivation of this study is derived from a complex trade-off at the core of many scheduling problem in practice. This trade-off involves balancing the machine efficiency of long production runs of a similar jobs, against the customers' satisfaction gained from completing the jobs before or by their due dates. At one extreme, if efficiency is more important, we find that the batch sizes tend to be large to allow many jobs to run on a similar set up. However, if resources

are committed to long production runs, other jobs tend to get delayed, thus not achieving their due dates. At the other extreme, when a good due date performance is warranted, the batch sizes are kept small so that priorities can be shifted. This allows jobs that face the most urgent due date pressures to be completed ahead of the other jobs. By doing so, it is observed that this shifting may require a number of set ups, and lead to a loss of productive efficiency. In the long run, this efficiency loss will in turn lead to a diminished ability to meet due dates. Thus, there is an obvious inherent conflict between efficiency and due date performance. This conflict represents a challenge to any scheduling procedure used for short-term scheduling of production batches.

In this study, the aim is to develop a MultiCrossover Genetic Algorithm (MXGA) that utilises the multicrossover operator to achieve better solution quality compared to a standard genetic algorithm and other local search algorithms namely Tabu Search and Descent Method. We use a standard 1-point or F-point crossover strategy to produce two temporary offspring. Detailed descriptions of the proposed multicrossover operator are discussed in Section 5.4. Various techniques are introduced into the MXGA to further enhance the solutions. The architecture of the MXGA used in this chapter is based on the framework discussed in Section 4.9.

In next section, we review some approaches used for the SMFSP with setup times. Some properties of the Earliest Due Date rules for SMFSP are stated in Section 5.3. The developments of the MXGA for solving SMFSP with sequence independent setup times are the focus of Section 5.4. Some of the main components in the MXGA are discussed in detail. Section 5.5 provides an insight into the local search algorithms we designed specifically for comparison purposes with the MXGA. Extensive computational experiments are carried out in Section 5.6. We end this chapter by giving some concluding remarks in Section 5.7.

5.2 Approaches to Single Machine Family Scheduling Problem

In this section, we concentrate on the review of the literature for the SMFSP with the presence of the family setup times. Various approaches, namely exact approaches, heuristic and local search algorithms have been proposed to solve the problem. Some excellent and comprehensive reviews of the SMFSP which involving setup consideration and batching can be found in Potts and Van Wassenhove [234], Webster and Baker [275], Liaee and Emmons [187], Allahverdi *et al.* [6], Yang and Liao [286], and Potts and Kovalyov [233].

It is worth mentioning that, in the event where all the setup times are zero, the problem of minimising the maximum lateness and minimising the total (weighted) completion time are solved in $O(N \log N)$ time by Jackson's Earliest Due Date (EDD) rule (see, Jackson [156]) and Smith's Shortest (Weighted) Processing Time (SWPT) rule (see, Smith [256]) respectively.

5.2.1 Exact Approaches

Maximum Lateness

In 1989, Monma and Potts [214] consider a variety of SMFSPs under the assumption that sequence dependent setup times s_{fg}, for families satisfy the '*triangle inequality*': the setup time associated with a changeover from family f to h is assumed to take no longer than that for the changeover from family f to g, followed by a changeover from family g to h. Using dynamic programming (DP), they show the problems with the objective (minimisation) $L_{\max}$, $\sum_j w_j C_j$, and $\sum_j U_j$ to be efficiently solvable for a fixed number of batches. Their DP approach solve $1|s_{fg}|L_{\max}$ and $1|s_{fg}|\sum w_j C_j$ in $O(F^2 N^{F^2+2F})$ time, and $1|s_f|L_{\max}$ and $1|s_f|\sum w_j C_j$ in $O(F^2 N^{2F})$ time. Thus, the DP algorithms are polynomial time bounded by the number of jobs but exponentially bounded in the number of families. Potts [232] shows that the time complexity for $1|s_f|L_{\max}$ and $1|s_f|\sum w_j C_j$

can be reduced to $O(N^3)$ when $F = 2$. Unfortunately, this DP approach is not of practical use unless F is very small.

An improved backward DP approach with job insertion which schedules the jobs from the back to the front (i.e. in non-decreasing order of their indices within the families) is proposed by Ghosh and Gupta [111] for $1|s_{fg}|L_{\max}$, which requires $O(F^2N^F)$ time. However, their approach is only practical when F is very small.

In 1996, Schutten *et al.* [250] develop a branch and bound (B&B) approach for the problem of $1|r_j, s_f|L_{\max}$. In the presence of release dates r_j, no results are known about the order of jobs within a family. A key component of their algorithm is the use of dummy jobs to represent setups. A lower bound is obtained by relaxing setups and solving the corresponding preemptive problem, and the approach uses a forward branching rule. Computational results show that the algorithm is effective in solving instances for up to about 40 jobs.

A year later, Hariri and Potts [139] develop a B&B approach where all jobs are ready at time zero for the problem $1|s_f|L_{\max}$. They first obtain an initial lower bound by ignoring setup, except for those associated with the first job in each family, and solved the resulting problem with EDD rule. This lower bound is then improved by a limited enumeration that considers whether or not certain families are split into at most two batches. Their B&B algorithm optimally solved problems with up to 50 jobs.

In 1997, Pan and Su [225] develop a B&B algorithm for problem $1|s_f|L_{\max}$. They derive some fundamental properties of an optimal schedule to simplify the problem. Several dominance criteria and a lower bound of the optimal lateness are also developed to construct the B&B algorithm. The computational results reveal that the proposed algorithm effectively solves problems up to 30 jobs.

In 2000, Baker and Magazine [22] provide an algorithm that uses a B&B approach combined with dominance properties which reduced the effective problem size to solve the problem of $1|s|L_{\max}$ (s = identical setup time). They establish that the size of the problems that can be solved is a function of the number of

families, the number of jobs per families, the relative size of the setup time and the relative due date range. The identification of composite jobs allows the effective problem size to be reduced before the enumeration begins. For the most difficult categories, they solve problems for up to 60 jobs.

Total (Weighted) Completion Time

In 1991, Mason and Anderson [208] define the changeover for a job in a new family as the set down operation from the previous family and a set up operation for the new family. When the setdown times are all zero, they show that the changeover structure is equivalent to sequence independent family setups. They derive various dominance rules and constructed a B&B algorithm for $1|s_f|\sum w_jC_j$. Their lower bound is derived using objective splitting: the total weighted completion time can be partitioned into contribution from the processing times and from the setup times, which are optimised separately. Their algorithm is able to solve problems up to 30 jobs. However, the algorithm can only be effective when the number of families are small compared with the number of jobs.

Crauwels *et al.* [59] propose a B&B approach for problem of $1|s_f|\sum w_jC_j$. They obtain a lower bound by performing a Lagrangian relaxation of the machine capacity constraints in a time-indexed formulation of the problem. Their first algorithm uses a forward branching rules and multiplier adjustment method for obtaining the lower bound, while the second algorithm uses a binary branching rule and subgradient optimisation method for computing the lower bound. Computational results show that the first algorithm solves problems with up to 70 jobs, and is more efficient than both Mason and Anderson's algorithm, and also their second algorithm.

In 2000, Dunstall *et al.* [81] introduce two new lower bounds for problem $1|s_f|\sum w_jC_j$. These lower bounds are shown analytically to dominate Mason and Anderson's lower bound and can be computed more efficiently than the Lagrangian lower bound of Crauwels *et al.* [59]. An improved B&B algorithm of Mason and Anderson [208] through the addition of a new dominance rule and the

substitution of the lower bounds is constructed. Their algorithm efficiently solves problems with 50 or more jobs, depending on the values of setup times.

Other Objective Functions

Chen [47] proposes a polynomial DP algorithm for solving the problem of earliness-tardiness penalties for two criteria. The first criterion minimises the total tardiness and earliness penalties, while the second extends the first criterion to include the total due date penalty. He shows that the algorithm has a running time polynomial with respect to the number of jobs but is exponential with the number of batches.

5.2.2 Heuristics and Local Search Algorithms

Maximum Lateness

Problem $1|s_f|L_{\max}$ has been an interest of many researchers in recent years. In 1991, Zdrzałka [287] proposed heuristic methods for $1|s_f|L_{\max}$ in which there are unit setup times. To facilitate the worst-case analysis, he assumes that all due dates are non-positive. When all jobs of a family are scheduled contiguously, the resulting schedule is shown to have a maximum lateness which does not exceed twice the optimal value. He also suggests an improvement which allows each family to be split into at most two batches. The improved heuristic requires $O(N^2)$ time and generates a schedule for which the maximum lateness does not exceed $\frac{5}{3}$ times that of an optimal schedule.

Four years later, Zdrzałka [288] designed two approximation algorithms for the problem without the unit setup time assumption and with non-positive due dates. The algorithm starts with a schedule in which each batch contains all jobs from a family, and allows each family to be split into at most two batches. The algorithm requires $O(N^2)$ time, and it generates a schedule with maximum lateness that is no more than $\frac{3}{2}$ times the optimal value. His algorithm can be adapted for the problem of $1|r_j, s_f|C_{\max}$ to generate a schedule of maximum lateness that is no more than $\frac{5}{2}$ times the optimal value.

Hariri and Potts [139] propose a single batch heuristic in which all jobs of a family form a batch, and a double batch heuristic in which each family is partitioned into at most two batches according to the due dates of its jobs. They show that both heuristics require $O(N \log N)$ time. They also show that the single batch heuristic has a worst-case performance ratio of $2 - \frac{1}{F}$, whereas a composite batch heuristic which selects the better of the schedules generated by the single and double batch heuristic has a worst case analysis of $\frac{5}{3}$ for arbitrary F.

Woeginger [282] investigates a SMFSP in which each job has a processing time and a delivery time. The objective is to find a schedule of jobs that minimises the time by which all jobs are delivered with the presence of the sequence independent family setup times. This problem is equivalent to $1|s_f|L_{\max}$. Woeginger formulates the problem using DP and applies the Trimming-The-State-Space technique to cut the state space down to polynomial size and simultaneously demonstrate the existence of a Polynomial-Time Approximation Scheme (PTAS) for the problem.

Baker [21] develops a procedure called Gap Heuristic that exploits a splitting condition while adding jobs, one at a time, to a schedule for the problem of $1|s|L_{\max}$ (s = identical setup time). Its computational requirement is $O(N^2 \log N)$ since each iteration inserts one job into the schedule, and there could be reordering of the batches with each insertion so that their batch due dates are in order. He suggests two heuristic procedures that use neighbourhood search routines to improve the existing heuristic. He defines a $C-$neighbourhood by choosing a batch and combining it with the next earlier batch of the same family, and a $S-$neighbourhood which involves splitting off the last job from a batch and inserting it later in the schedule, possibly as a separate batch. Computational results indicate that his hybrid heuristic (on average) produces results where the difference between the heuristic solution and an optimal solution is approximately equal to the average job processing time.

Pan *et al.* [224] propose a mathematical programming model for $1|s_f|L_{\max}$. The heuristic algorithm solves the problem by first finding an initial schedule and then applying merging properties (forward and backward mergers) to improve the initial

schedule. Their proposed algorithm can be modified to find approximate solutions that minimise the maximum tardiness. The computational results reveal that the proposed method produces more accurate solutions for maximum tardiness problems than for maximum lateness problems and is efficient in solving problems of up to 1000 jobs.

Shin *et al.* [253] propose a tabu search (TS) for the problem of $1|r_j, s_{fg}|L_{\max}$. The tabu search is composed of two parts: a MATCS (Modified Apparent Tardiness Cost with Setups) rule for finding an efficient initial solution, and a tabu search approach to seek a near optimal solution from the initial solution. They also develop a restricted neighbourhood generation scheme to find a better neighbourhood schedule more efficiently. They explore a hybrid move operator which alternates insert move and swap move as the search progresses. They compare the TS with the RHP (Rolling Horizon Procedure) heuristic proposed by Ovacik and Uzsoy [223] for problem instances up to 100 jobs. The computational experiments show that the TS outperforms RHP heuristic in terms of the computational time and solution quality.

In 2004, Schultz *et al.* [249] propose a new neighbourhood search heuristic for solving problem $1|s_{fg}|L_{\max}$ based on the properties and theorems presented by Hariri and Potts [139] and Baker [21]. Of particular interest is Hariri and Potts' problem reduction procedure that identifies the condition under which two jobs from the same family must be scheduled contiguously and can thus be replaced by a single composite job, therefore reducing the overall problem size. The procedure is shown to be effective, producing optimal/near optimal solutions over a wide range of problem instances and is computationally efficient for large problems (500 jobs).

Total (Weighted) Completion Time

Gupta [132] and Ahn and Hyun [2] present heuristic methods for solving the problem of $1|s_{fg}|\sum C_j$. Gupta's method constructs a partial schedule using the earliest completion time rule: the job which is appended to the current partial sequence is chosen so that its completion time is as small as possible. Ahn and Hyun [2] sug-

gest an improvement heuristic which attempts to reduce the total completion time of the current sequence by shifting contiguously scheduled jobs from the same family to another position. Computational results show that the improved heuristic generates better solutions compared to Gupta's method.

Mason [207] design a genetic algorithm (GA) for problem $1|s_f|\sum w_jC_j$ using a binary representation of solutions. Each element in the representation indicates whether or not the corresponding job starts a batch. He uses standard genetic operators in the algorithm.

Herrmann and Lee [144] study problem of $1|\overline{d}_j, s_f|\sum C_j$ by introducing an extended heuristic for the Constrained Flowtime with Setup problem (CFTS). The Multiple-Pass Minimum Waste heuristic performs well at minimising the total flowtime of CFTS. They use a GA to improve the solution quality by adjusting the inputs of the heuristic. This GA includes a penalty function for infeasible points that increases the cost of tardiness as the search progresses.

Williams and Wirth [281] propose a new heuristic for $1|s_f|\sum C_j$ solved in $O(N^4)$ time, based on the properties derived by Mason and Anderson [208] for an optimal schedule. Their heuristic performs well when tested against the no family splitting heuristic and Gupta's heuristic for problems of 50 jobs.

Crauwels *et al.* [61] investigate four local search heuristics: descent method, simulated annealing, threshold accepting, and TS for a problem of $1|s_f|\sum w_jC_j$. They use the neighbourhood search procedures proposed by Ahn and Hyun [2] in their local search heuristics. All four heuristics are reported to yield less than 0.4% deviation from the optimal solution for problems with up to 50 jobs. The best results are obtained with a hybrid method which uses the multistart version of a TS when the number of families is small, and uses Mason's GA for a large number of families.

Wang *et al.* [273] design a GA based on fundamental runs theory for the problem of $1|s_f|\sum C_j$. The numerical results show that the computational performance of the GA depends on the number of 'fundamental' runs, and not the

number of jobs. When the number of groups is much less than the number of jobs, the number of fundamental runs is usually much less than the number of jobs.

Other Objective Functions

Crauwels *et al.* [60] propose multistart descent method, simulated annealing, TS, and GA for the problem of $1|s_f|\sum U_j$. The neighbourhood search algorithms use either job or batch neighbourhood. Computational results for problems up to 50 jobs show that the GA performs the best compared to other local search algorithms.

Nowicki and Zdrzałka [217] propose a general TS approach for solving any general cost functions on a single machine with major sequence independent family setup times and minor setup times for jobs within families. They evaluate the approach computationally for the objectives of minimising the maximum weighted lateness and total weighted tardiness on the problem instances ranging between 40 and 200 jobs.

Webster *et al.* [276] propose and investigate a GA for scheduling jobs with an unrestricted common due date. The objective is to minimise total earliness and tardiness cost where early and tardy penalty rates are allowed to be arbitrary for each job. They compare the computational results of a GA with a B&B procedure on problem instances up to 30 jobs.

A Lagrangian relaxation based approach is developed by Sun *et al.* [259] for problem $1|s_{fg}|\sum w_j T_j^2$. The primal problem is decomposed into job level subproblems which are solved optimally and an approximate dual problem is then solved using subgradient technique. The result of the relaxation is a list of jobs sequenced by starting times that is then improved via a three way swap.

Armentano and Mazzini [13], design a GA for problem $1|s_{fg}|\sum T_j$. They compare the test problems with those obtained by the CPLEX software and the ATCS (Apparent Tardiness Cost with Setups) heuristic. For small problems, their proposed GA yield near optimal solutions for most of the problems tested. For larger

problems, the GA outperforms ATCS in 93% of the test problems using reasonable computational time.

Suriyaarachchi and Wirth [260] introduce some properties of an optimal schedule for the problem of $1|s_f|\sum E_j, \sum T_j$. They also present a fast heuristic procedure for the problem based on the proposed properties. Its performance is compared with a lower bound, a greedy heuristic, a genetic algorithm, and for small problems, the optimal solution.

5.3 Earliest Due Date (EDD)

To specify the problem of SMFSP with family setup time more formally, consider N jobs that are divided into F families. Each family f, for $1 \leq f \leq F$, contains n_f jobs where $n_1 + n_2 + \cdots + n_F = N$. We used the subscript pair (f, j) to identify the jth job from family f. Each job becomes available for processing at time zero, and is to be scheduled on a single machine. Let p_{fj} denote the processing time of job (f, j), for $1 \leq j \leq n_f$, and d_{fj} is its due date. A sequence independent family setup time s_f is required at the start of the schedule if a job of family f is the first to be processed and on the occasion when there is a switch in the processing of jobs from one family to family f.

In the event when there is no setup time in the problem, we can minimise the $L_{\max}$ by sequencing jobs using the EDD rule (see Jackson [156]).

Property 1 **(EDD Rule)**
For a given set of N jobs, with known processing times and due dates, the minimum value of $L_{\max}$ is achieved by sequencing the jobs in non-decreasing order according to their due dates.

According to Monma and Potts [214], there exists an optimal solution in which jobs within each family are sequenced in non-decreasing order of the due dates, that is in EDD order. We restate their result below.

Property 2: **(EDD within family)**
There is an optimal schedule such that the jobs within each family are sequenced in non-decreasing order of their due dates.

The practical implication of Property 2 is to focus scheduling decisions on choosing a family rather than choosing a job. The rule is aimed at determining which family is most critical. Within that family, the job with the earliest due date should come next. This means that the jobs in a family should appear in EDD order ($d_{fj} \leq d_{fj+1}$). Suppose that all jobs in family f are processed as a single batch, and the family due date d_f is defined as:

$$d_f = \min_{j=1,2,\ldots,n_f} \{d_{fj} + q_{fj}\} \tag{5.1}$$

where q_{fj} represents the processing time in family f that occurs after job (f,j) and is sometimes called the '*tail*' of job (f,j),

$$q_{fj} = \sum_{i=1}^{n_f} p_{fi} - (p_{f1} + p_{f2} + p_{f3} + \cdots + p_{fj}). \tag{5.2}$$

With this, one can construct an optimal schedule by sequencing the jobs in a non-decreasing order according to their due dates within each family and then continue with sequencing the families in a non-decreasing order of their family due dates. The concept of sequencing according to family due dates has broader applicability in schedules where families are split into two or more batches. A batch is a maximal group of contiguously scheduled jobs within a family. Let $(f,h),\ldots,(f,k)$ be the jobs of an arbitrary batch b, and the batch due date δ_b is defined as:

$$\delta_b = \min_{j=h,\ldots,k} \{d_{fj} + q_{fj}\} \text{ where } q_{fj} = \sum_{i=h}^{k} p_{fi} - (p_{fh} + \cdots + p_{fj}). \tag{5.3}$$

These batch-related parameters help the EDD rule adapt to sequencing the batches in the way as explained in Baker [21].

Property 3 **(EDD Rule for batches)**
There exists an optimal schedule where the batches are sequenced in a non-decreasing order according to their due dates.

This rule should be followed whenever one is considering scheduling in batches. Due dates may not always be relevant but generally, the meeting of deadlines is a major concern in scheduling problems. Scheduling to minimise lateness is a common way of making job completion times conform to due dates.

In an optimal schedule, jobs within each batch are sequenced using the EDD rule according to their job due date d_{bj}. Then, the batches in the schedule are sequenced in a non-decreasing order according to their batch due dates δ_b (i.e. $\delta_b \leq \delta_{b+1}$). A sequence independent family setup time s_f, is added before the start of each batch. The maximum lateness $L_{\max}$, of the schedule is

$$L_{\max} = \max_j \{L_j\} \tag{5.4}$$

where $L_j = C_j - d_j$, C_j = completion time of job j.

5.4 MultiCrossover Genetic Algorithm

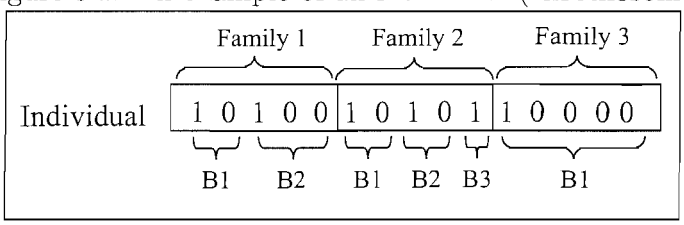
In this section, we propose a MXGA for solving a SMFSP to minimise the maximum lateness. To the best of our knowledge, no research has been carried out on the application of GAs for the problem of $1|s_f|L_{\max}$. In the following subsections, we will discuss some of the main components in the MXGA based on the architecture described in Section 4.9. A general framework of the proposed MXGA is summarised in Figure 4.20.

5.4.1 Representation

The proposed MXGA is developed using binary {0,1} representation to define the partition of families into batches, where '1' means the first job in a batch and '0' means a contiguously sequenced job in a batch. This representation is used by

Mason [207] in his GA for solving the problem of $1|s_f|\sum w_jCj$. The genes can be selected freely except the first gene (job $(f,1)$) in each family f, where '1' is placed to indicate the start of a family. The length of the chromosome (individual) corresponds to the number of jobs N, to be scheduled. Figure 5.1 shows an example of the gene representation for an individual with 15 jobs in 3 families. Note that the gene representation for the first job $(f,1)$ in each family f is always '1'. In this example, we have two batches in family 1, three batches in family 2 and one batch in family 3.

Figure 5.1: An example of an individual (chromosome)



During the decoding stage, genes in each individual will be decoded into a sequence of batches. Having calculated the batch due date using (5.3), the batches are scheduled in a non-decreasing order of their batch due dates (property 3).

5.4.2 MultiCrossover

Multicrossover is considered as the primary genetic operator used in the MXGA. Based on a crossover probability, p_c, two offspring will be produced from a pair of selected parents. As a result of the job permutation used as the gene representation in most previous studies, specially designed crossover operator such as Partially Mapped Crossover (PMX) ([252]) and Order Crossover (OX) ([182]) are used to generate feasible solutions. Unlike others, our proposed crossover operator uses standard 1-point or F-point crossover to produce two temporary offspring by crossing two selected parents in each cycle of steps. In this case, F defines the total number of families in the schedule. Thus, every family in the parent is involved in the crossover. The process of F-point crossover strategy is as follows.

S 1: Select randomly a crossover point in family 1 to be used in both P1 and P2 (Parent 1 and 2).

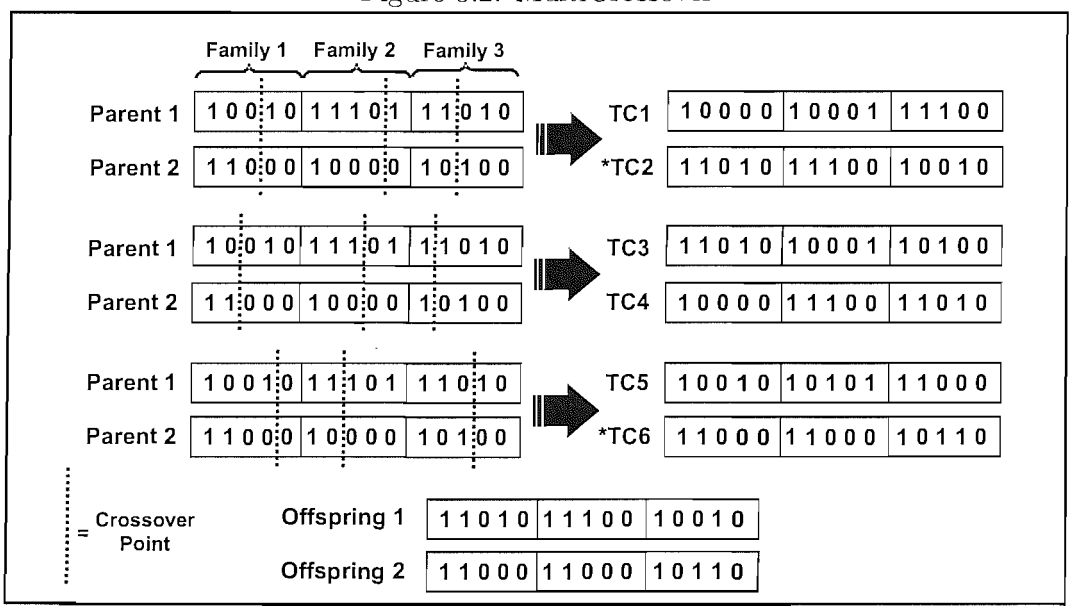
S 2: Exchange the '*tails*' of family 1 in both P1 and P2 to form two new temporary family partitions.

S 3: A randomly generated number will determine the assignment of the new temporary family partitions in TC1 or TC2 (Temporary Offspring 1 and 2).

S 4: Repeat **S 1 – S 3** for each family f ($f = 2, 3, \ldots, F$) in both P1 and P2 until two complete TC1 and TC2 are formed.

The steps above will be repeated t times to produce $2t$ temporary offspring. The best and a selected temporary offspring (using the probabilistic binary tournament selection mechanism) are then chosen to be the offspring for the current generation (refer to Figure 5.2 for an example with $t = 3$). Note that the steps can be easily modified to complement a 1-point crossover strategy.

Figure 5.2: MultiCrossover



5.4.3 Swap

As described in Section 4.9, the swap operator is applied to produce two new offspring when the multicrossover is not applied to the parents. This is achieved by doing the following:

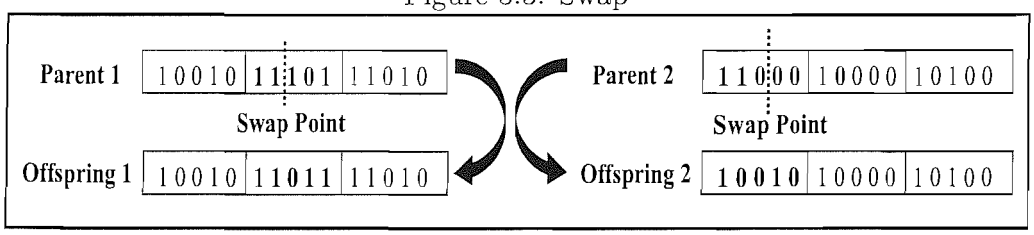
S 1: Select randomly a family and a swap point within the family from a parent to form two sub-genes.

S 2: Swap the position of the sub-genes (except the first job in the selected family) with the swap point as the point of exchange.

S 3: The other genes from the other family remain unchanged.

The steps above are repeated for the second parent to create a second offspring. In Figure 5.3, a swap point is chosen randomly between the second gene and the third gene from family 2 in parent 1. Two sub-genes ({1}, {1,0,1}) are formed in family 2. Note that the first gene (job (2, 1)) in family 2 is not in the list of the sub-genes and it will remain unchanged. We then swap the sub-genes in family 2 while the genes from the other families (1 and 3) remain unchanged. This results in a completely new offspring from the parent. Similarly, offspring 2 is formed from parent 2 where a swap point is chosen in family 1 in parent 2. Note that swap will result in a new structure of the gene representation but not the structure of the jobs. The genes will only be decoded into jobs in the decoding stage.

Figure 5.3: Swap



5.4.4 Mutation

After a crossover or swap procedure is performed, mutation takes place. We used two mutation operators in our MXGA. First, an offspring is selected for the procedure of the *gene mutation* based on an *individual mutation* probability, p_M. Then, each gene of the selected offspring (except the first gene in each family) is visited and flipping the '1' to '0' or vice versa with a given gene mutation probability, p_m.

When a gene is flipped from '1' to '0', it means we combine two contiguously scheduled batches into one single batch with the total number of the jobs in the new batch equal to the sum of the jobs in the previous two separated batches. We split a single batch into two separated batches if the gene is flipped from '0' to '1'. The flipped gene will be the first job in the second batch.

Figure 5.4: Job Mutation

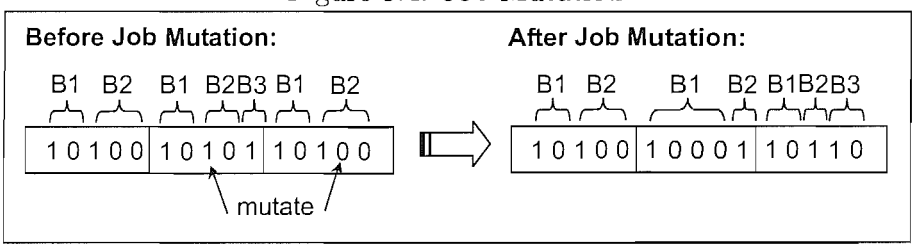


Figure 5.4 shows an example of gene mutation of an individual at the third gene in family 2 and the fourth gene in family 3. With the given P_m, we flipped the gene for job $(2, 3)$ from '1' to '0' and from '0' to '1' in the gene for job $(3, 4)$. Note that the number of batches in family 2 has been reduced to two with four jobs and one job in the first and second batch respectively. The number of batches in family 3 increased from two to three, where there are two, one and two jobs in each batch respectively.

5.5 Competitors – Performance Measure

In order to measure the effectiveness and quality of the MXGA in solving the problem, we compare the MXGA to those of a standard GA (SGA), a Tabu Search (TS) and a Descent Method (DM). All the local search algorithms adopt the binary representation used in the MXGA. To form an initial solution, the jobs representation are uniformly randomly generated except for the first job (i.e. job $(f, 1)$) in each family f, where '1' is placed to indicate the start of a family.

We first consider two neighbourhood approaches suggested by Ahn and Hyun [2] for problem $1|s_{fg}|\sum C_j$. They suggest that a neighbour can be constructed by shifting a job or a sub-batch forward or backward, while maintaining the Shortest Weighted Processing Time (SWPT) property for jobs within each family. Their approaches have been successfully applied by Crauwels *et al.* [61] within descent, simulated annealing, threshold accepting, and tabu search for problem $1|s_f|\sum w_j C_j$. Since the approaches proved to be very efficient in finding near-optimal solutions for the problem of minimising the completion time, we believe that the approaches will also be useful for our problem of minimising the maximum lateness. Detailed descriptions of the neighbourhood approaches can be found in Crauwels *et al.* [61]. It has been observed by Ahn and Hyun [2] that the *shift job* neighbourhood is smaller than the more general *shift sub-batch* neighbourhood. Conventionally, the complete neighbourhood is searched at each iteration to find the best possible move in DM and non-tabu move in TS. Thus, it is advantageous to choose a small neighbourhood. Therefore, we use the *shift-job* neighbourhood in preference to *shift sub-batch* neighbourhood.

It is convenient to describe the *shift job* neighbourhood using an example. Consider a sequence

$$S = (\ 1\ 0\ 1\ 0\ 0\ |\ 1\ 0\ 1\ 0\ 1\ |\ 1\ 0\ 0\ 0\ 1\ )$$

which comprises 7 batches in 3 families ($f = 1, 2, 3$) (vertical line " | " divides the jobs into families). For a *forward shift* of a single job, we select the first job of a

batch and swap its job representation with the second job in the same batch. For instance, consider the batches in family 1, by swapping the job representation of the third job (first job of batch 2) and the fourth job (second job of batch 2) in family 1, we obtain the sequence

$$S_1 = (\ 1\ 0\ 0\ 1\ 0\ |\ 1\ 0\ 1\ 0\ 1\ |\ 1\ 0\ 0\ 0\ 1\ ).$$

As a result, the third job in family 1 is now the last job in batch 1, and the fourth job in family 1 has became the first job in the second batch of family 1. Note that the actual sequence of the batches in an optimal schedule is determined during the decoding stage where the batches are sequenced in a non-decreasing order according to their batch due dates.

Similarly, for a *backward shift* of a single job, we select the first job of a batch and swap its job representation with the last job from the previous batch. Consider again the batches in family 1, by swapping the job representation of the third job (first job of batch 2) and second job (last job of batch 1) in family 1, we obtain the sequence

$$S_2 = (\ 1\ 1\ 0\ 0\ 0\ |\ 1\ 0\ 1\ 0\ 1\ |\ 1\ 0\ 0\ 0\ 1\ ).$$

In this case, the third job in family 1 is now the second job of batch 2 while the second job in family 1 has became the first job in the second batch of family 1.

It is worth mentioning that the *shift job* neighbourhood discussed earlier does not create any extra batches in the schedule. However, we extend it so that it can create an extra batch in consisting of a single job. For a *forward shift*, we select the second job in a batch and alter the job representation from '0' to '1', leaving the first job of the batch to form an extra batch by itself. Consider again the second batch in family 1 from sequence S, by altering the job representation of the fourth job in family 1 (second job in batch 2) from '0' to '1', we obtain the sequence

$$S_3 = (\ 1\ 0\ 1\ 1\ 0\ |\ 1\ 0\ 1\ 0\ 1\ |\ 1\ 0\ 0\ 0\ 1\ ).$$

Similarly, for a *backward shift*, we select the last job of a batch and alter the job representation from '0' to '1'. By doing this, the selected job has became an extra

batch by itself. Consider again the sequence S, by altering the job representation of the second job in family 1 (last job in batch 1) from '0' to '1', we get the following sequence

$$S_4 = (\ 1\ \ 1\ \ 1\ \ 0\ \ 0\ \mid\ 1\ \ 0\ \ 1\ \ 0\ \ 1\ \mid\ 1\ \ 0\ \ 0\ \ 0\ \ 1\ ).$$

5.5.1 Dynamic Length Tabu Search

A dynamic length tabu list of tabu search (DLTS) is designed for our problem using *shift job* neighbourhood. The tabu list length is dynamically controlled during implementation in order to achieve better solution quality. Such processes can have an important influence on which moves are available to be selected at a given iteration.

The basic role of the tabu list is to prevent cycling. If the length of the tabu list is too short, tabu search may keep returning to the same local optimum, thus preventing the search process from exploring a wide area of the solution space. Conversely, a tabu list that is too long creates too many restrictions. It also results in excessive computational time to search the tabu list to determine if a move is tabu. As a result, less time is available for the procedure to explore in the solution space within a given computational time limit. Therefore, the length of the tabu list should be as short as possible but long enough to allow the search to move away from the local optimum. An effective way of overcoming this difficulty is to use a variable length tabu list where each element of the list is active for a number of iterations, that is bounded by the given maximum and minimum values.

In the DLTS, a tabu list is created to prevent moves that shift certain jobs. After a move is executed, the job that is shifted is stored in the tabu list, or both jobs are stored if the move is effectively the transpose of adjacent jobs. Thus, a neighbour is tabu if it is generated by shifting one of the jobs in the tabu list. As in the standard tabu list procedure, whenever the list becomes full and a new entry is to be added, the oldest element is overwritten.

We also introduced an aspiration criterion (as described in Section 2.5.2) into the DLTS to prevent the occasional loss of good solutions due to the tabu list. If the solution value of a tabu neighbour is better than that for all solutions generated thus far, then its tabu status is overridden.

5.5.2 Randomised Steepest Descent Method

Having successfully developed the DLTS using *shift job* neighbourhood search, a steepest descent method (SDM) using the same neighbourhood search procedure is developed. It adopts an acceptance rule that allows neutral moves to be made for up to M consecutive iterations (where M is a parameter, e.g. $M = 1000$) before terminating the algorithm. The SDM is known to be a very greedy neighbourhood search method which finds the local optimum quickly. But, the risk of visiting the same solutions previously found thus creating a cycle within this solutions set is also high. To remedy this drawback, we introduced a randomisation strategy into the algorithm when there are multiple identical good solutions (i.e. improving and neutral moves) found in a single iteration. A move is selected randomly from the list of the identical good solutions. We believed the strategy will help the search to escape from falling into the same local optimum and continue its search in the solution space. Also note that deteriorating moves are not considered in the algorithm. In other words, the algorithm will terminate once the best schedule found in the current iteration is worse than the best schedule found so far.

5.6 Computational Experience

In this section, we report on computational results of our proposed local search algorithms. For TS and DM, we present results which show how the choice of parameters affect solution quality. For example, we investigate the different range of tabu list length in DLTS and the performance of the randomisation in our proposed RSDM compared to the standard DM. We also present results of our

proposed MXGA at the different stages of development. Although many additional parameter setting tests were performed to obtain a 'good' implementation of each algorithm, only the most significant are reported. Having found suitable parameter settings for each method, we complete this section by presenting extensive computational results for the different local search algorithms proposed in the previous sections.

5.6.1 Experimental Design

Problem instances with 50 and 100 jobs, and with 4, 8 and 12 families are generated. Jobs are distributed uniformly across families, so that each family contains $\lfloor N/F \rfloor$ or $\lceil N/F \rceil$ jobs. In each problem, processing times are randomly generated integers from an uniform distribution defined on [1,100]. Having generated processing times and computed $P = \sum_{f=1}^{F} \sum_{j=1}^{n_f} p_{fj}$, five sets of integer due dates are generated from the uniform distribution [0,αP], where $\alpha \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$. We label each set of due dates range as follows:

D1: [0,0.2P]
D2: [0,0.4P]
D3: [0,0.6P]
D4: [0,0.8P]
D5: [0,1.0P]

Setup times are integers from the following uniform distributions (based on Hariri and Potts [139]):

Class A: [1,100] (medium);
Class B: [1,20] (small);
Class C: [101,200] (large).

