UNIVERSITY OF SOUTHAMPTON

FACULTY OF ENGINEERING, SCIENCE & MATHEMATICS

School of Mathematics

Approaches for solving some scheduling and routing problems

by

Andrew John Drake

Thesis for the degree of Doctor of Philosophy

February 2009

University of Southampton

Abstract

Faculty of Engineering, Science and Mathematics
School of Mathematics

Doctor of Philosophy

Approaches for solving some scheduling and routing problems

by Andrew John Drake

We study approaches for finding good solutions, and lower bounds, for three difficult combinatorial optimisation problems.

The *supply ship travelling salesman problem* is a simplification of a situation faced by a naval logistics coordinator who must direct a support vessel tasked with resupplying ships in a fleet. It is a generalisation of the travelling salesman problem in which the nodes are in motion, each following some predetermined route. We apply dynamic programming state-space relaxation techniques, producing lower bounds for the problem that are 73% to 84% of the best solution, on average. We also apply heuristics to find good solutions to this NP-hard problem, showing that restricted dynamic programming approaches outperform simple 2-opt and 3-opt local search procedures for instances with 20 nodes.

We introduce the *supply ship scheduling problem*, another problem inspired by a support vessel environment. We wish to minimise the number of mobile machines required to process a set of jobs; each job is in a different stationary location and features a fixed start time. Jobs may be simultaneously processed by multiple machines, obtaining a speed-up in processing time. We represent the problem as a directed graph and use the minimum flow in a transformed network to determine the minimum number of machines. We present a neighbourhood structure based on the maximum cut, applying it within descent and tabu search procedures. We construct a restricted dynamic programming based approach, but this is outperformed by the tabu search algorithm.

The *task allocation problem*, arising in distributed computing, is to assign a set of tasks to a set of processors so that the overall cost is minimised. Costs are incurred from processor usage, interprocessor communication and task execution. We construct, and try to improve, semidefinite programming relaxations to find lower bounds for variants of this NP-hard problem. We develop a branch-and-bound approach to find optimal solutions, but this is only effective for small instances.

Contents

List of figures

List of tables

Declaration of authorship

I, Andrew John Drake, declare that the thesis entitled 'Approaches for solving some scheduling and routing problems' and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. 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 been previously 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: .....

Acknowledgements

I would like to thank my supervisors and advisors Professor Chris Potts, Dr. Jonathan Whitehead and Dr. Miguel Anjos for all their help and advice.

Chapter 1

Introduction

This thesis focuses on approaches for three combinatorial optimisation problems. Combinatorial optimisation problems are concerned with finding the best solution from among a discrete set of solutions. Evaluation of all possible solutions, known as complete enumeration, is usually time-consuming so methods that quickly find optimal or near-optimal solutions are desirable. Some methods rely on effective bounds on the optimal value, so techniques to determine tight lower bounds can prove useful.

Two of our combinatorial optimisation problems are complex scheduling and routing problems. Both have been inspired by MSc projects for the Defence Science and Technology Laboratory. Scheduling and routing are widely studied areas and feature important classes of problems. Scheduling is applied in many situations where a sensible ordering of activities and allocation of resources is required [103]; examples include manufacturing, computer processing and exam timetabling. Routing deals with the selection of paths within networks and is commonly applied to telephone, data or transport networks. A classical problem within these related fields is the travelling salesman problem [60]: a salesman, starting at his home city, must

visit each city within a given set exactly once before returning home, while minimising the total distance he must travel. Although simply described, this problem is not always easy to solve. It belongs to the NP-hard class of problems. Problems in this class are usually tackled by heuristic algorithms that obtain near-optimal solutions within a reasonable amount of time.

The first of our problems is a special variant of the travelling salesman problem in which the cities are in motion, so distances vary with time. The following problem we tackle is a new scheduling and routing problem with an interesting combination of features. The final problem studied, although not closely related to the previous two, is an important one within the world of distributed computing and micro-processor systems.

1.1 Problem areas

1.1.1 The supply ship travelling salesman problem

The supply ship travelling salesman problem is the name we have given to a generalisation of the travelling salesman problem in which the nodes are in motion. Each node, or warship, follows some predetermined route. A supply ship must visit and resupply all the warships in the fleet while they patrol an area of sea. We wish to find near-optimal solutions to this NP-hard problem using heuristic methods.

We first adapt dynamic programming state-space relaxation techniques for the classical travelling salesman problem [27] to produce lower bounds for the supply ship travelling salesman problem. The bounds found are reasonable, but not strong enough to embed in an effective branch-and-bound scheme.

A restricted dynamic programming heuristic for the time dependent travelling salesman problem [92] is applied to the supply ship travelling salesman problem. We suggest an improved measurement for the evaluation of partial tours. Our proposal for another measure, that predicts final tour costs using the nearest neighbour heuristic, is also considered.

Simple 2-opt and 3-opt local search approaches are tested, demonstrating that our variant of the restricted dynamic programming heuristic produces superior results.

1.1.2 The supply ship scheduling problem

We introduce the supply ship scheduling problem. This problem is concerned with minimising the number of mobile machines (supply ships) required to process a set of jobs (warships/docks). Each job is in a fixed location and must commence at a fixed start time. Jobs may be simultaneously processed by multiple machines, obtaining a speed-up in processing time.

The problem may be represented as a directed graph. Objective values are calculated by determining the minimum flow in a transformed network. We present a neighbourhood structure that restricts the search to moves likely to improve the solution, based on information provided by a maximum cut in the network. This structure is successfully applied within a number of simple local search procedures to produce solutions rapidly. The neighbourhood structure is then utilised within a tabu search metaheuristic with encouraging results.

In an alternative approach, we construct a heuristic for the supply ship scheduling problem based on the ideas behind restricted dynamic programming. Our initial formulation is less effective than the tabu search,

but may be used to find machine assignments to jobs when the total number of machines is fixed. We indicate an improved formulation, but this has not been implemented.

1.1.3 The task allocation problem

Task allocation problems arise in distributed computing (including air defence radar systems [89]) and micro-processor sub-systems in car manufacturing [106]. In a task allocation problem, a set of tasks must be assigned to a set of processors so that the overall cost is minimised. Costs are incurred through the use of processors, interprocessor communication and the execution of a task on a particular processor. A number of NP-hard variants of this problem appear in the literature, where there has been some success in using linear programming relaxations to find lower bounds [47].

We construct semidefinite programming relaxations to find lower bounds for some of these related problems. A number of heuristics for selection of tasks to use in a partial higher lifting approach [6] are proposed, but we find that the technique does not provide significant improvements to our bound.

We implement a branch-and-bound approach to find global lower bounds, proposing some heuristics for the selection of branching variable. Those heuristics based on analysis of communication costs were found to be the most effective.

1.2 Organisation of the thesis

The remainder of this thesis is organised as follows. Chapter 2 provides an overview of some of the methods used within the field of combinatorial optimisation. Chapter 3 concerns the supply ship travelling salesman problem while Chapter 4 deals with the supply ship scheduling problem. Chapter 5 tackles the task allocation problem using semidefinite programming. Finally in Chapter 6, we draw together our conclusions on the approaches for the problems we have studied.

Chapter 2

Combinatorial optimisation methods

2.1 Introduction

In an optimisation problem we wish to find feasible solutions that minimise or maximise the value of an objective function. In a *combinatorial optimisation* problem feasible solutions are made up of a number of discrete choices; there are a countable number of alternatives. Thus in a minimisation problem with objective function $f : X \to \mathbb{R}$, where X is the discrete set of solutions, we wish to find $x^* \in X$ such that $f(x^*) \leq f(x)$ for all $x \in X$.

In this chapter, we give a brief overview of some of the methods used to solve combinatorial optimisation problems. *Exact methods* may be used to find optimal solutions, but in some cases it is more appropriate to determine near-optimal solutions using *heuristic* algorithms. The theoretical complexity of a combinatorial optimisation problem can help us determine whether it is appropriate to apply exact approaches.

2.2 Computational complexity theory

Complexity theory seeks to classify problems, discovering relationships between them to gain insight into their difficulty. The computational complexity of a problem can indicate whether we should try to find an efficient algorithm to solve it, or instead devote our efforts to developing effective procedures to obtain good, but not necessarily optimal, solutions.

An algorithm's *time complexity function*, $\tau(n)$, gives the largest amount of time the algorithm requires to solve a problem for each problem size n. We say such a function is $O(p(n))$ whenever there exists a constant c such that

$$|\tau(n)| \leq c \cdot |p(n)|$$

for all values of $n \geq 0$. If p is a polynomial we say that the problem may be solved in *polynomial time*, otherwise we describe the algorithm as *exponential*. We usually regard polynomial time algorithms as more desirable; execution times for exponential time algorithms may suffer from explosive growth as input length increases. However, time complexity is a worst case measure and a few exponential algorithms are useful in practice. The empirical running time for an exponential algorithm may be better than that for an algorithm with polynomial worst case behaviour; for example, the polynomial time *ellipsoid algorithm* for linear programming is very slow in practice and does not compete well with running times for the exponential *simplex algorithm*.

Solving an instance of a *decision problem* results in one of only two possible answers: "yes" or "no". Algorithms for solving a problem may be modelled using a *Turing machine*. A *deterministic* Turing machine allows a single calculation at a time. A *non-deterministic* Turing machine allows an exponential number of parallel calculations. The class P contains decision problems that may be solved in polynomial time by a deterministic Turing

machine. The class NP contains decision problems that may be solved in polynomial time by a non-deterministic Turing machine. If it is possible to verify in polynomial time that a guess solution results in a "yes" or "no" answer then the problem belongs to NP.

It is clear that $P \subseteq NP$ since any problem solvable by a polynomial time deterministic algorithm must also be solvable by a polynomial time non-deterministic algorithm. It is widely believed that $P \neq NP$, but this conjecture has not been proven.

The class of NP-*complete* decision problems consists of the "hardest" members of NP: every problem in NP can be "reduced" to a problem in the NP-complete class using some polynomial time transformation (*Cook's theorem*). Such a transformation maps any instance of the NP problem into an instance of the NP-complete problem. This means that if a polynomial time algorithm is found that can solve an NP-complete problem, all problems in NP may be solved using a polynomial time algorithm, proving that $P = NP$. If it is shown that a problem in NP cannot be solved by any polynomial time algorithm, then no NP-complete problem can be solved by a polynomial time algorithm. That is, if $P \neq NP$ then NP-complete problems belong to $NP - P$.

The *SAT* problem was shown to be NP-complete by Cook [53]. We may prove a decision problem in NP is NP-complete by giving a *polynomial transformation* to it from an existing NP-complete problem. Any decision problem that can be polynomially transformed from an NP-complete problem may be described as NP-*hard*, whether it is in NP or not. Such a problem is *at least as hard* as the NP-complete problems. An NP-hard problem cannot be solved in polynomial time unless $P = NP$.