For each combination of N, F, α and setup times class, five problem instances are created. The algorithms are coded in ANSI-C using Microsoft Visual C++ 6.0 as the compiler and run on a Pentium 4, 2.0 GHz computer with 512MB memory. Since the optimal solutions are not known, we use a lower bound to assess the quality of solutions generated by the algorithms. The lower bound used for each test problem is presented in Section 3.3.2. Algorithms are compared by

listing, for each combination of value N, F, α and setup times class, the average relative percentage deviation (ARD) (equation (5.5)) and the maximum relative percentage deviation (MRD) (equation (5.6)) of the heuristic solution value from the lower bound.

$$ARD = \frac{\sum_{i=1}^{I}\sum_{r=1}^{R}\left(\frac{UB_{ir}-LB_i}{LB_i}\times 100\%\right)}{I\cdot R}, \tag{5.5}$$

$$MRD = \max_{\substack{i=1,2,\ldots,I\\ r=1,2,\ldots,R}}\left\{\frac{UB_{ir}-LB_i}{LB_i}\times 100\%\right\}, \text{ where} \tag{5.6}$$

I = number of problem instances with the relevant combination of parameters;

R = number of repeated runs for problem instance i ($i = 1, 2, \ldots, I$);

UB_{ir} = heuristic solution found in rth run of problem instance i;

LB_i = lower bound of the problem instance i.

We adopt the following abbreviations for the remaining subsections:

1P	: 1 point crossover
FP	: F point crossover
SGA	: Standard Genetic Algorithm
MXGA	: MultiCrossover Genetic Algorithm
STS	: Standard Tabu Search
DLTS	: Dynamic Length Tabu Search
SSDM	: Standard Steepest Descent Method
RSDM	: Randomised Steepest Descent Method
Elite & Filter	: Elitism replacement and Filtration strategy

The specific values for the generic design variables in SGA and MXGA are summarised in Table 5.1. Initial computational experiments are performed to determine the size of the candidate list of temporary offspring. Five values of t ($t = 3, 5, 7, 9, 10$) are tested and results show that $t = 5$ gives the best result within a reasonable computation time.

Table 5.1: Implementation of generic design variables for SGA and MXGA

variable	value
chromosome length, L	N
population size, P_{pop}	100
crossover operator	1-point and F-point (F = no. of families)
crossover rate, p_c	0.75
multicrossover, t (MXGA only)	5 (=10 temporary offspring)
individual mutation rate, p_M	0.25
gene mutation rate, p_m	$1/N$
filtration rate, R	every 50 generations
selection mechanism	probabilistic binary tournament

5.6.2 Standard Steepest Descent Method vs. Randomised Steepest Descent Method

Table 5.2 presents results comparing the SSDM with our proposed RSDM. Five problem instances with 100 jobs in 4 families are generated and setup class A ([1,100]) is used in this experiment. For each problem instance, a total of 30 runs are performed to obtain an average value. A duration of 20000 iterations for each run is performed.

The first column gives the due date combination among D1–D5. Columns two and four refer to the ARD (equation (5.5)), while columns three and five refer to the MRD (equation (5.6)) of SSDM and RSDM, respectively. For each algorithm, the entries report the average values computed over the five problem instances (in this case, $I = 5$, $R = 30$). The final line of the Table 5.2 gives the overall average value over all five combination of due dates.

Table 5.2: Comparison of SSDM with RSDM (20000 iterations per run)

Due Date	SSDM		RSDM	
	ARD	MRD	ARD	MRD
D1	0.25	0.80	0.18	0.55
D2	1.84	3.93	1.71	2.84
D3	9.12	12.41	8.61	11.31
D4	19.60	27.78	19.25	25.88
D5	66.73	140.93	65.64	139.60
Average	19.51	37.17	**19.08**	**36.04**

We first observe that the performance of both algorithms in D5 are unimpressive, with relatively large deviations of the heuristic solution value from that of the lower bound. Among the due date combinations, D5 proved to be the most difficult to achieve a value close to the lower bound. The solution quality significantly improves if the range of the due date is small. A comparison of all corresponding due date ranges for both algorithms show that RSDM is slightly the better of the two. It is worth mentioning that the RSDM takes slightly longer computation time to reach a better local optimum. This is due to the acceptance rule applied in the RSDM which allows neutral moves during the execution. The SSDM terminates once no improving move is found. Thus, for a descent method, we subsequently concentrate on the RSDM.

5.6.3 Standard Tabu Search vs. Dynamic Length Tabu Search

Table 5.3 gives results for the different settings of the tabu list length of the TS approach. The problem instances generated from Section 5.6.2 are used in this experiment. As in Section 5.6.2, for each problem instance, 30 runs were performed with a duration of 20000 iterations per each run. The different settings of the tabu list length in Table 5.3 are as follow:

$x \leftrightarrow y$: $x, y \in \{10 \leftrightarrow 75, 25 \leftrightarrow 75\}$.
Starts with a tabu list length of x. Increase the length by 5 after 100 non-improving moves. Decrease the length by 5 once an improving move is found. Dynamically control the length of the tabu list within the range ($[x, y]$)throughout the run.

$y \leftrightarrow x$: $y, x \in \{75 \leftrightarrow 10, 75 \leftrightarrow 25\}$.
Starts with a tabu list length of y. Decrease the length by 5 after 100 non-improving moves. Increase the length by 5 once an improving move is found. Dynamically control the length of the tabu list within the range ($[y, x]$)throughout the run.

z : $z \in \{10, 25, 50, 75, 100\}$.
Fixed tabu list length at z throughout the run.

Table 5.3: Comparison of DLTS with STS (20000 iterations per run)

Due Date		DLTS				STS				
		10↔75	75↔10	25↔75	75↔25	10	25	50	75	100
ARD	D1	0.00	0.00	0.00	0.00	0.25	0.22	0.25	0.26	0.18
	D2	0.98	0.99	0.99	1.00	1.67	1.67	1.62	1.64	1.68
	D3	6.44	6.35	6.58	6.20	7.73	7.61	7.67	7.73	8.69
	D4	16.21	15.94	16.83	15.64	18.76	18.39	18.48	18.60	19.42
	D5	55.19	57.98	56.42	56.36	59.09	58.45	59.91	58.88	58.68
Average		**15.76**	16.25	16.16	15.84	17.50	17.27	17.39	17.42	17.71
MRD	D1	0.00	0.00	0.00	0.00	0.92	0.82	0.98	0.98	0.93
	D2	1.95	1.95	1.95	1.95	2.95	2.92	2.59	2.59	2.47
	D3	8.71	8.21	8.72	8.06	10.30	10.21	9.96	9.95	9.97
	D4	23.73	22.53	24.34	21.87	26.56	25.98	25.73	26.56	25.44
	D5	113.41	121.52	115.72	119.51	124.32	121.00	122.96	122.70	121.34
Average		**29.56**	30.84	30.15	30.28	33.01	32.19	32.44	32.56	32.03

As for the descent method, we observe that the solution quality is high when the due date range is small. Note that the dynamically controlled tabu list length versions of TS achieved the lower bound for due date D1 for all five problem instances in every run (i.e. 150 runs where $I = 5$, and $R = 30$). Comparing the results for the dynamic and the fixed length versions of tabu list of the algorithms, better solution quality is generated for the former version in every due date range with the '$10 \leftrightarrow 75$' version performing the best. This improved performance is explained by the observation that the short tabu list is needed at the beginning of the run to allow the search to fully exploit the neighbourhood. A longer tabu list

is needed at the later stage of the run to allow the search into 'interesting' regions of the solution space which would not otherwise be explored. Thus, the TS with '10 $\leftrightarrow$ 75' version is preferred.

5.6.4 Initial Investigation of MultiCrossover Genetic Algorithm

During the development of the proposed MXGA, we made some decisions on the design at different stages. We gradually construct the proposed MXGA from the SGA. For the initial investigation in this subsection, five problem instances with five (D1-D5) combinations of due dates are generated and setup class A ([1,100]) is used in this experiment. For each combination of the problem instance and due date range, a total of 30 runs are performed to obtain the average value. A fixed time limit of 15 CPU seconds per run is imposed.

Table 5.4 shows results for the different replacement strategies that we employed in the SGA at the early stages of development of the MXGA. We compare the well known steady-state replacement strategy (see Section 4.8) with our proposed elitism replacement and filtration strategy described in Section 4.9.6 using standard 1-point and F-point crossover operators. For each algorithm, the entries report the average values (ARD and MRD) computed over the five problem instances with five combinations of due dates (i.e. 750 runs). The final line of Table 5.4 gives the overall average value.

Table 5.4: Comparison of Steady-State Replacement with Elitism Replacement and Filtration Strategies in SGA (15 CPU seconds per run)

N	F	Steady State + 1P		Steady State + FP		Elite & Filter + 1P		Elite & Filter + FP	
		ARD	MRD	ARD	MRD	ARD	MRD	ARD	MRD
50	4	25.87	79.53	24.90	78.65	25.27	79.05	24.89	77.23
	8	18.01	48.21	17.56	46.88	16.95	46.10	16.90	45.15
100	4	21.93	53.68	21.38	51.16	20.80	51.29	20.32	48.41
	8	23.34	54.65	22.87	50.27	22.41	51.12	22.04	48.82
Average		22.29	59.02	21.68	56.74	21.36	56.89	**21.04**	**54.90**

The results achieved by F-point crossover in both replacement strategies clearly outperform the 1-point crossover, although fewer generations are created within the time limit. Table 5.4 also shows that the elitism replacement and filtration strategy yield better results in both crossover operators compared to their counterparts in the steady state replacement strategy. A comparison of the corresponding replacement strategies and crossover points for the algorithms shows that the elitism replacement and filtration strategy with F-point crossover is the best of the four. The fact that the elitism replacement and filtration strategy with F-point crossover outperforms the other algorithms for every single test case shows that they can search the solution space in a more efficient manner. Thus, for the replacement strategy, we subsequently concentrate on the elitism replacement and filtration strategy in the MXGA.

Table 5.5 shows results of the multicrossover operator compared to the standard crossover operator using 1-point and F-point crossover strategies. The standard crossover operator can be regarded as a special case of the multicrossover operator where the steps described in Section 5.4.2 are used to produce exactly two offspring (i.e. $t = 1$). This operator is used to investigate whether it is advantageous to produce multiple offspring during multicrossover. Five problem instances with five combinations of due dates are generated and setup class A ([1,100]) is used in this experiment.

Table 5.5: Comparison Between Crossover Operators (15 CPU seconds per run)

N	F	**Elite & Filter + 1P**				**Elite & Filter + FP**			
		Standard		MultiCrossover		Standard		MultiCrossover	
		ARD	MRD	ARD	MRD	ARD	MRD	ARD	MRD
50	4	25.15	78.77	24.22	75.99	24.76	77.13	24.01	75.51
	8	16.98	45.83	16.35	44.01	16.85	45.20	16.12	43.89
100	4	20.75	51.05	17.79	43.65	20.05	49.67	17.47	43.15
	8	22.45	50.67	19.35	43.89	21.91	48.23	19.01	43.51
Average		21.33	56.58	19.43	51.89	20.90	55.06	**19.15**	**51.52**

Our first observation from Table 5.5 is that the standard crossover operators give poorer results than the multicrossover operator with both 1-point and F-point crossover strategies. As suggested, the F-point crossover strategy does perform

better compared to 1-point crossover strategy in both different crossover operators. It is clear that better solution quality is obtained under the multicrossover operator in both cases although fewer generations are created within the time limit. This superiority is more pronounced when the number of jobs is large.

We conclude that the 1-point and F-point multicrossover operators are the preferred versions of crossover operators, with the latter performing marginally better. Consequently, the F-point multicrossover operator is used in the proposed MXGA.

Table 5.6 shows the computational results of the proposed MXGA using the swap operator explained in Section 5.4.3. The purpose of this experiment is to investigate the effect on the solution quality when using the swap operator in our proposed MXGA.

Table 5.6: Results of Swap (15 CPU seconds per run)

N	F	**MXGA (1P)**				**MXGA (FP)**			
		with Swap		without Swap		with Swap		without Swap	
		ARD	MRD	ARD	MRD	ARD	MRD	ARD	MRD
50	4	23.89	74.51	24.65	76.05	23.23	73.21	24.09	75.34
	8	16.05	42.19	16.98	44.15	15.76	41.34	16.31	43.67
100	4	17.11	41.76	18.01	43.44	16.45	40.97	17.34	43.21
	8	19.09	42.29	19.74	43.67	18.76	41.19	19.25	43.19
Average		19.04	50.19	19.85	51.83	**18.55**	**49.18**	19.25	51.35

It is clear from Table 5.6 that the swap operator yields better results in the MXGA compared to the algorithms without the swap operator. This matches our intuition that the swap operator manages to create more diversity in the population and lead the search into the more 'interesting' regions to explore better local optima. Analysing the results obtained by the algorithms, we can conclude that the presence of the swap operator in the proposed MXGA improves the solution quality with the F-point crossover version performing the best.

Table 5.7 reports the results of the proposed MXGA using the mutation operator as described in Section 5.4.4. As for the swap operator, we observe that the presence of the mutation operator in the MXGA improves the solution qual-

ity. This improved performance is explained by the observation that the mutation operator is able to help the MXGA explore unknown regions.

The results of the computational experiments in this subsection provide guidelines for the design of the proposed MXGA. The elitism replacement and filtration strategy clearly outperform the steady state replacement strategy. The high selection pressure cause by the elitism scheme makes the population fall into premature convergence. But it can be overcome by introducing the filtration strategy into MXGA. By removing and replacing the identical individuals with randomly generated individuals in every R generations, this will help the population explore more unknown regions in the search space. The exploration technique used in the standard mutation operator can be further enhanced by introducing the swap operator into the MXGA, while the exploitation technique can be improved by using the multicrossover operator as described in Section 5.4.2. Consequently, this final version of the proposed MXGA is used in our comparative tests in the subsection.

Table 5.7: Results of Mutation (15 CPU seconds per run)

N	F	MXGA (1P)				MXGA (FP)			
		with Mutation		without Mutation		with Mutation		without Mutation	
		ARD	MRD	ARD	MRD	ARD	MRD	ARD	MRD
50	4	22.76	72.31	23.92	74.62	21.24	70.29	23.31	73.41
	8	14.95	40.77	15.99	42.01	13.98	38.76	15.87	41.65
100	4	16.21	39.64	17.32	41.92	15.64	38.21	16.41	40.99
	8	18.34	40.78	18.89	41.95	17.45	39.44	18.65	41.01
Average		18.07	48.38	19.03	50.13	**17.08**	**46.68**	18.56	49.27

5.6.5 A Comparison of different Local Search Algorithms

In this subsection, we present results of tests that compare the DLTS, RSDM, and MXGA with each other. We also compare our 'good' implementations with standard TS and GA. The differences between the MXGA and SGA are with regards to the use of the crossover operator, reproduction procedure and the replacement scheme. The SGA applies the standard F-point crossover operator to produce two offspring from two selected parents. In the case of SGA, the steps explained

in Section 5.4.2 are used only once (i.e. $t = 1$) to generate exactly two offspring. The SGA uses the reproduction procedure instead of a swap operator when the crossover does not apply to the selected parents. The replacement strategy employed in the SGA is the steady-state replacement strategy.

For this final experiment, we use the problem instances described in Section 5.6.1. For each combination of problem instance, 30 runs were performed. In order to have a fair comparison between different algorithms in this experiment, we employed a duration of 15 CPU seconds per run. Results are listed in Table 5.8. For each algorithm, the entries report the average values (ARD and MRD) computed over the five problem instances with five combinations of due dates (i.e. 750 runs). The final line of Table 5.8 gives the overall average value.

Table 5.8: Comparative Computational Results (15 CPU seconds per run)

Setup Class	N	F	SGA		MXGA		STS		DLTS		RSDM	
			ARD	MRD	ARD	MRD	ARD	MRD	ARD	MRD	ARD	MRD
A	50	4	18.58	79.63	13.54	71.56	25.34	133.93	13.76	72.31	17.03	88.60
		8	17.85	69.24	13.76	60.06	23.10	77.21	14.82	62.82	18.71	69.20
		12	12.38	41.35	9.31	36.71	15.33	50.48	10.93	39.37	13.48	44.01
	100	4	21.12	150.38	15.73	119.68	17.35	126.67	15.74	113.06	18.96	135.78
		8	26.26	138.34	18.75	106.15	23.65	129.21	20.80	114.30	24.24	125.23
		12	34.91	124.86	19.28	82.83	23.02	99.61	20.50	87.28	23.97	100.47
	Average		21.85	100.64	**15.06**	**79.50**	21.30	102.86	16.09	81.52	19.40	93.88
B	50	4	6.72	54.79	4.65	42.53	9.93	81.00	4.81	42.87	6.03	51.34
		8	10.30	72.03	7.74	59.31	12.84	82.93	8.52	64.07	9.46	67.68
		12	8.35	50.78	5.51	39.51	10.91	62.37	7.72	50.61	8.40	53.12
	100	4	9.78	90.00	7.11	69.12	8.07	87.09	7.47	76.14	8.34	81.78
		8	13.69	123.20	9.64	91.07	11.85	115.56	11.08	97.63	11.85	102.99
		12	17.44	113.88	9.52	72.46	11.37	86.34	10.89	79.57	12.01	88.80
	Average		11.05	84.12	**7.37**	**62.34**	10.83	85.89	8.42	68.48	9.35	74.29
C	50	4	28.09	74.80	17.05	57.75	33.17	86.38	17.29	59.33	22.11	68.93
		8	22.56	51.45	15.42	41.56	27.04	58.37	16.00	44.72	20.59	60.11
		12	13.40	30.32	9.27	25.26	14.91	35.06	10.07	26.31	14.18	34.29
	100	4	44.18	165.15	25.60	99.87	25.85	103.67	25.72	103.06	30.57	131.31
		8	39.39	112.15	25.60	83.38	29.92	98.94	26.82	89.00	31.76	102.47
		12	55.27	108.72	22.43	60.90	26.26	70.58	23.08	62.56	27.76	78.24
	Average		33.82	90.43	**19.23**	**61.45**	26.19	75.50	19.83	64.17	24.50	79.23
AVERAGE			22.24	91.73	**13.89**	**67.76**	19.44	88.08	14.78	71.39	17.75	82.47

We first observed that the MXGA performs significantly better than the SGA. This shows that MXGA is able to produce better solution quality compared to SGA. There is clear evidence from Table 5.8 that, on average, the MXGA is the best algorithm followed by the DLTS, RSDM, STS and finally the SGA. Also, our proposed DLTS and RSDM outperformed the STS and SGA.

We have found that computational difficulty as measured by relative deviation from the lower bound increases with problem size. With other things being equal, when we increase the number of jobs, then both ARD and MRD will increase. Note that for all the local search algorithms, fewer generations (or iterations) are executed within the time limit as the number of jobs or families become larger.

The algorithms find problems of setup class C (large setup time) to be the most challenging. This is due to the large setup time, as it contributes substantially to the maximum lateness of an optimal schedule. Jobs tend to form a larger batch size, with more jobs in a batch, to reduce the need of setup time between batches from different families. As a result, more jobs will miss their assigned due dates. From the manufacturer's point of view, the only solution to the large setup time is to form large batch sizes to allow many jobs to run on a similar setup. When the setup time is small (i.e. setup class B), more batches are formed which means fewer jobs are to be processed per batch, and hence more jobs will meet their respective due dates. Results from Table 5.8 suggest that the problem instances with setup class B is relatively easier to solve compared to other setup classes.

5.7 Conclusions and Remarks

In this chapter, a single machine family scheduling problem with family setup times to minimise the maximum lateness is studied. We have also described the EDD properties for jobs within a family and for batches.

We have designed a genetic algorithm which uses a multicrossover operator in an effort to achieve better solutions quality. Various techniques have also been introduced into the proposed algorithm to further enhance the solutions quality. Extensive computational experiments show that the proposed multicrossover genetic algorithm (MXGA) achieves better results compared to a standard genetic algorithm, both standard and dynamic length tabu search and a randomised steepest descent method. The development of MXGA for other optimality criterion such as minimising the total (weighted) tardiness/earliness is worthy of future research.

Chapter 6

Non-Oriented Two-Dimensional Rectangular Single Bin Size Bin Packing Problem

6.1 Introduction

In this chapter, we concentrate on a non-oriented two-dimensional rectangular single bin size bin packing problem (2DRSBSBPP) (based on Wäscher *et al.*'s Typology). According to Lodi *et al.* [194], the problem can be defined as follows: "Given are n rectangles, each characterised by a height h_j, and a width w_j, for $j = 1, 2, 3, \ldots, n$ and an unlimited number of identical rectangular bins, each having height H, and width W. The objective of the 2DRSBSBPP is to pack each rectangle into a bin so that no two rectangles overlap and the number of required bins is minimised."

This problem is classified as a class of NP-hard problem by Garey and Johnson [108].

The general assumptions of the 2DRSBSBPP in this study are:

1. all input data are positive integers;
2. $w_j \leq W, h_j \leq H$ $(j = 1, 2, \ldots, n)$;
3. a set of rectangular items, which may contain identical rectangles;
4. a set of identical objects (bins);
5. rectangle j may be rotated by 90° where $\max\{w_j, h_j\} \leq \min\{W, H\}$;
6. rectangles are packed in *non-guillotine cuts pattern* in the bin;
7. rectangles are packed in an *orthogonal packing pattern*: the edges of the rectangles are parallel to those of the bins.

One of the objectives in this study is to develop a new heuristic placement routine that can be used with our proposed MultiCrossover Genetic Algorithm (MXGA). The proposed placement routine was inspired by the best-fit heuristic placement routine designed by Burke *et al.* [40] and Whitwell [280] for solving the two dimensional rectangular stock cutting problem.

In this study, we propose a MXGA that utilises the multicrossover operator to solve the 2DRSBSBPP. We use a standard 1-point or 2-point crossover to produce two temporary offspring. Detailed descriptions of the proposed multicrossover operator are discussed in Section 6.3.4. The architecture of the MXGA used in this chapter is based on the framework discussed in Section 4.9.

This study will look at a new variant of the 2DRSBSBPP by including a due date for each rectangle and a fixed processing time for each bin. As a result, the problem becomes a bicriteria optimisation problem where the objective function is to find an optimal solution for minimising the maximum lateness of the rectangles and minimising the number of bins used. This extension has practical applications in the wood and metal industries. This problem can also be treated as a batching machine scheduling problem where a machine can process several jobs simultaneously. Section 6.4 will address this problem in more detail.

The motivation of this extension came from the dilemma faced in the industrial manufacturing applications which involved the trade-off between the customers' satisfaction (meeting customers' due date on the order placed) and the manufacturer's efficiency (minimising the wastage of material used).

In the next section, we present a new heuristic placement routine for 2DRSBSBPP in more detail. The developments of the MXGA for solving the 2DRSBSBPP are the focus of Section 6.3. Some of the main components of the MXGA are discussed in detail. A new variant of the 2DRSBSBPP which involves rectangle due date and fixed bin processing time is addressed in Section 6.4 and a new lower bound of the maximum lateness for the 2DRSBSBPP with due dates is then proposed in Section 6.5 in order to measure the performance of the heuristic solution found when the exact solution is unknown. Section 6.6 provides an insight into the local search algorithms we designed specifically for comparison purposes with the proposed MXGA. To end this chapter, extensive computational experiments are conducted for the proposed placement routine, the classic 2DRSBSBPP and the new 2DRSBSBPP with due dates in Section 6.7. Some concluding remarks are given in Section 6.8.

6.2 Lowest Gap Fill

Inspired by the Bottom-Left Fill (BLF) routine, Burke *et al.* [40] and Whitwell [280] propose a *best-fit* heuristic placement routine for the two-dimensional stock cutting problem that is effective in filling the available gaps in the partial layout by dynamically selecting the best rectangle for placement during the packing stage. Unlike the Bottom-Left (BL) and BLF routines that place the rectangles based on the sequence of rectangles supplied, their proposed routine would make informed decisions about which rectangle should be packed next and where it should be placed. Their extensive computational results show that the proposed heuristic is able to outperform the currently published and established heuristic and meta-heuristic methods to produce solutions that are very close to optimal with very

small computational time. Due to the excellent results achieved by the best-fit heuristic in stock cutting problem, we are intrigued to use their ideas to design a new placement routine to suit our problem. In the following, we briefly describe the *best-fit* heuristic placement routine for the cutting stock problem. Detailed descriptions of the routine can be found in Burke *et al.* [40] and Whitwell [280].

According to Burke *et al.* [40] and Whitwell [280], the *best-fit* heuristic is a greedy algorithm that attempts to produce a high quality packing layout by examining the available gap within the stock sheet and then placing the rectangle that best fits the lowest gap available. Every time a rectangle is placed, the lowest available gap will change with respect to its location and size. They define a *niche placement policy* for the case when the best fit rectangle does not completely fill the gap. This policy describes how a rectangle should be placed within the gap.

The *best-fit* placement routine is implemented in three stages: *preprocessing* stage, *packing* stage, and *postprocessing* stage. In the preprocessing stage, the rectangles are initially arranged following a horizontal orientation and sorted in non-increasing order of their width, breaking ties by non-increasing height. The stock sheet is represented as a linear array in which the number of elements is equal to the width of the stock sheet (x-coordinate). Each element of the array holds the total height of the packing at that x-coordinate of the stock sheet. Therefore, the coordinate of the lowest gap can be determined by locating the smallest value of the array and the width of the gap is the length of the consecutive array of equal value.

During the packing stage, a list of rectangles is examined and the best fitting rectangle returned. This rectangle is then placed within the gap depending on the current *niche placement policy.* The rectangle is assigned coordinates and removed from the rectangle list. The relevant stock sheet array elements are incremented by the rectangle height. The process continue until every rectangle is packed in the stock sheet.

In the postprocessing stage, the quality of solution is improved by repacking any rectangle that creates '*towers*' in the layout. Towers are created when long thin (i.e. height > width) rectangles are protruding from the top of the packing layout. The tower is removed from the packing and then rotated 90° before being repacked in the new orientation on top of the packing layout. If the solution quality is improved, the prosess is repeated on a new 'tower'. This process continues until there is no improvement in the solution quality.

Based on the ideas presented above, we propose a new heuristic placement routine for the 2DRSBSBPP, called the Lowest Gap Fill (LGF). This placement routine consists of two stages: *preprocessing* stage and *packing* stage. As in the *best-fit* placement routine, before the start of the LGF placement routine, the rectangles are initially arranged following a horizontal orientation (where its longest edge is parallel to the bottom of the bin) and sorted in a non-increasing order of their width (breaking ties by non-increasing height).

We employ a best-fit type strategy by examining the lowest available gap in the current bin and then placing the rectangle that best fits the gap available. This placement routine not only keeps track of the free position in the layout, but also of the dimensions of the available gap at the respective position. When no remaining rectangle can fit into any of the available gaps in the current bin, the bin is closed and a new empty bin is initialised to replace the closed bin as the current bin. Any unfilled space in the closed bin will be regarded as wastage. Also note that the proposed routine only concentrates on one bin during the packing process. This differs from the BL and BLF routines, where a list of bins that has been created needs to be maintained. The routine continues until all the rectangles in the list have been packed into a minimum number of bins. The details of the implementation stages will be discussed in the next subsection.

Some of the algorithms such as BL and BLF may require a costly *overlap* evaluation test. This evaluation test performs an overlap test between the rectangle and each of the rectangles that have already been packed in the current bin. It is obvious that the more rectangles that have been packed, the more overlap tests

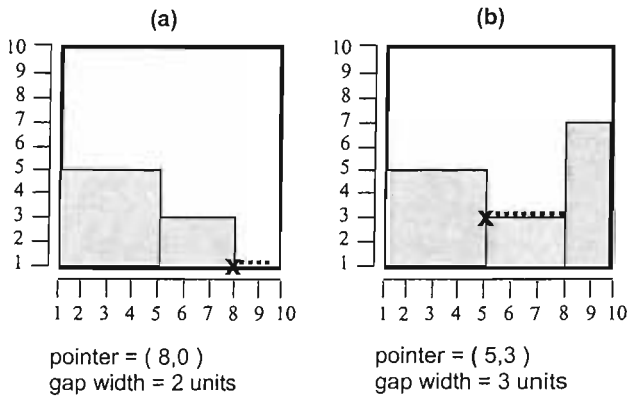
have to be performed. The process becoming increasingly slower as each rectangle is placed. With the best-fit approach and the implementation strategy of the LGF (presented in next subsection), this costly evaluation test is not needed as the rectangle packed will not overlap with other rectangles already packed in the current bin.

6.2.1 Implementation

Preprocessing Stage

Instead of maintaining a linear array as in the *best-fit* placement routine proposed by Whitwell [280], we use a pointer (x, y) to indicate the position of the lowest available gap in the bin during the packing stage. The pointer is determined by locating the lowest *free* position in a bin, left justified. A *free* position is where a rectangle can be placed without overlapping with other rectangles that are already packed. The width of the gap can be found by measuring the length on the x-coordinate starting from the pointer until it touches either the right edge of the bin (Figure 6.1 (a)) or the left edge of a tall rectangle (Figure 6.1 (b)). Note that the difference between the height of the bin and the y-coordinate of the pointer gives the height of the available gap. As a result, we obtain both dimensions of the available gap. This will ensure that the rectangle to be placed next will not overlap with the bin or any other rectangles that are already packed in the bin.

Figure 6.1: Examples of pointer and gap



Due to the best-fit strategy that we employ in our proposed routine, we must examine all of the rectangles to be sure that the selected rectangle to be placed next is the largest available rectangle that can be fit in the gap at each placement. However, we can reduce the number of rectangles we need to examine by sorting the list of rectangles once before packing commences.

This can be done by first rotating any rectangle for which the height is greater than the width so that we get a list of rectangles with their longest edge parallel to the bottom of the bin. For example, by denoting each rectangle by a (width, height) pair, the rectangle list of
{(2,2), (5,9), (1,2), (7,2), (3,5)} becomes {(2,2), (9,5), (2,1), (7,2), (5,3)}.

Then, the list of rectangles is sorted in non-increasing order of width (breaking ties by non-increasing height). From the previous example,
{(2,2), (9,5), (2,1), (7,2), (5,3)} becomes {(9,5), (7,2), (5,3), (2,2), (2,1)}. This preprocessing stage required $O(n \log n)$ time.

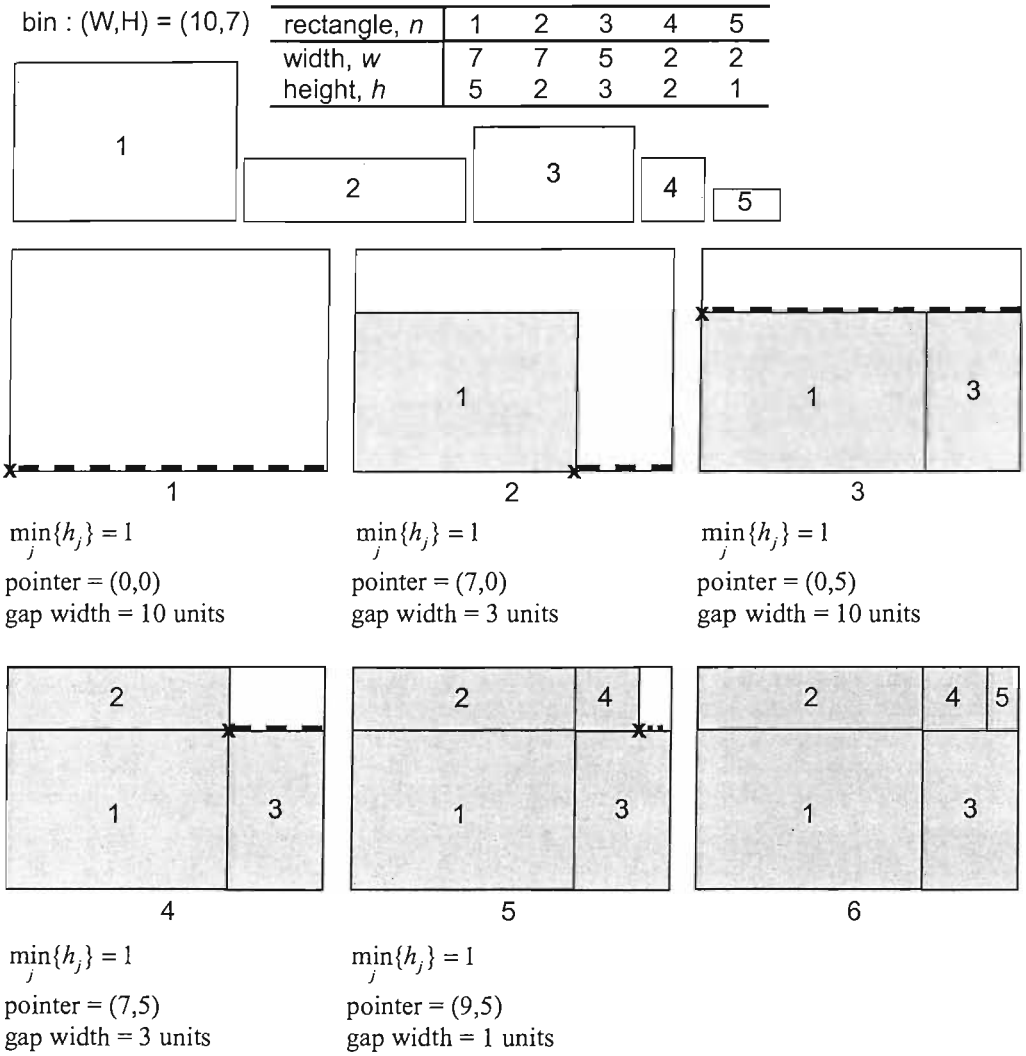
The new list of rectangles can now be examined for the best fitting rectangle without the need to search the entire list during the packing stage. For instance, suppose we found a gap of 6 units in the bin. The first rectangle (9,5) in the list is examined. Note that it could fill 5 units if rotated. It does not fit the gap exactly, so we continue the search. The second rectangle (7,2) is examined. It can fill 2 units if rotated. We must continue because there may be a rectangle with a width of 6 units that can fit the gap exactly. The third rectangle (5,3) is examined. It can fill the same number of units as the first rectangle. Since we prefer to pack the larger rectangle first, the first rectangle would be returned as the best fitting rectangle. We know we can terminate our search, as all remaining rectangles have dimensions of less than or equal to 5. Also note that we terminate the search as soon as a rectangle that fits exactly is found. This will reduce the search time of the process. In general, it is better to place a rectangle with larger dimensions earlier in the packing than towards the end of the packing, when smaller rectangles are easier to fit into any gaps within the bin.

Packing Stage

During the packing stage, the smallest dimension of height for the available rectangles in the rectangle list (i.e. $\min_{i=1,2,\ldots,j}\{h_i\}$, where j = number of the remaining rectangles in the rectangle list) is stored. This value is used to compare with the width of the gap in the current bin. The value of $\min_{j}\{h_j\}$ will only be updated if the rectangle with the smallest dimension of height has been packed into the current bin. The pointer and the corresponding gap width will also be maintained during the packing stage.

At first, an empty bin is initialised as the current bin, where the pointer is at the bottom-left corner ($x = 0, y = 0$) of the current bin, with a gap width of the entire bin width W. The first rectangle in the rectangle list is then placed at the bottom left of the current bin. The placed rectangle is then removed from the rectangle list. The pointer and gap width are updated according to the dimensions of the packed rectangle. Next, the rectangle list is examined again and the best fitting rectangle returned. The selected rectangle will be placed in the current bin to fill the gap, with the bottom-left corner of the selected rectangle placed at the position of the corresponding pointer. This will ensure that the current bin is systematically filled from the bottom-left corner of the bin. The rectangle is removed from the rectangle list and the value of the pointer and gap width are updated. If the best fitting rectangle does not completely fill the gap, then there is no need to locate or update the new pointer for the next rectangle. Only the gap width needs to be updated, where it is a portion of the recent gap. Figure 6.2 shows the stages of packing the rectangles into an empty bin using the LGF placement routine. Note that rectangle 3 and 5 have been rotated 90°.

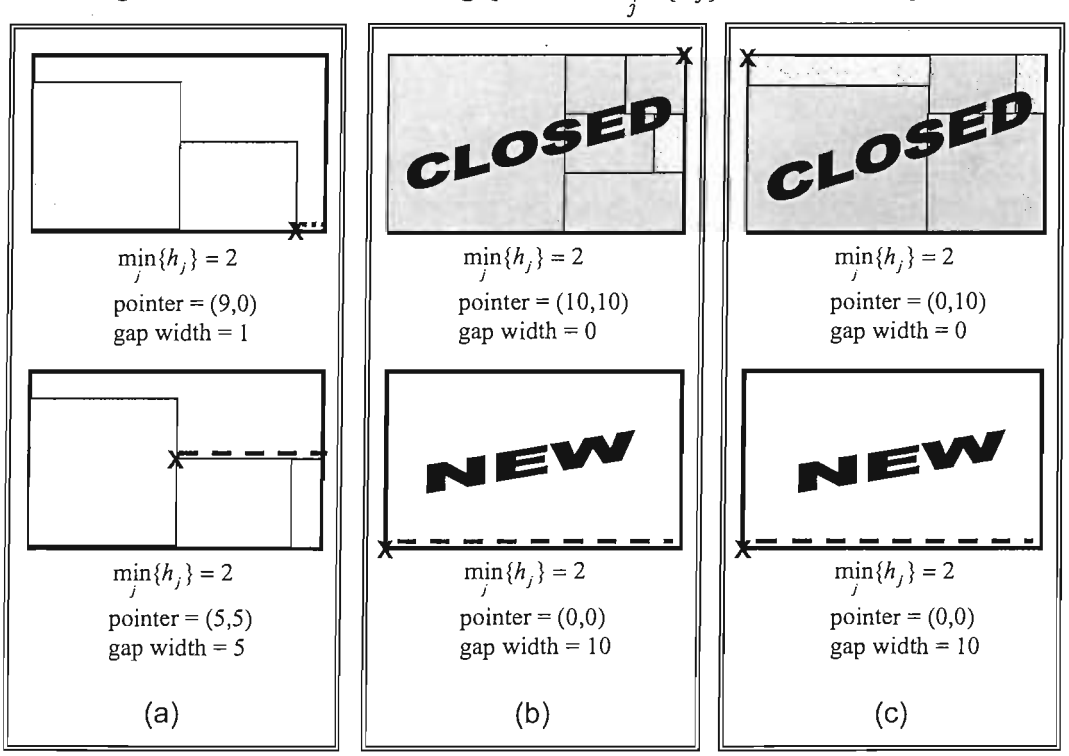
Figure 6.2: Packing the rectangles into a bin (LGF routine)



If the gap width at the corresponding pointer is smaller than the current value of $\min_j \{h_j\}$, we can regard the relevant space as wastage. The reason behind this is that if the gap cannot be filled now, it will not be able to be filled later in the process. So, the pointer will be raised up to the next lowest point where the corresponding gap width is at least as big as the value of $\min_j \{h_j\}$. For example (Figure 6.3 (a)), assume that there is a gap width of 1 unit at the corresponding pointer in the current bin, but the $\min_j \{h_j\} = 2$, which means that none of the remaining rectangles in the list can fit into the gap. The pointer is then raised to the next lowest point $(5, 3)$ where the corresponding gap width is at least 2 (in this

case is 5). When the current bin is full (Figure 6.3 (b)) or the pointer has been raised to the top of the current bin (i.e. $y = H$) (Figure 6.3 (c)), the bin is closed and removed. A new empty bin is initialised as the current bin and the process continues until all of the rectangles in the list are packed. Only one bin is open at a time. In order to reduce the time spent searching through all the bins that have been created during the process one by one, it is advantageous to close and remove the bin once none of the remaining gaps can be filled. This will reduce the processing time. This packing stage requires $O(n^2)$ time.

Figure 6.3: Scenarios where gap size $< \min_j \{h_j\}$, bin full and $y = H$



6.3 MultiCrossover Genetic Algorithm

In this section, we propose a MXGA for solving the 2DRSBSBPP. The proposed MXGA utilises a multicrossover operator in an effort to achieve enhanced solutions for a better quality of packing pattern in the bins. A common feature found in most genetic algorithms (GAs) developed for 2DRSBSBPP is their two stage approach, where a GA is used to explore and find good solutions in the search space. Then, a placement routine is needed to decode the solutions generated by the GA into the corresponding packing pattern for the evaluation of their layout quality.

In our proposed MXGA, we employ LGF as our placement routine. In the remaining section, we will discuss some of the main components in our proposed MXGA. Note that the general architecture of the MXGA for 2DRSBSBPP is the same framework as we described in Section 4.9.

6.3.1 Search Space

It would be beneficial if the search space is sufficiently large to allow the search process to explore a large range of layout patterns before it started converging. But, a very large search space may contain a high number of layout configurations which do not contribute to the search process due to their low quality. Therefore, it is advantageous to limit the search space. The search space for this implementation is limited with respect to the feasibility of the solutions. The feasibility of all solutions in the search space is guaranteed by the decoding procedure which only produce non-overlapping solutions.

6.3.2 Representation

Most researchers have used items permutation to represent an individual. Each rectangle only appeared once in the individual and is not repeated. The rectangles are then packed into bins according to the sequence in which they appeared. Since

we are using the LGF placement routine in the decoding stage (presented in next subsection), the sequence of the rectangles becomes irrelevant.

In our proposed MXGA, the complete set of rectangles n, forms the length of the chromosome (individual). The genes are represented by a uniform random permutation of the integer numbers of bins in the interval $[1,LB_o]$, where LB_o is the overall lower bound described in Section 3.4.3, equation (3.47). Thus, a solution to the packing problem in this case consists of a sequence of positive integer numbers indicating the bin number, in which the rectangles are placed into the bin. The exact location in the layout is then determined by a placement routine.

Figure 6.4 shows an example of the gene representation for an individual with 8 (n) rectangles attempt to pack into 3 (LB_o) bins. The individual will attempt to pack rectangles {3,5,7} into bin 1, rectangles {1,4} into bin 2, and rectangles {2,6,8} into bin 3 if it fits during the decoding stage using the LGF placement routine. Any rectangle that cannot be feasibly packed into the assigned bin are dealt with the strategies suggested in the next subsection. Also note that every rectangle only appears once in the individual.

Figure 6.4: An example of an individual (chromosome)

item's no.	1	2	3	4	5	6	7	8
bin's no.	2	3	1	2	1	3	1	3

6.3.3 Decoding

The LGF heuristic placement routine is used to decode the *genotype* (gene representation) of an individual into *phenotype* (packing layout). The heuristic placement routine is designed such that the decoding only results in valid packing layouts. Before the placement routine takes place, the individual needs to be decoded into a sequence of rectangles which are grouped by the bin numbers that are associated with each rectangle. This is achieved by the following two step procedure:

S1 : Sort the rectangles in a increasing order of their associated bin number.

S2 : Groups the rectangles with the same bin number.

Also note that, based on the requirement of the placement routine, we need to pre-order the rectangles within the group before the routine commences. During the process of packing the rectangles into the bin within the group, any rectangle that cannot be feasibly packed will be regarded as an unassigned rectangle and kept in a list. After placement routine has been applied to all groups of rectangles, a strategy is needed to pack the rectangle(s) in the unassigned rectangle list into the bins already used or into new bins, so that a complete set of rectangles are packed without overlapping. After the strategy has been applied, the genotype of the individual is updated with the new packing layout. We suggest the following three strategies to deal with the problem. Note that the rectangle(s) in the unassigned rectangle list are sorted in order of non-increasing size before the strategy commences.

***pack_extra*:**
Pack all the rectangle(s) in the unassigned rectangle list into one or more new bins without considering the bins already in use. This strategy is more likely to yield poor results, since more bins will be required than previously assigned. However, its computation time is very short in comparison to other strategies and therefore may be preferred.

***pack_above*:**
Pack the unassigned rectangle(s) above the other items already packed in the bins, without permitting overflowing of the bin or overlapping of the rectangles, until all the rectangle(s) in the list are packed. Initialise a new bin if none of the bins can accommodate the unassigned rectangle(s). This strategy might produce a better result compared to the first strategy. However, the resulting packing layout tends to produce waste in the bin by not taking into consideration filling the empty space in between the rectangles or between the rectangles and the bin.

***repack*:**

Pack any unassigned rectangles from the list into the bins already used. This is a more powerful but computationally more expensive strategy. The idea is to first unpack the rectangles already packed in the selected bin and repack it again after adding a rectangle from the unassigned rectangle list into the group. This method may result in a better quality in the packing layout of the bin compared to the former strategies. The procedure is as follows:

S1 : Sort the bins in non-decreasing order of their bin utilisation (equation (6.1)).

S2 : Start from the first bin in the list (i.e. lowest bin utilisation), unpack the rectangles from the selected bin by emptying the contents of the bin.

S3 : Repack the selected bin using the LGF with the rectangles previously assigned to the bin plus the first rectangle (i.e. largest) from the unassigned rectangle list. Any left over or unfit rectangle(s) will became unassigned at this stage.

S4 : Calculate the bin utilisation (equation (6.1)) for the selected bin.

- If the value of the bin utilisation has increased, the selected bin is updated with the new packing layout, bin utilisation, and the list of rectangle in the bin. The unassigned rectangle list will also be updated (reorder) with any left over rectangle(s) from **S3**. If the unassigned rectangle list is empty, **STOP**; else, go to **S1**.
- Else (i.e. decreased or equal value), the selected bin is not updated (i.e. the rectangles previously packed in the selected bin and the status of the unassigned rectangle remain unchanged). If the selected bin is the last bin in the list, go to **S5**; else, go to **S3** and pack the same unassigned rectangle into the next bin in the list.

S5 : Pack any unassigned rectangle(s) in the list into one or more new bins until the complete set of rectangles is packed without overlapping.

Step **S1** is inspired by the ideas of bin utilisation, where low bin utilisation means the rectangles in that particular bin are packed loosely. It could also mean

less rectangles were packed in the bin. This also means that there are more chances for the unassigned rectangles to be packed in the bin with some minor modification to the packing layout. Note that by the end of the process, we might have increased the number of bins used from the ones originally assigned in the genotype. But, we also might have increased the quality of the individual bin utilisation.

6.3.4 MultiCrossover

Most researchers have used specific crossover operator such as cycle crossover (CX), partially matched crossover (PMX)([151, 154]), order based crossover (OBX), or order crossover (OX)([158, 192, 255]) in their proposed GA when dealing with the 2DRSBSBPP. This is due to the fact that they use items permutation in the encoding stage, where the individual is represented by a sequence of permutations of the rectangles. In order to ensure the rectangles only appeared once and are not repeated in the offspring, they have to use the specific crossover to check the validity of the offspring before the decoding commences. Unlike others, our proposed multicrossover operator uses standard 1-point or 2-point crossover to produce two temporary offspring. The idea behind this is that our individual is represented by a sequence of the bin number with the length of the complete set of rectangles. Every gene represents a bin number in which the rectangle will be packed if it fits. Thus, the constraint for the rectangles to appear only once in the solution is already fulfilled before the crossover commences. The process for a 2-point crossover is as follows:

S 1: Select randomly 2 crossover points in both P1 and P2 (Parent 1 & 2).

S 2: Form 3 pairs of sub-genes (head, body, tail) which are separated by the crossover points in both P1 and P2.

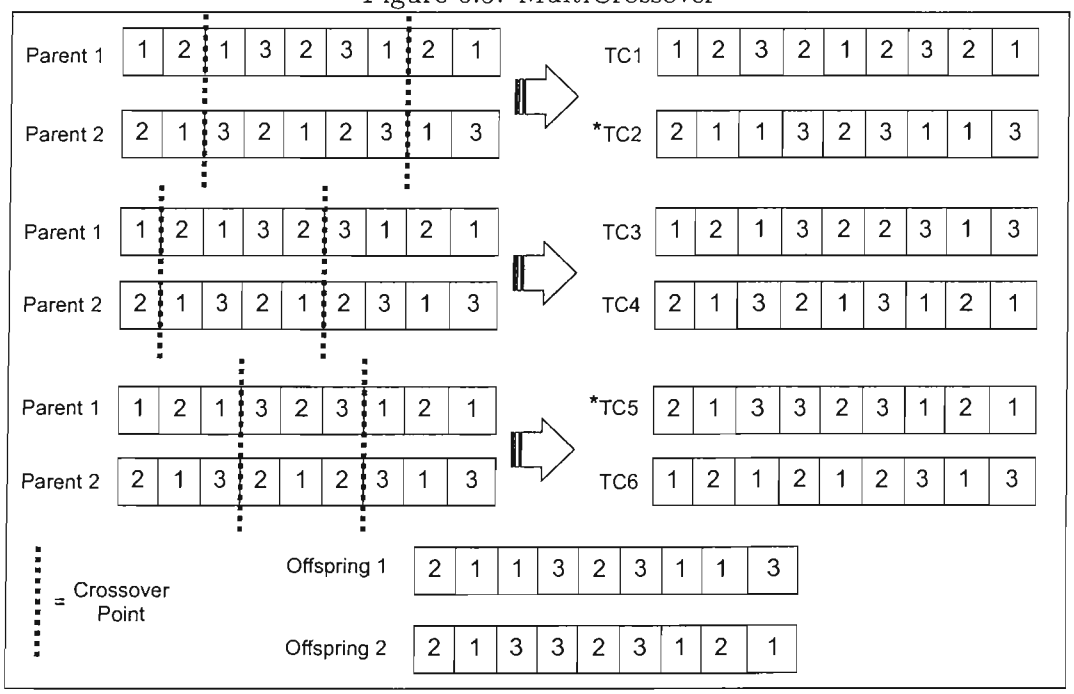
S 3: A randomly generated number will determine the allocation of each pair (e.g. head) in TC1 and TC2 (Temporary Offspring 1 & 2).

S 4: Repeat **S 3** for the other two pairs (body, tail) until 2 complete TC1 and TC2 are formed.

The steps above will be repeated t times to produce $2t$ temporary offspring. The best and a selected temporary offspring using probabilistic binary tournament selection mechanism are then chosen to be the offspring for the current generation. The steps can be easily modified to suit the 1-point crossover strategy. Figure 6.5 shows an example of the multicrossover process with $t = 3$. Recall that the individual is represented by a sequence of random permutations of the bin numbers, with length equal to the complete set of rectangles.

As a result of the computational complexity of the decoding strategies described in Section 6.3.3, the evaluation of the temporary offspring is conducted using the *pack_extra* strategy . Although the *pack_above* and *repack* strategies may generate a better solution quality, they require more computational effort in comparison to the *pack_extra* strategy.

Figure 6.5: MultiCrossover



6.3.5 Swap

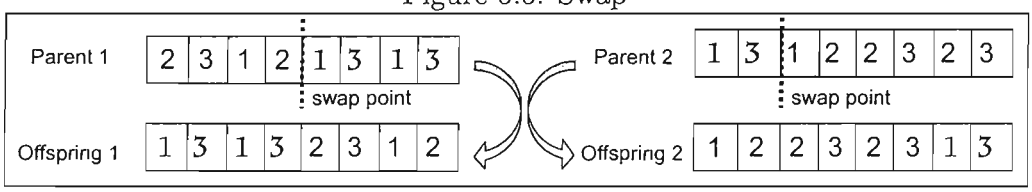
As a result of the crossover probability, p_c, the crossover may not be applicable to the selected parents. Instead of '*reproduction*' as in a standard GA, a new operator called '*swap*' was used to produce new offspring and introduce more diversity into the population. This can be achieved by:

S 1: Select randomly a swap point in a parent to form two sub-genes.

S 2: Swap the position of the sub-genes to form a new offspring.

The steps above are repeated for the second parent to create a second offspring. Figure 6.6 shows the process of swap on two selected parents. Before the swap commences, Parent 1 will pack rectangles {3,5,7} in bin 1, rectangles {1,4} in bin 2, and rectangles {2,6,8} in bin 3. After a swap point (between rectangles 4 and 5) has been selected in Parent 1, two sub genes consist of bins {2,3,1,2} (rectangle $1-4$) and bins {1,3,1,3} (rectangle $5-8$) are formed. Then, the sub genes will swap position to form a new offspring (Offspring 1) which will pack rectangles {1,3,7} in bin 1, rectangles {5,8} in bin 2, and rectangles {2,4,6} in bin 3. Similarly, Offspring 2 is created by using the same steps as explained above.

Figure 6.6: Swap



6.3.6 Mutation

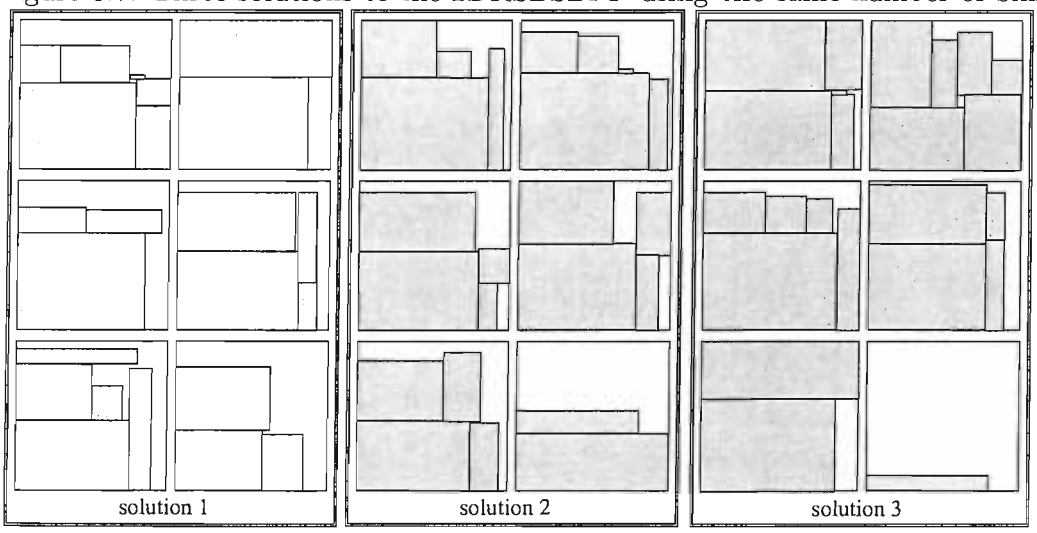
After a crossover or swap is performed, mutation takes place. This operator prevents all solutions in the population from converging on same local optima. We used two mutation operators in our MXGA. First of all, we select a subset of individuals from the new offspring population with a given *individual mutation* probability, p_M. An offspring is selected for individual mutation if the random number r assigned to the offspring is less than or equal to the individual mutation

probability (i.e $r \leq p_M$). Then, the selected offspring will go through the *gene mutation* process. Each gene in the selected offspring is visited and if $k \leq p_m$, the bin number is randomly changed in the interval [1,LB_o], where k is a random number and p_m is the *gene mutation* probability.

6.3.7 Fitness Evaluation

The main objective function in our proposed MXGA for the classic 2DRSBSBPP is to minimise the number of bins used to pack the complete set of rectangles. This is correct from the problem objective point of view. However, this is not sufficient to guide the search process since a large number of solutions will result in the same total number of bins used. Figure 6.7 shows three solutions to a problem instance involving 23 rectangles, which use the same number of bins. The bins in solution 1 and solution 2 are not very densely packed and show many empty areas enclosed between the rectangles. Solution 3 is the favoured one since the first five bins are very densely packed. The last bin contains only one rectangle, which one of the other bins may easily accommodate after some minor modifications to the packing layout. This solution is therefore a better than the other two and the objective function needs to indicate this via the fitness value.

Figure 6.7: Three solutions to the 2DRSBSBPP using the same number of bins



The quality of a packing pattern is determined by the utilisation of each bin as defined in equation (6.1), where j indicates the number of rectangles in the respective bin, A_{item_i} is the area of the rectangle i ($i = 1, 2, \ldots, j$) in the bin, and A_{object} is the capacity of the bin. The denser the rectangles are packed, the less waste is produced and the higher the bin utilisation.

$$Bin\ Utilisation = \frac{\sum_{i=1}^{j} A_{item_i}}{A_{object}}. \tag{6.1}$$

Next we consider the problem of aggregating the bin utilisation over all the bins. Falkenauer and Delchambre [90] suggest a fitness function for the 1DRSBSBPP that includes the capacity of the individual bins. In their work, equation (6.2) is used in a GA for a one-dimensional problem with M being the number of bins used. A_i is the area of all the items in a certain bin with the capacity of A_{object_i}. The exponent k is a constant ($k > 1$) and puts a higher weight on bins with higher utilisation. They experimented with several values for k and settled on $k = 2$.

$$Fitness = \frac{\sum_{i=1}^{M} (\frac{A_i}{A_{object_i}})^k}{M}\ ;\ k = 2. \tag{6.2}$$

From the mathematical point of view, equation (6.2) favours solutions where the rectangles are evenly packed across the bins, than solutions where rectangles are packed very densely in all bins except one bin, which is loosely packed in most cases. Refer back to Figure 6.7, and by applying equation (6.2), we will select the solution from solution 1 as having the highest fitness value in the solution space rather than solution 3. In solution 1, the items are packed quite evenly in every bin. While in solution 3, the first five bin are very densely packed but the last bin has a very small bin utilisation which will reduce the overall fitness value significantly.

In order to overcome these disadvantages, we propose an improved fitness function where we permit one bin to have a poor individual bin utilisation. The rectangle(s) in this bin has (have) a much higher chance of being removed from the bin and packed in other bins. To achieve this, we first sort the bins in non-increasing

order of their bin utilisation. Then, the fitness value is calculated, ignoring the last bin, using the following equation:

$$Fitness^* = \frac{\sum_{i=1}^{M-1}(\frac{A_i}{A_{object_i}})^2}{M-1}, M > 1. \quad (6.3)$$

For a single bin solution, the fitness value is calculated using equation (6.2). Note that equation(6.3) does not represent the '*true*' value of the overall bin utilisation. Equation (6.3) puts more pressure on bins with higher bin utilisation by discarding the bin with the lowest bin utilisation. The idea behind this is to always select the solutions which contain bins with high individual bin utilisation plus a bin with a very low individual bin utilisation. We hope that the bin with the lowest bin utilisation can easily be removed in the next iteration/generation.

It is worth mentioning that the fitness value obtained from equation 6.3 is only used during the execution of the algorithm. Once the stopping criterion is reached, the '*true*' overall bin utilisation for the best solution found is calculated using equation (6.2).

6.4 2DRSBSBPP with Due Dates

Having successfully developed the LGF placement routine and our proposed MXGA, we now extend our study on 2DRSBSBPP by assigning due dates to each rectangle that is to be packed and a fixed processing time for each bin. We define the problem as follows:

"Given are n rectangles, each characterised by a height h_j, a width w_j, and a due date d_j, for $j = 1, 2, 3, \ldots, n$ and an unlimited number of identical rectangular bins, each having a height H, a width W, and a fixed processing time P. The 2DRSBSBPP with due date has the objective of minimising the maximum lateness of the n rectangles by packing them, without overlap, and minimising the number of bins."

The aim of this study is to solve the dilemma often faced in the industrial application which involves the trade-off between the customers' satisfaction and

the manufacturer's efficiency. In a classic 2DRSBSBPP, we either assume the rectangles due dates are not considered during packing stage, or the productions are scheduled to complete within a single production period. These assumptions render time to not be an issue in the packing stage, with focus only on the quality of the packing.

This extension has practical applications in the wood and metal industries. In the metal industry for instance, suppose that the bins used are the metal sheets with fixed dimensions, and the rectangles placed in a bin are the rectangular shapes to be cut from a metal sheet. Each metal sheet requires a fixed processing time on a cutting machine to cut all the shapes, where each shape has a due date by which it should be completed.

Suppose that a group of customers place orders (i.e. shapes) with different due dates by which they should be completed. A decision has to be made on either to satisfy the customers by meeting the due dates of the orders placed, or to increase the packing efficiency by mixing the customers orders and therefore minimise the wastage of the metal sheets used.

From meeting the customers' satisfaction point of view, metal sheets which contain shapes with small due dates are ideally cut earlier. By doing this, shapes with small due dates are placed together in one or more metal sheets. Depending on the dimensions of the shapes, the metal sheets may not be fully utilised, where the packing layout will create waste. This solution becomes less appealing to the manufacturer if the wastage cannot be recycled rendering the cost of production to increase.

From the manufacturer's point of view, mixing the customers orders can increase the packing efficiency if the shapes of an order can be use to fill in the gaps created by the shapes of another order on the metal sheets. However, depending on the shapes' due dates, the approach of mixing the customers orders to achieve a higher packing efficiency may result in missing the due dates of the customers orders with small due dates.

However, due dates and packing efficiency may not be a direct trade-off. In a special case, mixing the customers orders with similar due dates might simultaneously increase the packing efficiency and meet customers due dates, since shapes with small dimensions are able to fill in the gaps between the larger shapes from other customers orders. By doing this, the metal sheets can be fully utilised where the metal sheets are densely placed and therefore creating less wastage. At the same time, this approach will meet the customers due dates by placing the shapes with similar due dates together in one or more metal sheets.

This problem can also be treated as a batching machine scheduling problem where a machine can processed several jobs simultaneously. A bin can be viewed as a single machine with a fixed capacity. Rectangles packed in the bin are the jobs processed in the machine. In this case, the jobs are processed in the machine with a fixed processing time. Thus, the problem can be transformed into a batching machine scheduling problem where the objective is to find a schedule which minimise the maximum lateness of the group jobs.

In order to deal with the 2DRSBSBPP with due dates, we first define two distinct objective functions for the problem to be solved:

1. minimise the maximum lateness of the rectangles to be packed with a secondary objective of minimising the number of bins used (ideal for customers' satisfaction);
2. minimise the number of bins used with a secondary objective of minimising the maximum lateness of the rectangles packed (ideal for manufacturer's efficiency).

Each of the objective function can be viewed as a *hierarchical optimisation* approach. This approach first optimises the primary objective, then the secondary objective is optimised subject to the additional constraint that the solution value of the primary objective is optimum. By alternating the objective functions in every G generations (or I iterations) during the execution of the local search algorithms, we will be able to find a good balance of the trade-off between the customers' satisfaction and salesman's routing efficiency.

The maximum lateness, $L_{\max}$ of the rectangles packed is calculated as follows: let B be the number of bins used, P is the fixed bin processing time, and d_i is the due date of rectangle i. Assume r_b is the number of rectangles packed in bin b, then the bin due date, δ_b, is obtained using the following equation

$$\delta_b = \min_{i=1,2,\ldots,r_b} \{d_i\}. \tag{6.4}$$

The bins are reindexed in a non-decreasing order of their bin due dates (i.e. $\delta_b \leq \delta_{b+1}$). Thus, the maximum lateness of the rectangles packed is

$$L_{\max} = \max_{b=1,2,\ldots,B} \{(P \times b) - \delta_b\}. \tag{6.5}$$

6.5 Lower Bound for 2DRSBSBPP with Due Dates

In this section, we derive a simple lower bound of the maximum lateness for the 2DRSBSBPP with due dates. The lower bound (for the number of bins used in non-oriented 2DRSBSBPP) proposed by Dell'Amico *et al.* [69] is used in the derivation of this lower bound.

As we do not know which rectangle has the maximum value of lateness, we first sort the n rectangles in EDD order (i.e. non-decreasing order of their due dates) to identify the rectangle with the largest due date. Then, by using the lower bound discussed in Section 3.4.3.2, we obtain the number of bins used for the n rectangles. As mentioned earlier, this problem uses a fixed processing time P, for the bins. The completion time for the bins is the product of the fixed processing time and the number of bins used. Thus, the lower bound on the maximum lateness for the n rectangles is the difference between the completion time and the highest due date. The steps (summarised below) are repeated by removing the rectangle with the largest due date from the list until only one rectangle is left to be packed into the bin.

S 1: Sort the n rectangles in a non-decreasing order of their associated rectangle due date d_i, so that $d_1 \leq d_2 \leq \ldots \leq d_n$. Set $j = n$.

S 2: Calculate the lower bound for the number of bins used, LB^j_{Bin} (as explained in Section 3.4.3.2) for the j rectangles.

S 3: Compute a lower bound on the maximum lateness of the j rectangles, $L^j = (P \times LB^j_{\text{Bin}}) - d_j$; where d_j = largest due date.

S 4: Reduce j by 1 (i.e. $j := j - 1$), and repeat **S 2** - **S 3** until $j = 1$.

Thus, the lower bound of the maximum lateness for 2DRSBSBPP with due dates is:

$$LB_{L_{\max}} = \max_{j=1,2,3,\ldots,n} \{L^j\}. \tag{6.6}$$

To justify that the above procedure generates a valid lower bound, consider rectangles $1, 2, \ldots, j$. The number of bins required to pack these rectangles is at least LB^j_{Bin} as shown by Dell'Amico *et al.* [69] in Section 3.4.3.2. In an optimal solution, define rectangle i so that none of the rectangles $1, \ldots, i-1, i+1, \ldots, j$ appears in a later bin than rectangle i. If rectangle i is in bin b, then

$$\begin{aligned} LB_{L_{\max}} &\geq (P \times b) - d_i \\ &\geq (P \times LB^j_{\text{Bin}}) - d_i \\ &\geq (P \times LB^j_{\text{Bin}}) - d_j, \end{aligned} \tag{6.7}$$

as defined in **S 3**.

6.6 Competitors - Performance Measure

6.6.1 Unified Tabu Search

In this subsection, we briefly describe the Tabu Search used in the computational experiments. It is based on the Unified Tabu Search (UTS) developed by Lodi *et al.* [194]. The choice was made based on the effectiveness of the algorithm in generating high quality solutions. For more details about the framework of the UTS, see Lodi *et al.* [193, 194, 196, 198].

The main feature of the framework is the use of a unified parametric neighbourhood, whose size and structure are dynamically varied during the search. The algorithm also adopts a search scheme which is independent of the specific packing problem to be solved.

Given a current solution, the neighbourhood is searched through *moves* which consist in modifying the solution by changing the packing of a subset of rectangle S, in an attempt to empty a specific *target bin*. Subset S is defined so as to include one rectangle, j, from the target bin and the current contents of k other bins. The new packing for S is obtained by executing a heuristic placement routine A on S. The value of parameter k, which defines the size and the structure of the current neighbourhood, is automatically updated during the search. The algorithm maintains k distinct tabu lists.

The target bin is selected as the one minimising, over all current bin i, the *filling function*

$$\varphi(S_i) = \alpha \frac{\sum_{j \in S_i} w_j h_j}{WH} - \frac{|S_i|}{n}, \tag{6.8}$$

where S_i denotes the set of rectangles currently packed in bin i, and α is a user specified positive weight. The resulting choice favours the selection of target bins with a low bin utilisation (breaking ties by bins packing a relatively large number of rectangles).

The overall algorithm is briefly stated in Figure 6.8. An initial incumbent solution is obtained by executing routine A on the complete instance, while the initial tabu search solution consists of packing one rectangle per bin. At each iteration, a target bin is selected, and a sequence of moves, each performed within the procedure SEARCH (Figure 6.9), tries to empty it. The procedure SEARCH also updates the value of parameter k and, in special cases, may perform the procedure DIVERSIFICATION (given in figure 6.10). The execution is halted as soon as a proven optimal solution is found, or a time limit is reached.

Figure 6.8: Unified Tabu Search Framework (Lodi *et al.* [198])

```
algorithm TSpack:
    z* := A({1, ..., n}) (comment: incumbent solution value);
    let L be a lower bound on the optimal solution value;
    if z* = L then stop;
    initialise all tabu lists to empty;
    pack each item into a separate bin;
    z := n (comment: Tabu Search solution value);
    d := 1;
    determine the target bin t;
    while time (or iteration) limit is not reached do
        diversify := false; k := 1;
        while diversify = false; and z* > L do
            k_in := k;
            call SEARCH(t, k, diversify, z);
            z* := min{z*, z};
            if k < k_in then determine the new target bin t
        end while;
        if z* = L then stop
        else call DIVERSIFICATION(d, z, t)
    end while
end.
```

Figure 6.9: Unified Tabu Search: Procedure SEARCH (Lodi *et al.* [198])

```
procedure SEARCH(t, k, diversify, z):
    penalty* := +∞;
    for each j ∈ S_t do
        for each k-tuple K of bins not including t do
            S := {j} ∪ (⋃_{i∈K} S_i);
            penalty := +∞;
            case
                A(S) < k:
                    execute the move and update the solution value z;
                    k := max{1, k − 1};
                    return;
                A(S) = k:
                    if the move is not tabu or S_t ≡ {j} then
                        execute the move and update the solution value z;
                        if S_t ≡ {j} then k := max{1, k − 1};
                        return
                    end if;
                A(S) = k + 1 and k > 1:
                    let I be the set of k + 1 bins used by A;
                    t̄ := arg min_{i∈I}{φ(Si)}, T := (S_t\{j}) ∪ S_t̄;
                    if A(T) = 1 and the move is not tabu then
                        penalty := min{φ(T), min_{i∈I\{t̄}}{φ(S_i)}}
            end case;
            penalty* := min{penalty*, penalty};
        end for;
    end for;
    if penalty* ≠ +∞ then execute the move corresponding to penalty*
    else if k = k_max then diversify := true else k := k + 1
return.
```

The value of k is updated as follows. When a move in the SEARCH procedure decreases the number of k bins ($A(S) < k$), or when a non-tabu move removes rectangle j from target bin t by packing the set S into exactly k bins ($A(S) = k$), the move is immediately performed and the neighbourhood size is reduced by one unit (i.e. $k := k - 1$). Its value is increased by one unit (i.e. $k := k + 1$), if the neighbourhood has been completely searched without finding an acceptable move. If k has already reached a maximum prefixed value $k_{\max}$, the procedure DIVERSIFICATION, as given in Figure 6.10, is performed.

Figure 6.10: Unified Tabu Search: Procedure DIVERSIFICATION (Lodi *et al.* [198])

```
procedure DIVERSIFICATION(d, z, t)
    if d ≤ z and d < d_max then
        d := d + 1;
        let t be the bin with d-th smallest value of φ(·);
    else
        remove from the solution the ⌊z/2⌋ bins with smallest φ(·) value;
        pack into a separate bin each item currently packed in a removed bin;
        reset all tabu list to empty;
        d := 1
return.
```

A move that is not immediately performed is evaluated through a *penalty*. The *penalty* is infinity if the move is tabu, or if routine A used at least two extra bins (i.e. $A(S) > k + 1$)), or if $k = 1$. Otherwise, the *penalty* is obtained as follows. A local target bin, $\bar{t}$ is determined among the $k + 1$ bins produced by A. Routine A is executed on the rectangles set T with the rectangles in bin $\bar{t}$ plus the residual rectangles in the target bin t, in an attempt to get a single bin solution. If this happens, the *penalty* of the overall move is the minimum among the filling function values obtained for the $k + 1$ resulting bins. Otherwise, the move is not acceptable and its *penalty* is set to infinity. The move with the minimum *penalty* (if any) is performed when the entire neighbourhood is searched without finding an acceptable move.

As mentioned earlier, each neighbourhood has a tabu list and a tabu tenure τ_k ($k = 1, \ldots, k_{\max}$). For $k > 1$, each list stores the *penalty** values corresponding to the last τ_k moves performed in the corresponding neighbourhood. For $k = 1$,

the tabu list stores the values of the filling function, $\varphi(\cdot)$, corresponding to the last τ_1 sets for which a move has been performed.

Having constructed the UTS using LGF as the placement routine, we incorporate the idea of finding the minimum of the maximum lateness for the rectangles to be packed into the design of the UTS. Since the UTS already is a very efficient algorithm, we only need some minor modifications to the original code to deal with the alternative objective function. Recall that, from the study of the 2DRSBSBPP with due dates, we have the following two distinct objective functions:

1. minimise the maximum lateness of the rectangles packed with a secondary objective of minimising the number of bins used;
2. minimise the number of bins used with a secondary objective of minimising the maximum lateness of the rectangles packed.

In order to find a good balance between the trade-off of the objective functions (i.e. customers' satisfaction and packing efficiency), we alternate between the objective functions every I iterations by introducing a new SEARCH_1 procedure (given in Figure 6.11) into the main algorithm of the UTS. Note that the SEARCH procedure developed by Lodi *et al.* [194] favoured the second objective where packing efficiency is very high.

Inspired by the *target bin t*, we defined the *weakest bin, l* which contains rectangles with small due dates but high lateness in current solution as follows. Let Q_j be the set of rectangles i, each having a due date d_i, currently packed into bin j with a fixed processing time P, and B is the number of bins used in the current solution. Bin l is the one maximising, over all current bin j ($j = 1, \ldots, B$), the following *lateness equation*

$$L_j = (P \times j) - \delta_j \ ; \ \text{where } \delta_j = \min_{i \in Q_j}\{d_i\}. \tag{6.9}$$

The moves try to remove the rectangle(s) with the smallest due date from the *weakest bin* using the procedure SEARCH_1 as explained in Figure 6.11. Variable F (in Figure 6.11) denotes the maximum lateness value obtained from a new

packing layout when a rectangle from the *weakest bin* is removed and placed in one of the k bins. The solution is considered "acceptable" if: (i) the maximum lateness over all bins is decreased (i.e. $F < L_{\max}$) or (ii) the number of bins used does not exceed the current solution value while maintaining the maximum lateness value (i.e. $F = L_{\max}$ and $A(Q) \leq k$). We allow the number of bins used to increase if it results in the decrease of the maximum lateness.

Figure 6.11: Unified Tabu Search: Procedure SEARCH_1

```
procedure SEARCH_1(l, k, diversify, y, z):
    penalty* := +∞;
    L_max := y;
    for each j ∈ Q_l do
        for each k-tuple K of bins not including l do
            Q := {j} ∪ (⋃_{i∈K} Q_i);
            penalty := +∞;
            case
                F < L_max:
                    execute the move and update the solution value y and z;
                    k := max{1, k − 1};
                    return;
                F = L_max and A(Q) ≤ k:
                    if the move is not tabu or Q_l ≡ {j} then
                        execute the move and update the solution value z;
                        if Q_l ≡ {j} then k := max{1, k − 1};
                        return
                    end if;
                F > L_max and A(Q) ≤ k:
                    if the move is not tabu then
                        penalty := F
            end case;
            penalty* := min{penalty*, penalty};
        end for;
    end for;
    if penalty* ≠ +∞ then execute the move corresponding to penalty*
    else if k = k_max then diversify := true else k := k + 1
return.
```

As in the procedure SEARCH, the value of k in the SEARCH_1 procedure is updated as follows. When either (i) or (ii) is applied, the move is immediately performed and the neighbourhood size is reduced by one unit (i.e. $k := k - 1$). Its value is increased by one unit (i.e. $k := k + 1$), if the neighbourhood has been completely searched without finding an acceptable move. If k has already reached a maximum prefixed value $k_{\max}$, the procedure DIVERSIFICATION (Figure 6.10) is performed. But, when neither (i) nor (ii) apply, a penalty is associated with the move. The penalty is infinity if the move is tabu, or if the maximum lateness value

obtained from the new packing is higher than the current solution when routine A uses at least one extra bin (i.e. $F > L_{\max}$ and $A(Q) > k$). Otherwise, the penalty takes the value of F.