An algorithm runs in pseudo-polynomial time if its running time is polynomial in the numeric value of the input. An NP-hard problem is called

weakly NP-hard if it may be solved by a pseudo-polynomial time algorithm. If it is proven that an NP-hard problem cannot be solved by a pseudo-polynomial time algorithm, it is *strongly NP-hard*.

In an *optimisation problem* the goal is to find the best solution from all feasible solutions. Each optimisation problem has a corresponding decision problem which asks whether there is a feasible solution matching a particular measure. An optimisation problem is NP-hard if its corresponding decision problem is NP-complete. Thus NP-hard combinatorial optimisation problems are at least as hard as NP-complete decision problems.

When faced with an NP-hard optimisation problem, it is unlikely that we will find a polynomial time exact algorithm to compute the optimal solution. For all but the smallest problems we settle for a solution that is close to optimal. Although we may use heuristics for this purpose, *approximation algorithms* produce a solution that is guaranteed to be within some factor of the optimal solution. To prove that the *approximation factor* is valid it is important to derive good bounds: lower bounds if it is a minimisation problem and upper bounds if it is a maximisation problem.

Extensive descriptions regarding complexity theory may be found in the texts of Garey and Johnson [53] and Papadimitriou [101].

2.3 Exact approaches

In a *complete enumeration* every possible solution to the problem is evaluated; this method is impractical for all but the smallest problems. By using information specific to the problem instance the number of evaluations may be reduced to a manageable level.

2.3.1 Branch-and-bound

A combinatorial optimisation problem may be conceptualised as a a decision tree. The root node represents the problem and its set of possible solutions. *Branches* from a node represent a choice. *Nodes* represent sub-problems resulting from the choices made on the branches forming the path from the root to the node. The further down the tree, the smaller the sub-problem. A node on the final level, a *leaf* node, represents a sub-problem where all decisions have been made; there is a single solution and no further branching from the node.

To avoid complete enumeration, a *branch-and-bound* method [77] begins with the root node and looks at its immediate branches and nodes, placing the sub-problems in a list that gives the order in which they will be investigated. A bound on the objective value for a sub-problem is calculated as the node is investigated (an upper bound for a maximisation problem and lower bound for a minimisation problem). This bound is compared to the objective value of a *trial solution*. If the bound reveals that the sub-problem must have an optimal solution worse than the objective value of the trial solution, the node is *pruned*: further branching from the node may be ignored and the sub-problem is said to be *fathomed*. Otherwise, the branches from the current node are followed and its immediate sub-problems are added to the 'order of investigation' list. The process continues until the list is empty. The trial solution used is replaced whenever a better solution is discovered.

The efficiency of the method relies on the tightness of the bound given by the bounding procedure, the ease with which it is calculated, the quality of the trial solution and the strategy employed to order new nodes within the investigation list.

2.3.2 Dynamic programming

In a *dynamic programming* approach [14] the problem is broken into sub-problems. By recursively building on the optimal solutions to the smallest sub-problems an optimal solution to the problem is reached. Dynamic programming approaches feature the following characteristics [104, 116]:

- The problem may be divided into *stages*, each requiring a *decision*.
- For each stage there a number of *states*.
- The *decision* transforms a state into a state belonging to another stage.
- Given the current state, optimal future decisions do not depend on the decisions made previously. (*Principle of optimality*).
- A *recursion* identifies optimal decisions for the current stage by relating it to stages that have already been solved.
- An initial stage is easily solved.

There may be several possible formulations for a problem. While *forward recursions*, starting at the first stage and working forward through decisions to the last stage, may seem more natural, *backward* procedures may also be constructed.

2.3.3 Integer programming

Many combinatorial optimisation problems may be formulated as *mathematical programs* in which the variables are constrained to be integer.

A column vector, $\mathbf{x}$, of n variables may be used to formulate a *linear* integer program with m equality constraints [5]:

$$\begin{aligned} \min \quad & \mathbf{c}^T\mathbf{x} \\ \text{s.t.} \quad & A\mathbf{x} = \mathbf{b} \\ & \mathbf{x} \geq 0 \\ & x_j \text{ integer for some (or all) } j \end{aligned}$$

where A is an $m \times n$ matrix of real numbers, and $\mathbf{b}$ and $\mathbf{c}$ are column vectors of real numbers.

If the integer constraints on the variables are removed we obtain the *linear programming (LP) relaxation*. The LP relaxation may be solved using an algorithm such as the *simplex* method, but this will not always provide an optimal solution that is integer. The optimal value for the LP relaxation provides a lower bound (for a minimisation problem) on the optimal value for the integer program.

Integer solutions may be found by constructing a branch-and-bound tree. At each node a *candidate problem* is solved consisting of the original LP relaxation plus constraints generated by branching. Nodes are pruned if their candidate problems are found to be infeasible or their optimal values are worse than the current best upper bound (for a minimisation problem) on the true optimal value.

The *branch-and-cut* method [94] is an extension to the branch-and-bound method that applies *cutting planes* to the candidate problems. By generating cuts and resolving the LP relaxation at each node, it is hoped

that this results in a shorter search tree. Cutting planes are linear inequalities that are included in the set of constraints in order to cut off (or *separate*) the optimal solution to the LP relaxation from the integer feasible set. It is often possible to find good cuts by exploiting the structure of the problem [5]. *Gomory cuts* are cutting planes that may be generated systematically by manipulating equations from the *optimal dictionary* associated with the LP relaxation. The optimal solution may be found after generating a finite number of Gomory cuts, but since this is potentially a large number, these cuts are well suited for use in a branch-and-cut method.

Branch-and-price [110] is another generalisation of the branch-and-bound method. This approach is designed for integer programs with a huge number of variables. To improve efficiency, the size of the LP relaxation is reduced by leaving out columns of A; many of the associated variables may be equal to zero in an optimal solution anyway. To verify the optimality of the LP solution, a sub-problem, known as the *pricing problem*, is solved to indicate columns with a profitable *reduced cost*. If such columns are discovered the linear program is reoptimised. If no profitable columns are found and the LP solution is not integer, branching occurs. *Column generation* is applied at every node of the branch-and-bound tree.

2.4 Heuristics

A *heuristic* method is a technique used to find near-optimal solutions to a problem for a reasonable computational cost. Heuristics do not guarantee the feasibility or optimality of the solutions they produce. By exploiting the structure of a problem, reasonable solutions may be found. A heuristic approach should be employed when exact solution methods require impractical computational requirements; large NP-hard problems are usually tackled with heuristic techniques.

Heuristics are also rules or strategies used within other solution methods to improve their effectiveness, for example, the choice of branching variable in the branch-and-bound method.

Although many heuristics are specific to the problem they are designed for, some categories of heuristics [104] include:

- **Construction:** Solutions are generated by adding one component of the solution at a time.
- **Improvement:** From a starting solution, apply a sequence of transformations to improve on it. This includes the important class of *local search* heuristics.
- **Partitioning:** The problem is split into sub-problems which are solved independently. The fragment solutions are combined together to form a solution to the original problem.
- **Aggregation:** Entities are grouped together to form a smaller problem.
- **Relaxation:** Some constraints are relaxed to make the problem easier to solve. Transformations must be performed on infeasible solutions to obtain a feasible answer.
- **Restriction:** The solution space is restricted to make the problem easier to solve.

A heuristic that guides subordinate heuristics is known as a *metaheuristic*.

2.5 Local search

Local search heuristics iteratively improve on feasible solutions by generating *neighbour* solutions within the search space. A starting solution, $x_0 \in X$, may be generated randomly or through the use of some construction heuristic. The *neighbourhood* $N(x_i)$ of a solution $x_i \in X$ is a set of solutions that may be found by applying specified modifications to x_i. A solution $x_{i+1} \in N(x_i)$ is usually selected by comparing its objective value, $f(x_{i+1})$, with $f(x_i)$. The iterative process continues with the generation of neighbours for x_{i+1}. The algorithm ends when a termination test is satisfied. For a survey of local search methods, see the book edited by Aarts and Lenstra [1].

For a minimisation problem, an *improving move* is one in which the objective value of the selected neighbour x_{i+1} is less than the objective value of the current solution x_i. In a *neutral move* the objective values are the same. In a *deteriorating move* the objective value of the neighbour is greater than the objective value of the current solution.

2.5.1 Descent

Descent procedures form the simplest local search heuristics. In a *first improvement descent* the neighbours of the current solution are searched until an improving move is found. The improved solution is accepted as the new current solution and the algorithm continues. The algorithm terminates when there are no more improving moves in the current neighbourhood; the solution is a local optimum. The order in which neighbours are searched may impact on the resulting solution. In a *steepest descent* all neighbours of the current solution are searched and then the solution that gives the largest improvement is selected. Larger neighbourhoods, and high computational

costs for exploring each solution, may negatively impact the efficiency of steepest descent. If measures are taken to prevent cycling, a descent algorithm can be modified to accept neutral moves and explore a larger area of the search space.

The solution provided by a descent algorithm may depend on the starting solution; ordinary descent cannot escape local minima. To avoid slipping into a single local optimum, a *multi-start descent* performs several individual descents from different starting solutions. Starting solutions may be generated randomly or by use of a construction heuristic with a random element. These constructive approaches are known as *greedy randomised adaptive search procedures* (GRASP) [48].

2.5.2 Iterated descent

Having found a local optimum it can prove useful to keep the good characteristics already discovered rather than start again from scratch. In an *iterated descent* [12] a *kick* is applied to the local optimum when it is reached. The kick may take the form of a move to a solution within an expanded or different neighbourhood, or a series of moves within the same neighbourhood. If the resulting solution is satisfactory, another descent begins using this as the starting point. Starting solutions may be generated from the result of the previous descent or from the best local optimum so far. The kick should be large enough to create a solution that does not swiftly descend back to the same local optimum, but not so large that the good features of the solution are lost. The iterative local optimum search process continues until a stopping criterion is met, for example, the completion of a fixed number of iterations.

2.5.3 Variable neighbourhood search

Variable neighbourhood search (VNS) procedures [66, 67, 95] exploit three observations:

1. A local minimum with respect to one neighbourhood structure need not be a local minimum with respect to another.
2. A global minimum is a local minimum for all possible neighbourhood structures.
3. In many problems, local minima are relatively close to one another [67]. This suggests that a local optimum provides information about the global optimum and a study of its neighbourhood may reveal better solutions.

Variable neighbourhood descent combines descent heuristics by stipulating a set of neighbourhood structures, N_l for $l = 1, \ldots, l_{max}$. A descent is repeatedly performed using N_1 until no improving move is found. Upon reaching a local minimum with respect to this first structure, the algorithm applies each subsequent neighbourhood structure until an improving move is found or all structures have been applied to the current solution. Whenever an improving move is found, the original descent begins anew from the improved solution.

Forming an ingredient for other VNS procedures, *reduced variable neighbourhood search* uses a set of neighbourhood structures, N_k for $k = 1, \ldots, k_{max}$. Usually, each subsequent neighbourhood contains the previous one. From an initial solution, a *shaking* procedure is applied that generates a solution at random from the first neighbourhood. Whenever any improvement is discovered, shaking continues from the new solution by generating a neighbour using the first structure. Otherwise, the algorithm

cycles through the neighbourhoods in the set, utilising each for a single shaking step. The process returns to the first neighbourhood after each improvement, continuing until a stopping condition is reached.

A *basic variable neighbourhood search* proceeds in a similar way to the reduced VNS described above, cycling through a set of neighbourhood structures, N_k for $k = 1, \ldots, k_{max}$, and stopping when a condition is met. The shaking procedure is used to generate neighbours of the incumbent solution, x. The difference between this and the reduced VNS is that a local search algorithm is applied to each of the generated neighbours; the resulting local optimum is compared to x and becomes the new incumbent solution if it is an improvement. If it is not an improvement, shaking continues from x using the next neighbourhood, with the local search procedure being used to try to improve each result.

A *general variable neighbourhood search* first uses a *reduced VNS* to improve the initial solution, then applies a *basic VNS* where *variable neighbourhood descent* is the featured local search procedure.

Variable neighbourhood search usually gives better solutions than multi-start, especially when there are several clustered local optima [67].

2.5.4 Tabu search

The basic idea behind the *tabu search* [55] metaheuristic is to prevent a search algorithm from moving back to solutions or characteristics that have already been explored, encouraging the search into new areas. A *tabu list* that contains forbidden moves, solutions or characteristics is compiled and updated with each move selected. In a similar fashion to the steepest descent method, the entire neighbourhood of a solution is explored before a move is made; however, in a tabu search the best available move not on the

tabu list is selected, even if it does not provide an improvement in the objective value. To mitigate the effect of the tabu list preventing moves to good solutions, the procedure may also include an *aspiration criterion*: this overrides the restriction of the tabu list when the search encounters an influential or quality solution. A move to a solution with a better objective value than previously found will be allowed by an aspiration for quality. An aspiration for influence enables moves to solutions that have a sufficiently different structure, driving the search into new areas of the space. The length of the tabu list is an important factor in the effectiveness of the algorithm; a shorter list provides a greater risk of cycling through solutions, while a longer list is more likely to cut off access to good solutions.

2.5.5 Simulated annealing

Simulated annealing [75] local search algorithms are inspired by a model of a physical annealing process. The model represents the controlled cooling of a heated substance from a liquid to solid state. The local search algorithm proceeds in a similar way to a descent method but includes an important difference: there is a probability that deteriorating moves will be accepted. While improving and neutral moves are automatically accepted, a deteriorating move is accepted with probability $e^{-\Delta/T}$, where Δ is the difference between the objective values and T is a control parameter known as the *temperature*. The value of T changes as the algorithm progresses according to a *cooling schedule*; the probability of accepting a deteriorating move decreases over time. As a result, the search is sometimes able to escape quickly discovered local optima and continue the quest for the global optimum.

2.5.6 Genetic algorithms

Genetic algorithms [35] are inspired by the theory of evolution; the fittest organisms within a population survive to pass their characteristics on to the next generation. In a genetic algorithm, solutions are represented as strings. An initial population is generated, randomly or by construction, and their individual fitnesses are evaluated. A new population representing the next generation is created with each iteration of the algorithm. In the classical version, two offspring at a time are created from two parent solutions. Selection of parents and breeding of offspring continues until the new population is complete. Parents are selected at random with probabilities based on their *fitness value*, a function of their objective value. The probability of a *crossover* determines whether the two offspring are identical to their parents or formed by swapping sections of their strings. There is also a probability that offspring will undergo *mutation*, where an element of the solution string is randomly altered. The algorithm terminates when the stopping criterion is satisfied, for example, after completion of a fixed number of generations. Some procedures apply a local search algorithm to improve offspring before determining fitness values.

2.5.7 Ant colony optimisation

Ant colony optimisation [39] is inspired by the behaviour of ants foraging for food. As an ant travels between food and the nest it deposits a *pheromone trail*. This trail is detectable by other ants; they may choose to follow it and deposit more of their own pheromone along the way. As the pheromone trail grows in strength, more and more ants are attracted to the route. The pheromone decays over time, so if the number of ants walking the trail decreases, fewer ants will be attracted to it.

The optimisation model first transforms the problem into the problem of finding the best path on a weighted graph. During each iteration of the algorithm, *artificial ants* are placed randomly on the graph. They wander the edges of the graph, choosing a route from each vertex based on probabilities given by the amount of pheromone laid on each incident edge. In addition to the influence of the pheromone on edge selection, the artificial ants may see the length of an edge and will remember the vertices they have already visited. Each ant incrementally builds a solution to the problem. Once the ants have completed their route, an amount of pheromone evaporates from each edge. Then, extra pheromone is added to the edges of the best solutions discovered. Iterations continue, using the updated pheromone values, until a stopping condition is met. Optionally, solutions are improved using a local search algorithm before pheromone alterations are implemented.

2.6 Methods used in this thesis

Some of the methods summarised above have been shown to be very effective for finding solutions to certain combinatorial optimisation problems. Two of the problems we study in this thesis are NP-hard (the complexity classification of the third problem has not yet been determined), so truly efficient algorithms for solving them are unlikely to be discovered. Heuristic methods therefore represent the best approaches unless dealing with small instances. We employ restricted variations on the exact methods of branch-and-bound and dynamic programming. We also use descent procedures and a simple tabu search method.

Chapter 3

The supply ship travelling salesman problem

3.1 Introduction

The supply ship travelling salesman problem is a variation on the travelling salesman problem (TSP) where the nodes (or *warships*) are in motion. The *supply ship* must visit and resupply all the warships. During replenishment the supply ship and warship travel side by side. This problem is most closely related to the time dependent TSP [91], the moving-target TSP [71] and the non-stationary TSP [74].

The TSP is among the most well known combinatorial optimisation problems and has been well studied over the past few decades [60]. It involves finding a shortest route for a travelling salesperson that starts at a home city, visits a prescribed set of other cities and returns to the starting city. The problem is to find an optimal ordering of the cities, equivalent to visiting each city exactly once while minimising the total distance travelled.

In the supply ship TSP, the cities are replaced by warships that may continuously move location. The problem assumes all ships in the group require resupply as soon as possible. As in the TSP, each warship must be visited once by the supply ship. However, the cost (in time) of travelling from one warship to another depends on the movement of the ships and thus the time at which the transition is made. Each ship has a *replenishment time*, the amount of time required by the supply ship to finish restocking the warship. The supply ship remains with a warship while it is being replenished. Upon completion of a warship's replenishment, the supply ship may move on to the next warship. The supply ship's starting position is given by a depot location. The supply ship returns to the depot once all warships are replenished. The objective of the supply ship TSP is to minimise the time taken to complete replenishment of all ships and return to the depot location.

In this chapter, we compare the effectiveness of a number of heuristic methods applicable to the supply ship TSP. State-space relaxation methods for dynamic programming are used to provide lower bounds; our suggestion to equate the values of the state-space modifiers with warship replenishment times provides a significant improvement over iterative methods. We apply restricted dynamic programming approaches, presenting alternative valuations for selection of partial tours to be retained. We implement a number of simple 2-opt and 3-opt local search heuristics for the supply ship TSP, demonstrating that they are outperformed by our best restricted dynamic programming approach.

Section 3.2 provides an overview of the key literature for the TSP and some of its variants. Section 3.3 describes the supply ship TSP, while Section 3.4 outlines how arc costs are calculated. A dynamic programming formulation for the supply ship TSP, based on a variant of the time dependent TSP, is shown in Section 3.5. We apply state-space relaxation methods to find lower

bounds using dynamic programming in Section 3.6. In Section 3.7 we apply restricted dynamic programming to the supply ship TSP. Section 3.8 describes 2-opt and 3-opt local search procedures for the supply ship TSP. Our computational experience, and a comparison of the heuristic methods described, are provided in Section 3.9. Suggestions for further study are provided in Section 3.10. We conclude the chapter in Section 3.11.

3.2 Literature

3.2.1 The classical travelling salesman problem

According to Schrijver [112] the travelling salesman problem was first formulated in a German manual for the successful travelling salesman in 1832; it was presented as a research problem by Menger in the 1930s.

In the classical travelling salesman problem (TSP) we are given a set, $\{1, 2, \ldots, n\}$, of *cities*, and *distances*, c_{ij}, for each pair of distinct cities. The goal is to find an ordering of the cities, $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$, that minimises the *tour length*

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

In the *symmetric* TSP the distances satisfy $c_{ij} = c_{ji}$ for $1 \leq i, j \leq n$. Extensive studies of the TSP may be found in [1, 60].

The TSP is an NP-hard problem [53] so algorithms to find optimal tours must "work well on real-world rather than worst-case instances" [1]. Alternatively, we use heuristics to find near-optimal tours quickly.

Exact methods

Bellman [15] and Held and Karp [68] present the TSP in *dynamic programming* (DP) terms. These approaches are efficient for small instances of the TSP. Bellman notes that the DP approach can easily "incorporate all types of realistic constraints involving the order in which cities can be visited" [15]. He also states that the method may be used to provide approximate solutions to large-scale problems by grouping subtours as one new distance.

Dantzig et al. [33, 34] introduce an approach that iteratively improves linear programming relaxations. This *cutting plane method* uses the simplex method to move from a starting solution to new solutions, adding new constraints to cut out solutions that are not tours.

Early *branch-and-bound* procedures for the TSP were developed by Eastman [43] and Little et al. [84]. Optimal solutions to corresponding assignment problems are used as lower bounds by Eastman, while Little et al. calculate rapid lower bounds from the distance matrix. Held and Karp [69] introduce an effective branch-and bound algorithm, exploiting the relationship between the TSP and the minimum spanning tree problem to derive strong lower bounds.

Applegate et al. [8] trace the development of the many refinements to the linear programming relaxation method of Dantzig et al. [33], stating, "their approach remains the only known tool for solving TSP instances with more than several hundred cities" [8]. They use a variant of the branch-and-bound method known as *branch-and-cut* [100]: it applies the cutting plane method to each linear programming relaxation before branching. The *Concorde TSP solver* [9, 121] includes their efficient implementation of this approach.

Tour construction heuristics

Tour construction procedures build a tour by successively adding a new node or edge at each step, terminating on construction of a feasible tour. These heuristics perform well in practice, "the best typically getting within roughly 10-15% of optimal in relatively little time" [1]. Important tour construction heuristics include:

- *Nearest neighbour*: This starts with a partial tour consisting of a single node. With each step it adds the unvisited node that is closest to the last node added to the partial tour.
- *Greedy*: Starting with an empty set of edges, this algorithm adds the shortest unused edge from the graph at each step to the set. The selection is restricted so that the edges form a valid tour in the final step.
- *Savings* [30]: This looks for the best shortcuts in an initial pseudo-tour.
- *Christofides* [26]: This constructs a tour based on a minimum spanning tree of the graph.