When the neighbourhood has been searched entirely without detecting cases (i) and (ii), the move having the minimum finite penalty (if any) is performed and the control returns to the main algorithm. As previously mentioned, there is a tabu list and a tabu tenure τ_k ($k = 1, \ldots, k_{\max}$) for each neighbourhood. Each list keeps a memory of the *penalty** values corresponding to the last τ_k moves performed.

6.6.2 Randomised Descent Method

The Randomised Descent Method (RDM) we employed in the computational experiments has a similar framework as in the UTS discussed in the previous subsection. The main difference lies in the use of the tabu list in the UTS and the acceptance rule and randomisation introduced in the RDM.

The SEARCH and SEARCH_1 procedures discussed in the previous subsection also have been modified to suit the framework of the algorithm. While the main features of the UTS (i.e. unified parametric neighbourhood, stopping criteria, and diversification) are used in the RDM, we also adopted an acceptance rule which allows the neutral move solutions up to R consecutive iterations before terminating the algorithm. When there are multiple identical neutral moves found during the SEARCH procedure (given in Figure 6.12) or SEARCH_1 procedure (given in Figure 6.13) in a single iteration, randomisation is used to randomly select a move from the list of identical moves. Consequently, the procedure can escape from falling into the same local optimum and continue its search. Note that the deteriorating move (i.e. $[A(S) = k + 1$ and $k > 1]$ in Figure 6.9 and $[F > L_{\max}$ and $A(Q) < k]$ in Figure 6.11) is not considered in the RDM.

As in UTS for 2DRSBSBPP with due dates, we alternate the objective functions (discussed in Section 6.4) in every I iterations and employ *first improve* strategy for the RDM.

Figure 6.12: Randomised Descent Method: Procedure SEARCH

```
procedure SEARCH(t, k, diversify, z):
    U := 0;
    for each j ∈ S_t do
        for each k-tuple K of bins not including t do
            S := {j} ∪ (⋃_{i∈K} S_i);
            case
                A(S) < k:
                    execute the move and update the solution value z;
                    k := max{1, k − 1};
                    return;
                A(S) = k:
                    if S_t ≡ {j} then
                        execute the move and update the solution value z;
                        k := max{1, k − 1};
                        return
                    else
                        U := U + 1;
                        if U ≤ T then update the move into temp list
            end case;
        end for;
    end for;
    if U ≠ 0 then randomly execute a move from the temp list
    else if k = k_max then diversify := true else k := k + 1
return.
```

Figure 6.13: Randomised Descent Method: Procedure SEARCH_1

```
procedure SEARCH_1(l, k, diversify, y, z):
    U := 0;
    L_max := y;
    for each j ∈ Q_l do
        for each k-tuple K of bins not including l do
            Q := {j} ∪ (⋃_{i∈K} Q_i);
            case
                F < L_max:
                    execute the move and update the solution value y and z;
                    k := max{1, k − 1};
                    return;
                F = L_max and A(Q) ≤ k:
                    U := U + 1;
                    if U ≤ T then update the move into temp list
                    end if;
            end case;
        end for;
    end for;
    if U ≠ 0 then randomly execute a move from the temp list
    else if k = k_max then diversify := true else k := k + 1
return.
```

6.7 Computational Experience

We are now in position to give computational results on the performance of the LGF and the proposed local search algorithms. Having explained the experimental design for our computational experiments, we conduct the computational experiments in four parts. First, we compare the LGF with some well known heuristic placement routines, namely: BLF, Touching Perimeter (TP) and Floor Ceiling (FC), as described in Section 3.4.2.1. Then, we compare the results obtained by using TP and LGF as the placement routine (i.e. inner heuristic) in the Unified Tabu Search developed by Lodi *et al.* [194]. In the third part of the computational experiments, we give some computational results of our proposed MXGA at different stages of development. We present the final results of our extensive computational experiments for different local search algorithms in two different scenarios. First, we show the final results of comparing our proposed MXGA with standard GA (SGA), UTS and RDM where the objective function of the problem to be solved is minimising the number of bins used (and therefore maximising the overall bin utilisation). Then, comparisons of different local search algorithms based on the objective functions discussed in Section 6.4 are presented for 2DRS-BSBPP with the inclusion of rectangle due dates and a fixed bin processing time.

6.7.1 Experimental Design

The algorithms are coded in ANSI-C using Microsoft Visual C++ 6.0 as the compiler and run on a Pentium 4, 2.0 GHz computer with 512 MB memory. We use problem instances taken from the literature. We consider ten different classes of problem instances. The first six classes (*I* - *VI*) are proposed by Berkey and Wang [32] as in Table 6.1. In each of the six classes, all of the rectangle sizes are generated in the same interval. Martello and Vigo [203] propose the next four classes (*VII* - *X*)(Table 6.1), where a more realistic situation is considered. The rectangles are classified into four types:

Type 1: w_j uniformly random in $[\frac{2}{3}W, W]$, h_j uniformly random in $[1, \frac{1}{2}H]$;
Type 2: w_j uniformly random in $[1, \frac{1}{2}W]$, h_j uniformly random in $[\frac{2}{3}H, H]$;
Type 3: w_j uniformly random in $[\frac{1}{2}W, W]$, h_j uniformly random in $[\frac{1}{2}H, H]$;
Type 4: w_j uniformly random in $[1, \frac{1}{2}W]$, h_j uniformly random in $[1, \frac{1}{2}H]$;

Table 6.1: Classes for the Problem Instances (Lodi *et al.* [194])

Class	Bin ($W \times H$)	Item (w_j and h_j)
I	10×10	uniformly random in [1,10]
II	30×30	uniformly random in [1,10]
III	40×40	uniformly random in [1,35]
IV	100×100	uniformly random in [1,35]
V	100×100	uniformly random in[1,100]
VI	300×300	uniformly random in [1,100]
VII	100×100	*Type 1* with probability 70%, *Type 2, 3, 4* with probability 10% each
VIII	100×100	*Type 2* with probability 70%, *Type 1, 3, 4* with probability 10% each
IX	100×100	*Type 3* with probability 70%, *Type 1, 2, 4* with probability 10% each
X	100×100	*Type 4* with probability 70%, *Type 1, 2, 3* with probability 10% each

For each class, five values of n : $20, 40, 60, 80, 100$ are considered. For each combination of class and value of n, 10 problem instances are generated. The problem instances are provided by Lodi *et al.* [194] and are publicly available on the web [64]. We adopt the following abbreviations for the remaining subsections:

1P	: 1-point crossover
2P	: 2-point crossover
SGA	: Standard Genetic Algorithm
MXGA	: MultiCrossover Genetic Algorithm
$\mathbf{MXGA}_1$	: MXGA with *pack_extra* strategy in the decoding stage
$\mathbf{MXGA}_2$	: MXGA with *pack_above* strategy in the decoding stage
$\mathbf{MXGA}_3$	: MXGA with *repack* strategy in the decoding stage
$\mathbf{MXGA}_F$	: final version of MXGA
$\mathbf{UTS}_{TP}$	: Unified Tabu Search with TP as the inner heuristic
$\mathbf{UTS}_{LGF}$	: Unified Tabu Search with LGF as the inner heuristic
RDM	: Randomised Descent Method

Since the optimal solutions for the problem instances are not known, we use the lower bound proposed by Dell'Amico *et al.* [69] for the number of bins used, as described in Section 3.4.3.2. We compare the performance of the various heuristic placement routines and local search algorithms on the basis of the following statistics:

$$\text{Average Ratio, } Ratio = \frac{\sum_{i=1}^{K} \left(\frac{UB_{\text{Bin}_i}}{LB_{\text{Bin}_i}} \right)}{K}, \tag{6.10}$$

$$\text{Overall Bin Utilisation, } OBU = \frac{\sum_{i=1}^{K} U_i}{K}, \text{ where} \tag{6.11}$$

$$U_i = \frac{\sum_{j=1}^{UB_{\text{Bin}_i}} (\frac{A_j}{W \times H})^2}{UB_{\text{Bin}_i}}, \text{ and} \tag{6.12}$$

$$\text{Average Relative Percentage Deviation, } ARD = \frac{\sum_{i=1}^{K} D_i}{K}, \text{ where} \tag{6.13}$$

$$D_i = \frac{UB_{L_{\max_i}} - LB_{L_{\max_i}}}{LB_{L_{\max_i}}} \times 100\%. \tag{6.14}$$

Note that parameter K in equations (6.10), (6.11), and (6.13) takes the value of the number of problem instances tested for each combination of class and value of n. In this case $K = 10$. Also note that the variables UB_{Bin_i} and $UB_{L_{\max_i}}$ represent the heuristic solutions found in instance i for the number of bins used and the maximum lateness respectively. Similarly, LB_{Bin_i} and $LB_{L_{\max_i}}$ represent the lower bound of the problem instance i for number of bins used and the maximum lateness respectively. Equation (6.12) is the overall bin utilisation as explained in equation (6.2), where A_j is the total area of all the rectangles in bin j ($j = 1, 2, \ldots, UB_{\text{Bin}_i}$).

The specific values for the generic design variables in MXGA, UTS and RDM are summarised in Table 6.2 and 6.3 respectively. Initial computational experiments are performed to determine the size of the candidate list of temporary offspring. Five values of t ($t = 3, 5, 7, 9, 10$) are tested and results show that $t = 5$ gives the best result within a reasonable computation time.

Table 6.2: Implementation of generic design variables for MXGA and SGA

variable	value
chromosome length, L	n (number of rectangles)
population size, P_{pop}	100
crossover operator	1-point and 2-point
crossover rate, p_c	0.75
multicrossover, t (MXGA only)	5 (= 10 temporary offspring)
individual mutation rate, p_M	0.25
gene mutation rate, p_m	$1/n$
selection mechanism	probabilistic binary tournament
filtration rate, F (MXGA only)	every 50 generations

Table 6.3: Implementation of generic design variables for UTS_{TP}, UTS_{LGF} and RDM

variable	value
tabu tenure for all k, τ_k (UTS_{TP} and UTS_{LGF} only)	3
max. number of distinct tabu lists, $k_{\max}$ (UTS_{TP} and UTS_{LGF} only)	3
α (equation (6.8))	20
max. value of the differentiation counter d, $d_{\max}$	50
iterations executed before alternating the objective function, I	100
max. no. of consecutive neutral move allowed per run, R (RDM only)	1000

In the final part of the computational experiments, having computed the completion time of LB_{Bin_i}, $C_{\text{LB}_i} = P \times LB_{\text{Bin}_i}$ ($P = 100$), for each problem instance, we generate three sets of integer due dates from the uniform distribution of $[101, \beta C_{\text{LB}_i}]$, where $\beta \in \{0.6, 0.8, 1.0\}$. We label each set of due date class as follows (assuming $LB_{\text{Bin}_i} > 1$):

Class A: $[101, 0.6C_{\text{LB}_i}]$;

Class B: $[101, 0.8C_{\text{LB}_i}]$;

Class C: $[101, 1.0C_{\text{LB}_i}]$.

6.7.2 A Comparison of Different Heuristic Placement Routines

In this subsection, we first compare the LGF with the BLF as explained in Section 3.4.2.1. Literature suggests that the BLF routine outperforms the Bottom-Left (BL) routine, although it has a longer execution time. Also, the use of the preordering sequence by non-increasing width, height or area gives better quality solutions. Hence, we concentrate only on the BLF routine. This is also due to the fact that both placement routines have the same time complexity of $O(n^2)$.

The results of the BLF placement routine are generated using the following five preordering sequences of the rectangles:

DW : **D**ecreasing **W**idth, breaking ties by decreasing height.
DH : **D**ecreasing **H**eight, breaking ties by decreasing width.
DA(W): **D**ecreasing **A**rea, breaking ties by decreasing **W**idth.
DA(H) : **D**ecreasing **A**rea, breaking ties by decreasing **H**eight.
R : **R**andom permutation.

As we want to extensively test the LGF placement routine, we compare our results with the TP and FC placement routines developed by Lodi *et al.* [194] using the problem instances provided in [64]. In this experiment, we use DA(W) preordering sequence in the BLF placement routine to generate the solutions. Note that we do not give the execution times for both experiments, as these are negligible (never exceeding 0.1 seconds per execution).

LGF vs. BLF

The computational results comparing LGF with BLF placement routine are presented in Table 6.4. The first two columns give the class and the value of n. The next five columns refer to the BLF placement routine with five different preordering sequences (DW, DH, DA(W), DA(H), R). The last column refers to the LGF placement routine.

For each algorithm, the entries report the average ratio (equation (6.10)), computed over the 20 randomly generated problem instances. For each class, the final line gives the average over all values of n. The final line of Table 6.4 gives the overall average value over all classes. The bold face figures represent the best solution obtained for each class.

We first observe that the solution quality of LGF placement routine outperforms the BLF placement routine in all five preordering categories. The best average value found in BLF placement routine is with DA(W) preordering sequence. By considering in each class, we see that the LGF routine produces the best results in 9 out of 10 classes. There is clear evidence that filling the gaps in the partial layout by dynamically selecting the best rectangle is better than based on the sequence of the rectangles supplied.

Table 6.4: Comparison of BLF Routine with LGF Routine (Execution Time: less than 0.1 CPU second)

Class	n	BLF					LGF
		DW	DH	DA(W)	DA(H)	R	
I	20	1.170	1.170	1.148	1.148	1.251	1.117
	40	1.115	1.115	1.109	1.115	1.180	1.067
	60	1.123	1.130	1.125	1.130	1.188	1.085
	80	1.130	1.130	1.124	1.124	1.174	1.082
	100	1.132	1.132	1.126	1.126	1.155	1.060
Average		1.136	1.136	1.126	1.129	1.190	**1.082**
II	20	1.000	1.000	1.000	1.000	1.100	1.000
	40	1.025	1.025	1.025	1.025	1.025	1.000
	60	1.075	1.075	1.075	1.075	1.250	1.050
	80	1.017	1.033	1.033	1.033	1.150	1.000
	100	1.042	1.046	1.042	1.046	1.083	1.000
Average		1.032	1.036	1.035	1.036	1.122	**1.010**
III	20	1.265	1.265	1.265	1.265	1.333	1.167
	40	1.248	1.248	1.229	1.229	1.339	1.145
	60	1.220	1.220	1.194	1.194	1.259	1.130
	80	1.232	1.232	1.216	1.216	1.263	1.118
	100	1.259	1.255	1.233	1.233	1.275	1.100
Average		1.245	1.244	1.227	1.227	1.294	**1.132**
IV	20	1.100	1.100	1.050	1.050	1.350	1.050
	40	1.000	1.000	1.000	1.000	1.000	1.000
	60	1.100	1.100	1.125	1.125	1.200	1.075
	80	1.017	1.050	1.033	1.050	1.200	1.000
	100	1.017	1.017	1.050	1.050	1.104	1.017
Average		1.047	1.053	1.052	1.055	1.171	**1.028**
V	20	1.213	1.201	1.173	1.173	1.265	1.137
	40	1.186	1.186	1.149	1.149	1.278	1.154
	60	1.181	1.181	1.156	1.181	1.235	1.135
	80	1.202	1.202	1.181	1.181	1.255	1.144
	100	1.170	1.172	1.144	1.172	1.219	1.123
Average		1.190	1.188	1.161	1.171	1.250	**1.138**
VI	20	1.000	1.000	1.000	1.000	1.050	1.000
	40	1.300	1.350	1.350	1.350	1.350	1.250
	60	1.125	1.125	1.075	1.125	1.300	1.075
	80	1.092	1.108	1.108	1.108	1.158	1.075
	100	1.167	1.167	1.183	1.183	1.250	1.100
Average		1.137	1.150	1.143	1.153	1.222	**1.100**
VII	20	1.252	1.252	1.185	1.185	1.283	1.213
	40	1.234	1.234	1.194	1.194	1.312	1.199
	60	1.218	1.218	1.194	1.218	1.284	1.163
	80	1.229	1.229	1.202	1.202	1.320	1.168
	100	1.204	1.204	1.182	1.204	1.290	1.149
Average		1.227	1.227	1.191	1.201	1.298	**1.178**
VIII	20	1.276	1.276	1.215	1.276	1.358	1.203
	40	1.239	1.239	1.191	1.191	1.316	1.176
	60	1.231	1.231	1.200	1.231	1.313	1.174
	80	1.234	1.236	1.205	1.205	1.315	1.189
	100	1.203	1.203	1.179	1.203	1.281	1.145
Average		1.237	1.237	1.198	1.221	1.317	**1.177**
IX	20	1.014	1.014	1.003	1.003	1.014	1.014
	40	1.007	1.007	1.000	1.007	1.007	1.007
	60	1.005	1.005	1.000	1.005	1.002	1.005
	80	1.005	1.005	1.001	1.005	1.003	1.005
	100	1.006	1.006	1.001	1.001	1.003	1.006
Average		1.007	1.007	**1.001**	1.004	1.006	1.007
X	20	1.246	1.246	1.213	1.246	1.488	1.167
	40	1.157	1.149	1.149	1.157	1.408	1.117
	60	1.162	1.169	1.157	1.162	1.376	1.092
	80	1.136	1.152	1.132	1.152	1.397	1.082
	100	1.144	1.148	1.121	1.121	1.407	1.093
Average		1.169	1.173	1.154	1.168	1.415	**1.110**
AVERAGE		1.143	1.145	1.129	1.137	1.228	**1.096**

LGF vs. BLF vs. FC vs. TP

Table 6.5 summarises the results comparing the BLF with DA(W) preordering sequence, FC, and TP placement routines with LGF. The decision of using the preordering sequence of DA(W) in the BLF placement routine is based on the computational results reported in the previous experiment. For each placement routine, the entries report the average ratio (equation (6.10)), computed over the 10 problem instances given in [64]. For each class, the final line gives the average overall values of n. The final line of Table 6.5 gives the overall average value over all classes. The bold face figures represent the best solution found in each class.

By considering, for each class, the average value computed over all values of n, we see that the BLF placement routine with DA(W) preordering sequence always produces the worst results among the four placement routines. Neither of the placement routines for LGF, FC and TP can be classified as the clear winner in this experiment as they produce mixed degrees of success in terms of the solution quality in each class. In terms of the solution quality for the average value over all classes, LGF only underperforms TP by 0.3% (7.6% over the lower bound for TP and 7.9% over the lower bound for LGF) but it outperforms FC by 2% (9.9% over the lower bound). However, if we recall that both FC and TP have a time complexity of $O(n^3)$ for their algorithm, while LGF only has a time complexity of $O(n^2)$, then it is advantageous to employ the LGF as the heuristic placement routine for 2DRSBSBPP.

Table 6.5: Comparison of LGF with BLF, FC, and TP (Lodi *et al* [194]) (Execution Time: less than 0.1 CPU second)

Class	n	**BLF**	**LGF**	**FC**	**TP**	Class	n	**BLF**	**LGF**	**FC**	**TP**
	20	1.09	1.03	1.06	1.05		20	1.00	1.00	1.00	1.00
	40	1.12	1.04	1.08	1.06		40	1.40	1.40	1.40	1.40
I	60	1.13	1.05	1.09	1.05	VI	60	1.10	1.05	1.05	1.05
	80	1.15	1.06	1.09	1.06		80	1.00	1.00	1.00	1.00
	100	1.12	1.04	1.07	1.03		100	1.13	1.07	1.07	1.07
Average		1.122	**1.044**	1.078	1.050	Average		1.127	**1.103**	1.104	1.104
	20	1.00	1.00	1.00	1.00		20	1.22	1.19	1.19	1.13
	40	1.10	1.10	1.10	1.10		40	1.20	1.12	1.17	1.10
II	60	1.10	1.05	1.05	1.00	VII	60	1.20	1.10	1.18	1.12
	80	1.07	1.07	1.03	1.07		80	1.20	1.10	1.17	1.11
	100	1.06	1.03	1.03	1.00		100	1.19	1.09	1.17	1.11
Average		1.065	1.050	1.042	**1.034**	Average		1.202	1.119	1.176	**1.114**
	20	1.20	1.06	1.18	1.06		20	1.23	1.15	1.16	1.16
	40	1.22	1.13	1.16	1.11		40	1.22	1.16	1.19	1.16
III	60	1.26	1.10	1.19	1.11	VIII	60	1.19	1.09	1.18	1.11
	80	1.27	1.10	1.15	1.10		80	1.19	1.10	1.16	1.11
	100	1.23	1.08	1.13	1.08		100	1.19	1.09	1.17	1.12
Average		1.239	1.093	1.162	**1.092**	Average		1.204	**1.116**	1.172	1.132
	20	1.00	1.00	1.00	1.00		20	1.01	1.01	1.00	1.01
	40	1.00	1.00	1.00	1.00		40	1.02	1.02	1.01	1.02
IV	60	1.10	1.15	1.10	1.10	IX	60	1.01	1.01	1.01	1.01
	80	1.10	1.10	1.10	1.07		80	1.01	1.01	1.01	1.01
	100	1.13	1.07	1.07	1.03		100	1.01	1.01	1.01	1.01
Average		1.065	1.063	1.054	**1.040**	Average		1.011	1.011	**1.008**	1.012
	20	1.15	1.09	1.08	1.06		20	1.15	1.20	1.15	1.20
	40	1.18	1.10	1.10	1.11		40	1.13	1.07	1.09	1.08
V	60	1.16	1.09	1.11	1.08	X	60	1.14	1.08	1.09	1.09
	80	1.17	1.09	1.11	1.08		80	1.14	1.06	1.06	1.06
	100	1.16	1.08	1.10	1.08		100	1.11	1.07	1.07	1.06
Average		1.165	1.092	1.100	**1.082**	Average		1.135	1.098	**1.092**	1.098
						AVERAGE		1.133	1.079	1.099	**1.076**

6.7.3 Unified Tabu Search

This subsection compares the empirical performance of UTS using TP and LGF as the inner heuristic. Note that the results of UTS_{TP} are extracted from Lodi *et al.* [194] while the results of UTS_{LGF} are generated from the problem instances given in [64]. This is a comparison of a classic 2DRSBSBPP. In each algorithm, the results are obtained from the average of 10 (i.e. K) problem instances. The execution in both algorithms are halted as soon as a time limit of 60 CPU seconds per instance is reached.

Table 6.6 gives the results of the comparison between the UTS_{TP} and UTS_{LGF}. As explained previously, the first and third pairs of columns correspond to the class and the value of n. The second and last pairs of columns refer to the average ratio (equation (6.10)) computed by UTS_{TP} and UTS_{LGF} respectively. In this table too, for each class, the final line gives the average over all values of n. The final line of Table 6.6 gives the overall average value over all classes. The bold face figures represent the best solutions obtained for each class.

Based on the average value computed over all values of n, we can observe that both UTS_{TP} and UTS_{LGF} generally improve the initial deterministic solution produced by the inner heuristic (i.e. TP and LGF respectively in Table 6.5), except Class IV and VI produced by UTS_{TP}. With the help of the tabu search approach, we further improved the solution quality of the LGF from 7.9% to 6.0% over the lower bound. The approach also improved the solution quality of the TP from 7.6% to just 6.0% over the lower bound. Note that both algorithms achieve the same result (i.e. 1.060). We can conclude that, the unified tabu search is an effective approach to 2DRSBSBPP regardless of the specific placement routine used in the SEARCH procedure.

Table 6.6: Comparison of UTS_{TP} (Lodi *et al.* [194]) with UTS_{LGF} (60 CPU seconds per run)

Class	n	$\mathbf{UTS}_{TP}$	$\mathbf{UTS}_{LGF}$	Class	n	$\mathbf{UTS}_{TP}$	$\mathbf{UTS}_{LGF}$
	20	1.05	1.03		20	1.00	1.00
	40	1.04	1.04		40	1.40	1.30
I	60	1.04	1.04	VI	60	1.05	1.05
	80	1.06	1.06		80	1.00	1.00
	100	1.03	1.03		100	1.07	1.07
Average		1.044	**1.039**	Average		1.104	**1.083**
	20	1.00	1.00		20	1.11	1.13
	40	1.10	1.10		40	1.08	1.08
II	60	1.00	1.00	VII	60	1.06	1.07
	80	1.03	1.00		80	1.10	1.10
	100	1.00	1.00		100	1.08	1.08
Average		1.026	**1.020**	Average		**1.086**	1.093
	20	1.06	1.04		20	1.10	1.10
	40	1.09	1.10		40	1.10	1.13
III	60	1.08	1.09	VIII	60	1.07	1.07
	80	1.07	1.09		80	1.08	1.09
	100	1.07	1.06		100	1.09	1.08
Average		**1.074**	1.077	Average		**1.088**	1.094
	20	1.00	1.00		20	1.00	1.00
	40	1.00	1.00		40	1.01	1.01
IV	60	1.10	1.10	IX	60	1.01	1.01
	80	1.07	1.03		80	1.01	1.01
	100	1.03	1.03		100	1.01	1.01
Average		1.040	**1.033**	Average		1.008	**1.007**
	20	1.04	1.06		20	1.12	1.15
	40	1.07	1.09		40	1.06	1.06
V	60	1.06	1.09	X	60	1.06	1.06
	80	1.07	1.08		80	1.05	1.05
	100	1.07	1.08		100	1.05	1.04
Average		**1.062**	1.079	Average		**1.068**	1.072
				AVERAGE		**1.060**	**1.060**

6.7.4 Initial Investigation of MultiCrossover Genetic Algorithm

In this subsection, we report on computational results of the MXGA at different stages of development. Although many additional parameter setting tests are performed to obtain a '*good*' implementation of the MXGA, only the most significant are reported. As for the proposed MXGA for the single machine family scheduling problem described in Chapter 5, we believed that the elitism replacement and filtration strategy also improved the proposed MXGA for the 2DRSBSBPP.

As for the initial development, we consider only the first four classes of problem instances described in Table 6.1. For each class, we consider three values of n : $20, 40, 80$, and for each combination of class and n, 10 problem instances are tested. We use the problem instances provided by Lodi *et al.* [194]. In the remaining tables (Table 6.7–6.9), the results generated by the LGF placement routine are used as the benchmark for the development of the MXGA. The execution of the MXGA algorithms are halted as soon as a time limit of 120 CPU seconds per instance is reached.

We first compare different decoding strategies in the MXGA as suggested in Section 6.3.3 where the results are reported in Table 6.7. We employ 2-point multicrossover in this experiment with the *pack_extra* strategy as the decoding strategy for the temporary offspring. The choice is made based on the low time complexity of this decoding strategy compared to others. The first two columns give the class and the value of n. The next pair of columns refer to the results generated by heuristic placement routine, LGF. The following three pairs of columns refer to the results generated by the MXGA with three different decoding strategies: *pack_extra*, *pack_above* and *repack* strategies. For each algorithm, the entries in the first column report the average ratio (equation (6.10)) while the second column give the average overall bin utilisation (equation (6.11)). For each class, the final line gives the average over all values of n. The final line of Table 6.7 gives the overall average value of all classes.

We first observe that the solution quality of all three MXGAs outperform the LGF. Among the classes, Class III proved to be the most difficult to solve. As suggested, the *repack* strategy proved to be the most powerful tool among the three, followed by the *pack_above* strategy in generating good results. The *pack_extra* strategy produced the least impressive results. It is clear that the better solution quality is obtained under the *repack* strategy, although it is computationally more expensive. Consequently, the *repack* strategy is used in the decoding stage.

Table 6.7: Comparison of LGF with $MXGA_{1,2,3}$

Class	n	[a]LGF		[b]$MXGA_1$		[b]$MXGA_2$		[b]$MXGA_3$	
		Ratio	OBU	Ratio	OBU	Ratio	OBU	Ratio	OBU
I	20	1.0268	80.54	1.0268	80.54	1.0268	80.54	1.0268	80.54
	40	1.0379	84.95	1.0379	84.95	1.0379	84.95	1.0379	84.95
	80	1.0626	86.22	1.0626	86.22	1.0626	86.22	1.0626	86.22
Average		1.0424	83.90	1.0424	83.90	1.0424	83.90	1.0424	83.90
II	20	1.0000	42.40	1.0000	42.40	1.0000	42.40	1.0000	42.40
	40	1.1000	54.72	1.1000	54.72	1.1000	54.72	1.1000	54.72
	80	1.0667	78.48	1.0633	78.84	1.0567	79.26	1.0499	79.84
Average		1.0556	58.53	1.0544	58.65	1.0522	58.79	1.0500	58.99
III	20	1.0567	65.70	1.0529	66.01	1.0510	66.38	1.0499	66.77
	40	1.1266	69.26	1.1199	70.24	1.1067	71.64	1.1000	72.54
	80	1.0982	78.47	1.0967	78.86	1.0899	79.03	1.0831	79.25
Average		1.0938	71.14	1.0898	71.70	1.0825	72.35	1.0777	72.85
IV	20	1.0000	38.36	1.0000	38.36	1.0000	38.36	1.0000	38.36
	40	1.0000	56.71	1.0000	56.71	1.0000	56.71	1.0000	56.71
	80	1.1000	74.13	1.1000	74.13	1.1000	74.13	1.1000	74.13
Average		1.0333	56.40	1.0333	56.40	1.0333	56.40	1.0333	56.40
AVERAGE		1.0563	67.50	1.0550	67.67	1.0526	67.86	**1.0509**	**68.04**

[a] deterministic algorithm with execution time of less than 0.1 CPU second.
[b] stopping criterion of 120 CPU seconds per run.

Table 6.8 examines the effect on solution quality for the proposed MXGA when we incorporate the swap procedure instead of the reproduction procedure as in a SGA if the multicrossover does not apply to the selected parents. As in the previous experiment, we employ 2-point multicrossover with the *pack_extra* decoding strategy for the temporary offspring. From the results we achieved from the previous experiment, we apply the *repack* strategy during the decoding stage for the offspring in this experiment. The table gives the same information for the first two pairs of columns as in Table 6.7, while the following two pairs of columns

refer to the results generated by the MXGA for the cases of without and with the swap procedure respectively.

Our first observation from Table 6.8 is that the swap procedure yields better results in the MXGA compared to the algorithm which is without the procedure. As the values indicate, the swap procedure has a beneficial influence on the solution quality although it requires a slightly longer computation time and therefore fewer generations will be evaluated within the time limit. This indicates that, besides the mutation operator, the swap procedure also helps the algorithm to further explore the search space. Analysing the results obtained by the algorithms, we can conclude that the presence of the swap procedure in our proposed MXGA does improve the solution quality.

Table 6.8: Comparison of LGF with $MXGA_3$ (with and without swap)

Class	n	[c]LGF		[d]$MXGA_3$			
				without Swap		with Swap	
		Ratio	OBU	Ratio	OBU	Ratio	OBU
I	20	1.0268	80.54	1.0268	80.54	1.0268	80.54
	40	1.0379	84.95	1.0379	84.95	1.0379	84.95
	80	1.0626	86.22	1.0626	86.22	1.0626	86.22
Average		1.0424	83.90	1.0424	83.90	1.0424	83.90
II	20	1.0000	42.40	1.0000	42.40	1.0000	42.40
	40	1.1000	54.72	1.1000	54.72	1.1000	54.72
	80	1.0667	78.48	1.0499	79.84	1.0499	79.84
Average		1.0556	58.53	1.0500	58.99	1.0500	58.99
III	20	1.0567	65.70	1.0499	66.77	1.0379	67.67
	40	1.1266	69.26	1.1000	72.54	1.1000	71.64
	80	1.0982	78.47	1.0805	80.34	1.0769	81.03
Average		1.0938	71.14	1.0768	73.22	1.0716	73.45
IV	20	1.0000	38.36	1.0000	38.36	1.0000	38.36
	40	1.0000	56.71	1.0000	56.71	1.0000	56.71
	80	1.1000	74.13	1.1000	74.13	1.0921	74.89
Average		1.0333	56.40	1.0333	56.40	1.0307	56.65
AVERAGE		1.0563	67.50	1.0506	68.13	**1.0487**	**68.25**

[c] deterministic algorithm with execution time of less than 0.1 CPU second.
[d] stopping criterion of 120 CPU seconds per run.

In the next experiment, we investigate the impact of the mutation operator in the MXGA. It has been suggested that the mutation operator might deteriorate the solution quality by randomly assigning the item into a random bin. In order

to analyse the impact of the mutation operator on the final outcome, a MXGA has been applied without using the mutation operator. Note that the multicrossover operators used in this experiment are the 1-point and 2-point crossover strategies described in Section 6.3.4. Once again, the *pack_extra* decoding strategy is used for the temporary offspring. Table 6.9 summarises the computational results of the MXGA using the mutation operator. The table gives the same information for the first two pairs of columns as in Table 6.7, while the following four pairs of columns refer to the results generated by the MXGA for the cases of with or without the mutation operator using 1-point and 2-point multicrossover respectively.

Table 6.9: Comparison of LGF with $MXGA_3$ (with and without mutation)

Class	n	[e]LGF		[f]MXGA$_3$							
				1P + Mutation		1P - Mutation		2P + Mutation		2P - Mutation	
		Ratio	OBU	Ratio	OBU	Ratio	OBU	Ratio	OBU	Ratio	OBU
I	20	1.0268	80.54	1.0268	80.54	1.0268	80.54	1.0268	80.54	1.0268	80.54
	40	1.0379	84.95	1.0379	84.95	1.0379	84.95	1.0379	84.95	1.0379	84.95
	80	1.0626	86.22	1.0626	86.22	1.0626	86.22	1.0626	86.22	1.0626	86.22
Average		1.0424	83.90	1.0424	83.90	1.0424	83.90	1.0424	83.90	1.0424	83.90
II	20	1.0000	42.40	1.0000	42.40	1.0000	42.40	1.0000	42.40	1.0000	42.40
	40	1.1000	54.72	1.1000	54.72	1.1000	54.72	1.1000	54.72	1.1000	54.72
	80	1.0667	78.48	1.0333	81.02	1.0499	79.84	1.0333	81.02	1.0499	79.84
Average		1.0556	58.53	1.0444	59.38	1.0500	58.99	1.0444	59.38	1.0500	58.99
III	20	1.0567	65.70	1.0367	68.91	1.0499	66.89	1.0367	68.91	1.0499	66.89
	40	1.1266	69.26	1.0921	74.33	1.1000	71.64	1.0921	74.33	1.1000	71.64
	80	1.0982	78.47	1.0731	81.23	1.0835	80.12	1.0731	81.23	1.0835	80.12
Average		1.0938	71.14	1.0673	74.82	1.0778	72.88	1.0673	74.82	1.0778	72.88
IV	20	1.0000	38.36	1.0000	38.36	1.0000	38.36	1.0000	38.36	1.0000	38.36
	40	1.0000	56.71	1.0000	56.71	1.0000	56.71	1.0000	56.71	1.0000	56.71
	80	1.1000	74.13	1.0834	75.67	1.1000	74.13	1.0834	75.67	1.1000	74.13
Average		1.0333	56.40	1.0278	56.91	1.0333	56.40	1.0278	56.91	1.0333	56.40
AVERAGE		1.0563	67.50	**1.0455**	**68.76**	1.0509	68.04	**1.0455**	**68.76**	1.0509	68.04

[e] deterministic algorithm with execution time of less than 0.1 CPU second.
[f] stopping criterion of 120 CPU seconds per run.

The results achieved by the MXGA with the mutation operator in both 1-point and 2-point multicrossover operators clearly outperform the MXGA without the mutation operator. Also note that the MXGA with mutation operator in both the 1-point and 2-point multicrossover operators achieved the same solution values in every combination of class i ($i = 1, 2, 3, 4$) and value n ($n = 20, 40, 80$). Thus,

the mutation operator is used in our proposed MXGA. In the next subsections, this final version of the MXGA is compared to other local search approaches. The results of the tests that are described in the previous experiments provide guidelines for the design of the MXGA.

6.7.5 A Comparison of different Local Search Algorithms

In this subsection, we present the results of an extensive computational experiment that compares our proposed MXGA with the SGA, UTS and RDM on solving the classic 2DRSBSBPP. Since the MXGA performs equally well for both 1-point and 2-point multicrossover operators, we decide to test the performance of both with the SGA. For this experiment, problem instances provided by Lodi *et al.* [194] are used. For each class, we consider five values of n : $20, 40, 60, 80, 100$. For each combination of class and value of n, 10 problem instances are tested. For a fair comparison between different algorithms in this experiment, we employ the stopping criterion of 120 CPU seconds per instance. The specific values for the generic design variables in MXGA, SGA, UTS and RDM are summarised in Table 6.2 and Table 6.3.