The theoretical performance of these heuristics (and others) is discussed by Johnson and McGeoch [1]. Empirical performance of heuristics may be evaluated by comparing their results to the *Held-Karp lower bound* on the optimal tour length [69].

Tour improvement heuristics

Tour improvement procedures start from an initial tour and seek a better one by moving from one solution to another within a neighbourhood structure.

For a local search approach, the starting tour may be found using a tour construction heuristic or through some other method. Modifications are made to the initial tour to examine the neighbourhood and one of these neighbour tours is selected. The search continues iteratively from this tour, with neighbours being investigated and selected until a local optimum is found.

Neighbourhood structures for local search approaches to the TSP are based on edge-exchange and node-insertion procedures. The k-opt algorithms are classic local search methods for the TSP. A neighbour of the current tour is found by deleting k edges, breaking the tour into k segments, then reconnecting these segments in a different order using k edges. The neighbourhood of the tour may be generated by applying this modification for each possible combination of k edges. For an n-node instance, discovering all neighbours of a solution requires $O(n^k)$ time.

Croes [32] introduces the *2-opt* algorithm. A 2-opt move deletes two edges, breaking the tour into two paths, then reconnects the paths in the other possible way. Bock [20] and Lin [82] present the *3-opt* method. In a 3-opt move, three edges are deleted and the tour is broken into three paths. There are then eight ways to reconnect the paths to form a tour. Johnson and McGeoch [1] discuss theoretical bounds and experimental results for these local search algorithms.

The *Lin-Kernighan* algorithm [83] is an effective method that generalises k-opt, "the value of k is dynamically determined using a sequence of 2-opt moves" [1]. The algorithm has been modified and improved by Helsgaun [70] and Applegate et al. [10].

Voudouris and Tsang [119] apply guided local search (GLS) and fast local search (FLS) to the TSP. GLS augments the objective function "with a set of penalty terms which are dynamically manipulated during the search process to steer the heuristic to be guided" [119]. FLS narrows the search of the neighbourhood by splitting it into active and inactive sub-neighbourhoods.

According to Fredman et al., "the choice of data structure for tour representation plays a critical role in the efficiency of local improvement heuristics for the travelling salesman problem" [52]. They consider alternative tree-based tour data structures which are useful for large TSPs.

Johnson and McGeoch [61] perform extensive experimentation with a number of heuristics for real-world applications of the TSP, discovering that, "heuristics can provide surprisingly good results in reasonable amounts of time" [61].

3.2.2 Variations of the TSP

The supply ship TSP is a variation of the TSP in which the nodes are in motion, each following a predetermined route. Thus each inter-node distance or *arc cost* is a known continuous function of time. We now look at *static* TSP variants that share some aspect of this characteristic. (In static problems, as opposed to *dynamic* ones, costs and requirements are known before runtime).

Asymmetric TSP

In the *asymmetric TSP* (ATSP), the distance c_{ij} from city i to city j need not equal the reverse distance c_{ji}.

A number of heuristic classes are tested on real-world instances of the ATSP by Johnson et al. [28, 62]: classical tour construction such as nearest neighbour; local search such as 3-opt; cycle cover; and repeated local search.

Burke et al. [21] present HyperOpt, a variable neighbourhood search for the ATSP. They propose a hybrid of this method with 3-opt that yields good results.

Carpaneto et al. [23] use a branch-and-bound algorithm to find exact solutions to large-scale ATSPs in reasonable times.

Time-dependent TSP

The *time-dependent TSP* (TDTSP) is a generalisation of the classical TSP where, in the standard version, the cost of any given arc is dependent on its position in the tour. This has applications to one-machine scheduling with sequence dependent set-up times.

Gouveia and Voß [57] present a number of linear programming formulations for the TDTSP, including those of Picard and Queyranne [102], Fox, Gavish and Graves [49] and a formulation based on the quadratic assignment problem [78]. The branch-and-bound approach of Picard and Queyranne [102] is an effective method for solving TDTSPs with up to 20 nodes.

Another effective tree-search based method is presented by Lucena [86] for the *deliveryman problem*, a TDTSP that seeks to minimise the average arrival time at each location.

Vander Wiel and Sahanidis [123] present a mixed integer linear program formulation. They develop a heuristic based on Lin-Kernighan, producing solutions within 4.4% of optimum, on average.

Time dependent TSP for a congested urban environment

Malandraki and Daskin [91] introduce an alternative version of the TDTSP, where edge costs represent travel times that depend on both the distance between two locations and the time of day the trip is made. Some form of "rush hour" affects the travel time. Treating travel time functions as step functions, they present a mixed integer linear programming formulation for a time dependent vehicle routing problem, of which the TDTSP is a special case. They give a probabilistic nearest neighbour heuristic for the TDTSP and briefly describe a cutting plane heuristic based on their linear programming formulation (with further details in [90]). The algorithms are tested on randomly generated problems with 10 to 25 nodes, and 2 or 3 time periods per edge. The cutting plane algorithm produces better tours than the nearest neighbour heuristic but is much more computationally expensive.

Malandraki and Dial [92] introduce a restricted dynamic programming heuristic for the TDTSP. The heuristic provides a middle ground between an optimal dynamic programming algorithm and the nearest neighbour heuristic by "retaining only the H most promising partial tours" [92]. They state that the algorithm relies heavily on its sorting procedure. The problem instances they generate and test contain 10 to 55 nodes and use step functions with 2 or 3 periods to represent each edge cost. They present results for $H \in \{1, 100, 1000, 5000, 15000\}$, showing the heuristic provides significant improvements over nearest neighbour. Marginal improvements diminish quickly as H increases. They suggest that good solutions can be obtained for 200-node problems in reasonable computational times.

Schneider [111] applies simulated annealing to solve special cases of a TDTSP in which some subset of cities fall within a traffic zone. Travel times between cities in the zone are increased by a constant factor after the start of the rush hour in the afternoon.

Moving-target TSP

In the *moving-target TSP* (or *kinetic TSP*), a pursuer must intercept in minimum time a set of targets which move with constant velocities from the origin.

Hammar and Nilsson [65] study the approximation complexity of variants of the kinetic TSP where targets move with fixed constant speeds in fixed directions, starting from the origin. They show that, if all the targets move with the same velocity, there is a polynomial time approximation scheme.

Helvig et al. [71] propose approximate and exact algorithms for variants of the moving-target TSP. They consider the situation where targets are confined to a single line, deducing an algorithm to find an optimal tour with

quadratic runtime. They also provide a heuristic for the case in which only a few targets move. Another variant they consider is the *moving-target TSP with resupply after intercepts*, where the pursuer must return to the origin after intercepting each target.

Non-stationary TSP

The *non-stationary TSP* is similar to the moving-target TSP. A pursuer must intercept in minimum time a set of targets which move with constant velocities; however, each target has an initial starting position, while the pursuer starts at the origin.

Jiang et al. [74] introduce the non-stationary TSP and apply a *genetic algorithm* with two different crossovers.

3.2.3 Support ship routing in a deployed task group

As part of an MSc dissertation, Hewitt [72] studies *'Support ship routing in a deployed task group'* on behalf of Dstl. A task group is a collection of ships deployed to perform a specific function. There may be between 11 and 29 combat ships in the group. The deployment may involve transiting between locations or patrolling an area of sea. During transit the configuration of zones for the combat ships is maintained, with individual ships moving within their designated zone in the group. During the deployment the combat ships must be resupplied by the task group's support ships. A replenishment-at-sea is accomplished with the supply ship and combat ship travelling side by side.

Hewitt developed a scenario generator to approximate realistic deployments of task groups. Individual ships are restricted to zones relative to the centre

of the task group as it transits between locations; whatever a ship's individual movements, its position relative to the other ships in the group is constrained. The heuristic approaches that Hewitt tests include: nearest neighbour; cheapest insertion; ant systems; 3-opt (without reversed segements) and restricted enumeration. The algorithms are implemented in Microsoft Excel, using Visual Basic for Applications. Restricted enumeration is highlighted as a good approach. The 3-opt approaches produce the best tours but running times are significantly longer than those for restricted enumeration. The inclusion of a stopping condition for 3-opt demonstrates that restricted enumeration performs better over similar time frames. The restricted enumeration algorithm is an attempt to apply the restricted dynamic programming approach of Malandraki and Dial [92], but fails to apply the principle of optimality in order to retain only the best state from a set of identical states with different objective values.

It is Hewitt's project that has lead us to tackle the supply ship TSP. We have generalised warship movements, eliminated task group movement and simplified patrol zone structure. A key difference is the change in the number of allowable ships per patrol zone.

3.2.4 Conclusions

The strongest approaches for the TSP rely on being able to compute a minimum spanning tree [26, 69] or the capture of "good" sub-sequences that can be switched around to produce better solutions [83]. These ideas do not translate well to the supply ship TSP since edge costs may change whenever the solution sequence changes. The approaches developed for the more unusual TSP variations may depend on specific restrictions to the structure of the problem. We will test heuristics that do not rely too heavily on step functions or node movement restrictions.

3.3 The supply ship travelling salesman problem

The supply ship travelling salesman problem involves the minimisation of the total time required by a *supply ship* to visit and replenish a set of *warships* while they 'patrol' an area of sea. The positions and movements of all warships are known throughout the patrol time period. Each warship has an associated *replenishment time*, the time duration required to complete transfer of supplies. During replenishment the supply ship and warship travel side by side, sharing the warship's position, course and speed.

The supply ship begins at node 0, the *depot*, at time t_0. Let $N = \{1, \ldots, n\}$ be the set of nodes representing the n warships to be replenished, while r_k is the replenishment time at node k. The *travel time*, or arc cost, from node i to node j is given by $c_{ij}(t_i)$, where t_i is the time of departure from node i.

The goal is to find an ordering of the nodes of N, $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$, that minimises the supply ship's return time to the depot:

$$T^* = t_0 + c_{0\pi(1)}(t_0) + r_{\pi(1)} + \sum_{i=1}^{n-1} \left(c_{\pi(i)\pi(i+1)}(t_{\pi(i)}) + r_{\pi(i+1)} \right) + c_{\pi(n)0}(t_{\pi(n)})$$

where

$$t_{\pi(1)} = t_0 + c_{0\pi(1)}(t_0) + r_{\pi(1)}$$

$$t_{\pi(i)} = t_{\pi(i-1)} + c_{\pi(i-1)\pi(i)}(t_{\pi(i-1)}) + r_{\pi(i)} \quad \text{for } 2 \leq i \leq n$$

This is equivalent to minimising the supply ship's total travel time between warships:

$$c_{0\pi(1)}(t_0) + \sum_{i=1}^{n-1} \left(c_{\pi(i)\pi(i+1)}(t_{\pi(i)}) \right) + c_{\pi(n)0}(t_{\pi(n)})$$

where $t_{\pi(i)}$ is as described above.