The differences between the MXGA and SGA employ in this experiments are with regards to the use of the crossover operator, reproduction procedure and the replacement scheme. The SGA applies the standard crossover operator to produce two offspring from two selected parents. In the case of SGA, the steps explained in Section 6.3.4 are used only once (i.e. $t = 1$) to generate exactly two offspring. The SGA uses the reproduction procedure instead of a swap operator when the crossover does not apply to the selected parents. The replacement strategy employ in the SGA is the steady-state replacement strategy.

The results are presented in Table 6.10. The first two columns give the class and the value of n. The next two pairs of columns refer to the final version of MXGA, MXGA_F, and give the results for the 1-point and 2-point multicrossover respectively. The following two pairs of columns refer to the SGA, and give results

for the 1-point and 2-point crossover respectively. The last two pairs of columns give the results of the UTS_{LGF} and RDM respectively.

For each algorithm, the entries in the first column report the average ratio (equation (6.10)), while the entries in the second column give the average overall bin utilisation (equation (6.11)), computed over the ten generated instances. For each class, the final line gives the average over all values of n. The final line of Table 6.10 gives the overall average value over all classes. It is worth mentioning again that the placement routine used in all of the algorithms is the LGF placement routine we developed in Section 6.2.

By considering for each class, the average values computed over all values of n, we see that the MXGA_F, in both the 1-point and 2-point multicrossover, perform reasonably well compared to other algorithms except in Class IV and VI where UTS_{LGF} performs the best. In general, SGA produced the least impressive results where the SGA with the 2-point crossover operator performs slightly better (i.e. 0.1%) than the SGA with 1-point crossover operator in terms of the overall average ratio over all classes. Only a small fraction of improvement (i.e. 0.6%) is achieved in the SGA compared to the LGF heuristic routine. This may suggest that either the SGA is not an ideal choice of algorithm to be used in the bin packing problem or the LGF placement routine is itself already a powerful heuristic for the packing problem.

A closer look at the results of the UTS_{LGF} and RDM show that both algorithms also performed quite well, with UTS_{LGF} performing marginally better. This supports the idea that acceptance rule and randomisation procedure introduced in the RDM, are comparable with the ideas of tabu lists and tabu tenure used in the UTS_{LGF} in generating high quality solutions. This indicates that the randomisation procedure is capable of directing the moves to escape from local optima.

Improvements of 2.5% and 2.2% in the 1-point and 2-point MXGA_F respectively as compared to LGF placement routine show that the MXGA_F is able to produce better solution quality. A significant improvement of 2% (1-point) and

1.6% (2-point) in the $MXGA_F$ compared to SGA is obtained from the overall results. This suggests that the various techniques used in the $MXGA_F$ are capable of improving the results although less generations are generated within the time limit. The overall results show that the $MXGA_F$ algorithm is the preferred choice followed by the UTS_{LGF}, RDM and SGA.

Table 6.10: A Comparison of $MXGA_F$ with the SGA, UTS_{LGF} and RDM (120 CPU seconds per run)

Class	n	$MXGA_F$				SGA				UTS_{LGF}		RDM	
		1P		2P		1P		2P					
		Ratio	OBU	Ratio	OBU	Ratio	OBU	Ratio	OBU	Ratio	OBU	Ratio	OBU
	20	1.03	81.48	1.03	81.49	1.03	81.11	1.03	81.04	1.03	81.31	1.05	80.69
	40	1.04	85.94	1.03	86.91	1.04	85.79	1.05	84.42	1.04	85.75	1.05	84.36
I	60	1.04	88.40	1.04	88.25	1.05	86.63	1.04	87.85	1.04	88.05	1.04	87.73
	80	1.06	87.40	1.06	87.20	1.06	87.00	1.06	86.93	1.06	87.25	1.06	87.23
	100	1.02	93.43	1.03	92.71	1.03	92.26	1.03	92.59	1.03	92.45	1.03	92.67
Average		1.037	**87.33**	**1.036**	87.31	1.041	86.56	1.040	86.57	1.039	86.96	1.046	86.54
	20	1.00	42.40	1.00	42.40	1.00	42.40	1.00	42.40	1.00	42.40	1.00	42.40
	40	1.10	56.07	1.10	56.07	1.10	54.80	1.10	55.03	1.10	56.07	1.10	56.07
II	60	1.00	76.54	1.00	76.81	1.20	67.24	1.17	70.38	1.00	76.81	1.00	76.81
	80	1.00	83.51	1.03	81.07	1.07	78.00	1.07	78.00	1.00	83.45	1.03	81.34
	100	1.00	81.48	1.00	81.36	1.03	78.21	1.03	78.05	1.00	81.48	1.03	78.09
Average		**1.020**	68.00	1.027	67.54	1.080	64.13	1.073	64.77	**1.020**	**68.04**	1.032	66.94
	20	1.04	68.91	1.04	68.96	1.06	66.18	1.06	66.21	1.04	68.91	1.06	66.02
	40	1.09	74.33	1.09	74.37	1.09	73.20	1.11	71.15	1.10	72.87	1.09	74.11
III	60	1.08	81.76	1.09	80.72	1.10	77.71	1.09	78.35	1.09	80.23	1.09	80.25
	80	1.06	83.33	1.06	83.21	1.09	79.25	1.09	79.17	1.09	79.82	1.09	79.10
	100	1.06	85.27	1.06	83.98	1.08	81.10	1.09	80.39	1.06	83.34	1.07	83.01
Average		**1.065**	**78.72**	1.068	78.24	1.085	75.49	1.087	75.06	1.077	77.03	1.080	76.50
	20	1.00	38.36	1.00	38.36	1.00	38.36	1.00	38.36	1.00	38.36	1.00	38.36
	40	1.00	56.71	1.00	56.56	1.00	55.07	1.00	55.01	1.00	56.71	1.00	56.71
IV	60	1.10	71.46	1.10	71.33	1.10	70.07	1.10	69.64	1.10	71.05	1.10	71.07
	80	1.07	76.49	1.10	74.15	1.10	72.96	1.10	72.86	1.03	79.25	1.07	76.24
	100	1.03	79.30	1.03	79.02	1.07	75.75	1.07	75.37	1.03	79.15	1.03	79.15
Average		1.040	64.47	1.047	63.88	1.053	62.44	1.053	62.25	**1.033**	**64.90**	1.040	64.31
	20	1.04	70.58	1.04	70.51	1.06	68.03	1.06	68.06	1.06	68.34	1.04	70.44
	40	1.06	76.54	1.06	76.51	1.08	73.74	1.08	73.36	1.09	72.89	1.07	75.21
V	60	1.07	78.23	1.07	78.06	1.09	75.37	1.07	76.33	1.09	75.46	1.07	77.65
	80	1.07	78.96	1.07	78.93	1.08	76.35	1.08	76.40	1.08	76.63	1.07	78.85
	100	1.05	83.93	1.07	82.44	1.07	80.98	1.08	79.81	1.08	79.44	1.07	81.89
Average		**1.058**	**77.65**	1.061	77.29	1.076	74.89	1.075	74.79	1.079	74.55	1.064	76.81
	20	1.00	29.23	1.00	29.23	1.00	29.23	1.00	29.23	1.00	29.23	1.00	29.23
	40	1.40	49.09	1.40	49.13	1.40	47.40	1.40	47.42	1.30	50.12	1.40	48.25
VI	60	1.00	70.03	1.00	70.17	1.05	66.22	1.05	65.96	1.05	66.00	1.03	68.34
	80	1.00	68.67	1.00	67.86	1.00	66.66	1.00	67.00	1.00	68.67	1.00	68.67
	100	1.07	75.99	1.07	75.40	1.10	72.60	1.10	72.40	1.07	75.34	1.07	75.25
Average		1.093	**58.60**	1.093	58.36	1.110	56.42	1.110	56.40	**1.083**	57.87	1.100	57.95
	20	1.11	71.94	1.11	71.89	1.13	68.96	1.13	68.79	1.13	68.51	1.13	68.61
	40	1.07	80.43	1.07	80.27	1.09	77.16	1.12	75.28	1.08	78.97	1.08	78.89
VII	60	1.05	85.22	1.06	83.68	1.08	80.60	1.08	80.68	1.07	82.74	1.06	83.24
	80	1.08	84.79	1.09	83.45	1.10	81.23	1.09	82.22	1.10	81.14	1.10	81.33
	100	1.07	86.30	1.07	85.81	1.10	82.00	1.09	82.20	1.08	84.36	1.08	84.11
Average		**1.075**	**81.74**	1.080	81.02	1.101	77.99	1.103	77.83	1.093	79.14	1.090	79.24
	20	1.10	72.14	1.10	72.14	1.12	69.27	1.12	69.01	1.10	72.14	1.10	72.14
	40	1.09	80.23	1.09	79.84	1.11	76.55	1.09	78.41	1.13	74.95	1.11	76.25
VIII	60	1.06	84.93	1.06	84.36	1.10	79.96	1.10	79.67	1.07	83.27	1.07	83.01
	80	1.07	85.21	1.08	83.93	1.10	81.22	1.11	80.62	1.09	82.63	1.10	81.11
	100	1.06	86.55	1.07	86.10	1.09	82.31	1.09	82.79	1.08	84.74	1.08	84.56
Average		**1.078**	**81.81**	1.081	81.27	1.105	77.86	1.101	78.10	1.094	79.55	1.092	79.41
	20	1.00	43.57	1.00	43.57	1.01	43.03	1.01	43.03	1.00	43.57	1.00	43.57
	40	1.01	45.75	1.01	45.75	1.01	45.75	1.01	45.75	1.01	45.75	1.01	45.75
IX	60	1.01	43.56	1.01	43.56	1.01	43.56	1.01	43.56	1.01	43.56	1.01	43.56
	80	1.01	45.12	1.01	45.12	1.01	45.12	1.01	45.12	1.01	45.12	1.01	45.12
	100	1.01	46.10	1.01	46.10	1.01	46.10	1.01	46.10	1.01	46.10	1.01	46.10
Average		**1.007**	**44.82**	**1.007**	**44.82**	1.008	44.71	1.008	44.71	**1.007**	**44.82**	**1.007**	**44.82**
	20	1.13	68.40	1.13	68.38	1.13	67.13	1.13	67.01	1.15	66.34	1.13	67.01
	40	1.06	79.87	1.06	79.68	1.06	78.17	1.06	77.87	1.06	79.73	1.06	79.55
X	60	1.07	84.42	1.08	83.41	1.10	79.99	1.08	80.59	1.06	85.21	1.07	84.67
	80	1.06	85.83	1.06	85.51	1.07	82.66	1.06	83.17	1.05	89.14	1.05	89.00
	100	1.04	87.85	1.04	87.11	1.07	82.80	1.07	83.36	1.04	86.89	1.05	85.65
Average		**1.070**	81.27	1.072	80.82	1.086	78.15	1.080	78.40	1.072	**81.46**	1.072	81.18
Average		**1.054**	**72.44**	1.057	72.06	1.074	69.86	1.073	69.88	1.060	71.43	1.062	71.36

6.7.6 A Comparison of different Local Search Algorithms (with due dates)

In this final section of computational experiments, we compare the results of different local search algorithms on solving the 2DRSBSBPP with due dates. Both algorithms UTS_{LGF} and RDM use SEARCH and SEARCH_1 procedures during the execution.

Once again, we use the problem instances provided by Lodi *et al.* [194]. For each class, we consider five values of n : $20, 40, 60, 80, 100$, and for each combination of class and value of n, 10 problem instances are tested. As described in Section 6.7.1, we further categorise the data set into three separate groups by allocating the due dates to the rectangles. In order to have a fair comparison between different algorithms in this experiment, we employ the stopping criterion of 120 CPU seconds (2 minutes) per instance.

Recall that, we optimise the bicriteria objective function of the problem by alternating between optimising each of the objective function discussed in Section 6.4, through a *hierarchical optimisation* approach in every I iterations (in this case = 100 for UTS_{LGF} and RDM), and G generations (in this case = 100 for MXGA_F and SGA). By alternating the objective functions during the execution of the algorithms, we are solving the problem using a *simultaneous optimisation* approach. Under this approach, both objective functions are treated as equally important. As a result, a set of *Pareto optimal* solutions consisting of both objective functions is obtained, where a *trade-off curve* and an *efficient frontier* for the problem can be formed. Note that the trade-off curve and the efficient frontier are equal only if the trade-off curve is convex.

It is worth mentioning that there is no suitable way of constructing a single composite objective function to represent the bicriteria objective function of the problem. This is due to the incomparability of the unit used (i.e. time, number of bins) in both performance criteria which result in the computationally inaccessibility for optimising the single composite objective function in a direct manner.

Both MXGA$_F$ and SGA algorithms do not require any major modifications to suit the objective functions. Both algorithms are only required to include the objective functions discussed in Section 6.4 as part of the fitness evaluation procedure. As suggested from the previous results, we employ the 1-point crossover operator in both algorithms as it produced better results within a fixed computation time compared to the 2-point crossover operator.

We present only the results of the two extreme points of the efficient frontier. The computational results of the first and second objective functions are presented in Table 6.11 and Table 6.12 respectively. In both tables, the first two columns give the due date class (discussed in Section 6.7.1) and the problem class. For each algorithm, the entries in the first column report the overall average ratio computed over all values of n (in this case, $K = 50$ in equation (6.10)). The entries in the second and third columns give the average overall bin utilisation (equation (6.11) with $K = 50$) and the overall average relative percentage deviation (equation (6.13) with $K = 50$) respectively. For each due date class, the final line gives the average value over all classes. The final line of each table gives the overall average value over all due date classes. Table 6.13 gives a summary of Table 6.11 and Table 6.12.

By considering the overall average value found in each due date class in both Table 6.11 and Table 6.12, we see that the problem instances in due date class **C** to be the most challenging. The overall average ratio reported in Table 6.11 clearly shows that the trade-off begins to show effect on the solution quality when the range of the rectangle due date increased. It is clear to see that the MXGA obtained better results compared to other algorithms, although the results obtained in 'Ratio' (equation (6.10)) and 'OBU' (equation (6.11)) are worse than the results for the classic 2DRSBSBPP (refer Table 6.10). We notice that the RDM performs better than UTS$_{LGF}$ in all performance measures except in class **A**, where UTS$_{LGF}$ performs marginally better than RDM in both 'Ratio' and 'OBU'.

It is interesting to see that in Table 6.12, the computational results generated by SGA (1.065 on average) and UTS_{LGF} (1.052 on average) in terms of 'Ratio' are generally better than the results obtained from the classic 2DRSBSBPP (from Table 6.10: 1.074 [SGA] and 1.060 [UTS_{LGF}]). However, MXGA and RDM fail to improve the solution quality of the 'Ratio' when compared with their counterpart in the classic problem. In this table, RDM clearly performs better than UTS_{LGF} in terms of 'ARD'. This may indicate that UTS_{LGF} is a very effective approach in finding the minimum number of bins used, but less powerful in minimising the maximum lateness of the rectangles packed. There is no huge surprise at this outcome as the UTS_{LGF} is initially designed specifically for the classic 2DRSBSBPP where the '*only*' objective is to minimise the number of bins used.

In this case (i.e. second objective in Section 6.4), MXGA generated less impressive results compared to UTS_{LGF} in term of 'Ratio' and 'OBU'. But, the 'ARD' obtained by the MXGA clearly outperforms other algorithms.

Table 6.11: A Comparison of Different Local Search Algorithms (objective function: minimise the $L_{\max}$ with a secondary objective of minimising the number of bins used) (120 CPU seconds per run)

Due Date Class	Data Class	SGA			MXGA_F			UTS_{LGF}			RDM		
		Ratio	OBU	ARD	Ratio	OBU	ARD	Ratio	OBU	ARD	Ratio	OBU	ARD
A	I	1.056	83.10	16.58	1.042	85.26	12.37	1.053	83.42	16.02	1.088	78.73	22.27
	II	1.033	63.69	17.38	1.020	66.19	11.15	1.025	64.92	13.17	1.025	65.36	12.00
	III	1.109	71.36	30.86	1.078	75.40	22.00	1.084	74.51	27.90	1.092	73.23	26.59
	IV	1.047	60.68	21.74	1.047	61.65	17.29	1.033	62.25	19.09	1.040	61.77	18.95
	V	1.087	72.45	24.24	1.070	74.46	18.00	1.077	73.61	21.97	1.076	73.53	21.73
	VI	1.110	54.51	23.23	1.093	56.01	16.66	1.110	54.41	21.49	1.103	55.34	19.34
	VII	1.120	74.45	33.48	1.090	78.54	23.52	1.107	76.70	29.67	1.099	77.10	29.46
	VIII	1.125	74.14	33.96	1.089	78.79	23.31	1.102	77.26	29.99	1.103	76.41	29.03
	IX	1.007	44.07	1.68	1.007	44.10	1.68	1.007	42.92	1.74	1.007	43.17	2.12
	X	1.099	74.96	27.90	1.080	77.27	23.89	1.089	76.59	32.05	1.093	74.93	27.54
Average		1.079	67.34	23.10	**1.062**	**69.77**	**16.99**	1.069	68.66	21.31	1.073	67.96	20.90
B	I	1.065	81.82	34.93	1.046	84.73	24.17	1.069	81.58	31.78	1.088	78.46	38.27
	II	1.033	63.61	47.72	1.027	65.52	33.98	1.038	64.05	39.68	1.032	63.68	33.46
	III	1.132	68.91	66.78	1.088	73.90	46.21	1.128	69.99	64.99	1.107	71.50	56.46
	IV	1.060	59.27	53.45	1.047	61.70	35.98	1.063	59.58	49.09	1.060	59.22	45.72
	V	1.113	69.66	48.58	1.080	73.43	35.51	1.104	70.91	48.33	1.094	71.59	40.41
	VI	1.110	54.34	48.85	1.110	54.93	37.73	1.090	55.34	46.41	1.097	55.00	42.01
	VII	1.133	72.88	71.94	1.102	76.80	52.17	1.135	73.47	65.82	1.122	74.28	58.16
	VIII	1.143	72.19	72.72	1.099	77.38	49.41	1.122	75.08	67.28	1.118	74.27	60.49
	IX	1.007	43.84	2.42	1.007	43.97	2.42	1.007	43.09	2.53	1.007	43.30	3.79
	X	1.113	73.38	67.45	1.087	76.31	53.48	1.125	72.90	81.02	1.110	73.23	64.39
Average		1.091	65.99	51.48	**1.069**	**68.87**	**37.11**	1.088	66.60	49.69	1.084	66.45	44.32
C	I	1.085	79.30	136.69	1.054	83.50	92.98	1.083	79.76	115.41	1.104	76.50	128.02
	II	1.050	61.80	232.20	1.040	64.02	149.48	1.048	62.60	165.41	1.040	62.44	179.75
	III	1.164	65.80	180.45	1.093	73.28	124.96	1.148	68.01	173.81	1.127	69.10	148.03
	IV	1.070	58.68	223.21	1.053	60.59	153.24	1.063	60.12	210.69	1.063	59.19	183.06
	V	1.134	67.32	149.25	1.088	72.38	105.04	1.134	68.20	142.07	1.106	69.88	121.12
	VI	1.110	54.34	274.92	1.110	54.43	241.31	1.110	54.42	264.36	1.117	53.73	251.38
	VII	1.161	70.18	296.58	1.106	76.20	209.59	1.164	70.42	261.95	1.134	71.77	227.27
	VIII	1.153	70.86	421.53	1.101	76.79	273.28	1.172	69.72	387.14	1.135	72.15	320.40
	IX	1.007	43.71	9.93	1.007	43.81	9.93	1.008	43.14	15.13	1.008	43.29	18.72
	X	1.131	71.33	396.65	1.100	75.24	318.50	1.148	70.83	412.62	1.134	70.87	345.31
Average		1.107	64.33	232.14	**1.075**	**68.02**	**167.83**	1.108	64.72	214.86	1.097	64.89	192.31

Table 6.12: A Comparison of Different Local Search Algorithms (objective function: minimise the number of bins used with a secondary objective of minimising the $L_{\max}$) (120 CPU seconds per run)

Due Date Class	Data Class	**SGA**			**MXGA$_F$**			**UTS$_{LGF}$**			**RDM**		
		Ratio	OBU	ARD	Ratio	OBU	ARD	Ratio	OBU	ARD	Ratio	OBU	ARD
A	I	1.038	85.73	22.37	1.036	86.13	13.96	1.038	85.63	18.83	1.084	79.08	22.58
	II	1.033	63.69	17.38	1.020	66.19	11.15	1.020	65.31	13.20	1.020	65.73	12.19
	III	1.079	74.82	38.33	1.070	76.35	23.83	1.060	77.30	30.30	1.079	74.75	29.02
	IV	1.047	60.68	21.74	1.047	61.65	17.29	1.033	62.25	19.09	1.040	61.77	18.95
	V	1.068	74.55	27.57	1.062	75.53	20.28	1.055	76.24	26.34	1.073	73.85	22.01
	VI	1.110	54.51	23.23	1.093	56.01	16.66	1.110	54.41	21.49	1.103	55.34	19.34
	VII	1.091	77.89	39.45	1.077	80.20	28.44	1.068	81.43	34.14	1.094	77.66	30.64
	VIII	1.089	78.23	39.93	1.079	80.05	28.68	1.074	80.80	33.40	1.096	77.15	29.67
	IX	1.007	44.07	1.68	1.007	44.10	1.68	1.007	42.91	1.74	1.007	43.17	2.12
	X	1.079	77.22	30.07	1.073	78.28	26.56	1.064	79.73	35.09	1.083	75.93	28.52
Average		1.064	69.14	26.18	1.056	70.45	**18.85**	**1.053**	**70.60**	23.36	1.068	68.44	21.51
B	I	1.041	85.19	46.29	1.036	86.24	29.00	1.035	85.95	39.96	1.081	79.42	42.80
	II	1.033	63.61	47.72	1.020	66.02	34.34	1.020	65.52	40.23	1.020	64.75	33.68
	III	1.078	74.88	89.86	1.071	75.99	55.05	1.064	77.05	76.45	1.076	74.90	68.20
	IV	1.053	59.72	53.68	1.047	61.70	35.98	1.045	61.16	49.59	1.047	60.44	45.85
	V	1.068	74.44	63.55	1.062	75.60	40.84	1.057	76.03	56.22	1.075	73.56	48.26
	VI	1.110	54.34	48.85	1.103	55.40	37.80	1.083	55.83	46.48	1.090	55.43	42.10
	VII	1.090	77.85	91.03	1.081	79.34	66.72	1.067	81.24	82.04	1.094	77.33	65.92
	VIII	1.089	78.19	89.91	1.081	79.56	59.48	1.069	81.35	82.18	1.097	76.58	68.15
	IX	1.007	43.84	2.42	1.007	43.97	2.42	1.007	43.09	2.53	1.007	43.30	3.79
	X	1.079	77.28	75.08	1.074	78.02	60.04	1.073	78.42	92.67	1.080	76.41	73.31
Average		1.065	68.93	60.84	1.058	70.18	**42.17**	**1.052**	**70.56**	56.83	1.067	68.21	49.21
C	I	1.041	85.17	189.73	1.036	85.96	124.60	1.037	85.67	149.20	1.089	78.17	138.09
	II	1.037	62.99	234.99	1.020	65.65	152.94	1.020	64.95	168.62	1.020	64.21	181.97
	III	1.076	75.08	253.78	1.072	75.69	147.83	1.054	78.04	227.23	1.081	73.73	182.49
	IV	1.053	59.99	225.52	1.047	61.13	154.41	1.040	61.79	213.44	1.047	60.49	186.81
	V	1.072	73.99	200.62	1.062	75.48	145.06	1.056	76.14	184.62	1.078	72.59	146.44
	VI	1.110	54.34	274.92	1.110	54.43	241.31	1.103	55.15	264.76	1.103	54.78	252.29
	VII	1.088	78.04	386.60	1.079	79.43	263.25	1.067	81.28	332.09	1.107	74.73	267.94
	VIII	1.091	77.85	509.33	1.081	79.39	328.74	1.069	81.03	467.79	1.099	75.80	450.80
	IX	1.007	43.71	9.93	1.007	43.81	9.93	1.007	43.23	16.61	1.008	43.31	21.83
	X	1.074	77.73	497.09	1.071	78.29	418.83	1.066	78.99	487.26	1.085	76.24	441.32
Average		1.065	68.89	278.25	1.058	69.93	**198.69**	**1.052**	**70.63**	251.16	1.072	67.41	227.00

Table 6.13: Comparative Computational Results (120 CPU seconds per run)

Due Date Class	**SGA**			**MXGA$_F$**			**UTS$_{LGF}$**			**RDM**		
Lmax / Bin	Ratio	OBU	ARD	Ratio	OBU	ARD	Ratio	OBU	ARD	Ratio	OBU	ARD
A	1.079	67.34	23.10	1.062	69.77	16.99	1.069	68.66	21.31	1.073	67.96	20.90
B	1.091	65.99	51.48	1.069	68.87	37.11	1.088	66.60	49.69	1.084	66.45	44.32
C	1.107	64.33	232.14	1.075	68.02	167.83	1.108	64.72	214.86	1.097	64.89	192.31
AVERAGE	1.092	65.89	102.24	**1.069**	**68.89**	**73.98**	1.088	66.66	95.29	1.085	66.43	85.84
Bin / Lmax	Ratio	OBU	ARD	Ratio	OBU	ARD	Ratio	OBU	ARD	Ratio	OBU	ARD
A	1.064	69.14	26.18	1.056	70.45	18.85	1.053	70.60	23.36	1.068	68.44	21.51
B	1.065	68.93	60.84	1.058	70.18	42.17	1.052	70.56	56.83	1.067	68.21	49.21
C	1.065	68.89	278.25	1.058	69.93	198.69	1.052	70.63	251.16	1.072	67.41	227.00
AVERAGE	1.065	68.99	121.76	1.057	70.19	**86.57**	**1.052**	**70.60**	110.45	1.069	68.02	99.24

6.8 Conclusions and Remarks

In this chapter, a non-oriented two-dimensional rectangular single bin size bin packing problem with the objective of minimising the number of bins used is defined. We have developed a heuristic placement routine called Lowest Gap Fill (LGF) that is effective in filling the available gaps in the partial layout by dynamically selecting the best rectangle for placement during the packing stage. The routine requires $O(n^2)$ time. We compare the placement routine with some well known heuristics reported in the literature. Computational results shown that our proposed placement routine is capable of producing high quality solutions.

A MultiCrossover Genetic Algorithm (MXGA) has been proposed to solve the non-oriented 2DRSBSBPP in this chapter. Various techniques have been introduced into the MXGA to further enhance the solutions. We compared the MXGA with the well known Unified Tabu Search (UTS) proposed by Lodi *et al.* [194]. Extensive computational experiments show that the MXGA achieves better results compared to a standard genetic algorithm, UTS and randomised descent method.

We also introduced a new variant of the 2DRSBSBPP where each rectangle has a due date and there is a fixed processing time for the bins used. The objective of this problem variant is to minimise the maximum lateness of the rectangles and minimising the number of bins used. A lower bound to the problem is also proposed in this chapter. Extensive computational experiments have been carried out to solve the 2DRSBSBPP with due dates. All the local search algorithms previously designed have been modified to suit the problem. Comparative computational results shown that our proposed MXGA achieved a mixed degree of success compared to UTS.

The applications of MXGA and LGF for other cutting and packing problems such as open dimension problem and stock cutting problem are worthy of future research.

Chapter 7

Symmetric Travelling Salesman Problem with Due Dates

7.1 Introduction

In this chapter, we study a new variant of the symmetric version of the Time Constrained Travelling Salesman Problem (TCTSP), called the Travelling Salesman Problem with Due Dates (TSPDD). Other TCTSP in literature are TSP with Deadlines (TSPD), TSP with Target Times (TSPTT) and TSP with Time Windows (TSPTW). The TSPDD has important practical applications in bank or postal deliveries and scheduling deliveries.

The TSPDD can be defined as follows:
"Given a set $\{1, 2, \ldots, n\}$ of *cities*, there exist a *distance* (or *cost*) c_{ij}, and a *travel time* t_{ij}, for each pair $i, j \in n$ of distinct cities. Assume that city 1 is a depot and the tour must visit every city exactly once, starting and ending at the depot. For each city i (except city 1), there is a *due date* d_i. This problem is best treated as a bicriteria optimisation problem where the objective is to find an ordering of the cities that starts and ends at the depot which minimises the *maximum lateness* $L_{\max}$, and the total *tour length* of the cities."

The main motivation of this study is derived from the trade-off between the customers' (to be served in the cities) satisfaction and the routing (travelling cost/time) efficiency of the salesman.

In this study, we propose a MultiCrossover Genetic Algorithm (MXGA) that utilises the multicrossover operator to solve the TSPDD. We introduce a new variant of a subtour based crossover, where the constraint on sharing the common subtours in both parents is relaxed. Detailed descriptions of the operator are given in Section 7.5.2. Once again, the architecture of the MXGA uses in this chapter is based on the framework design discussed in Section 4.9. Various techniques will be introduced to further enhance the solution quality.

In the next section, we concentrate on the study of the TCTSP from literature. To the best of our knowledge, there is no literature on the TSPDD. Hence, we focus on problems which are closely related to TSPDD, namely TSPD, TSPTT and TSPTW. Extensive search on the literature shows that both TSPD and TSPTT receive little attention from the research community. Literature on the TSPTW will be the focus in Section 7.2.

TSPTW consists of finding the minimum-cost tour of a set of cities where each city is visited exactly once. To be feasible, the tour must start and end at an unique depot within a certain time window and each city must be visited within its own time window. According to Balas and Simonetti [23], the problem can be defined more formally as follows:
"Given a set $\{1, 2, \ldots, n\}$ of *cities*, there exist a *distance/cost* c_{ij}, and a *travel time* t_{ij}, for each pair $i, j \in n$ of distinct cities. For each city i, there is a *time window* $[a_i, b_i]$ where a_i and b_i are the earliest and latest bound of the time window. The time window indicates that city i has to be visited not earlier than a_i and not later than b_i. Early arrival is allowed, where there exists a *waiting time* w_i until a_i. The TSPTW is:
– *hard* : if the late arrival is not allowed, or
– *soft* : if the late arrival is allowed by adding a penalty to the objective function.

The objective is to find a minimum cost tour, where the cost of a tour may be the total distance travelled (in which case the waiting time w_i, is ignored) or the total time it takes to complete the tour (in which case the waiting time w_i is added to the travel time t_{ij})."

The problem is NP-hard and Savelsbergh [246] has shown that even finding a feasible solution to the TSPTW is an NP-complete problem. TSPTW can be viewed as a subproblem of the Vehicle Routing Problem with Time Window (VRPTW).

Section 7.3 addresses the TSPDD in more detail and a new lower bound of the maximum lateness for the TSPDD is then proposed in Section 7.4. The developments of the MXGA for solving the TSPDD are the focus of Section 7.5. Some of the main components in the MXGA are discussed in detail. Section 7.6 provides an insight into the local search algorithms designed specifically for comparison purposes with the proposed MXGA. Computational experiments are conducted in Section 7.7 to assess the merit of the proposed algorithms. To end this chapter, we give some concluding remarks in Section 7.8.

7.2 Time Constrained Travelling Salesman Problem

Bansal *et al.* [24] consider the TSPD which they define as a problem of finding a tour in a set of n cities, starting at a city r (i.e. depot), that visits as many cities as possible by their deadlines. This problem has a practical application on the point-to-point orienteering problem and machine scheduling problem with sequence dependent setup times. The authors give an $O(\log n)$ approximation algorithm for the problem. They also extend their study in VRPTW and give an $O(\log^2 n)$ approximation to the problem. No computational results are reported in their study.

Campbell and Thomas [43] address the stochastic version of the TSPD where each city i to be visited is based on a given probability p_i. They present three different models to represent three different ways in which deadline violations can be measured and addressed in a stochastic environment. No computational results are reported in their study.

Balas and Simonetti [23] introduce the TSPTT which is applicable to Just-In-Time scheduling problems. This model defines a target time for each city, rather than a time window. The objective is to minimise the maximum deviation between the target time and the actual service time over all cities. A secondary objective could be the minimisation of the total time needed to complete the tour. They propose a Dynamic Programme (DP) which is initially used in their research to solve the TSPTW. Computational results for problem instances from the literature for up to 46 cities, where the target times are defined as the time window mid-points, show that the optimal solution is found in most cases within a reasonable time. To the best of our knowledge, no further research has been done on this problem variant.

The first approaches for the TSPTW can be attributed to Christofides *et al.* [53] and Baker [17]. Both papers present a Branch and Bound (B&B) approach. Christofides *et al.* [53] describe the B&B approach in which the lower bound computation is performed via a state-space relaxation in a DP scheme. Solutions of problem instances of up to 50 cities with 'moderately tight' time windows are reported. Baker [17] exploits a time constrained critical path formulation in the lower bound computation. The algorithm performs well on problems of up to 50 cities when only a small percentage of the time windows overlap.

Langevin *et al.* [175] address the problem using a two-commodity flow formulation within a B&B scheme. Computational results for problems of up to 60 cities are reported. More recently, Ascheuer *et al.* [14] consider several formulations for the asymmetric version of the problem and compares them within a branch and cut scheme. The framework incorporates techniques tailored for the asymmetric TSPTW such as data preprocessing, primal heuristics, local search, and variable

fixing. Their algorithm solves instances in the range of 50–70 cities to optimality. They also tested their algorithm on real-world problem instances with sizes of up to 250 cities.

Dumas *et al.* [80] propose a DP approach for the TSPTW that extensively exploits elimination tests to reduce the state space. They report solving instances with up to 200 cities with 'fairly wide' time windows. Mingozzi *et al.* [211] propose a DP derived by through a generalisation of the state-space relaxation scheme developed by Christofides *et al.* [53]. Their proposed algorithm can be also applied to TSPTW problems with precedence constraints. They present computational results for instances of up to 120 cities.

More recently, Balas and Simonetti [23] present a new DP algorithm that can be applied to a wide class of restricted TSP. This approach yields good results on the asymmetric version of TSPTW in cases where the number of overlapping time windows are small.

Recently, TSPTW has drawn interest from the Constraint Logic Programming (CLP) community. Pesant *et al.* [227] propose a CLP which incorporates arc elimination and time window reduction rules previously proposed by Langevin *et al.* [175] and Desrochers *et al.* [71] respectively. A year later, Pesant *et al.* [228] show the flexibility of the CLP by solving a new variant of the TSPTW, called TSPTW with Multiple Time Windows, using the same algorithm as for the original problem with some minor modifications.

Focacci *et al.* [95] view the TSPTW as a model combining TSP and a scheduling problem. They use a set of propagation techniques based on feasibility reasoning and exploiting information on costs. In addition, they also propose two branching strategies that can be used to solve the problem. Computational results of problem instances from the literature show that the algorithm is a viable choice for solving TSPTW. Moreover, they use the TSPTW as a case study and present a series of papers on CLP based on the general framework presented in [95] (see [96, 97, 98]).

Because of the limitations associated with exact formulation, there exists a body of research focusing on heuristic techniques for the TSPTW. Carlton and Barnes [44] solve the TSPTW with a reactive tabu search approach that permits infeasible solutions in its search neighbourhood through the implementation of a static penalty function. Computational results on 145 problem instances of up to 200 cities from the literature show that optimal or near-optimal solutions can be obtained within reasonable computation times.

Gendreau *et al.* [110] present an algorithm based on the GENIUS procedure (as discussed in Section 3.5.1.3). The computational results for problem instances of up to 100 cities from the literature, indicate that optimal or near- optimal solutions can be obtained in most cases within reasonable computation times.

Calvo [42] presents a three-stage procedures to solve the TSPTW. It starts by solving an assignment problem with a particular objective function. This objective function takes into account the scheduling constraints, which are relaxed to obtain a pure assignment problem from the original formulation. By doing this, a solution close to feasible is obtained where there is a set of subtours and a long main tour starting from and ending at the depot. An insertion heuristic is then applied to insert the subtours into the main tour to obtain a good initial feasible solution for the problem. The solution is further improved by a local search scheme based on a k-Opt exchange procedure. Computational results on 395 benchmark problems indicate that optimal or near-optimal solutions can be obtained in most cases within reasonable times.

Ohlmann and Thomas [219] apply a variant of simulated annealing, called compressed annealing, which incorporates a variable penalty method with stochastic search to solve the TSPTW. The performance of the compressed annealing algorithm is tested on 400 problem instances from the literature. The authors demonstrate that compressed annealing consistently converges to good solutions, and exhibits potential for particularly large instances with *wide* time windows. They obtain new best known solution for a number of instances and match the previously best known solution in most of the benchmark problem instances.

Nygard and Yang [218] propose a Genetic Algorithm (GA) to solve the TSPTW. They develop a new crossover operator, called the Earliest Closing Time Crossover operator. Each offspring is carefully constructed by selecting the next city to be visited, based on the latest service time of the next potential city of both parents. The GA consistently provides good quality solutions for the problem instances from literature.

A year later, Yang and Nygard [285] solve the TSPTW using a GA by introducing a new class of crossover operator, called edge-type crossover, and a heuristically selected initial population. The initial population of the tour is constructed by first ordering the cities with time windows according to their latest service time to form a partial tour. Then, the cities with no time windows are inserted in between the partial tour to form a complete route. The edge-type crossover consists of a class of five different operators (i.e. shorter edge, longer edge, most cities, randomly combined, and nearest cities), which are modifications of the greedy crossover operator of Grenfenstette *et al.*[131]. Computational results of the problem instances from literature demonstrate that the proposed crossover operators are effective. Moreover, the use of a heuristic method to generate the initial population greatly reduces the computation time.