In the problem instances we consider, each warship's journey is described by a series of *legs*. Each leg is described by a starting position and the direction

in which the ship travels at its constant speed until the start of the next leg. We assume that the supply ship travels at a constant speed greater than that of any warship, but matches the course and speed of a warship during replenishment.

The supply ship TSP could be considered to be a special case of an asymmetric time dependent TSP where costs may vary continuously with time. Both the moving-target TSP [71] and non-stationary TSP [74] are similar to the supply ship TSP when each warship is restricted to one journey leg.

Only some of the approaches for the TSP mentioned in Section 3.2 are able to handle the supply ship TSP's time dependency. Promising heuristic methods include *nearest neighbour* and *3-opt* local search for their simplicity, while the adaptability of *restricted dynamic programming* indicates it to be a worthwhile candidate.

3.4 Calculation of arc costs

It is clear that, because the warships are in motion and their movements are known, the travel time between any two warships depends on the time at which the journey begins. We calculate travel time, or arc cost, by solving a quadratic equation based on the warships' movement information.

We wish to find $c_{ij}(t_i)$, the travel time for the supply ship to reach warship j when departing from warship i at time t_i. We know the position of the supply ship at time t_i: it is at the same position as warship i. Let (x_i, y_i) be the supply ship's coordinates at time t_i. Let (x_j, y_j) be the coordinates of warship j at time t_i and (w_x, w_y) be the components of its velocity in the x- and y- directions during its current leg. Let s be the speed of the supply

ship and (s_x, s_y) be the unknown components of its velocity in the x- and y-directions.

We must solve a set of 3 equations. To simplify the notation in these equations, let $t = c_{ij}(t_i)$, $x = x_j - x_i$ and $y = y_j - y_i$. We have

$$s_x t = x + w_x t \tag{3.1}$$

$$s_y t = y + w_y t \tag{3.2}$$

$$s_x^2 + s_y^2 = s^2 \tag{3.3}$$

where s_x, s_y and t are unknowns. Equations (3.1) and (3.2) determine the intersection time for the journeys of the supply ship and warship. Equation (3.3) applies Pythagoras' theorem to constrain the velocity of the supply ship in the x- and y- directions. We are primarily interested in finding a non-negative real value for t. We may eliminate s_x and s_y by multiplying (3.3) by t^2 and squaring each of (3.1) and (3.2).

$$\begin{aligned} s_x^2 t^2 &= x^2 + 2xw_x t + w_x^2 t^2 \\ s_y^2 t^2 &= y^2 + 2yw_y t + w_y^2 t^2 \\ s_x^2 t^2 + s_y^2 t^2 &= s^2 t^2 \end{aligned}$$

The resulting quadratic equation in t is

$$x^2 + 2xw_x t + w_x^2 t^2 + y^2 + 2yw_y t + w_y^2 t^2 = s^2 t^2$$

which we may simplify to

$$at^2 + bt + c = 0$$

where

$$\begin{aligned} a &= w_x^2 + w_y^2 - s^2 \\ b &= 2(xw_x + yw_y) \\ c &= x^2 + y^2 \end{aligned}$$

(this is equivalent to the quadratic equation applied by [72]).

The value of the smallest non-negative solution to the equation may reveal that the supply ship cannot reach the warship during its current leg. In this case, we solve an adjusted quadratic equation using the starting position and velocity components that describe warship j's next leg. If the start of the new leg is at time t_{leg} we redefine (x_j, y_j) to be the position of the warship j at time t_{leg}. (x_i, y_i) remains as the position of the warship i at time t_i. We begin with the following set of equations:

$$\begin{aligned} s_x t &= x + w_x(t-h) \\ s_y t &= y + w_y(t-h) \\ s_x^2 + s_y^2 &= s^2 \end{aligned}$$

where $h = t_{leg} - t_i$. The components of the resulting quadratic equation become:

$$\begin{aligned} a &= w_x^2 + w_y^2 - s^2 \\ b &= 2(xw_x + yw_y) - 2h(w_x^2 + w_y^2) \\ c &= x^2 + y^2 - 2h(xw_x + yw_y) + h^2(w_x^2 + w_y^2) \end{aligned}$$

We continue to re-solve the quadratic equation, updating t_{leg} to mark the start of the next leg with each failure, until a valid value for t is found. Thus $c_{ij}(t_i)$ has time complexity $O(L_j)$, where L_j is the maximum number of legs for warship j. Letting $L = \max_{j \in N} \{L_j\}$, we may say that the time complexity of the travel time functions is $O(L)$.

3.5 Dynamic programming

The following dynamic programming exact algorithm finds a tour of the warships with the shortest return time to the supply ship's depot location.

The formulation is equivalent to that presented by Malandraki and Dial [92] for the time dependent TSP.

Recall that $N = \{1, \ldots, n\}$ is the set of nodes representing n warships to be replenished, while a node 0 is the supply ship depot location. We have defined r_p to be the replenishment time at node p, while $c_{pk}(t_p)$ is the travel time for the supply ship to reach node k when departing from node p at time t_p.

Given a set of nodes $S \subseteq N$, and $k \in S$, let $T(S, k)$ be the minimum time needed to start from node 0, visit all the nodes in S and arrive at node k.

First, we find $T(S, k)$ for $|S| = 1$, so $S = \{k\}$. We have

$$T(\{k\}, k) = t_0 + c_{0k}(t_0) \qquad \text{for all } k \in N,$$

where t_0 is the supply ship's departure time from node 0.

For $|S| > 1$, the algorithm considers visiting k immediately after $p \in S - \{k\}$ and looks up the value of $T(S - \{k\}, p)$ from the preceding computations. We have

$$T(S, k) = \min_{p \in S - \{k\}} \Big\{ T(S - \{k\}, p) + r_p + c_{pk}\big(T(S - \{k\}, p) + r_p\big) \Big\} \tag{3.4}$$

for all $k \in S$.

The minimum return time to the supply ship's depot location for a complete tour is given by

$$T^* = \min_{p \in N} \Big\{ T(N, p) + r_p + c_{p0}\big(T(N, p) + r_p\big) \Big\}.$$

The dynamic programming algorithm will only produce an optimum tour for travel time functions with the following property [92]:

$$\text{If} \quad t_i' \leq t_i'' \quad \text{then} \quad t_i' + c_{ij}(t_i') \leq t_i'' + c_{ij}(t_i'') \quad \text{for all } i, j \in N.$$

This states that the supply ship must always arrive at its destination warship earlier than if it had set off at a later time. This condition will hold for the supply ship TSP if the supply ship knows the course its destination warship will take over the period of time it takes to intercept it. With this information it can set a heading, travel in a straight line and arrive at the warship at the earliest possible time. If the above condition was not true then the principle of optimality would not hold, since a partial path of minimum arrival time would not necessarily lead to a minimum tour. The algorithm would need to be extended to include waiting times at each node.

The dynamic programming algorithm guarantees optimality but has exponential time and computer memory requirements; it is only effective when applied to small problems.

Time complexity: There are 2^n possibilities for S (a node is either in S, or it is not in S) and at most n values for k. Therefore, the total number of states to be stored is bounded by $n2^n$. Since the maximum number of steps per state grows linearly with n, we may say that the algorithm has time complexity $O(n^2 2^n)$. The time complexity of the dynamic programming algorithm becomes $O(Ln^2 2^n)$ if we include the complexity of calculating travel times for each state.

The formulation will be used as the basis for finding lower bounds on the optimal solution (Section 3.6) and also for heuristic methods to find good solutions (Section 3.7).

3.6 Dynamic programming state-space relaxation

Dynamic programming state-space relaxation (DPSSR) techniques may be used to obtain lower bounds on the optimal value. In addition to its use as a measure of the effectiveness of heuristic solutions, a good lower bounding method may be embedded within a branch-and-bound scheme to fathom and prune nodes. The DPSSR method was developed for routing problems by Christofides et al. [27]. A relaxed problem is obtained from the dynamic programming formulation by mapping the original state-space onto a smaller state-space. Suitable dynamic programming recursions are then performed using the smaller state-space.

The state space is relaxed through the use of a *separable mapping function* [27]. This function, $g(.)$, maps the domain (S, k) to a smaller space $(g(S), k)$. We use the mapping functions for the classical TSP in [27] to produce recursions for the supply ship TSP, modifying the recursion in Equation (3.4).

3.6.1 An n-path relaxation

Define $g(S) = |S|$. We then have

$$g(S - \{k\}) = g(S) - 1$$

and, letting $s \equiv |S|$, the recursion on the relaxed state-space becomes

$$T(s,k) = \min_{p \in N - \{k\}} \Big\{ T(s-1,p) + r_p + c_{pk}\big(T(s-1,p) + r_p\big) \Big\} \quad \text{for all } k \in N.$$

Notice that the set of possible candidates for p in Equation (3.4) has been expanded to $p \in N - \{k\}$ from $p \in S - \{k\}$ as the set of visited nodes is unknown.

Initialisation:

$$T(1,k) = t_0 + c_{0k}(t_0) \quad \text{for all } k \in N.$$

Termination:

$$T^*_{LB} = \min_{p \in N} \Big\{ T(n,p) + r_p + c_{p0}\big(T(n,p) + r_p\big) \Big\}.$$

There are only n states (each requiring a comparison of $n-1$ values to find the minimum) in each of the n stages so the time complexity of the algorithm is $O(n^3)$ (if we include complexity of travel times this is $O(Ln^3)$). The shortest path provides a lower bound on the true optimal value. The sequence of n nodes corresponding to the shortest path may contain repetitions of the same node.

3.6.2 A q-path relaxation

Associate an integer number $q_k \geq 1$ with every node $k \in N$. Define

$$g(S) = \sum_{i \in S} q_i.$$

We then have

$$g(S - \{k\}) = g(S) - q_k.$$

Letting $Q \equiv \sum_{i \in S} q_i$, the recursion becomes

$$T(Q,k) = \min_{p \in N(Q,k)} \Big\{ T(Q - q_k, p) + r_p + c_{pk}\big(T(Q - q_k, p) + r_p\big) \Big\}$$

for all $k \in N$, where

$$N(Q,k) = \{x |\ x \in N, x \neq k, q_x \leq Q - q_k\}.$$

Initialisation:

$$T(Q,k) = \begin{cases} t_0 + c_{0k}(t_0), & \text{if } Q = q_k \\ \infty, & \text{if } Q \neq q_k \end{cases} \quad \text{for all } k \in N.$$

Termination: Let $\mathbf{Q} = \sum_{k \in N} q_k$, then

$$T^*_{LB} = \min_{p \in N} \Big\{ T(\mathbf{Q}, p) + r_p + c_{p0}\big(T(\mathbf{Q}, p) + r_p\big) \Big\}.$$

The n-path relaxation is a special case of the q-path, where $q_k = 1$ for all $k \in N$. Appropriate selection of these state-space modifiers, q_k, can reduce the number of repeated visits to a node, improving the 'feasibility' of the resulting sequence and thus improving the bound. If any $q_k > 1$, the node sequence corresponding to the shortest path will not necessarily contain n nodes. There are now $\mathbf{Q}$ stages in the algorithm, so the time complexity is $O(\mathbf{Q}n^2)$ (if we include complexity of travel times this is $O(L\mathbf{Q}n^2)$).

3.6.3 A q-q-path relaxation

Associate two integer numbers q'_k, q''_k, where $q'_k + q''_k \geq 1$, with every node $k \in N$. Define $g(S)$ as the vector

$$(Q', Q'') = \left(\sum_{i \in S} q'_i, \sum_{i \in S} q''_i \right).$$

We then have

$$g(S - \{k\}) = (Q' - q'_k, Q'' - q''_k).$$

The recursion becomes

$$T(Q', Q'', k) = \min_{p \in N(Q', Q'', k)} \Big\{ T(Q' - q'_k, Q'' - q''_k, p) + r_p + c_{pk}(t_p) \Big\}$$

for all $k \in N$, where

$$t_p = T(Q' - q'_k, Q'' - q''_k, p) + r_p$$

and

$$N(Q', Q'', k) = \{x |\ x \in N, x \neq k, q'_x \leq Q' - q'_k, q''_x \leq Q'' - q''_k\}$$

Initialisation:

$$T(Q', Q'', k) = \begin{cases} t_0 + c_{0k}(t_0), & \text{if } Q' = q'_k \text{ and } Q'' = q''_k \\ \infty, & \text{otherwise} \end{cases}$$

for all $k \in N$.

Termination: Let $(\mathbf{Q}', \mathbf{Q}'') = \left(\sum_{k \in N} q'_k, \sum_{k \in N} q''_k \right)$, then

$$T^*_{LB} = \min_{p \in N} \Big\{ T(\mathbf{Q}', \mathbf{Q}'', p) + r_p + c_{p0}\big(T(\mathbf{Q}', \mathbf{Q}'', p) + r_p\big) \Big\}.$$

The q-path is a special case of the q-q-path where $q'_k = q_k$ and $q''_k = 0$ for all $k \in N$. The additional set of state-space modifiers is included to allow further flexibility and improve the quality of the relaxation. The number of sets of such modifiers can be increased, but will result in significant increases in the size of the state-space. With $\mathbf{Q}_{qq} = \max\{\mathbf{Q}' \times \mathbf{Q}'', \mathbf{Q}', \mathbf{Q}''\}$ stages, the time complexity of the algorithm is $O(L\mathbf{Q}_{qq}n^2)$.

3.6.4 Selection of state-space modifiers

The integers, q_k, in the q-path and q-q-path relaxations are called the *state-space modifiers* [2]. Our aim for these modifiers is to force the shortest path to define a feasible sequence.

Replenishment times as state-space modifiers

In the supply ship TSP each warship has a replenishment time. During replenishment the supply ship and warship travel side by side. Total replenishment time across all warships may make up a large proportion of the total time to complete a tour. The state-space modifiers should reflect the replenishment times for the warships, so the node sequence provided by the relaxation includes an accurate contribution to the total tour time from

the total replenishment time. Otherwise, the sequence may repeatedly visit the warships with smallest replenishment times in order to arrive at a smaller total tour time.

We propose that for a q-path relaxation, we let $q_k = r_k$ for all $k \in N$. Note that this will only work if $r_k \in \mathbb{N}$. If any of the replenishment times are not positive integers, the modifiers should represent these values as an integer; for example, let $q_k = \lceil r_k \rceil$ for all $k \in N$.

The q-q-path relaxation is an extension of the q-path. It uses two sets of modifiers. In this relaxation, a modifier may be zero so long as the corresponding modifier in the other set is an integer greater than zero. If the first set of modifiers, q'_k, are used to reflect the replenishment times for the warships, the second set, q''_k, may be altered to improve the bound using the methods described below.

Iterative methods for state-space modifiers

We follow the two iterative methods proposed by Abdul-Razaq and Potts [2] to find values for a set of modifiers, q_k for all $k \in N$.

- q-path: initially set $q_k^{(0)} = 1$ for all $k \in N$.
- q-q-path: the first set of modifiers are selected using some other method. Denote the second set of modifiers by q_k. Assuming the first set of modifiers have values of at least 1, the second set are initially set to $q_k^{(0)} = 0$ for all $k \in N$. Iterative modifications are performed on this second set.

At iteration $i-1$ the DPSSR lower bound is obtained with its corresponding sequence of nodes. Let $m_k^{(i-1)}$ be the number of times that node k occurs in this sequence. If $m_k^{(i-1)} = 1$ for all $k \in N$, then every node

has been visited exactly once and the sequence provides a feasible tour. Such a tour must be optimal.

Method 1:

If the sequence produced is not feasible, we wish to find a node $p \in N$ for which $m_p^{(i-1)} > 1$, and increase the modifier for node p by one with the hope of obtaining $m_p^{(i)} = 1$ in the next iteration. Select a node p satisfying

$$(m_p^{(i-1)} - 1)(q_p^{(i-1)} + a) = \max_{k \in N}\{(m_k^{(i-1)} - 1)(q_k^{(i-1)} + a)\},$$

where $a \geq 0$ is an integer parameter. Update the modifiers using

$$q_k^{(i)} = \begin{cases} q_k^{(i-1)} & \text{for } k \neq p \\ q_k^{(i-1)} + 1 & \text{for } k = p \end{cases} \quad \text{for all } k \in N.$$

The use of this method results in $\mathbf{Q}^{(i)} = \sum_{k \in N} q_k^{(i)} = i$ for the q-q-path. (More DP stages must be calculated as $\mathbf{Q}^{(i)}$ increases). Abdul-Razaq and Potts [2] found that using this formula with $a = 2$ produced satisfactory results for the single-machine scheduling problem.

Method 2:

Update the modifiers using

$$q_k^{(i)} = \max\{q_k^{(i-1)} + (m_k^{(i-1)} - 1), \delta\} \text{ for } k \in N$$

where $\delta = 1$ for the q-path, and $\delta = 0$ for the q-q-path. $\mathbf{Q}^{(i)}$ is typically larger (in earlier iterations) than if Method 1 is used, so more DP stages must be calculated, but the modifiers should approach their optimal values more quickly. This method has similarities to the subgradient optimisation technique also used in [2] to update *penalties*.

Penalties

A useful technique to improve the bounds provided by state-space relaxations is the inclusion of *penalties*. A penalty λ_k is defined as an additional cost that is incurred when node k is visited. Since the cost of any complete tour would be increased by $\lambda = \sum_{k \in N} \lambda_k$, the introduction of penalties provides an equivalent problem. However, in a state-space relaxation the sequences of nodes that do not define a complete tour would increase in cost by varying amounts. It is desirable to find penalty values that force the sequence with the smallest cost to form a complete tour. These penalties are analogous to the multipliers used in Lagrangian relaxation for integer programming.

Unfortunately, penalties may not be used so easily for our relaxation of the supply ship TSP. The cost involved for the measurement of a tour is 'time for completion'. Since the travel time of the supply ship between warships depends on the time of departure, using penalties would interfere with the function calculating the travel time.

3.7 Restricted dynamic programming

Before looking at the restricted dynamic program, let us briefly review the *nearest neighbour* heuristic. The nearest neighbour algorithm follows the behaviour of a traveller whose rule is to always go to the nearest unvisited location. For the supply ship TSP the nearest neighbour tour begins at the depot, then repeatedly adds the warship not yet in the tour that takes the shortest time to reach until all warships have been added. Once all the warships are in the tour, it returns to the depot.

The restricted dynamic programming heuristic [92] provides a middle

ground between the dynamic programming exact algorithm and the nearest neighbour heuristic. This modification of the dynamic programming algorithm can avoid the exponential explosion of time and storage requirements by retaining only the H most promising partial tours at each stage, where H is a parameter specified by the user. When $H = 1$, the algorithm is equivalent to the nearest neighbour heuristic. On average, higher values for H will produce better solutions, but at a greater computational cost.

3.7.1 Retaining partial tours

Partial tours, or states, are generated by adding a single node $k \in N - S$ to each partial tour, (S, p), in the previous stage. The principle of optimality is applied so that inferior duplicate states are eliminated. Although there are $i\binom{n}{i}$ distinct states available in stage $i \equiv |S|$, only H states are retained by the restricted dynamic programming algorithm. The partial tours are judged by an associated cost value, with lower costs being preferred. The natural cost measure (used in [92] for the time dependent TSP) is $T(S, k)$, the total time to arrive at the last node in the partial tour after reaching and replenishing all preceding nodes.

As alternatives to using $T(S, k)$ to determine which H tours to retain, we propose the following strategies:

- **Ignore replenishment:** Compare the total travel time of the supply ship *between* nodes, ignoring replenishment times and time of departure from the depot. This eliminates any bias towards retaining partial tours that schedule nodes with the shortest replenishment times first. (During computational testing we designated this approach *RDP2*).

- **Predictive costing:** Use the cost of a corresponding complete tour. We calculate a predictive measure of the final cost of each partial tour by applying the nearest neighbour algorithm to complete the tour. (During computational testing we designated this approach as *Predictive RDP*). This idea is inspired by the *beam search* technique [97], an approximate branch-and-bound method where a simple heuristic algorithm is used to estimate complete solution values from partial solutions.

3.7.2 Computational complexity

The algorithm consists of n stages. A maximum of H partial tours are retained at each stage. Since a partial tour includes S, where $|S| \leq n$, the space complexity of retaining partial tours for all stages is $O(n^2H)$. By retaining only two stages at a time (the current stage and the previous stage from which it is being generated) the space requirements are reduced to $O(nH)$.

Up to n partial tours may be generated from each partial tour retained in the previous stage (each new tour requires a single use of the $O(L)$ travel time function). The number of steps required to identify whether a partial tour is to be retained depends only on the number of states already retained ($O(H)$). Thus the restricted dynamic programming algorithm for the supply ship TSP has a time complexity of $O(n^2H^2L)$.

The predictive costing approach applies the nearest neighbour heuristic to complete the tour whenever a new partial tour is generated. The nearest neighbour heuristic requires $O(n^2L)$ time, as up to n journeys to each unvisited node must be computed. The predictive restricted dynamic programming method therefore has a time complexity of $O(n^4H^2L^2)$.

3.8 Local search: k-opt

In this section, we briefly outline the application of k-opt algorithms to the supply ship TSP. It is important to note that the cost of travelling from warship i to warship j is not necessarily the same as the cost of travelling from warship j to warship i (as in the asymmetric TSP). These costs also depend on the time the supply ship leaves the origin warship, which in turn depends on the sequence of warships visited so far. Whenever a neighbour tour is generated, many arc costs must be recomputed; this requires $O(nL)$ time. A straightforward k-opt approach for the supply ship TSP therefore requires $O(n^{k+1}L)$ time to search for an improving move.