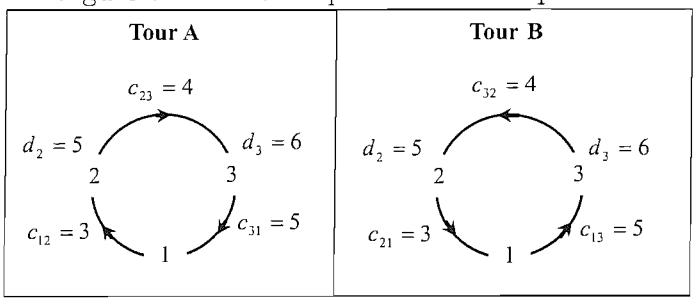
7.3 Travelling Salesman Problem with Due Dates

In this section, we introduce a new variant of the TCTSP, called Travelling Salesman Problem with Due Dates (TSPDD). The formal definition of the TSPDD is given in Section 7.1.

We concentrate our study on the symmetric version (i.e. $c_{ij} = c_{ji}$) of the problem. For simplicity, we also assume that $c_{ij} = t_{ij}$ for $i, j \in n$. It is worth mentioning that, although the total tour length of the cities is symmetric, the maximum lateness of the cities is not. We can demonstrate this by using the example in Figure 7.1 with a 3 cities ($i = 1, 2, 3$) problem. Note that the formulation and calculation of the maximum lateness of the cities will be given later in the section.

Let city 1 be the depot, city 2 and city 3 have a due date of 5 units and 6 units respectively. Suppose that $\{c_{12}, c_{23}, c_{31}\} = \{3, 4, 5\}$ and $c_{ij} = c_{ji}$ for each pair $i, j \in 3$ of distinct cities. Thus, for Tour A: (1 2 3), the total tour length is 12 units and the maximum lateness is 1 unit. On the other hand, Tour B: (1 3 2) gives the same tour length as Tour A but with the maximum lateness of 4 units. This result shows that the total tour length is symmetric but the maximum lateness is not.

Figure 7.1: An example of a 3 cities problem



The main motivation of this study is derived from the trade-off between the customers' (to be served in the cities) satisfaction and the routing (travelling cost) efficiency of the salesman. On one hand, if the routing efficiency is more important, the salesman tends to visit customers via the shortest route in between cities to reduce the travelling cost (e.g. time and fuel). On the other hand, to meet the customer due date (e.g. bank closing time), the original route has to be reassigned so that priorities can be shifted. This allows customers (i.e. cities) that have the shortest due date to be visited ahead of the other cities.

In order to deal with the problem, we first define two distinct objective functions for the problem to be solved:

1. minimise the maximum lateness of the cities to be visited with a secondary objective of minimising the total tour length (ideal for customers' satisfaction);
2. minimise the total tour length with a secondary objective of minimising the maximum lateness of the cities to be visited (ideal for salesman's routing efficiency).

Each of the objective function can be viewed as a *hierarchical optimisation* approach. This approach first optimises the primary objective, then the secondary

objective is optimised subject to the additional constraint that the solution value of the primary objective is optimum. By alternating the objective functions in every G generations (or I iterations) during the execution of the local search algorithms, we will be able to find a good balance of the trade-off between the customers' satisfaction and salesman's routing efficiency.

The maximum lateness $L_{\max}$, of the cities to be visited in the tour is calculated as follows: let $\pi(1), \pi(2), \ldots, \pi(n)$, where $\pi(1) =$ city 1, be the ordering of the cities to be visited in the tour and $d_{\pi(j)}$ is the due date of the jth city in the tour. We denote $t_{\pi(j)\pi(j+1)}$ as the travel time from jth city to $(j+1)$th city in the tour. The lateness of the jth city in the tour is obtained by the following:

$$L_{\pi(j)} = \sum_{i=1}^{j-1} t_{\pi(i)\pi(i+1)} - d_{\pi(j)}. \tag{7.1}$$

Thus, the maximum lateness of the cities to be visited in the tour is

$$L_{\max} = \max_{j=2,3\ldots,n} \{L_{\pi(j)}\}. \tag{7.2}$$

7.4 Lower Bound for TSPDD

In this section, we derive a simple lower bound of the maximum lateness for the TSPDD where the distance between the cities satisfy the '*triangle inequality*'. The Concorde software [57] which is developed by the research team lead by Applegate is used to generate the optimal tour length LB^n of the n cities.

We first sort cities $2, \ldots, n$ in EDD order (i.e. non-decreasing order of their due dates) and let city 1 be the depot at which the tour must start and end. By employing the Concorde software [57], we obtain the optimal tour length LB^n, of the problem. Since we assumed that $c_{ij} = t_{ij}$, then, LB^n is also the total time travelled by the cities in the optimal tour. The lower bound on the maximum lateness of the tour of n cities is the minimum value of

$$LB^n - t_{j1} - d_j; \text{ for } j = 2, 3, \ldots, n. \tag{7.3}$$

The logic is: one of the cities $2, 3, \ldots, n$, must be the last city in the tour. Suppose it is city j. Then travel time t_{j1} is included in the tour, and so a lower bound on the time that we can visit city j is $LB^n - t_{j1}$. We then subtract the due date of the city j to get the lower bound on the maximum lateness, $L^n = LB^n - t_{j1} - d_j$. As we do not know which city is last in the tour, we look at all possibilities and choose the smallest.

The steps of finding the optimal tour length and the lower bound on the maximum lateness of the tour are repeated by first removing the city with the largest due date from the list. The process is repeated until the list contains only the depot. More formally, the process can be summarised as follows:

S 1: Sort cities $2, \ldots, n$ in a non-decreasing order of their associated city due date d_i, so that $d_2 \leq d_3 \leq \ldots \leq d_n$ and let city 1 be the depot. Set $j = n$.

S 2: Find the optimal tour length LB^j, of j cities using the Concorde software.

S 3: Compute a lower bound on the maximum lateness of the j cities, $L^j = \min_{i=2,3,\ldots,j} \{LB^j - t_{i1} - d_i\}$.

S 4: Reduce j by 1 (i.e. $j := j - 1$), and repeat **S 2** - **S 3** until $j = 1$.

Thus, the lower bound of the maximum lateness for TSPDD is

$$LB_{L_{\max}} = \max_{j=2,3,\ldots,n} \{L^j\}. \tag{7.4}$$

To justify that the above procedure generates a valid lower bound, consider an optimal tour of n cities. Suppose that we delete cities $j+1, \ldots, n$ from this tour. From the triangle inequality, no city is visited later as a result of these deletion. Let T be the tour length, and h be the last city visited. Then

$$\begin{aligned} LB_{L_{\max}} &\geq T - t_{h1} - d_h \\ &\geq LB^j - t_{h1} - d_h \\ &\geq \min_{i=2,3,\ldots,j} \{LB^j - t_{i1} - d_i\} \end{aligned} \tag{7.5}$$

as defined in **S3**.

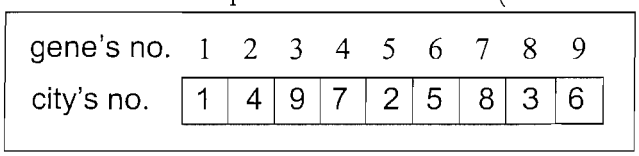
7.5 MultiCrossover Genetic Algorithm

In this section, we proposed a MultiCrossover Genetic Algorithm (MXGA) for solving the TSPDD. In the remaining subsections, we will discuss some of the main components of our proposed MXGA. The general architecture of the MXGA is shown in Section 4.9.

7.5.1 Representation

The proposed MXGA is developed using path representation (as discussed in Section 4.2). The complete set of cities n, forms the length of the chromosome (individual). In this representation, the n cities to be visited are sequenced in order according to a string of n genes, so that if city i is the jth gene of the string, city i is the jth city to be visited. Without loss of generality, the first gene in each chromosome is city 1 and indicates the start of the tour as a depot. Also note that, every city appears only once in the chromosome. Figure 7.2 shows an example of the representation of an individual with 9 cities to be visited. Hence, the tour starts at city 1, the depot, followed by city 4, city 9, city 7, etc. until it reaches city 6 before it returns to the depot.

Figure 7.2: An example of an individual (chromosome)



7.5.2 MultiCrossover

The proposed multicrossover operator is a variant of the subtour based crossover. The offspring generated from this operator inherit the subtours from their parents. Unlike the other subtour based crossover operators such as Subtour Exchange Crossover (SXX) [284], Complete Subtour Exchange Crossover (CSEX) [167] and

Subtour Preservation Crossover (SPX) [257], where the parents have to share the common subtours in order for the operator to be effective, our proposed crossover strategy relaxes this constraint.

This subtour based crossover strategy first randomly selects a subtour of size s from a parent P. Then, an arbitrary city is chosen within the subtour which indicates the position of the city in another parent Q where the subtour will be inserted. Any repeated city from parent Q is removed to form a valid temporary offspring. Similarly, we can generate the second temporary offspring by inserting a randomly selected subtour from parent Q to parent P. The size of the subtour is randomly selected within a bound of $[2, \alpha n]$, where n is the number of cities to be visited. Initial experiments suggest the value of α as 0.2. In other words, the size s of the subtour is between the range of 2 cities to 20% of the total cities to be visited in the tour. The process of generating two temporary offspring from a pair of parents is as follows (an example is given in Figure 7.3):

S 1: Select randomly a subtour of size s in both P1 and P2 (Parent 1 & 2).

S 2: Select randomly a city in the subtour of P2 as the identifier I.

S 3: Locate the position R of I in P1.

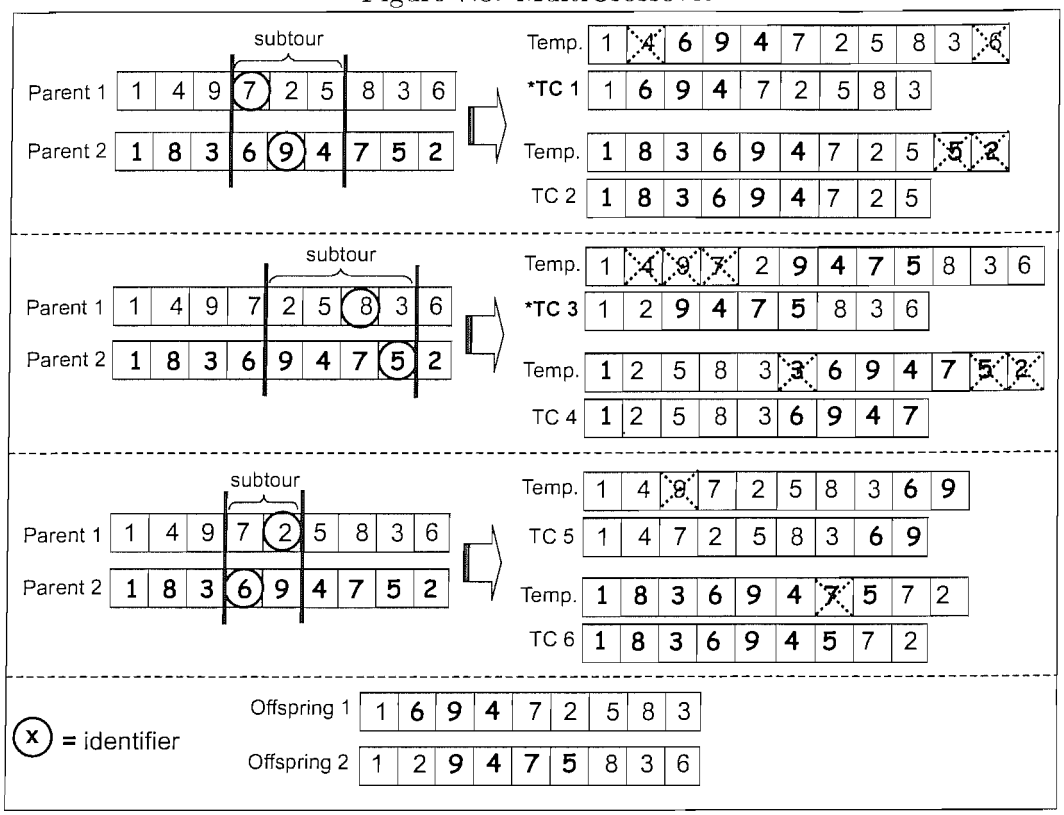
S 4: Insert the subtour of P2 into P1 at R.

S 5: Remove the duplicate cities from P1 by preserving the subtour of P2. The completed tour is denoted as TC1 (Temporary Offspring 1).

S 6: Repeat **S2** – **S5** to generate TC2 (Temporary Offspring 2) by inserting subtour from P1 into P2 analogously.

By repeating the steps above for r times, we produce a candidate list of $2r$ temporary offspring. The best and a selected temporary offspring found using the probabilistic binary tournament selection mechanism are then chosen to be the new offspring for the current generation. It is more convenient to describe the process using the example in Figure 7.3 with $r = 3$.

Figure 7.3: MultiCrossover



In Figure 7.3, we have two parent tours represented by Parent1 (P1) and Parent2 (P2) respectively as:

$$P1 : (1\ 4\ 9\ 7\ 2\ 5\ 8\ 3\ 6) \text{ and } P2 : (1\ 8\ 3\ 6\ 9\ 4\ 7\ 5\ 2).$$

Suppose that the cities between the 4th and 6th gene are selected as the subtour in both P1 (7 2 5) and P2 (6 9 4), and we randomly selected city 9 in P2 as the identifier I. Hence, the position R of I in P1 is at the 3rd gene. By inserting the subtour from P2 into P1 at the 3rd position, we get

$$\text{Temp.} : (1\ 4\ \mathbf{6}\ \mathbf{9}\ \mathbf{4}\ 7\ 2\ 5\ 8\ 3\ 6).$$

Note that the resulting chromosome is infeasible as city 4 and city 6 appear twice (i.e. visited twice in the tour). To deal with this infeasibility, city 4 and city 6 that previously appear in P1 are removed from the resulting chromosome. The aim is to always preserve the cities in the subtour from the other parent. The feasible chromosome is the new temporary offspring (TC1):

$$\text{TC1} : (1\ \ 6\ \ 9\ \ 4\ \ 7\ \ 2\ \ 5\ \ 8\ \ 3).$$

By repeating the steps above (i.e. subtour of P1 (7 2 5), I = city 7 in P1, and $R = 7$ in P2), we have the second temporary offspring (TC2):

$$\text{TC2} : (1\ \ 8\ \ 3\ \ 6\ \ 9\ \ 4\ \ 7\ \ 2\ \ 5).$$

The whole process of generating two temporary offspring is repeated for r (in this case $r = 3$) times to generate a candidate list of $2r$ temporary offspring. Then, the best and a selected temporary offspring (in this case are TC1 and TC3) found using the probabilistic binary tournament selection mechanism will be chosen to be the new offspring for the current generation.

The advantage of randomly choosing a city within the subtour as the identifier helps the crossover strategy to generate different offspring if the same parents and subtour are selected again later in the process. For instance, suppose that the two parents from Figure 7.3 are selected again for crossover, with a different identifier selected within the same subtour (i.e. (6 9)) in Parent 2, we produce different offspring:

– city 6 from Parent 2 as the identifier: Offspring: (1 4 7 2 5 8 3 **6** **9**);

– city 9 from Parent 2 as the identifier: Offspring: (1 4 **6** **9** 7 2 5 8 3).

7.5.3 Swap

The *swap* operator used in the MXGA is based on the idea of a 3-Opt move as described in Section 3.5.1.2. This operator is used to produce new offspring and introduce more diversity into the population when the multicrossover operator does not apply. This can be achieved by doing the following steps:

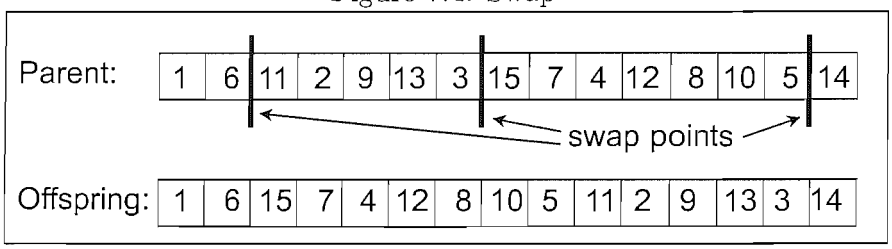
S 1: Select randomly 3 swap points in a parent.

S 2: Form 4 subtours which are separated by the swap points.

S 3: Swap the position of the 2nd and 3rd subtours to form a new offspring.

The steps above are repeated for the second parent to create a second offspring. Figure 7.4 shows the resulting offspring after the swap operator has been applied to a parent.

Figure 7.4: Swap



Note that we do not reverse the sequence of the genes within the subtours. It is considered that reversing the sequence of genes might produce deteriorating results when the city with the shorter due date is visited later in the tour.

7.5.4 Mutation

We use the Displacement Mutation (DM) operator proposed by Michalewicz [210] (as described in Section 4.7) to mutate the offspring. First of all, a subset of individuals from the new offspring population is selected with a given *individual mutation* probability p_M, where an offspring is selected for mutation if the random number m, assigned to the offspring is less or equal to the *individual mutation* probability (i.e. $m \leq p_M$). Then the selected offspring will go through the DM. The steps of the DM can be summarised as follows:

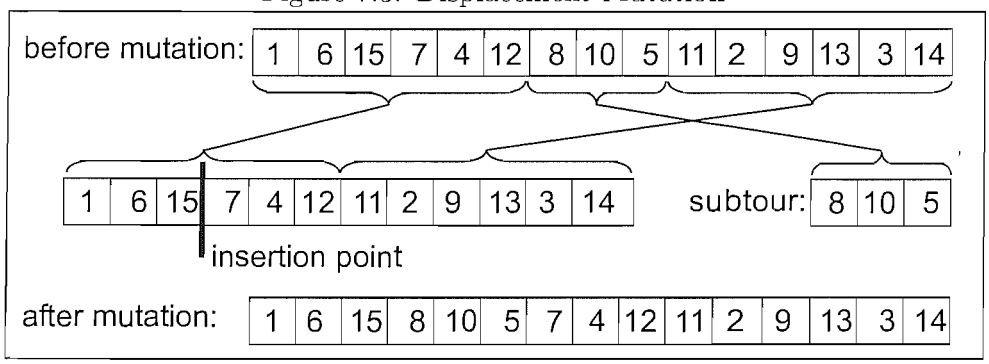
S 1: Select a subtour at random from the offspring.

S 2: Remove the subtour from the offspring.

S 3: Reinsert the subtour into the offspring at a randomly selected point.

For example, consider the offspring in Figure 7.5 and suppose that the subtour $(8 - 10 - 5)$ is selected. Let say we randomly select the 3rd gene to be the gene after which the subtour is inserted. This results in a new offspring as in Figure 7.5.

Figure 7.5: Displacement Mutation



7.6 Competitors - Performance Measure

7.6.1 Dynamic Length Tabu Search

In this subsection, we briefly describe the Tabu Search (TS) used in the computational experiments. We use the TS developed by Knox [170]. Several modifications have been considered to further enhance the efficiency of the algorithm. For instance, Knox uses the 2-Opt (2-edge exchange) procedure as the inner heuristic and a fixed tabu list length of $3n$ for the TS, where n is the number of cities to be visited. On the other hand, we employ the 3-Opt (3-edge exchange) procedure as the inner heuristic and a dynamic length of the tabu list for the algorithm.

The length of the tabu list is dynamically controlled during implementation in order to achieve better solution quality. Such processes can have an important influence on which moves are available to be selected at a given iteration. We observe that a short tabu list is needed at the beginning of the run to allow the search to fully *exploit* the neighbourhood. A longer tabu list is needed at the later stage of the run to allow the search to *explore* the other region of the search space.

Since applying the 3-Opt procedure would result in a huge search space, we use a *neighbour list* for each city to reduce the search space and improve the speed of the algorithm while maintaining the solution quality. The *neighbour list* implementation was proposed by Johnson and McGeoch [161]. The details of our proposed Dynamic Length Tabu Search (DLTS) algorithm can be described as follows.

Before the start of the algorithm, a *neighbour list* for each city i ($i = 1, 2, 3, \ldots, n$), which contains k (where k is a parameter, typically $k = 20$) nearest neighbours in a non-decreasing order of their distance/cost c_{ij}, for $i \neq j$; $j = 1, 2, 3, \ldots, n$ is generated. Two tabu lists (one for each of the bicriteria objective functions) with size of $\lceil n/8 \rceil$ are initialised (which are empty at the beginning of the search). However, throughout the implementation, the size of the tabu list is dynamically controlled within the range of $\lceil n/8 \rceil$ to $\lceil n/6 \rceil$ via the following systematic proce-

dure. Start with a tabu list length of $\lceil n/8 \rceil$. Increase the length by 2 after 100 non-improving moves. Decrease the length by 2 once an improving move is found.

The idea of a dynamic length tabu list is inspired by Tsubakitani and Evans [267] in their study to find an optimum size of the tabu list for STSP. We quote the following statement from their paper:

> *"If the tabu list is too short, the TS may keep returning to the same local optimum. This prevents the search process from exploring a wide area of the solution space. In contrast, if the tabu list is too long, it results in excessive computational time to search the tabu list to determine if a move is tabu. Thus, a longer time spent going through a tabu list provides less time for the procedure to explore in the solution space for a given computational time ..... Therefore, the tabu list size should be as small as possible but long enough to allow the search to move away from the local optimum."*

Based on their computational experiments, they recommend that the tabu list size is in the range of $\lceil n/8 \rceil$ to $\lceil n/6 \rceil$ for TS using 3-Opt moves.

The search begins with a randomly constructed initial tour except city 1, which is set to be the depot. At each iteration, a series of candidate 3-edge exchanges is evaluated and the *best* exchange candidate e_{best}, is identified and accepted as the new tour. An exchange candidate e, is one which is either not tabu or is able to override the tabu status by producing a new tour whose total tour length or maximum lateness (depending on the objective functions) is lower than the aspiration criterion value.

It is worth mentioning that, this implementation does not perform an exhaustive evaluation of all 3-edge exchange candidates on each iteration of the search. Instead, the *neighbour list* described earlier is used during the edge exchange procedure. Furthermore, the candidates for city $i+1$ are also subject to the due date constraint where the due date of city $i+1$ has to be greater than or equal to that of

city i. The logic is that travelling from city i to city $i+1$ with the shortest distance helps to reduce the total tour length and visiting cities in a non-decreasing order of their due date helps to reduce the maximum lateness.

We employ the same updating procedures suggested by Knox [170] for the tabu list and aspiration criterion. Note that Knox [170] employs the 2-edge exchange procedure as the inner heuristic. Thus, only two dropped edges are considered during updating procedures. In DLTS, the information recorded on the tabu lists consists of the three dropped edges of a 3-edge exchange candidate. The added edges are not recorded on the tabu lists. In other words, updating a tabu list involves placing the deleted edges of the 3-edge exchange on the list. If the list is full, the oldest elements of the tabu list are replaced by the new deleted edges information. An exchange candidate is classified as tabu only if all three added edges of the exchange are on the tabu lists. If one or more added edges are not on the tabu lists, then the candidate move is not classified as tabu. The value recorded on the aspiration criterion is the total tour length or the maximum lateness (depending on the objective functions) which exists prior to making the candidate exchange. The aspiration value associated with the three dropped edges of an exchange is the only one updated.

In order to find a good balance between the trade-off of the objective functions (i.e. customers' satisfaction and salesman's routing efficiency), we alternate the objective functions during the search in every I iterations the relevant tabu list and aspiration criterion.

7.6.2 Randomised Steepest Descent Method

The Randomised Steepest Descent Method (RSDM) we employed in the computational experiments has a similar framework (i.e. 3-Opt exchange and *neighbour list*) as in the DLTS. The main differences are: instead of using the tabu lists and the aspiration criterion to *explore* and *exploit* the search space in DLTS, we use the acceptance rule and randomisation to search the solution space in RSDM.

The acceptance rule we employ allows neutral moves of up to R consecutive iterations (where R is a parameter, e.g. 1000) before terminating the algorithm. This gives the algorithm more chance to *explore* and *exploit* the search space. We also introduce a randomisation strategy into the algorithm when there are multiple identical good tours, either improving or neutral moves, found in a single iteration. In order to prevent the tour from falling into the same local optimum, a move is randomly selected from the list of identical tours to be the new current tour.

Note that deteriorating moves are not allowed in RSDM. In other words, the algorithm will terminate once the best tour found at the end of the current iteration is worse than the best tour found so far. As in DLTS, we alternate the objective functions during the search every I iterations.

7.7 Computational Experience

In this section, we report on computational results of our proposed local search algorithms. Having explained the experimental design for the computational experiments, we present the performance effect of the swap and mutation operators in our proposed MXGA in solving a series of test problems from literature for the standard Symmetric Travelling Salesman Problem (STSP).

We complete this section by presenting the extensive computational results for different local search algorithms proposed in the previous sections for solving the symmetric version of TSPDD, based on the objective functions discussed in Section 7.3.

7.7.1 Experimental Design

All the reported computational results are generated on a Pentium IV 2.0GHz PC with 512 Mb memory. The algorithms are coded in ANSI-C using Microsoft Visual C++ 6.0 as the compiler. The problem instances used are from the TSPLIB

[266], with the number of cities varying from 17 to 200. We adopt the following abbreviations for the remaining subsections:

$T_{\min}$	: Minimum Tour Length
$L_{\max}$	: Maximum Lateness
SGA	: Standard Genetic Algorithm
MXGA	: MultiCrossover Genetic Algorithm
MXGA$_s$	: MXGA for STSP
DLTS	: Dynamic Length Tabu Search
RSDM	: Randomised Steepest Descent Method

We compare the performance of the various local search algorithms on the basis of the following statistics:

- Minimum Relative Percentage Deviation:

$$T_{\min} : MRD_T = \min_i \left\{ \frac{UB_{T_{\min_{i=1,2,\ldots,K}}} - OPT}{OPT} \times 100\% \right\}. \tag{7.6}$$

$$L_{\max} : MRD_L = \min_i \left\{ \frac{UB_{L_{\max_{i=1,2,\ldots,K}}} - LB_{L_{\max}}}{LB_{L_{\max}}} \times 100\% \right\}. \tag{7.7}$$

- Average Relative Percentage Deviation:

$$T_{\min} : ARD_T = \frac{\sum_{i=1}^{K} D_i}{K}, \text{ where } D_i = \frac{UB_{T_{\min_i}} - OPT}{OPT} \times 100\%. \tag{7.8}$$

$$L_{\max} : ARD_L = \frac{\sum_{i=1}^{K} E_i}{K}, \text{ where } E_i = \frac{UB_{L_{\max_i}} - LB_{L_{\max}}}{LB_{L_{\max}}} \times 100\%. \tag{7.9}$$

The variables used in the above equations take the following values:

- K = total number of repeated runs for each problem instance using different starting solution(s). In this case, $K = 10$.
- $UB_{T_{\min_i}}$ = heuristic solution found in i-th run for $T_{\min}$.
- $UB_{L_{\max_i}}$ = heuristic solution found in i-th run for $L_{\max}$.
- $LB_{L_{\max}}$ = lower bound of the $L_{\max}$ for the problem instance.
- OPT = optimal solution of the problem instance with respect to the tour length (as given in TSPLIB [266]).

The due date d_i, of city i is generated based on the idea of Solomon [258] in generating the center of the time windows for the vehicle routing problem with

time windows. He uses the interval $(e_1 + t_{1i}, l_1 - t_{i1} - s_i)$ as the center of the time windows where the variables take the following values:

- e_1 and l_1 = earliest and latest bound of the time windows for the depot,
- t_{1i} and t_{i1} = travel time from depot to city i and vice versa, and
- s_i = service time in city i.

As mentioned in Section 7.3, the depot (i.e. city 1) does not has a time window nor due date. For simplicity, we assume $e_1 = 1$ and $l_1 = OPT$. Note that $c_{ij} = t_{ij}$ for $i, j \in n$, and there is no service time in the cities. Hence, the integer due date of the city i, d_i, is generated from the uniform distribution of $[t_{1i}, OPT - t_{i1}]$.

The specific values for the generic design variables in MXGA, SGA, DLTS, and RSDM are summarised in Table 7.1 and 7.2 respectively. Initial computational experiments are performed to determine the size of the candidate list of temporary offspring. Five values of r ($r = 3, 5, 7, 9, 10$) are tested and results show that $r = 5$ gives the best result within a reasonable computation time.

Table 7.1: Implementation of generic design variables for MXGA and SGA

variable	value
crossover rate, p_c	0.75
multicrossover, r (MXGA and MXGA$_s$ only)	5 (= 10 temporary offspring)
individual mutation rate, p_M	0.25
filtration rate, F (MXGA and MXGA$_s$ only)	every 50 generations
selection mechanism	probabilistic binary tournament
chromosome length, L	number of cities
population size, N	100
alternating the objective functions, G	every 100 generations

Table 7.2: Implementation of generic design variables for DLTS and RSDM

variable	value
neighbour list, k	20
alternating the objective functions, I	100
max. number of consecutive neutral move allowed per run, R (RSDM only)	1000

7.7.2 Initial Investigation of MultiCrossover Genetic Algorithm

In this subsection, we report on computational results of the proposed MXGA for solving the standard STSP. This problem variant does not includes due dates to the cities. In other words, the only objective function to be considered is the minimum tour length of visiting each city exactly once. We believe that the results obtained from these experiments will give good indications on the performance of the MXGA. For the initial development, we consider only 15 problem instances containing between 17 and 100 cities, taken from the TSPLIB [266]. For each test problem, a total of 10 runs are performed to obtain an average value. A duration of maximum of 20000 generations per run are performed. Note that, the execution in each run of the $MXGA_s$ is halted as soon as the OPT for the problem instance is found, or when 1000 consecutive non-improving generations have been generated. By doing this, we can reduce the computation time spent on the execution of the algorithms. The logic is that, if an algorithm fails to improve the solution quality after a certain number of generations, the population is considered to have converged to a local optimum. Further exploration and exploitation on the search space are unlikely to improve the solution quality. Note that the average computation time for each problem instance is not reported in Table 7.3 and 7.4.

We incorporate the swap operator in the $MXGA_s$ instead of the reproduction procedure when the multicrossover operator does not apply to the selected parents. Table 7.3 examines the effect of that on solution quality for the proposed $MXGA_s$. The first column of Table 7.3 lists the names of the TSPLIB instances considered, where the number indicates the number of cities. The second column gives the optimal solution of the problem instances. The following two pairs of columns refer to the results obtained by the $MXGA_s$ for cases with and without the swap operator respectively. For each algorithm, the entries in the first column report the minimum relative percentage deviation (equation (7.6)) of the tour length while the second column gives the average relative percentage deviation (equation (7.8))

of the tour length. The final line of Table 7.3 gives the overall average value over all test problems.

We first observe that the swap procedure yields better results in the $MXGA_s$ compared to the algorithm without the procedure. Of the 15 problems tested, the $MXGA_s$ with the swap operator finds optimal solutions for six problems at least once in 10 runs, compared to five problems for $MXGA_s$ without the swap operator. As the values indicate, we believed that the swap operator procedure is able to help the algorithm to further explore the search space although a slightly longer computation time is required. Consequently, the swap operator is used in the proposed MXGA.

Table 7.3: Results of $MXGA_s$ (with and without Swap) (maximum of 20000 generations per run [a])

TSPLIB		**$MXGA_s$**			
		with Swap		without Swap	
Data Set	OPT	MRD_T	ARD_T	MRD_T	ARD_T
gr17	2085	0.00	0.71	0.00	1.74
gr21	2707	0.00	0.00	0.00	0.07
gr24	1272	0.00	1.01	0.00	1.21
fri26	937	0.00	0.00	0.00	0.64
bays29	2020	0.00	1.36	0.00	3.54
dantzig42	699	0.00	0.00	0.14	3.35
swiss42	1273	0.67	4.41	1.01	6.21
att48	10628	1.11	5.04	1.11	4.73
gr48	5046	1.34	5.83	2.11	8.02
hk48	11461	1.15	4.88	1.89	5.70
eil51	426	1.50	4.42	2.03	5.44
berlin52	7542	1.34	4.37	3.21	6.51
st70	675	2.96	8.61	5.48	10.81
pr76	108159	0.77	6.33	2.54	8.87
kroA100	21282	2.63	8.00	4.67	10.27
Average		**0.90**	**3.66**	1.61	5.14

[a] stopping criterion: 1000 consecutive non-improving generations or optimal solution is found.

In the next experiment, we will investigate the impact of the mutation operator in the $MXGA_s$. It has been suggested that the mutation operator might deteriorate the solution quality by randomly inserting the cities into the tour. To analyse this, a $MXGA_s$ has been applied without using the mutation operator. Note that the swap operator is used in the $MXGA_s$ for both cases (with and without

mutation operator). Table 7.4 summarises the computational experiments of these experiments. The table gives the same information for the first two columns as in Table 7.3. The following two pairs of columns refer to the results obtained by the MXGA$_s$ for the cases of with and without the mutation operators respectively.

Table 7.4: Results of MXGA$_s$ (with and without Mutation) (maximum of 20000 generations per run [b])

TSPLIB		**MXGA$_s$**			
		with Mutation		without Mutation	
Data Set	OPT	MRD_T	ARD_T	MRD_T	ARD_T
gr17	2085	0.00	0.00	0.00	0.48
gr21	2707	0.00	0.00	0.00	0.00
gr24	1272	0.00	0.00	0.00	0.94
fri26	937	0.00	0.00	0.00	0.00
bays29	2020	0.00	0.79	0.00	1.47
dantzig42	699	0.00	0.00	0.00	0.00
swiss42	1273	0.00	3.97	0.84	4.23
att48	10628	0.56	3.11	1.11	5.11
gr48	5046	0.00	3.06	1.23	5.79
hk48	11461	0.00	4.46	1.43	4.95
eil51	426	0.23	1.81	1.41	4.32
berlin52	7542	0.00	3.89	1.05	4.24
st70	675	0.74	4.56	3.05	8.54
pr76	108159	0.65	4.81	0.79	6.54
kroA100	21282	0.45	5.77	2.79	7.51
Average		**0.18**	**2.42**	0.91	3.61

[b] stopping criterion: 1000 consecutive non-improving generations or optimal solution is found.

Our first observation from Table 7.4 is that the results achieved by the MXGA$_s$ with the mutation operator clearly outperform the MXGA$_s$ without the mutation operator. Of 15 problems tested, the MXGA$_s$ with mutation operator finds optimal solutions of 10 problems at least once in 10 runs while the remaining five problems of near optimal solutions (i.e less than 1% over the optimal). As in the previous experiment, the MXGA$_s$ with mutation operator requires slightly longer computation time compared to the MXGA$_s$ without mutation operator. Thus, the mutation operator is used in the proposed MXGA. Consequently, this final version of the MXGA is used in the comparative tests in next subsection.

7.7.3 A Comparison of different Local Search Algorithms

In this subsection, we present the results of an extensive computational experiment that compare our proposed MXGA with the Standard GA (SGA), DLTS and RSDM described in the previous sections. The differences between the MXGA and SGA are with regards to the use of the subtour crossover operator, reproduction procedure and the replacement scheme. The SGA applies the subtour crossover operator to produce two offspring from two selected parents. In the case of SGA, the steps explained in Section 7.5.2 are used only once (i.e. $r = 1$) to generate exactly two offspring. The SGA uses the reproduction procedure instead of a swap operator when the crossover does not apply to the selected parents. The replacement strategy employ in the SGA is the steady-state replacement strategy.

For this final experiment, we use 29 problem instances containing between 51 and 200 cities, taken from the TSPLIB [266]. All the problem instances use a format of 'EUC_2D' for the distance between the cities. EUC_2D means the edge of a pair of cities is an Euclidean distance in two-dimensional. The choice is made due to the limitation of the Windows version of the Concorde software [57] in obtaining the optimal tour length which is then used in the calculation of the lower bound of maximum lateness in Section 7.4. This Windows version of software is constrained by the format of the distance between the cities.

For each test problem, a total of 10 runs are performed to obtain an average value. In order to have a fair comparison between the different algorithms in this experiments, we employ the stopping criterion of 300 CPU seconds (5 minutes) per run for problem instances with 100 cities or less, and 600 CPU seconds (10 minutes) per run for problem instances with cities between 101 and 200.

Recall that, we optimise the bicriteria objective function of the problem by alternating between optimising each of the objective function discussed in Section 7.3, through a *hierarchical optimisation* approach in every I iterations (in this case = 100) for DLTS and RSDM, and G generations (in this case = 100) for MXGA and SGA. By alternating the objective functions during the execution of

the algorithms, we are actually trying to solve the problem using a *simultaneous optimisation* approach. Under this approach, both objective functions are treated as equally important. As a result, a set of *Pareto optimal* solutions consisting of both objective functions is obtained, where a *trade-off curve* and an *efficient frontier* for the problem can be formed. Note that the trade-off curve and the efficient frontier are equal only if the trade-off curve is convex.

It is worth mentioning that there is no suitable way of constructing a single composite objective function to represent the bicriteria objective function of the problem. This is due to the incomparability of the unit used (i.e. time, tour length) in both performance criteria which result in the computationally inaccessibility for optimising the single composite objective function in a direct manner.