3.8.1 2-opt

2-opt is the simplest of the k-opt family of algorithms [32]. It modifies a tour by removing two edges, leaving two segments. The segments are reconnected by inserting two edges in the only other way possible to make a new tour. This is equivalent to reversing one of the segments. The arc costs for the reversed segment could change significantly.

3.8.2 3-opt

3-opt modifies a tour by removing three edges, leaving three segments [20]. The segments are reconnected by inserting three edges to make a new tour. There are seven ways to reconnect the segments to make a new tour; of these, three correspond to 2-opt moves, where one of the deleted edges is reinserted. Except for those arcs on the directed path leading from the depot node within the 'depot segment', all arc costs must be recalculated.

A 3-opt move allows segments to be reversed in the same way as 2-opt. There is only one way of reconnecting the segments to construct a tour that maintains their direction; it may be worth considering a restricted 3-opt strategy that considers only this swap, since this is effective for the asymmetric TSP [62].

3.8.3 Neighbourhood move acceptance strategies

Two classical ways of choosing the move to accept within the neighbourhood are:

- *First improvement descent:* Generate neighbour tours until one is found that has a lower cost than the current tour. This new tour becomes the current tour and now its neighbourhood is generated until a better one is found. This process continues until a tour is found that does not have any better tours in its neighbourhood.
- *Best improvement / steepest descent:* Generate all neighbour tours of the current tour. From this list, select the tour with the lowest cost. If its cost is lower than the current tour's, the new tour becomes the current tour and now its neighbourhood is generated. This process continues until a tour is found that does not have any better tours in its neighbourhood. This strategy may result in better tours being found, but exploration of each adjacent tour may increase running time.

3.9 Computational experience

The algorithms were coded in the C programming language and compiled using Microsoft Visual Studio .NET 2003. The computer used to test the algorithms featured a Pentium 4 processor (2.4 GHz with 504 MB of RAM).

3.9.1 Generating instances of the supply ship TSP

We generated random instances of the supply ship TSP using the following parameters. A 200×200 area of sea is split into four quadrants of 100×100 (a large simplification of the fixed patrol zone structure of a deployed task group). For an n-ship (or n-node) problem we allocate n warships evenly to the four zones (i.e. $\lfloor n/4 \rfloor$ to each zone, then each of the remaining warships to a different zone). Starting positions for each warship are randomly generated within their assigned quadrant. A fixed location for the supply ship's depot position is generated and may lie anywhere within the 200×200 area (the depot might also be considered to be a node, thus an "n-node" instance actually requires a tour containing $n + 1$ nodes).

The fixed patrol speed of each warship is selected at random from $\{2, 5, 10\}$. Similarly, the fixed pursuit speed of the supply ship is randomly selected from $\{12, 15, 20\}$, ensuring it is fast enough to catch any warship. Replenishment times associated with each warship are selected at random from $\{1, 2, 3\}$.

The warships' patrol movement is made in 'legs': a warship continues along a randomly generated bearing ($\{0,1,\ldots,359\}$ degrees) for a time interval randomly selected from $\{2, 5, 10\}$. We have not included specialised patrol strategies. Warships must remain inside their zone, so a leg is interrupted if the ship reaches the boundary; a new leg begins on a course that does not

cross the boundary. If a warship ends a leg within a 5 unit 'buffer' of a zone edge, the direction of its next leg will be set so that it does not cross the boundary. Legs continue to be generated until an upper bound on the time required by the supply ship is reached. Information about position changes and start/finish times of legs are stored so that we can calculate the position of a warship at any time.

An illustration of warship movements for an 8-node instance is shown in Figure 3.1. Each set of coloured edges and vertices represents the course of a warship over the full time period. This does not truly represent a set of realistic courses, but the random nature of these movements allows for fairly diverse instances within the structure we have set out. The grid structure arose in deference to the scenarios used in [72], but does not attempt to match the distinct layout or single-ship-per-zone limitations.

Given our 200×200 movement area, the longest possible trip for the supply ship between any two points is in a straight line across the diagonal. If s is the speed of the supply ship and $c_{max} = \frac{\sqrt{200^2+200^2}}{s}$, we have $c_{ij}(t_i) \leq c_{max}$ for all $i, j \in N$. Assuming that the supply ship must make only n journeys to warships, the time period over which warship movements must be known is $(c_{max} + r_{max}) \times n$ units long, where r_{max} the largest allowed replenishment time ($r_{max} = 3$ for our instances).

By fixing a minimum leg duration value, we may determine, L, the maximum possible number of legs needed by a warship to cover the time period. For our instances we have specified that no leg duration may be less than one time unit, so $L = (c_{max} + r_{max})n$. Therefore, in the worst case, $c_{ij}(t_i)$ requires $O(n)$ time as it proceeds through all possible legs. (Note that in the q-path relaxation there may be as many as $\mathbf{Q}$ trips, so there should be $(c_{max} + r_{max}) \times \mathbf{Q}$ legs).

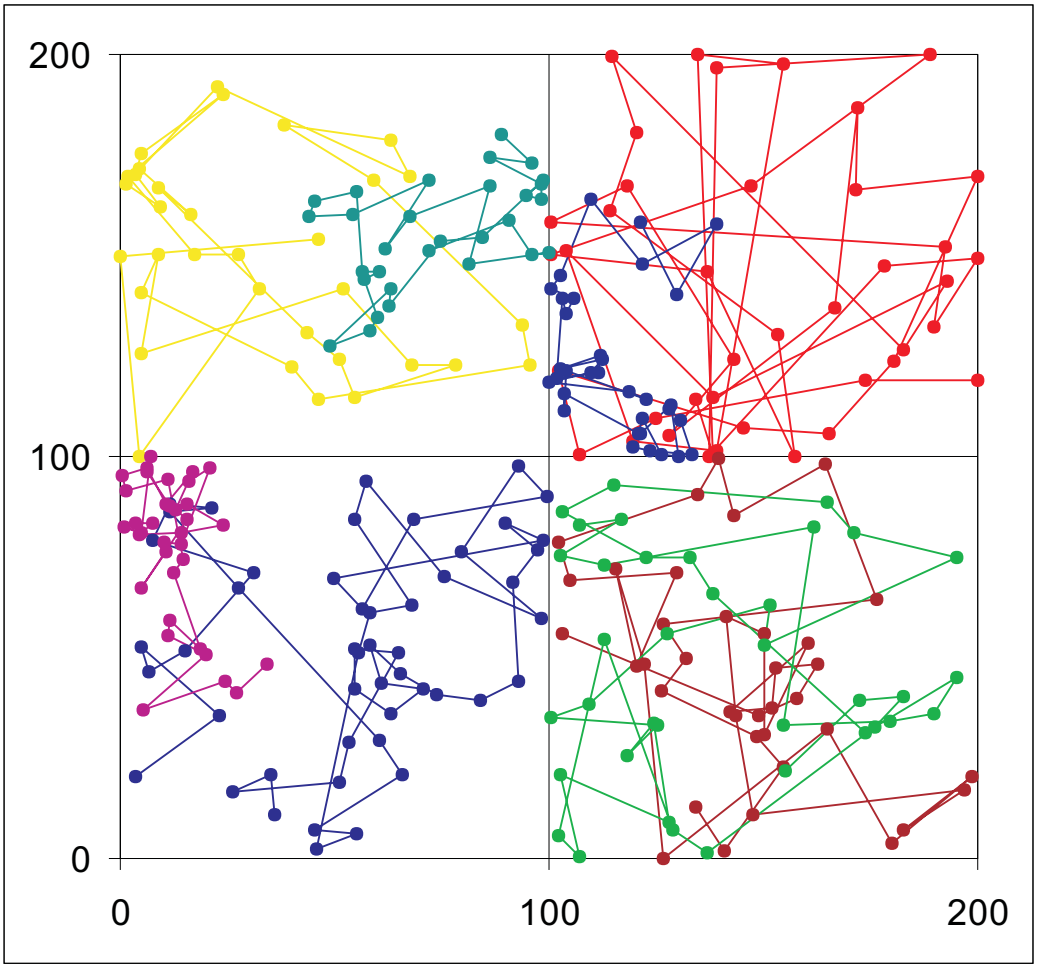

Figure 3.1: Example patrols for eight warships

3.9.2 Lower bounds from dynamic programming state-space relaxations

Experiments were conducted on twenty 20-node instances. Since optimal values are unknown, we used the solution value from RDP2 with $H = 10,000$ as an approximation (see Section 3.9.3). The percentage of this solution value achieved by the bound was calculated. That is, if α is the approximation to the optimal value and Λ is the lower bound value, we compute $\frac{\Lambda}{\alpha} \times 100$. We looked at the average of these relative values (ARV) across the twenty instances; the higher the ARV the better we consider the lower bound to be.

n-path relaxation

The ARV for the n-path relaxation (Section 3.6.1) was 63.4% and required 0.03 seconds to compute, on average.

q-path relaxations

The iterative methods described in Section 3.6.4 were used independently to find values for a single set of state-space modifiers, q_k, $k \in N$. Following [2], we set $a = 2$ for Method 1. For the starting iteration, $q_k = 1$ for every $k \in N$, thus the relaxations are equivalent to the n-path. Plots of ARV against average computation time are shown in Figure 3.2. The lower two lines display the average performance of Method 1 and Method 2 as iterations progress. On average, Method 1 produced greater increases in the bound within a shorter time than Method 2. Method 1 completed 270 iterations in 54 seconds on average, achieving an ARV of 74.8%, while Method 2 computed 311 iterations with an ARV of 73.8% in the same time.