In this section, we present only the results of the two extreme points of the efficient frontier. The computational results of the first and second objective functions are presented in Table 7.5 and Table 7.6 respectively. The first column in both tables lists the names of the TSPLIB instances considered, where the number indicates the number of cities. For each algorithm, the entries in the first two columns report the minimum relative percentage deviation (equation (7.6)) and the average relative percentage deviation (equation (7.8)) of the tour length respectively. The next two columns report the minimum relative percentage deviation (equation (7.7)) and the average relative percentage deviation (equation (7.9)) of the maximum lateness respectively. The final line of each table gives the overall average value over all test problems.

By considering the overall average value found over all test problems in both Table 7.5 and Table 7.6, the MXGA achieved better results compared to SGA, DLTS and RSDM. However, a one-to-one comparison on the test problems shows that the MXGA achieved a mixed degree of success compared to DLTS and RSDM. Note that, the results obtained for $L_{\max}$ in both tables are less impressive in all cases. This might be due to the simple lower bound derived previously. Hence, a better lower bound is needed to further justify the quality of the results. We also noticed that both DLTS and RSDM obtained similar results in most of the test

problems. This suggests that the randomisation used in the RSDM might have the same effectiveness as the tabu list in DLTS. The problem instances with prefix "pr" appear to be difficult to solve using DLTS and RSDM compared to MXGA and SGA.

The results of SGA in Table 7.5 appear comparable with both the DLTS and RSDM algorithms in most of the test problems, especially in finding the minimum tour length. However, SGA failed to compete with the other algorithms when the second objective is considered (in Table 7.6). Based on the computational results, we can conclude that a good balance between the trade-off of the maximum lateness and the minimum tour length is very difficult to achieve within a limited computation time.

Table 7.5: A Comparison of Different Local Search Algorithms [c] (objective function: minimise $L_{\max}$ with a secondary objective of minimising the total tour length)

TSPLIB	SGA				MXGA				DLTS				RSDM			
	Tmin		Lmax		Tmin		Lmax		Tmin		Lmax		Tmin		Lmax	
Data Set	MRD_T	ARD_T	MRD_L	ARD_L	MRD_T	ARD_T	MRD_L	ARD_L	MRD_T	ARD_T	MRD_L	ARD_L	MRD_T	ARD_T	MRD_L	ARD_L
eil51	15.82	17.57	61.33	70.11	15.59	19.10	45.74	51.19	20.85	23.35	52.41	62.95	17.04	17.77	44.29	47.29
berlin52	17.06	25.07	58.76	77.44	19.43	22.04	44.51	53.72	22.19	25.58	52.28	69.13	18.36	21.15	42.78	47.28
st70	26.07	28.10	85.10	106.00	22.13	27.41	69.37	80.02	25.27	29.66	70.21	81.77	23.05	24.80	66.78	70.62
eil76	19.29	27.13	110.81	140.96	23.75	25.23	81.77	94.45	22.42	26.25	79.76	92.96	20.19	22.24	71.46	79.48
pr76	13.50	18.86	40.69	52.24	10.32	14.17	31.92	37.74	21.39	27.95	37.28	59.08	16.28	21.60	28.62	43.67
rat99	28.93	35.35	121.88	144.91	20.38	22.70	82.44	96.95	29.48	31.80	88.63	98.10	26.36	29.29	85.05	90.78
kroA100	28.90	34.02	87.88	107.86	24.28	25.53	68.29	79.67	33.39	36.84	66.36	78.67	31.54	34.00	61.85	68.45
kroB100	33.55	37.14	112.00	132.07	18.55	22.19	67.98	81.50	26.59	32.83	63.81	73.23	25.21	29.37	61.71	67.25
kroC100	24.60	36.67	88.69	117.20	17.21	25.46	70.63	80.98	32.94	38.58	60.57	73.61	33.72	36.49	59.37	68.33
kroD100	26.71	41.00	99.60	120.32	22.28	24.92	73.86	82.12	33.28	39.41	64.56	78.54	33.87	35.61	65.11	72.23
kroE100	32.04	33.56	116.62	139.18	28.71	32.28	83.82	97.58	27.21	32.92	76.90	89.53	26.39	30.73	71.83	80.97
rd100	30.47	34.53	110.04	130.26	20.09	22.26	76.97	90.30	29.84	32.42	67.28	78.05	25.72	29.39	60.70	69.93
eil101	27.31	31.32	109.12	125.69	18.60	20.83	75.67	88.38	26.10	29.01	64.42	75.20	20.76	25.15	61.22	67.65
lin105	24.15	36.56	87.92	111.08	18.03	24.24	71.16	79.94	35.27	39.30	65.54	79.23	30.26	34.55	64.92	71.70
pr107	21.01	33.87	65.40	102.48	16.59	18.83	50.49	61.31	24.03	32.31	52.02	89.16	29.21	44.62	59.60	113.29
pr124	30.70	46.86	93.86	123.78	15.61	22.82	65.60	75.26	35.92	55.71	65.84	115.46	41.26	68.52	77.99	128.78
bier127	24.70	29.66	44.21	55.71	12.61	15.50	25.17	31.02	13.19	21.75	25.07	32.54	13.77	21.08	32.77	41.29
ch130	36.21	40.82	108.82	129.91	15.51	24.07	73.45	86.16	27.86	33.66	56.67	66.73	28.50	31.76	54.59	61.99
pr136	30.99	36.25	77.60	89.20	13.63	17.73	49.54	56.61	24.03	36.16	59.18	91.51	46.18	65.22	81.29	138.57
pr144	33.11	48.76	78.52	104.97	22.52	24.85	53.77	61.37	63.69	96.19	138.68	201.26	101.71	143.41	192.88	280.18
ch150	38.73	49.30	135.71	160.78	23.04	26.94	93.20	103.58	35.15	38.37	75.87	83.30	32.78	35.53	71.37	78.75
kroA150	46.22	49.16	115.34	138.05	26.53	28.39	77.29	89.36	36.99	40.80	59.09	68.29	36.60	40.52	61.59	68.18
kroB150	39.48	48.86	132.88	161.65	24.88	27.47	90.07	102.78	37.04	39.28	72.96	83.20	37.65	38.89	73.73	81.90
pr152	27.49	42.56	90.64	116.00	18.30	20.63	62.55	75.84	38.88	59.99	88.34	147.74	51.10	92.39	136.75	218.27
u159	42.72	52.60	118.26	145.43	21.14	19.75	63.16	72.72	29.99	36.18	56.51	67.86	28.56	35.39	55.66	65.35
rat195	57.80	66.11	230.26	257.20	39.78	36.94	148.91	165.52	34.27	37.27	107.60	120.11	31.18	36.04	107.21	118.40
d198	31.58	39.49	120.53	143.26	26.26	22.64	85.91	97.41	26.05	30.75	61.51	69.84	25.10	29.60	56.79	67.49
kroA200	58.20	64.99	161.84	187.91	34.80	38.40	106.44	121.09	36.81	40.68	69.73	75.50	36.76	40.67	66.51	76.18
kroB200	55.61	69.62	164.15	196.39	37.87	37.17	104.32	116.10	37.92	39.84	70.99	79.83	35.40	39.04	71.44	78.22
AVERAGE	31.83	39.85	104.43	127.17	**21.67**	**24.50**	72.21	**83.13**	30.62	37.41	**67.93**	85.60	31.88	39.82	70.55	88.36

[c] stopping criterion : 300 CPU seconds per run for problem instances with 100 cities or less.
: 600 CPU seconds per run for problem instances with cities between 101 and 200.

Table 7.6: A Comparison of Different Local Search Algorithms [d] (objective function: minimise the total tour length with a secondary objective of minimising L_{max})

TSPLIB	SGA				MXGA				DLTS				RSDM			
	Tmin		Lmax		Tmin		Lmax		Tmin		Lmax		Tmin		Lmax	
Data Set	MRD_T	ARD_T	MRD_L	ARD_L	MRD_T	ARD_T	MRD_L	ARD_L	MRD_T	ARD_T	MRD_L	ARD_L	MRD_T	ARD_T	MRD_L	ARD_L
eil51	4.37	7.85	96.10	101.43	1.17	2.54	128.69	116.21	2.44	5.77	140.17	121.50	0.52	1.08	124.93	121.70
berlin52	8.44	14.75	98.71	112.99	0.06	3.79	109.95	117.50	4.04	9.03	117.09	129.09	0.63	1.72	115.65	116.66
st70	8.59	17.16	134.48	134.63	2.31	4.59	127.18	136.14	3.35	6.83	139.29	147.92	1.54	3.11	132.91	138.93
eil76	12.53	19.16	154.59	181.73	2.57	5.27	163.84	168.61	3.98	6.13	181.43	171.98	1.34	3.12	172.48	183.73
pr76	6.80	13.09	52.96	60.06	1.17	4.24	50.08	55.65	12.57	17.88	49.43	71.61	5.05	11.70	45.71	60.01
rat99	19.54	28.82	138.92	175.58	6.87	10.03	158.35	165.14	6.00	8.46	159.38	160.94	3.67	5.53	154.72	155.49
kroA100	13.77	24.25	111.87	131.07	2.43	5.98	115.12	117.84	4.52	9.23	119.23	126.01	4.06	6.55	114.11	122.79
kroB100	19.93	29.20	138.60	156.75	4.15	7.38	122.16	134.22	3.68	6.99	130.48	131.85	3.14	4.89	122.70	125.76
kroC100	19.53	28.93	123.42	129.89	1.79	8.14	105.21	118.09	4.92	9.49	112.46	124.33	5.05	7.63	115.16	118.81
kroD100	18.64	29.08	109.20	134.99	3.89	7.67	120.92	122.48	4.07	8.77	117.66	125.62	3.63	6.02	119.12	123.69
kroE100	18.07	27.52	150.11	164.06	3.19	7.14	151.33	150.32	3.41	6.81	142.47	148.62	3.26	4.90	138.49	144.45
rd100	17.78	27.34	131.15	147.99	2.97	8.34	135.10	132.48	2.56	6.43	119.61	131.32	1.07	3.11	121.52	122.59
eil101	20.76	26.40	135.30	155.02	4.96	7.83	137.11	145.04	5.06	7.70	138.53	143.10	3.72	5.26	125.59	137.54
lin105	17.39	29.69	98.48	127.89	8.80	13.02	102.90	104.66	5.77	11.27	90.07	107.98	4.21	7.98	99.65	105.05
pr107	14.63	27.14	94.23	121.38	4.94	9.96	69.19	83.48	9.59	16.11	105.24	120.90	13.23	27.48	130.81	138.98
pr124	26.05	42.21	108.15	133.17	7.90	13.78	79.79	90.44	19.62	35.95	83.22	144.59	19.55	45.37	95.79	148.06
bier127	19.79	27.29	51.35	61.33	5.35	9.49	40.71	48.78	3.93	9.21	34.96	39.79	7.56	15.24	39.21	51.98
ch130	28.14	36.59	140.19	150.57	10.48	14.24	109.23	114.63	4.69	6.90	108.67	115.07	3.00	4.76	100.75	112.26
pr136	24.54	33.36	85.24	97.84	5.39	10.56	69.93	73.64	12.95	27.55	99.19	109.58	29.61	58.17	99.53	154.60
pr144	28.55	42.89	84.97	115.67	7.75	12.81	68.15	71.67	52.47	89.04	202.43	224.44	71.66	134.32	243.45	294.08
ch150	33.91	44.41	145.60	181.10	12.32	19.02	152.16	168.80	5.70	8.20	128.55	140.03	4.39	6.21	135.46	136.86
kroA150	34.45	44.24	146.42	149.52	11.31	17.01	96.64	109.66	6.61	9.89	102.69	109.97	7.04	9.93	101.85	109.11
kroB150	30.14	43.95	161.05	182.24	12.51	17.70	131.54	133.49	7.16	10.01	115.77	130.13	7.48	10.08	118.56	132.79
pr152	21.98	37.97	111.33	126.65	8.55	14.58	80.58	86.54	29.27	50.16	155.07	182.43	37.94	86.20	182.62	233.07
u159	36.51	48.57	147.10	157.31	6.75	11.69	83.81	87.31	4.43	9.24	99.16	105.92	4.04	8.11	90.03	101.75
rat195	51.39	62.07	254.20	282.43	24.91	28.65	195.64	200.17	6.49	10.22	173.80	182.95	7.33	9.60	175.64	182.43
d198	26.34	35.52	130.87	156.89	12.02	16.03	101.11	112.59	3.65	5.43	106.99	109.62	4.20	6.06	107.71	111.53
kroA200	52.64	61.84	192.82	199.99	23.68	29.43	129.08	140.49	6.19	8.80	118.01	120.95	7.26	9.23	118.49	120.62
kroB200	52.47	66.56	185.28	209.16	23.07	27.60	132.33	136.44	7.30	10.27	115.91	120.84	7.13	9.77	119.39	118.43
AVERAGE	23.71	33.72	128.02	146.18	**7.70**	**12.02**	**112.68**	**118.71**	8.50	14.75	120.93	131.00	9.39	17.69	122.83	135.30

[d] stopping criterion : 300 CPU seconds per run for problem instances with 100 cities or less.
: 600 CPU seconds per run for problem instances with cities between 101 and 200.

7.8 Conclusions and Remarks

In this chapter, the variants of the Time Constraint Travelling Salesman Problem (TCTSP) are studied. A new variant of TCTSP, called the Travelling Salesman Problem with Due Dates (TSPDD) is introduced, where each city to be visited has a due date. The objective is to find an ordering of the cities that starts and ends at the depot which minimises the maximum lateness and the total tour length of the cities. A lower bound of the maximum lateness to the problem is also proposed in this chapter.

A MultiCrossover Genetic Algorithm (MXGA) has been proposed to solve the TSPDD in this chapter. Various techniques have been introduced into the proposed MXGA to further enhanced the solutions quality. The computational results presented for several symmetric version of TSP instances have shown that the MXGA is able to produce high quality solutions.

A tabu search which dynamically control the length of the tabu list during the execution and a neighbour list of visiting the nearest cities is developed for the problem. A randomised steepest descent method is also developed. It randomly selects a tour when there are multiple identical good tours found in a single iteration. Extensive computational experiments have been carried out to solve the TSPDD. Comparative results show that the MXGA achieved better solution quality compared to a standard genetic algorithm, dynamic length tabu search and randomised steepest descent method. However, the results obtained in minimising the maximum lateness are less impressive.

There are several issues for future research. First, a better lower bound of the maximum lateness could be derived. Secondly, it would be interesting to investigate the performance of the local search algorithms on other TCTSP. Thirdly, further tests of the algorithms on other possibly more complex TSP instances are required to provide a detailed assessment of the merits of the proposed algorithms. For instance, different range of city due dates sets could be considered.

Chapter 8

Conclusions and Further Research

Throughout this thesis, we have considered the developments of a general framework for MultiCrossover Genetic Algorithms (MXGAs) for three specific variants of Combinatorial Optimisation Problems (COPs) and successfully applied the proposed MXGAs to each of the problems. We demonstrated that the proposed approach is general enough to be applicable to a diverse range of problems from the Single Machine Family Scheduling Problem (SMFSP) with family setup times to the Symmetric Travelling Salesman Problem with due dates (STSPDD). In this last chapter, we summarise the research conducted and discuss the prospects for future research on the subject.

8.1 Summaries of Research Conducted

The problems we studied are mainly motivated by the dilemma faced by manufacturing organisations which involves the trade-off between the manufacturer's efficiency and customers' satisfaction. By including the customers' due dates into the standard problems, we have created some new **NP**-hard problems which have these due dates as a common theme.

One of the main objectives of this research is to develop a general framework for the MXGA (Section 4.9) for solving the problems. The proposed MXGA utilises a

multicrosover operator that uses a simple yet effective standard crossover operator as the crossover strategy to generate offspring. Every time the proposed crossover strategy is executed, two temporary offspring are generated from the selected parents. The main feature of the multicrossover is that it first generates a candidate list of valid temporary offspring from a pair of selected parents through repeated applications of the proposed crossover strategy. Then, the best and a selected temporary offspring (using the probabilistic binary tournament selection mechanism) are chosen to be the offspring for the current generation. Furthermore, various operators such as swap and filtration techniques have been introduced into the MXGA to further enhance the solution quality. The efficiency of the MXGA developed in the thesis is measured through a comparison with other local search methods such as tabu search (TS) and a steepest descent method (SDM) using the same problem instances. Since the problems are **NP**-hard, the optimal solutions are not known. The next best form of validation of the algorithm is to compare the results with lower bounds.

The implementation of the MXGA starts in Chapter 5 with the SMFSP with family setup times. The objective is to find a schedule which minimises the maximum lateness of the jobs in the presence of the sequence independent family setup times. To the best of our knowledge, no research has been carried out on the application of a genetic algorithm (GA) for this specific problem. The proposed MXGA is developed using binary representation and uses the standard 1-point or F-point crossover operator (where F defines the total number of families in the schedule) as the crossover strategy to produce temporary offspring.

The performance of the MXGA is compared with an improved TS and SDM. A tabu search with a dynamic length tabu list (DLTS) is designed for the problem using the shift job neighbourhood (as explained in Section 5.5). The tabu list length is dynamically controlled during implementation in order to achieve better solution quality. Such a process has an important influence on which moves are available to be selected at a given iteration. An aspiration criterion is also introduced to prevent the occasional loss of good solutions due to the tabu list. In our

SDM, we considered a randomisation strategy when there are multiple identical good solutions (i.e. improving and neutral moves) found in a single iteration. A move is selected randomly from the list of the identical good solutions. We believed this strategy helps the search to escape from a local optimum and continue its search in other '*interesting*' regions of the search space. Extensive computational results show that the proposed MXGA performs better compared than the other local search methods, and in particularly significantly improves the solution quality compared to a standard GA (SGA).

In Chapter 6, we first studied the non-oriented Two-Dimensional Rectangular Single Bin Size Bin Packing Problem (2DRSBSBPP) and successfully developed a heuristic placement routine, called Lowest Gap Fill (LGF), that is effective in filling the gaps in a partial layout by dynamically selecting the best rectangle for placement. The LGF requires only $O(n^2)$ time (where n is the number of rectangles) during the packing stage. Promising results have been achieved (as in Table 6.4 and 6.5) and it is comparable with other higher complexity placement routines such as Floor Ceiling and Touching Perimeter (both routines have time complexity of $O(n^3)$).

We extended the problem by including a positive integer due date for each rectangle and a fixed processing time for the bins used. Hence, a new variant of the problem emerged and we referred to it as 2DRSBSBPP with due dates. The objective is to minimise the maximum lateness of the rectangles by packing them, without overlap, and minimising the number of bins. This new problem variant has practical industrial applications such as in the wood and metal industries. Section 6.4 explains in detail the problem to be solved. Since the optimal solution is not known, we derived a simple lower bound on the maximum lateness for the problem.

The MXGA proposed for both standard and extended 2DRSBSBPP are based on the general framework discussed in Section 4.9. An integer permutation representation is used where each gene indicates the bin number, in which the rectangles are placed into the bin. We used the standard 1-point or 2-point crossover operator

as the crossover strategy to produce two temporary offspring. The performance of the MXGA is compared with the Unified Tabu Search (UTS) developed by Lodi *et al.* [194] and a Randomised Descent Method (RDM) which uses a similar framework as in the UTS. All the local search algorithms use the LGF as the placement routine. Although the MXGA significantly outperforms the SGA and RDM in the extensive computational experiments, it achieves a mixed degrees of success compared to UTS.

The STSPDD is the main focus of Chapter 7. This is a new variant of the time-constrained TSP where each city has a due date by which it should ideally be visited. The objective is to minimise the maximum lateness and the total tour length of the cities to be visited. This extension has important practical applications in banks or postal deliveries, scheduling deliveries, etc. The proposed MXGA uses a path representation and subtour crossover operator. The proposed crossover strategy generates two temporary offspring through a process that inserts a subtour, from one parent to the other parent.

The performance of the MXGA is compared with a DLTS and RSDM. Both DLTS and RSDM use a 3-Opt move as the inner heuristic. Since applying the 3-Opt procedure would result in a huge search space, a *neighbour list* for each city is introduced to reduce the search space and increase the speedup of the algorithms while maintaining the solution quality. A *neighbour list* of a city is a list which contains a group of nearest neighbours from the city in a non-decreasing order of their distance. Comparative results show that the MXGA achieves better solution quality compared to the SGA, DLTS and RSDM. However, the results obtained in minimising the maximum lateness are less impressive for all the local search algorithms.

Unfortunately, the lack of suitable lower bounds for the maximum lateness makes it difficult to decide whether the solutions obtained from the proposed MXGAs and other local search methods in the previous problems are in fact of high quality. This could be the subject of future further research efforts.

8.2 Further Research

Our work has left some open ends, both theoretical and computational. The use of local search algorithms to tackle any kind of COPs is always an endless source for research. More classical local search algorithms such as simulated annealing, scatter search, variable neighbourhood search or ant colony optimisation could be implemented to further justify the merit of our proposed MXGAs in solving the problems. Further extensive computational tests of the MXGAs on other possibly more complex problem instances may also be required to provide a detailed assessment of the merits of the proposed algorithms. For instance, different ranges of customers' due dates sets could be considered.

Furthermore, it would be interesting to investigate the performance of the proposed MXGAs on other optimality criteria for the problems studied. For instance, the development of MXGAs for other optimality criterion such as minimising the total (weighted) tardiness/earliness in SMFSP, other cutting and packing problems such as open dimension problem and stock cutting problem, and other time-constrained TSP such as the TSP with time windows are worthy of future research.

The next issue that comes to mind is computing suitable lower bounds of the maximum lateness for the non-oriented 2DRSBSBPP with due dates and the TSPDD. Better lower bounds could be derived by using the dynamic programming formulation of the state-space relaxation technique. The search for lower bounds using other techniques is still an open area for future research.

In this thesis, we have demonstrated the effectiveness and efficiency of the MXGAs built up from the underlying general framework in solving COPs. We really hope that the studies in this thesis will be helpful for the developments of such algorithms and new variants of the COPs. It has certainly given us many possibilities for further research.

References

[1] E. Aarts and J.K. Lenstra (editor). *Local Search in Combinatorial Optimization.* John Wiley & Sons Ltd., UK, 1997.

[2] B.H. Ahn and J.H. Hyun. Single Facility Multi-Class Job Scheduling. *Computers & Operations Research*, 17(3):265–272, 1990.

[3] J.T. Alander. On Optimal Population Size of Genetic Algorithms. In *Proceedings of IEEE International Conference of Computer Systems and Software Engineering*, volume 92, pages 65–70, 1992.

[4] J.T. Alander. An Indexed Bibliography of Genetic Algorithms: 1957–1993. *Technical Report: 94-1, Department of Information Technology and Production Economics, University of Vaasa*, 13 Feb. 1994.

[5] J.T. Alander. An Indexed Bibliography of Genetic Algorithms and the Traveling Salesman Problem. *Technical Report: 94-1-TSP, Department of Information Technology and Production Economics, University of Vaasa*, 12 Dec. 2000.

[6] A. Allahverdi, J.N.D. Gupta, and T. Aldowaisan. A Review Of Scheduling Research Involving Setup Considerations. *OMEGA, International Journal of Management Science*, 27:219–239, 1999.

[7] B.K. Ambati, J. Ambati, and M.M. Mokhtar. Heuristic Combinatorial Optimization by Simulated Darwinian Evolution: A Polynomial Time Algorithm for the Traveling Salesman Problem. *Biological Cybernetics*, 65:31–35, 1991.

[8] E.J. Anderson, C.A. Glass, and C.N. Potts. Machine Scheduling. In E. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 361–414, UK, 1997. John Wiley & Sons Ltd.

[9] D.L. Applegate, R. Bixby, V. Chvátal, and W. Cook. On the Solution of Traveling Salesman Problems. *Documenta Mathematica. Extra Volumn ICM 1998*, 3:645–656, 1998.

[10] D.L. Applegate, R. Bixby, V. Chvátal, and W. Cook. Finding Tours in the TSP. *Report No. 99885, Research Institute for Discrete Mathematics, Universität Bonn, Bonn, Germany*, 1999.

[11] D.L. Applegate, L.S. Buriol, B.L. Dillard, D.S. Johnson, and P.W. Shor. The Cutting-Stock Approach to Bin Packing: Theory and Experiments. In R.E. Ladner, editor, *Proceedings of the Fifth Workshop on Algorithm Engineering and Experiments (ALENEX)*, Baltimore, Maryland, April 2003. SIAM (Society for Industrial and Applied Mathematics).

[12] D.L. Applegate, W. Cook, and A. Rohe. Chained Lin-Kernighan for Large Traveling Salesman Problems. *INFORMS Journal on Computing*, 15(1):82–92, Winter 2003.

[13] V.A. Armentano and R. Mazzini. A Genetic Algorithm for Scheduling on a Single Machine with Set-Up Times and Due Dates. *Production Planning & Control*, 11(7):713–720, 2000.

[14] N. Ascheuer, M. Fischetti, and M. Grötschel. Solving the Asymmetric Traveling Salesman Problem with Time Windows by Branch-and-Cut. *Mathematical Programming*, 90(3):475–506, May 2001.

[15] T. Bäck. Optimal Mutation Rates in Genetic Search. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithm*, pages 2–8, San Mateo, CA, USA, 1993. Morgan Kaufmann Publisher Inc.

[16] B.S. Baker, Jr.E.G. Coffman, and R.L. Rivest. Orthogonal Packing in Two Dimensions. *SIAM Journal on Computing*, 9(4):846–855, 1980.

[17] E.K. Baker. An Exact Algorithm for the Time-Constrained Traveling Salesman Problem. *Operations Research*, 31(5):938–945, Sept.–Oct. 1983.

[18] J.E. Baker. Adaptive Selection Methods for Genetic Algorithms. In J.J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 101–111, Mahwah, NJ, USA, 1985. Lawrence Erlbaum Associates, Inc.

[19] J.E. Baker. Reducing Bias and Inefficiency in the Selection Algorithm. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, pages 14–21, Hillsdale, New Jersey, 1987. Lawrence Erlbaum Associates.

[20] K.R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, New York, 1974.

[21] K.R. Baker. Heuristic Procedures for Scheduling Job Families with Setups and Due Dates. *Naval Research Logistic*, pages 976–991, 1999.

[22] K.R. Baker and M.J. Magazine. Minimizing Maximum Lateness With Job Families. *European Journal of Operational Research*, 127:126–139, 2000.

[23] E. Balas and N. Simonetti. Linear Time Dynamic-Programming Algorithms for New Classes of Restricted TSPs: A Computational Study. *INFORMS Journal on Computing*, 13(1):56–75, Winter 2001.

[24] N. Bansal, A. Blum, S. Chawla, and A. Meyerson. Approximation Algorithms for Deadline-TSP and Vehicle Routing with Time Windows. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, pages 166–174, Chicago, IL, USA, 13-16 June 2004. ACM Press, New York, USA.

[25] W. Banzhaf. The "Molecular" Traveling Salesman. *Biological Cybernetics*, 64(1):7–14, Nov. 1990.

[26] D. Beasley, D.R. Bull, and R.R. Martin. An Overview of Genetic Algorithms: Part 1, Fundamentals. *University Computing*, 15(2):58–69, 1993.

[27] D. Beasley, D.R. Bull, and R.R. Martin. An Overview of Genetic Algorithms: Part 2, Research Topics. *University Computing*, 15(4):170–181, 1993.

[28] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[29] R. Bellman. Dynamic Programming Treatment of the Traveling Salesman Problem. *Journal of the ACM*, 9(1):61–63, Jan. 1962.

[30] M. Bellmore and G.L. Nemhauser. The Traveling Salesman Problem: A Survey. *Operations Research*, 16(3):538–558, 1968.

[31] J.L. Bentley. Fast Algorithms for Geometric Traveling Salesman Problems. *ORSA Journal of Computing*, 4:387–411, 1992.

[32] J.O. Berkey and P.Y. Wang. Two-Dimensional Finite Bin-Packing Algorithms. *The Journal of the Operational Research Society*, 38(5):423–429, 1987.

[33] R.G. Bland and D.F. Shallcross. Large Traveling Salesman Problems arising from experiments in X-ray Crystallography: a preliminary report on computation. *Operation Research Letters*, 8:125–128, 1989.

[34] M.A. Boschetti and A. Mingozzi. The Two-Dimensional Finite Bin Packing Problem. Part I: New Lower Bounds for the Oriented Case. *4OR: Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1:27–42, 2003.

[35] M.A. Boschetti and A. Mingozzi. The Two-Dimensional Finite Bin Packing Problem. Part II: New Lower and Upper Bounds. *4OR: Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1:135–147, 2003.

[36] R.M. Brady. Optimization Strategies Gleaned from Biological Evolution. *Nature*, 137:804–806, 1985.

[37] B. Brugger, K.F. Doerner, R.F. Hartl, and M. Reimann. AntPacking – An Ant Colony Optimization Approach for the One-Dimensional Bin Packing Problem. In J. Gottlieb and G.R. Raidl, editors, *4th European Conference on Evolutionary Computation in Combinatorial Optimization: EvoCOP 2004*, volume 3004 of *LNCS*, pages 41–50, Coimbra, Portugal, 5-7 April 2004. Springer-Verleg Berlin Heidelberg.

[38] J. Bruno and P. Downey. Complexity of Task Sequencing with Deadlines, Set-up Times and Changeover Costs. *SIAM Journal on Computing*, 7(4):393–404, November 1978.

[39] E. K. Burke, P.I. Cowling, and R. Keuthen. Effective Local and Guided Variable Neighbourhood Search Methods for the Asymmetric Traveling Salesman Problem. In E.J.W. Boers, J. Gottlieb, P.L. Lanzi, R.E. Smith, S. Cagnoni, E. Hart, G.R. Raidl, and H. Tijink, editors, *Applications of Evolutionary Computing, Proceedings of the EvoWorkshops 2001, Lecture Notes in Computer Science*, volume 2037, pages 203–212, Berlin, 2001. Springer-Verlag.

[40] E. K. Burke, G. Kendall, and G. Whitwell. A New Placement Heuristic for the Orthogonal Stock-Cutting Problem. *Operations Research*, 52(4):655–671, July 2004.

[41] E.K. Burke and G.Kendall (eds.). *Search Methodologies. Introductory Tutorials in Optimization and Decision Support Techniques.* Springer, New York, USA, 2005.

[42] R.W. Calvo. A New Heuristic for the Traveling Salesman Problem with Time Windows. *Transportation Science*, 34(1):113–124, Feb. 2000.

[43] A.M. Campbell and B.W. Thomas. Probabilistic Traveling Salesman Problem with Deadlines. *Transportation Science*, (to appear).

[44] W.B. Carlton and J.W. Barnes. Solving the Traveling Salesman Problem with Time Windows using Tabu Search. *IIE Transactions*, 28:617–629, 1996.

[45] V. Černý. A Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, Jan. 1985.

[46] B. Chazelle. The Bottom-Left Bin Packing Heuristic: An Efficient Implementation. *IEEE Transactions on Computers*, 32(8):697–707, 1983.

[47] Z.L. Chen. Scheduling with Batch Setup Times and Earliness-Tardiness Penalties. *European Journal of Operational Research*, 96:518–537, 1997.

[48] T.C.E. Cheng, J.N.D. Gupta, and G. Wang. A Review of Flowshop Scheduling Research with Setup Times. *Production and Operations Management*, 9(3):262–282, Fall 2000.

[49] T.C.E. Cheng and C.C.S. Sin. A State-of-the-art Review of Parallel-machine Scheduling Research. *European Journal of Operational Research*, 47:271–292, 1990.

[50] I.C. Choi, S.I. Kim, and H.S. Kim. A Genetic Algorithm with a Mixed Region Search for the Asymmetric Traveling Salesman Problem. *Computers & Operations Research*, 30:773–786, 2003.

[51] P. Chrétienne, Jr. E.G. Coffman, J.K. Lenstra, and Z. Liu (eds.). *Scheduling Theory and Its Applications*. Wiley, Chichester, 1995.

[52] N. Christofides. Worst-case Analysis of a New Heuristic for the Traveling Salesman Problem. *Technical Report: 388, GSIA, Carnegie-Mellon University, Pittsburgh, PA*, 1976.

[53] N. Christofides, A. Mingozzi, and P. Toth. State Space Relaxation Procedures for the Computation of Bounds to Routing Problems. *Networks*, 11:145–164, 1981.

[54] F.K.R. Chung, M.R. Garey, and D.S. Johnson. On Packing Two-Dimensional Bins. *SIAM Journal of Algebraic Discrete Mathematics*, 3:66–76, 1982.

[55] G. Clarke and J.W. Wright. Scheduling of Vehicles from a Central Depot to a number of Delivery Points. *Operations Research*, 12(4):568–581, 1964.

[56] E.G. Coffman, M.R. Garey, and D.S. Johnson. Approximation Algorithms for Bin Packing. In G. Ausiello, N. Lucertini, and P. Serafini, editors, *Algorithm Design for Computer Systems Design*, pages 49–106, Springer, Vienna, 1984.

[57] Concorde. Website. http://www.tsp.gatech.edu/concorde/index.html, last accessed 04/07/05.

[58] R.W. Conway, W.L. Maxwell, and L.W. Miller. *Theory of Scheduling.* Addison-Wesley, Reading, M.A., 1967.

[59] H.A.J. Crauwels, A.M.A. Hariri, C.N. Potts, and L.N. Van Wassenhove. Branch and Bound Algorithm for Single Machine Scheduling with Batch Set-Up Times to Minimize Total Weighted Completion Time. *Annals of Operations Research*, 83:59–76, 1998.

[60] H.A.J. Crauwels, C.N. Potts, and L.N. Van Wassenhove. Local Search Heuristics for Single-Machine Scheduling with Batching to Minimize the Number of Late Jobs. *European Journal of Operational Research*, 90:200–213, 1996.

[61] H.A.J. Crauwels, C.N. Potts, and L.N. Van Wassenhove. Local Search Heuristics for Single Machine Scheduling with Batch Set-Up Times to Minimize Total Weighted Completion Time. *Annals of Operations Research*, 70:261–279, 1997.

[62] G.A. Croes. A Method for Solving Traveling Salesman Problem. *Operations Research*, 6(6):791–812, 1958.

[63] G.B. Dantzig, D.R. Fulkerson, and S.M. Johnson. Solution of a Large-scale Traveling-salesman Problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954.

[64] Data Set for 2DRSBSBPP. Website. http://www.or.deis.unibo.it/research_pages/ORinstance/2BP.html, last accessed 19/10/04.

[65] L. Davis. Job Shop Scheduling with Genetic Algorithms. In J.J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pages 136–140, Mahwah, NJ, USA, 1985. Lawrence Erlbaum Associates, Inc.

[66] L. Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, New York, USA, 1991.

[67] J.M. Valério de Carvalho. Exact Solution of Bin-Packing Problems using Column Generation and Branch and Bound. *Annals of Operations Research*, 86:629–659, Jan. 1999.

[68] J.M. Valério de Carvalho. LP models for Bin Packing and Cutting Stock Problems. *European Journal of Operational Research*, 141(2):253–273, 2002.

[69] M. Dell'Amico, S. Martello, and D. Vigo. A Lower Bound for the Non-Oriented Two-Dimensional Bin Packing Problem. *Discrete Applied Mathematics*, 118:13–24, 2002.

[70] E.V. Denardo. *Dynamic Programming: Models and Applications.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1982.

[71] M. Desrochers, J. Desrosiers, and M.M. Solomon. A New Optimization Algorithm for the Vehicle Routing Problem with Time Windows. *Operations Research*, 40(2):342–354, 1992.

[72] D.L. Applegate, R. Bixby, V. Chvátal, and W. Cook. Website. http://www.tsp.gatech.edu/history/milestone.html, last accessed 04/07/05.

[73] M. Dorigo. *Optimization, Learning, and Natural Algorithms.* PhD thesis, Politecnico di Milano, 1992.

[74] M. Dorigo, G. Di Caro, and L.M. Gambardella. Ant Algorithms for Discrete Optimization. *Artificial Life*, 5(3):137–172, 1999.

[75] K.A. Dowsland. Simulated Annealing. In C.R. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems (Advance Topics in Computer Science)*, pages 20–69, UK, 1995. McGRAW-HILL.

[76] K.A. Dowsland. GAs: a Tool for OR? *Journal of the Operational Research Society*, 47:550–561, 1996.

[77] K.A. Dowsland. Classical Techniques. In E.K. Burke and G.Kendall, editors, *Search Methodologies. Introductory Tutorials in Optimization and Decision Support Techniques*, pages 19–68, New York, USA, 2005. Springer.

[78] K.A. Dowsland and W.B. Dowsland. Packing Problems. *European Journal of Operational Research*, 56:2–14, 1992.

[79] R.A. Dudek, S.S. Panwalker, and M.L. Smith. The Lessons of Flowshop Scheduling Research. *Operations Research*, 40(1):7–13, 1992.

[80] Y. Dumas, J. Desrosiers, E. Gelinas, and M.M. Solomon. An Optimal Algorithm for the Traveling Salesman Problem with the Time Windows. *Operations Research*, 43(2):367–371, 1995.

[81] S. Dunstall, A. Wirth, and K. Baker. Lower Bounds and Algorithms for Flowtime Minimization on a Single Machine with Set-Up Times. *Journal of Scheduling*, 3:51–69, 2000.

[82] H. Dyckhoff. A Typology of Cutting and Packing Problems. *European Journal of Operational Research*, 44:145–159, 1990.

[83] H. Dyckhoff and U. Finke. *Cutting and Packing in Production and Distribution.* Physica Verlag, Heidelberg, 1992.

[84] H. Dyckhoff, G. Scheithauer, and J. Terno. Cutting and Packing (C&P). In M. Dell'Amico, F. Maffioli, and S. Martello, editors. *Annotated Bibliographies in Combinatorial Optimization*, pages 393–412, 1997. John Wiley & Sons, Chichester.

[85] W.L. Eastman. *Linear Programming with Pattern Constraints*. PhD thesis, Harvard University, Cambridge, MA., 1958.

[86] G.W. Evans. An Overview of Techniques for Solving Multiobjective Mathematical Programs. *Management Science*, 30:1268–1282, 1984.

[87] E. Falkenauer. A New Representation and Operators for Genetic Algorithms Applied to Grouping Problems. *Evolutionary Computation*, 2:123–144, 1994.

[88] E. Falkenauer. A Hybrid Grouping Genetic Algorithm for Bin Packing. *Journal of Heuristics*, 2:5–30, 1996.

[89] E. Falkenauer and S. Bouffouix. A Genetic Algorithm for Job Shop. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pages 824–829, Sacramento, California, April 1991.

[90] E. Falkenauer and A. Delchambre. A Genetic Algorithm for Bin-Packing and Line Balancing. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, volume 2, pages 1186–1192, Nice, France, 1992.

[91] S. Fekete and J. Schepers. On More-Dimensional Packing II: Bounds. *Technical Report: 97.289, Angewandte Mathematik und Informatik, Universität zu Köln*, 2000.

[92] S. Fekete and J. Schepers. New Classes of Fast Lower Bounds for Bin Packing Problems. *Mathematical Programming*, 91:11–31, 2001.

[93] T.A. Feo and M.G.C. Resende. A Probabilistic Heuristic for a computationally difficult Set Covering Problem. *Operations Research Letters*, 8:67–71, 1989.

[94] T.A. Feo and M.G.C. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[95] F. Focacci, M. Milano, and A. Lodi. Solving TSP with Time Windows with Constraints. In *Proceedings of the 1999 International Conference on Logic Programming*, pages 515–529, Las Cruces, New Mexico, USA, 1999. MIT Press.

[96] F. Focacci, M. Milano, and A. Lodi. Cutting Planes in Constraint Programming: A Hybrid Approach. In R. Dechter, editor, *LNCS: 6th International Conference of Principle and Practice of Constraint Programming – CP2000*, volume 1894, pages 187–201, Singapore, Sept. 2000. Springer-Verlag, Berlin Heidelberg.

[97] F. Focacci, M. Milano, and A. Lodi. A Hybrid Exact Algorithm for the TSPTW. *INFORMS Journal on Computing*, 14(4):403–417, Fall 2002.

[98] F. Focacci, M. Milano, and A. Lodi. Embedding Relaxations in Global Constraints for Solving TSP and TSPTW. *Annals of Mathematics and Artificial Intelligence*, 34(4):291–311, April 2002.

[99] T.C. Fogarty. Varying the Probability of Mutation in the Genetic Algorithm. In J.D. Schaffer, editor, *Proceedings of the Third International Conferences on Genetic Algorithms*, pages 104–109, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers.

[100] D.B. Fogel. An Evolutionary Approach to the Traveling Salesman Problem. *Biological Cybernetics*, 60:139–144, 1988.

[101] D.B. Fogel. A Parallel Processing Approach to a Multiple Traveling Salesman Problem Using Evolutionary Programming. In L. Canter, editor, *Proceedings on the Fourth Annual Parallel Processing Symposium*, pages 318–326, Fullterton, CA, 1990.

[102] D.B. Fogel. Applying Evolutionary Programming to Selected Traveling Salesman Problems. *Cybernetics and Systems*, 24:27–36, 1993.

[103] L.R. Foulds. The Heuristic Problem-Solving Approach. *Journal of the Operational Research Society*, 34(10):927–934, Oct. 1983.

[104] M.S. Fox and M.B. McMahon. Genetic Operators for Sequencing Problems. In G. Rawlings, editor, *Foundations of Genetic Algorithms: First Workshop on the Foundations of Genetic Algorithms and Classifier Systems*, pages 284–300, Los Altos, CA, 1987. Morgan Kaufmann Publishers.

[105] B. Freisleben and P. Merz. A Genetic Local Search Algorithm for Solving Symmetric and Asymmetric Traveling Salesman Problem. In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, pages 616–621, Nagoya, Japan, May 20-22 1996.

[106] J.B. Frenk and G.G. Galambos. Hybrid Next-Fit Algorithm for the Two-Dimensional rectangle Bin-Packing Problem. *Journal on Computing*, 39:201–217, 1987.

[107] T.D. Fry, R.D. Armstrong, and H. Lewis. A Framework for Single Machine Multiple Objective Sequencing Research. *OMEGA*, 17:595–607, 1989.

[108] M.R. Garey and D.S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, San Francisco, 1979.

[109] M. Gendreau, A. Hertz, and G. Laporte. New Insertion and Postoptimisation Procedures for the Traveling Salesman Problem. *Operations Research*, 40(6):1086–1094, Nov.-Dec. 1992.

[110] M. Gendreau, A. Hertz, G. Laporte, and M. Stan. A Generalized Insertion Heuristic for the Traveling Salesman Problem with Time Windows. *Operations Research*, 46(3):330–335, May.-June. 1998.

[111] J.B. Ghosh and J.N.D. Gupta. Batch Scheduling to Minimize Maximum Lateness. *Operations Research Letters*, 21:77–80, 1997.

[112] P.C. Gilmore and R.E. Gomory. A Linear Programming approach to the Cutting Stock Problem. *Operations Research*, 9(6):849–859, 1961.

[113] P.C. Gilmore and R.E. Gomory. A Linear Programming approach to the Cutting Stock Problem - Part II. *Operations Research*, 11(6):863–888, 1963.

[114] P.C. Gilmore and R.E. Gomory. Multistage Cutting Stock Problem of Two and more Dimensions. *Operations Research*, 13(1):94–120, 1965.

[115] F. Glover. Heuristics for Integer Programming using Surrogate Constraints. *Decision Sciences*, 8:156–166, 1977.

[116] F. Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers & Operations Research*, 13:533–549, 1986.

[117] F. Glover. Tabu Search - Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[118] F. Glover. Tabu Search - Part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

[119] F. Glover. A Template for Scatter Search and Path Relinking. In J.K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution, Lecture Notes in Computer Science*, volume 1363, pages 13–54. Springer, 1998.

[120] F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, Boston, 1997.

[121] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, USA, 1989.

[122] D.E. Goldberg. Sizing Populations for Serial and Parallel Genetic Algorithms. In J.D. Schaffer, editor, *Proceedings of the Third International Conferences on Genetic Algorithms*, pages 70–79, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers.

[123] D.E. Goldberg. A Note on Boltzmann Tournament Selection for Genetic Algorithms and Population-Oriented Simulated Annealing. *Complex Systems*, 4:445–460, 1990.

[124] D.E. Goldberg and K. Deb. A Comparative Analysis of Selection Schemes used in Genetic Algorithms. In G.J.E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991.

[125] D.E. Goldberg, B. Korb, and K. Deb. Messy Genetic Algorithms: Motivation, Analysis and First Results. *Complex Systems*, 3:493–530, 1989.

[126] D.E. Goldberg and R. Lingle. Alleles, Loci and the Traveling Salesman Problem. In *Proceedings of the International Conferences on Genetic Algorithms and Their Applications*, pages 154–159, 1985.

[127] B.L. Golden and W.R. Stewart. Empirical Analysis of Heuristics. In E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, pages 207–249, Chichester, UK, 1985. John Wiley & Sons Ltd.

[128] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and Approximation in Deterministic Machine Scheduling: A Survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[129] J.J. Grefenstette. Optimisation of Control Parameters for Genetic Algorithms. *IEEE Transactions on Systems, Man and Cybernatics*, 16(1):122–128, Jan./Feb. 1986.

[130] J.J. Grefenstette. Incorporating Problem Specific Knowledge into Genetic Algorithms. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, pages 42–60, Los Altos, CA., 1987. Morgan Kaufmann Publishers.

[131] J.J. Grefenstette, R. Gopal, B. Rosmaita, and D. Van Gucht. Genetic Algorithms for Traveling Salesman Problem. In J.J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and Their*

Applications, pages 160–168, Mahwah, NJ, USA, 1985. Lawrence Erlbaum Associates, Inc.

[132] J.N.D. Gupta. Single Facility Scheduling with Multiple Job Classes. *European Journal of Operational Research*, 33:42–45, 1988.

[133] S.K. Gupta and J. Kyparisis. Single Machine Scheduling Research. *OMEGA: International Journal of Management Science*, 15:207–227, 1987.

[134] P. Hansen and N. Mladenović. An Introduction to Variable Neighborhood Search. In S. Voß, S. Martello, I.H. Osman, and C. Roucairol, editors, *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, pages 433–458, Boston, MA., 1999. Kluwer Academic Publishers.

[135] P. Hansen and N. Mladenović. Developments of Variable Neighborhood Search. In C. Ribeiro and P. Hansen, editors, *Essays and Surveys in Metaheuristics*, pages 415–440, Dordrecht, 2001. Kluwer Academic Publishers.

[136] P. Hansen and N. Mladenović. Variable Neighborhood Search. *European Journal of Operational Research*, 130(3):449–467, May 2001.

[137] P. Hansen and N. Mladenović. Variable Neighborhood Search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 145–184, Dordrecht, 2003. Kluwer Academic Publishers.

[138] P. Hansen and N. Mladenović. Variable Neighborhood Search. In E.K. Burke and G. Kendall, editors, *Search Methodologies. Introductory Tutorials in Optimization and Decision Support Techniques*, pages 211–238, New York, USA, 2005. Springer.

[139] A.M.A. Hariri and C.N. Potts. Single Machine Scheduling with Batch Setup Time to Minimize Maximum Lateness. *Annals of Operations Research*, 70:75–92, 1997.

[140] J.P. Hart and A.W. Shogan. Semi-Greedy Heuristics: An Empirical Study. *Operations Research Letters*, 6:107–114, 1987.

[141] M. Held and R.M. Karp. A Dynamic Programming Approach to Sequencing Problems. *SIAM Journal on Applied Mathematics*, 10(1):196–210, March 1962.

[142] M. Held and R.M. Karp. The Traveling-Salesman Problem and Minimum Spanning Trees. *Operations Research*, 18(6):1138–1162, Nov.-Dec. 1970.

[143] M. Held and R.M. Karp. The Traveling-Salesman Problem and Minimum Spanning Trees: Part II. *Mathematical Programming*, 1:6–25, 1971.

[144] J.W. Herrmann and C.Y. Lee. Solving a Class Scheduling Problem with a Genetic Algorithm. *ORSA Journal on Computing*, 7(4):443–452, 1995.

[145] J. Hesser and R. Männer. Towards an Optimal Mutation Probability for Genetic Algorithms. In H.P. Schwefel and R. Männer, editors, *Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, pages 23–32, London, UK, 1991. Springer-Verlag.

[146] A.J. Hoffman and P.Wolfe. History. In E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, editors, *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, pages 207–249, Chichester, UK, 1985. John Wiley & Sons Ltd.

[147] J.H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, (reprinted by mit press, 1992) edition, 1975.

[148] A. Homaifar, S. Guan, and G.E. Liepins. A New Approach on the Traveling Salesman Problem by Genetic Algorithm. *Technical Report, North Carolina A&T State University*, 1991.

[149] H. Hoogeveen. Multicriteria Scheduling. *European Journal of Operational Research*, 167:592–623, 2005.

[150] J.A. Hoogeveen, J.K. Lenstra, and S.L. van de Velde. Sequencing and Scheduling: an annotated bibliography. *Memorandum COSOR 97-02, Eindhoven University of Technology*, 1997.

[151] E. Hopper and B.C.H. Turton. A Genetic Algorithm for a 2D Industrial Packing Problem. *Computers & Industrial Engineering*, 37:375–378, 1999.

[152] E. Hopper and B.C.H. Turton. A Review of the Application of Meta-Heuristic Algorithms to 2D Strip Packing Problems. *Artificial Intelligence Review*, 16:257–300, 2001.

[153] E. Hopper and B.C.H. Turton. An Empirical Investigation of Meta-Heuristic and Heuristic Algorithms for a 2D Packing Problem. *European Journal of Operational Research*, 128:34–57, 2001.

[154] S.M. Hwang, Y.K. Cheng, and J.T. Horng. On Solving Rectangle Bin Packing Problems using Genetic Algorithms. In *Proceedings of the 1994 IEEE International Conference on Systems, Man, and Cybernetics. Part 2 (of 3)*, pages 1583–1590, San Antonio, TX, USA, 1994.

[155] H. Iima and T. Yakawa. A New Design of Genetic Algorithm for Bin Packing. In *Proceedings of the 2003 Congress on Evolutionary Computations, CEC2003. Vol.1-4*, volume 2, pages 1044–1049, Canberra, Australia, Dec. 2003.

[156] J.R. Jackson. Scheduling a Production Line to Minimize Maximum Tardiness. *Research Report 43, Management Science Research Project, University of California, Los Angeles, CA*, 1955.

[157] A.S. Jain and S. Meeran. A State-Of-The-Art Review of Job-Shop Scheduling Techniques. *Technical Report: Department of Applied Physics, Electronic and Mechanical Engineering, University of Dundee, UK*, 1998.

[158] S. Jakobs. On Genetic Algorithms for the Packing of Polygons. *European Journal of Operational Research*, 88:165–181, 1996.

[159] G.A. Jayalakshmi, S. Sathiamoorthy, and R. Rajaram. A Hybrid Genetic Algorithm - a new Approach to solve Traveling Salesman Problem. *International Journal of Computational Engineering Science*, 2(2):339–355, 2001.

[160] P. Jog, J.Y. Suh, and D. Van Gucht. The Effects of Population Size, Heuristic Crossover and Local Improvement on a Genetic Algorithm for the Traveling Salesman Problem. In J. Schaffer, editor, *Proceedings of the Third International Conferences on Genetic Algorithms*, pages 110–115, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers.

[161] D.S. Johnson and L.A. McGeoch. The Traveling Salesman Problem: A Case Study in Local Optimization. In E. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310, UK, 1997. John Wiley & Sons Ltd.

[162] A. Jones and L.C. Rabelo. Survey of Job Shop Scheduling Techniques. *Technical Report: National Institute of Standards and Technology, Gaithersburg, MD*, 1998.

[163] K.A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, University of Michigan, 1975.

[164] M. Jünger, G. Reinelt, and G. Rinaldi. The Traveling Salesman Problem. *Report No.92.113, Angewandte Mathematik und Informatik, Universität zu Köln, Cologne, Germany*, 1994.

[165] T. Kämpke. Simulated Annealing: use of a new tool in Bin Packing. *Annals of Operations Research*, 16:327–332, 1988.

[166] R.M. Karp. A Patching Algorithm for the Nonsymmetric Traveling-Salesman Problem. *SIAM Journal on Computing*, 8(4):561–573, Nov. 1979.

[167] K. Katayama, H. Hirabayashi, and H. Narihisa. Performance Analysis of a New Genetic Cross-over for the Traveling-Salesman Problem. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E81-A(5):738–750, 1998.

[168] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 13 May 1983.

[169] J. Knox. *The Application of Tabu Search to the Symmetric Traveling Salesman Problem.* PhD thesis, College of Business and Administration, University of Colorado, Boulder, CO., 1989.

[170] J. Knox. Tabu Search Performance on Symmetric Traveling Salesman Problem. *Computers & Operations Research*, 21(8):867–876, 1994.

[171] J. Knox and F. Glover. Comparative testing of Traveling Salesman Heuristics derived from Tabu Search, Genetic Algorithms and Simulated Annealing. *Working Paper, California State Polytechnic University*, September 1989.

[172] B. Köger. Guillotineable Bin Packing: A Genetic Approach. *European Journal of Operational Research*, 84:645–661, 1995.

[173] W. Kubiak, C. Sriskandarajah, and K. Zaras. A note on the Complexity of Openshop Scheduling Problems. *INFOR*, 29:284–294, 1991.

[174] M. Laguna and R. Martí. *Scatter Search – Methodology and Implementations in C.* Kluwer Academic Publishers, Boston, 2003.

[175] A. Langevin, M. Desrochers, J. Desrosiers, and F. Soumis. A Two-Commodity Flow Formulation for the Traveling Salesman and Makespan Problem with Time Windows. *Networks*, 23:631–640, 1993.

[176] A. Langevin, F. Soumis, and J. Desrosiers. Classification of Traveling Salesman Problem Formulations. *Operations Research Letters*, 9:127–132, March 1990.

[177] G. Laporte. The Traveling Salesman Problem: An Overview of Exact and Approximate Algorithms. *European Journal of Operational Research*, 59:231–247, 1992.

[178] P. Larrañaga, C.M.H. Kuilpers, R.H. Murga, I. Inza, and S. Dizdarevic. Genetic Algorithms for Traveling Salesman Problem: A Review of Representations and Operators. *Artificial Intelligence Review*, 13:129–170, 1999.

[179] P. Larrañaga, C.M.H. Kuilpers, M. Poza, and R.H. Murga. Decomposing Bayesian Networks: Triangulation of the Moral Graph with Genetic Algorithms. *Statistics and Computing*, 7(1):19–34, Mar. 1997.

[180] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys (eds.). *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons Ltd., Chichester, UK, 1985.

[181] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. Sequencing and Scheduling: Algorithms and Complexity. In S.C. Graces, A.H.G. Rinnooy Kan, and P.H. Zipkin, editors, *Logistic of Production and Inventory, Handbooks in Operations Research and Management Science*, volume 4, pages 445–522, Amsterdam, North-Holland, 1993.

[182] C.Y. Lee and J.Y. Choi. A Genetic Algorithm for Job Sequencing Problems with Distinct Due Dates and General Early-Tardy Penalty Weights. *Computers & Operations Research*, 22(8):857–869, 1995.

[183] J.K. Lenstra and A.H.G. Rinnooy Kan. Some Simple Applications of the Traveling Salesman Problem. *Operational Research Quarterly*, 26:717–733, 1975.

[184] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of Machine Scheduling Problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[185] J.Y-T. Leung. *Handbook of Scheduling. Algorithms, Models, and Performance Analysis*. Chapman & Hall/CRC, US, 2004.

[186] J. Levine and F. Ducatelle. Ant Colony Optimisation and Local Search for Bin Packing and Cutting Stock Problems. *Journal of Operational Research Society*, 55(7):705–716, July 2004.

[187] M.M. Liaee and H. Emmons. Scheduling Families of Jobs with Setup Times. *International Journal of Production Economics*, 51:165–176, 1997.

[188] G.E. Liepins and M.R. Hilliard. GAs: Foundations and Applications. *Annals of Operations Research*, 21:31–58, 1989.

[189] S. Lin. Computer Solutions of the Traveling Salesman Problem. *Bell System Technical Journal*, 44:2245–2269, 1965.

[190] S. Lin and B.W. Kernighan. An Effective Heuristic Algorithm for the Traveling Salesman Problem. *Operations Research*, 21(2):498–516, 1973.

[191] J.D.C. Little, K.G. Murty, D.W. Sweeney, and C. Karel. An Algorithm for the Traveling Salesman Problem. *Operations Research*, 11(6):972–989, 1963.

[192] D. Liu and H. Teng. An Improved BL-Algorithm for Genetic Algorithm of the Orthogonal Packing of Rectangles. *European Journal of Operational Research*, 112:413–420, 1999.

[193] A. Lodi, S. Martello, and D. Vigo. Approximation Algorithms for the Oriented Two-Dimensional Bin Packing Problem. *European Journal of Operational Research*, 112:158–166, 1999.

[194] A. Lodi, S. Martello, and D. Vigo. Heuristic and Metaheuristic Approaches for a Class of Two-Dimensional Bin Packing Problems. *INFORMS Journal on Computing*, 11:345–357, 1999.

[195] A. Lodi, S. Martello, and D. Vigo. Recent Advances on Two-Dimensional Bin Packing Problems. *Technical Report OR/99/2, DEIS - Università di Bologna*, 1999.

[196] A. Lodi, S. Martello, and D. Vigo. Recent Advances on Two-Dimensional Bin Packing Problems. *Discrete Applied Mathematics*, 123:379–396, 2002.

[197] A. Lodi, S. Martello, and D. Vigo. Two-Dimensional Packing Problems: A Survey. *European Journal of Operational Research*, 141:241–252, 2002.

[198] A. Lodi, S. Martello, and D. Vigo. TSpack: A Unified Tabu Search Code for Multi-Dimensional Bin Packing Problems. *Annals of Operations Research*, 131(1-4):203–213, October 2004.

[199] H.R. Lourenço, O. Martin, and T. Stützle. Iterated Local Search. In F. Glover and G. Kochenberger, editors, *Handbook of Meta-Heuristics (International Series in Operations Research and Management Science)*, volume 57, pages 321–353, Norwell, MA, 2002. Kluwer Academic Publishers.

[200] S. Mahfoud. An Analysis of Boltzmann Tournament Selection. *IlliGAL Report 91007, University of Illinois at Urbana-Champaign*, October 1991.

[201] M. Malek, M. Guruswamy, and M. Pandya. Serial and Parallel Simulated Annealing and Tabu Search Algorithms for the Traveling Salesman Problem. *Annals of Operations Research*, 21:59–84, 1989.

[202] S. Martello and P. Toth. Lower Bounds and Reduction Procedures for the Bin Packing Problem. *Discrete Applied Mathematics*, 28(1):59–70, July 1990.

[203] S. Martello and D. Vigo. Exact Solution of the Two-Dimensional Finite Bin Packing Problem. *Management Science*, 44(3):388–399, 1998.

[204] R. Martí, M. Laguna, and F. Glover. Principles of Scatter Search. *European Journal of Operational Research*, 169(2):359–372, 2006.

[205] O. Martin, S.W. Otto, and E.W. Felten. Large-step Markov Chains for the Traveling Salesman Problem. *Complex Systems*, 5:299–326, 1991.

[206] O. Martin, S.W. Otto, and E.W. Felten. Large-step Markov Chains for the TSP incorporating Local Search Heuristics. *Operations Research Letters*, 11:219–224, 1992.

[207] A.J. Mason. *Genetic Algorithms and Scheduling Problems.* PhD thesis, Department of Engineering, University of Cambridge, UK, 1992.

[208] A.J. Mason and E.J. Anderson. Minimizing Flow Time on a Single Machine with Job Classes and Setup Times. *Naval Research Logistics*, 38:333–350, 1991.

[209] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[210] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer Verlag, Berlin, Heidelberg, 1992.

[211] A. Mingozzi, L. Bianco, and S. Ricciardelli. Dynamic Programming Strategies for the Traveling Salesman Problem with Time Windows and Precedence Constraints. *Operations Research*, 45(3):365–377, 1997.

[212] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, USA, 1996.

[213] N. Mladenović and P. Hansen. Variable Neighborhood Search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

[214] C.L. Monma and C.N. Potts. On the Complexity of Scheduling with Batch Setup Time. *Operations Research*, 37(5):798–804, 1989.

[215] H. Mühlenbein, M. Gorges-Schleuter, and O. Krämer. Evolution Algorithms in Combinatorial Optimization. *Parallel Computing*, 7:65–85, 1988.

[216] National TSP. Website. http://www.tsp.gatech.edu/world/countries.html, last accessed 04/07/05.

[217] E. Nowicki and S. Zdrzałka. Single Machine Scheduling with Major and Minor Setup Times: A Tabu Search Approach. *The Journal of Operational Research Society*, 47(8):1054–1064, Aug. 1996.

[218] K.E. Nygard and C.H. Yang. Genetic Algorithms for the Traveling Salesman Problem with Time Windows. In Sharda Balci and Zenios, editors, *Com-*

puter Science and Operations Research. New Development in their Interfaces, pages 411–423, Williamsburg, V.A., 8-10 Jan. 1992. Pergamon Press.

[219] J.W. Ohlmann and B.W. Thomas. A Compressed Annealing Approach to the Traveling Salesman Problem with Time Windows. *INFORMS Journal on Computing*, (to appear).

[220] I.M. Oliver, D.J. Smith, and J.R.C. Holland. A Study of Permutation Crossover Operators on the TSP. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms and Their Applications*, pages 224–230, Hillsdale, New Jersey, 1987. Lawrence Erlbaum Associates, Inc.

[221] I. Or. *Traveling Salesman-Type Combinatorial Problems and Their relation to the Logistics of Regional Blood Banking.* PhD thesis, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL, 1976.

[222] I.H. Osman and J.P. Kelly (editor). *Meta-Heuristic: Theory and Applications.* Kluwer Academic Publishers, Boston, MA., 1996.

[223] I.M. Ovacik and R. Uzsoy. Rolling Horizon Algorithms for a Single-Machine Dynamic Scheduling Problem with Sequence Dependent Setup Times. *International Journal of Production Research*, 32:1243–1263, 1994.

[224] J.C.H. Pan, J.S. Chen, and H.L. Cheng. A Heuristic approach for Single-Machine Scheduling with Due Dates and Class Setups. *Computers & Operations Research*, 28:1111–1130, 2001.

[225] J.C.H. Pan and C.S. Su. Single Machine Scheduling with Due Dates and Class Setups. *Journal of the Chinese Institute of Engineers*, 20(5):561–572, 1997.

[226] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, Reading, MA, 1994.

[227] G. Pesant, M. Genfreau, J.Y. Potvin, and J.M. Rousseau. An Exact Constraint Logic Programming Algorithm for the Traveling Salesman Problem with Time Windows. *Transportation Science*, 32(1):12–29, Feb. 1998.

[228] G. Pesant, M. Genfreau, J.Y. Potvin, and J.M. Rousseau. On the flexibility of Constraint Programming models: from Single to Multiple Time Windows for the Traveling Salesman Problem. *European Journal of Operational Research*, 117(2):253–263, 1999.

[229] M.L. Pinedo. *Scheduling: Theory , Algorithms, and Systems.* Prentice Hall, Englewood Cliffs, NJ., 1995.

[230] D. Pisinger and M. Sigurd. Using Decomposition Techniques and Constraint Programming for Solving the Two-Dimensional Bin Packing Problem. *Technical Report 03/01, University of Copenhagen, Denmark.*, 2003.

[231] G. Polya. *How to Solve it.* Princeton University Press, Princeton, 1945.

[232] C.N. Potts. Scheduling Two Job Classes on a Single Machine. *Computers & Operations Research*, 18:411–415, 1991.

[233] C.N. Potts and M.Y. Kovalyov. Scheduling With Batching: A Review. *European Journal of Operational Research*, 120:228–249, 2000.

[234] C.N. Potts and L.N. Van Wassenhove. Integrating Scheduling with Batching and Lot-Sizing: A Review of Algorithms and Complexity. *The Journal of the Operational Research Society*, 43(5):395–406, May 1992.

[235] J.Y. Potvin. Genetic Algorithms for the Traveling Salesman Problem. *Annals of Operations Research*, 63:339–370, 1996.

[236] J. Puchinger and G.R. Raidl. An Evolutionary Algorithm for Column Generation in Integer Programming: An Effective Approach for 2D Bin Packing. In X. Yao, editor, *Lecture Notes in Computer Science (8th International Conference on Parallel Problem Solving from Nature (PPSN VIII)*, volume 3242, pages 642–651, Birmingham, UK., Sept 18-22, 2004 2004.

[237] J. Puchinger and G.R. Raidl. Models and Algorithms for Three-Stage Two-Dimensional Bin Packing. *Technical Report TR-186-04-04, Technische Universität Wien*, Sept. 2004.

[238] C.R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems (Advance Topics in Computer Science)*. McGRAW-HILL, London, UK, 1995.

[239] C.R. Reeves. Genetic Algorithms for the Operations Researcher. *INFORMS Journal of Computing*, 9(3):231–250, 1997.

[240] C.R. Reeves and J.E. Beasley. Introduction. In C.R. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems (Advance Topics in Computer Science*, pages 1–19, London, UK, 1995. McGRAW-HILL.

[241] G. Reinelt. Fast Heuristics for Large Geometric Traveling Salesman Problems. *ORSA Journal of Computing*, 4:206–217, 1992.

[242] G. Reinelt. The Traveling Salesman Problems: Computational solutions for TSP Applications. *Lecture Notes in Computer Science*, 840, 1994. Springer-Verlag, Berlin.

[243] M.G.C. Resende. Greedy Randomized Adaptive Search Procedures (GRASP). In C. Floudas and P.M. Pardalos, editors, *Encyclopedia of Optimization*, volume 2, pages 373–382. Kluwer Academic Publishers, 2001.

[244] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis. An Analysis of several Heuristics for the Traveling Salesman Problem. *SIAM Journal of Computing*, 6(3):563–581, Sept. 1977.

[245] T.P. Runarsson, M.T. Jonsson, and P. Jensson. Dynamic Dual Bin Packing using Fuzzy Objectives. In *The 1996 IEEE International Conference on Evolutionary Computation, ICEC'96*, pages 219–222, Nagoya, Japan, May 20-22 1996 1996.

[246] M.W.P. Savelsbergh. Local Search in Routing Problem with Time Windows. *Annals of Operations Research*, 4:285–305, 1985.

[247] J.D. Schaffer, R.A. Caruana, L.J. Eshelman, and R. Das. A Study of Control Parameters affecting online performance of Genetic Algorithms for Function Optimization. In J.D. Schaffer, editor, *Proceedings of the Third International Conferences on Genetic Algorithms*, pages 51–60, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers.

[248] L.J. Schmitt and M.M. Amini. Performance Characteristics of Alternative Genetic Algorithmic Approaches to the Traveling Salesman Problem using Path Representation: An Empirical Study. *European Journal of Operational Research*, 108:551–570, 1998.

[249] S.R. Schultz, T.J. Hodgson, R.E. King, and M.R. Taner. Minimizing $L_{\max}$ for the Single Machine Scheduling problem with Family Set-ups. *International Journal of Production Research*, 42(20):4315–4330, October 2004.

[250] J.M.J. Schutten, S.L. van de Velde, and W.H.M. Zijm. Single-Machine Scheduling with Release Dates, Due Dates, and Family Setup Times. *Management Science*, 42(8):1165–1174, 1996.

[251] D. Seniw. *A Genetic Algorithm for the Traveling Salesman Problem.* MSc Thesis, University of North Carolina, Charlotte, 1991.

[252] M. Sevaux and S. Dauzère-Pérès. Genetic Algorithm to Minimize the Weighted Number of Late Jobs on a Single Machine. *European Journal of Operational Research*, 151:296–306, 2003.

[253] H.J. Shin, C.O. Kim, and S.S. Kim. A Tabu Search Algorithm for Single Machine Scheduling with Release Times, Due Dates, and Sequence-Dependent Set-Up Times. *The International Journal of Advanced Manufacturing Technology*, 19:859–866, 2002.

[254] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing, Cambridge, MA, 1997.

[255] D. Smith. Bin-Packing with Adaptive Search. In J.J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, pages 202–206, Mahwah, NJ, USA, 1985. Lawrence Erlbaum Associates, Inc.

[256] W.E. Smith. Various Optimizers for Single-Stage Production. *Naval Research Logistics Quarterly*, 3:59–66, 1956.

[257] S.M. Soak and B.H. Ahn. New Genetic Crossover Operator for the TSP. In L. Rutkowski, J. Siekmann, R. Tadeusiewics, and L.A. Zadeh, editors, *Proceedings of the 7th International Conference of Artificial Intelligence and Soft Computing: ICAISC 2004*, LNCS, pages 480–485, Zakopane, Poland, 7-11, June 2004.

[258] M.M. Solomon. Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraint. *Operations Research*, 35(2):254–265, Mar.-Apr. 1987.

[259] X.Q. Sun, J.S. Noble, and C.M. Klein. Single-Machine Scheduling with Sequence Dependent Setup to Minimize Total Weighted Squared Tardiness. *IIE Transactions*, 31(2):113–124, Feb. 1999.

[260] R.H. Suriyaarachchi and A. Wirth. Earliness/Tardiness Scheduling with a Common Due Date and Family Setups. *Technical Report, School of Computing and Information Technology, University of Western Sydney*, 2003.

[261] G. Syswerda. Uniform Crossover in Genetic Algorithms. In *Proceedings of the Third International Conferences on Genetic Algorithms*, pages 2–9, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers.

[262] G. Syswerda. Schedule Optimization Using Genetic Algorithms. In L. Davis, editor, *Handbook of Genetic Algorithms*, pages 332–349, New York, 1991. Van Nostrand Reinhold.

[263] R. Tanese. Distributed Genetic Algorithms. In *Proceedings of the Third International Conferences on Genetic Algorithms*, pages 434–439, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers.

[264] M. Tomassini. A Survey of Genetic Algorithms. In D. Stauffer, editor, *Annual Reviews of Computational Physics*, volume III, pages 87–118, Singapore, 1995. World Scientific.

[265] C.A. Tovey. Tutorial on Computational Complexity. *Interfaces*, 32(3):30–61, May-June 2002.

[266] TSPLIB. Website. http://www.iwr.uni-heidelberg.de/groups/comopt/soft/TSPLIB95/TSPLIB.html, last accessed 04/07/05.

[267] S. Tsubakitani and J.R. Evans. Optimizing Tabu List Size for the Traveling Salesman Problem. *Computers & Operations Research*, 25(2):91–97, 1998.

[268] R.J.M. Vaessens, E.H.L. Aarts, and J.K. Lenstra. Job Shop Scheduling by Local Search. *INFORMS Journal of Computing*, 8:302–317, 1996.

[269] F. Vanderbeck. Computational Study of a Column Generation Algorithm for Bin Packing and Cutting Stock Problems. *Mathematical Programming*, 86(3):565–594, Dec. 1999.

[270] VLSI TSP. Website. http://www.tsp.gatech.edu/vlsi/index.html, last accessed 04/07/05.

[271] S. Voß, S. Martello, I.H. Osman, and editors C. Roucairol. *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization.* Kluwer Academic Publishers, Boston, MA., 1999.

[272] C. Voudouris and E. Tsang. Guided Local Search and its Application to the Traveling Salesman Problem. *European Journal of Operational Research*, 113:469–499, 1999.

[273] D. Wang, M. Gen, and R. Cheng. Scheduling Grouped Jobs on Single Machine with Genetic Algorithm. *Computer and Industrial Engineering*, 36:309–324, 1999.

[274] G. Wäscher, H. Haußner, and H. Schumann. An Improved Typology of Cutting and Packing Problems. *Working Paper: No. 24. Otto von Guericke University*, last revision: 17th May 2005.

[275] S. Webster and K.R. Baker. Scheduling Groups of Jobs on a Single Machine. *Operations Research*, 43(4):692–703, 1995.

[276] S. Webster, P.D. Jog, and A. Gupta. A Genetic Algorithm for Scheduling Job Families on a Single Machine with Arbitrary Earliness/Tardiness Penalties and an unrestricted common Due Date. *International Journal of Production Research*, 36(9):2543–2551, 1998.

[277] D. Whitley. A Genetic Algorithm Tutorial. *Statistic and Computing*, 4:65–85, 1994.

[278] D. Whitley, T. Starkweather, and D. Fuquay. Scheduling Problems and Traveling Salesman: The Genetic Edge Recombination Operator. In *Proceedings of the Third International Conferences on Genetic Algorithms*, pages 133–140, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers.

[279] D. Whitley and J.P. Watson. Complexity Theory and the No Free Lunch Theorem. In E.K. Burke and G.Kendall, editors, *Search Methodologies. Introductory Tutorials in Optimization and Decision Support Techniques*, pages 317–339, New York, USA, 2005. Springer.

[280] G. Whitwell. *Novel Heuristic and Metaheuristic Approaches to Cutting and Packing.* PhD thesis, School of Computer Science and Information Technology, University of Nottingham, September 2004.

[281] D. Williams and A. Wirth. A New Heuristic for a Single Machine Scheduling Problem with Set-Up Times. *The Journal of the Operational Research Society*, 47(1):175–180, Jan. 1996.

[282] G.J. Woeginger. A Polynomial-Time Approximation Scheme for Single-Machine Sequencing with Delivery Times and Sequence-Independent Batch Set-Up Times. *Journal of Scheduling*, 1:79–87, 1998.

[283] World TSP. Website. http://www.tsp.gatech.edu/world/index.html, last accessed 04/07/05.

[284] M. Yamamura, I. Ono, and S. Kobayashi. Character-Preserving Genetic Algorithm for Traveling Salesman Problem. *Journal of Japanese Society for Artificial Intelligence*, 6:1049–1059, 1992. (in Japanese).

[285] C.H. Yang and K.E. Nygard. The Effect of Initial Population in Genetic Search for Time Constrained Traveling Salesman Problem. In *Proceedings of the 1993 ACM Conference on Computer Science*, pages 378–383, Indianapolis, Indiana, US, 1993. ACM, ACM Press, New York.

[286] W.H. Yang and G.J. Liao. Survey of Scheduling Research Involving Setup Times. *International Journal of System Science*, 30(2):143–155, 1999.

[287] S. Zdrzałka. Approximation Algorithms for Single Machine Sequencing with Delivery Times and Unit Batch Setup Times. *European Journal of Operational Research*, 51:199–209, 1991.

[288] S. Zdrzałka. Analysis of Approximation Algorithms for Single-Machine Scheduling with Delivery Times and Sequence Independent Batch Setup Times. *European Journal of Operational Research*, 80:371–380, 1995.