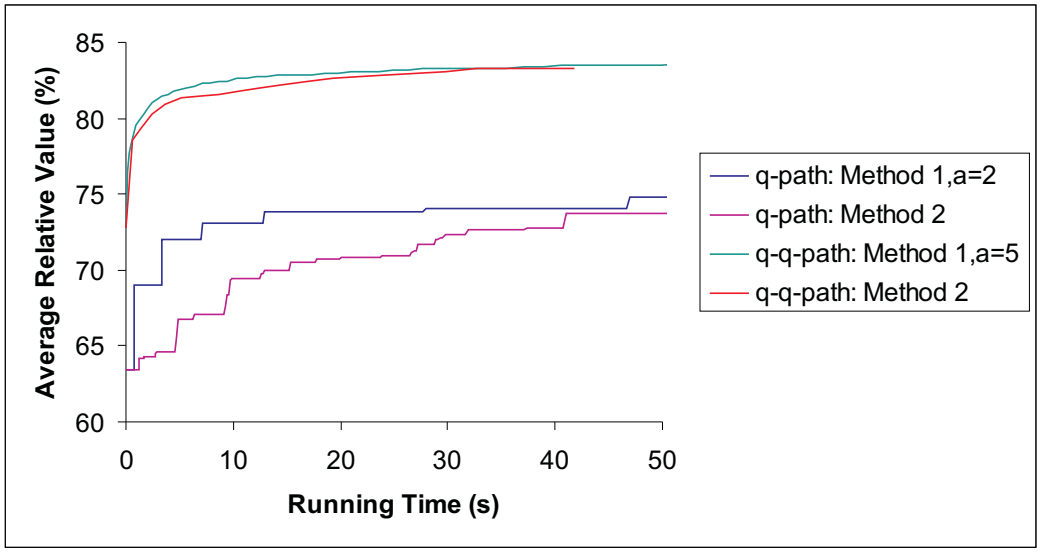

Figure 3.2: Average performance of iterative methods for selection of state-space modifiers over twenty 20-node problems

In Section 3.6.4, we proposed setting the state-space modifiers equal to the replenishment times for each node. The q-path relaxation with $q_k = r_k$ for all $k \in N$ provided an ARV of 72.8% and required 0.06 seconds. This represents a good improvement in the bound over the n-path, while computation time remains small. Setting modifiers in this way compares favourably with the iterative methods, since Method 1 required 7 seconds to match the ARV obtained, while Method 2 required 32 seconds.

q-q-path relaxations

A single set of modifiers, q'_k for $k \in N$, were set equal to the integer replenishment times for the nodes. Both iterative methods were used to find values for the second set of state-space modifiers, where initially, $q''_k = 0$ for all $k \in N$. Thus at the starting iteration, the relaxations are equivalent to the q-path with $q_k \equiv q'_k = r_k$ for all $k \in N$. Method 1 was tested using a number of parameter values: $a = 1, 2, 3, 4, 5$.

The performances of Method 1 (with $a = 5$) and Method 2 are included in Figure 3.2. Method 1 with $a = 5$ performed the best; ARV increased to 80.6% within 2 seconds. It computed 52 iterations within 54 seconds, with an ARV of 83.6%. The bounds found by Method 1 improved marginally as parameter a increased from 1 to 5.

Overall, Method 1 with its incremental approach was found to be superior to the subgradient approach of Method 2 for the supply ship TSP. Although our proposal for the use of replenishment times provides a significant improvement, the resulting bounds are probably not tight enough to underpin an efficient branch-and-bound scheme (but this has not been tested).

3.9.3 Restricted dynamic programming

Three versions of the restricted dynamic program were tested, each with a different criterion for retaining partial tours:

1. **RDP:** Retains partial tours with the smallest time cost, $T(S, k)$.
2. **RDP2:** Retains partial tours with the smallest time cost, ignoring contribution of replenishment times.
3. **Predictive RDP:** Retains partial tours with the smallest final time cost found using the nearest neighbour heuristic.

The idea behind RDP2 is to remove the bias of retaining those tours where the ships with shortest total replenishment times are placed earliest, as this may occur in RDP. The supply ship's total travel time *between* warships is used instead. Since the total replenishment time is fixed for a complete tour, our objective is to minimise this travel time in order to minimise tour completion time.

Results were calculated for three sizes of supply ship TSP problem: 10-node, 20-node and 30-node. A number of different values for the parameter H, the number of partial tours retained at each stage, were experimented with. The quality of a solution is measured using the percentage excess of the solution value over the solution found using RDP2 with $H = 20,000$, after subtracting total replenishment time (which is fixed) from both solutions. That is, if U is the objective value given by our heuristic method, B is the objective value we have found for the instance using RDP2 with $H = 20,000$, and R is the sum of each warship's replenishment, then we let

$$\text{Excess} = 100 \times \frac{((U - R) - (B - R))}{(B - R)} = \frac{100(U - B)}{B - R}.$$

By removing replenishment times, we may differentiate more clearly between the heuristics' ability to pick tours with the smallest total travel times. We consider $H = 20,000$ to be a suitably large parameter value against which results for smaller values may be meaningfully compared.

The 10-node problem instances proved easy to solve exactly using the RDP heuristic with $H = \max_{1 \le i \le 10} \{i\binom{10}{i}\} = 1260$. This parameter value effectively retains all partial tours and provides the dynamic programming exact solution. In this case, 50 randomly generated scenarios were solved exactly, requiring an average CPU time of 0.130 seconds. During experimentation, the optimal solution was produced for all 50 of the instances using RDP2 with $H = 840$, requiring a CPU time of 0.101 seconds, on average.

The 20-node problem instances could not be solved exactly by dynamic programming on our machine within a reasonable time; the algorithm would require $H = 1,847,560$. Our approximation for the optimal solution was provided by RDP2 with $H = 20,000$, which required an average CPU time of 277 seconds over 20 instances. Figure 3.3 compares the quality of solution against average CPU time over twenty instances. In the figure, the closer the points lie to the bottom-left, the better the method. Each point represents the results obtained using a particular value for H; as H

increases, the average excess over optimal decreases and average CPU times increase. RDP2 produces better solutions on average than both RDP and Predictive RDP at all running times. Predictive RDP outperforms RDP for running times greater than 20 seconds.

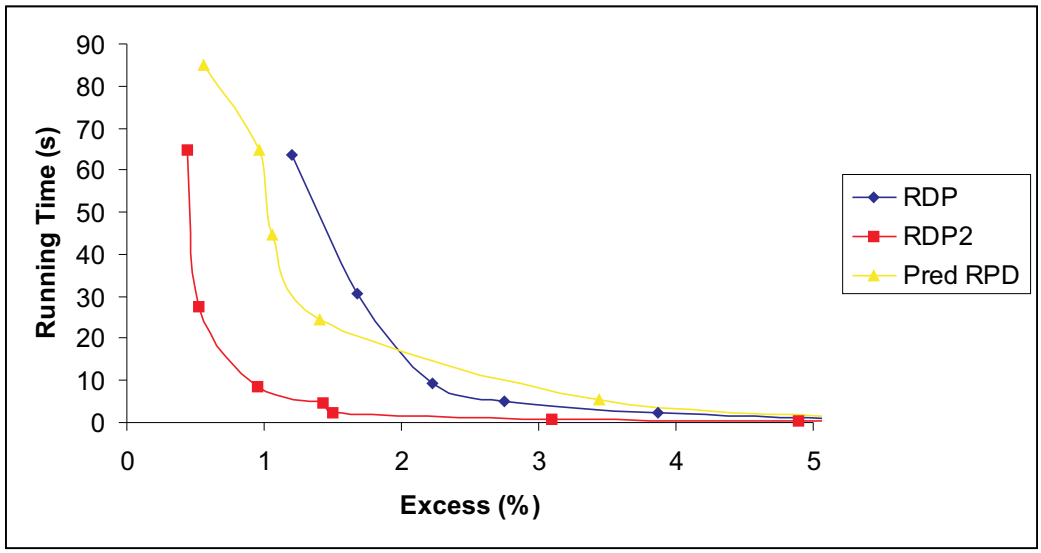

Figure 3.3: Comparison of restricted dynamic programming approaches to the 20-node supply ship TSP

Table 3.1 displays the excess percentage and running times for RDP2 with a number of values for the parameter H (some correspond to $i\binom{n}{i}$ for small i). When $H = 1$, RDP2 is equivalent to the nearest neighbour heuristic; this produced an excess of 27% but took almost zero time to compute. Perhaps the best value to use for a fast, quality solution is $H = 2000$, since this provides an excess of only 1.5 % in approximately 2 seconds.

The predictive costing approach used in Predictive RDP provided excellent improvements in the quality of solutions over RDP2 for each value of H (see Table 3.2), but the need to calculate a nearest neighbour tour for each partial tour as it was generated meant that CPU times did not compare favourably. Applying RDP2 with a larger value for H provides a better solution in a shorter time.

H	Excess (%)	CPU time (s)
1	27.012	0.002
20	14.156	0.016
50	10.263	0.036
100	7.883	0.070
200	6.464	0.149
300	5.802	0.234
380	5.702	0.299
400	5.307	0.316
500	4.888	0.400
1000	3.099	0.890
2000	1.500	2.204
3420	1.433	4.800
5000	0.957	8.530
10000	0.528	27.360
15000	0.437	64.818
20000	0.000	276.812

Table 3.1: Results using RDP2 for the 20-node problem

H	Excess (%)	CPU time (s)
1	13.037	0.083
20	5.834	1.312
100	3.438	5.427
500	1.400	24.420
1000	1.059	44.505
1500	0.965	64.608
2000	0.556	84.838

Table 3.2: Predictive RDP for the 20-node problem

The comparison of these restricted dynamic programming heuristics for twenty instances of the 30-node problem yielded similar patterns. The RDP2 heuristic proved the best for this larger problem. Results for RDP2 are shown in Table 3.3. Using a value of H between 2000 and 5000 may provide good results in a reasonable time.

H	Excess (%)	Time (s)
1	29.007	0.001
30	14.204	0.051
50	13.214	0.088
100	11.783	0.171
2000	10.400	0.345
300	8.946	0.530
400	8.525	0.727
500	8.088	0.931
870	7.227	1.759
1000	6.371	2.080
2000	4.622	4.938
5000	1.556	19.162
10000	0.517	60.724
12180	0.428	85.870
15000	0.352	146.908
20000	0.000	651.320

Table 3.3: Results using RDP2 for the 30-node problem

3.9.4 k-opt

Six varieties of a k-opt descent algorithm were tested on twenty instances of the 20-node problem and twenty instances of the 30-node problem (the same instances were used for the RDP testing).

1. **3-opt FI:** first improvement.
2. **3-opt BI:** best improvement.
3. **3-opt NR FI:** no reversed segment moves - first improvement.
4. **3-opt NR BI:** no reversed segment moves - best improvement.
5. **2-opt FI:** first improvement.
6. **2-opt BI:** best improvement.

In each case, the starting tour was generated using the nearest neighbour (NN) algorithm. Figure 3.4 and Table 3.4 display the quality of solution against average CPU time for the 20-node instances. In the figure, the closer the points lie to the bottom-left, the better the method. The figure also displays the performance of the RDP2 algorithm for comparative purposes. We see that none of these k-opt methods compete with the RDP2 method in terms of quality or running time. The nearest neighbour heuristic provided a starting solution with an excess of 27%. As expected, the 2-opt methods were fastest, reducing the excess percentage by 10 percentage points in very little time. The 2-opt best improvement method performed slightly better than the 2-opt first improvement method. The 3-opt NR best improvement had a slightly shorter running time than the 3-opt NR first improvement method, though both provided a similarly effective solution. The 3-opt FI and BI methods had much larger running times relatively, but produced better solutions on average: around 10% in excess of the

approximation for the optimum. The first improvement method ran longer than best improvement, but produced the best quality solution from the k-opt algorithms tested.

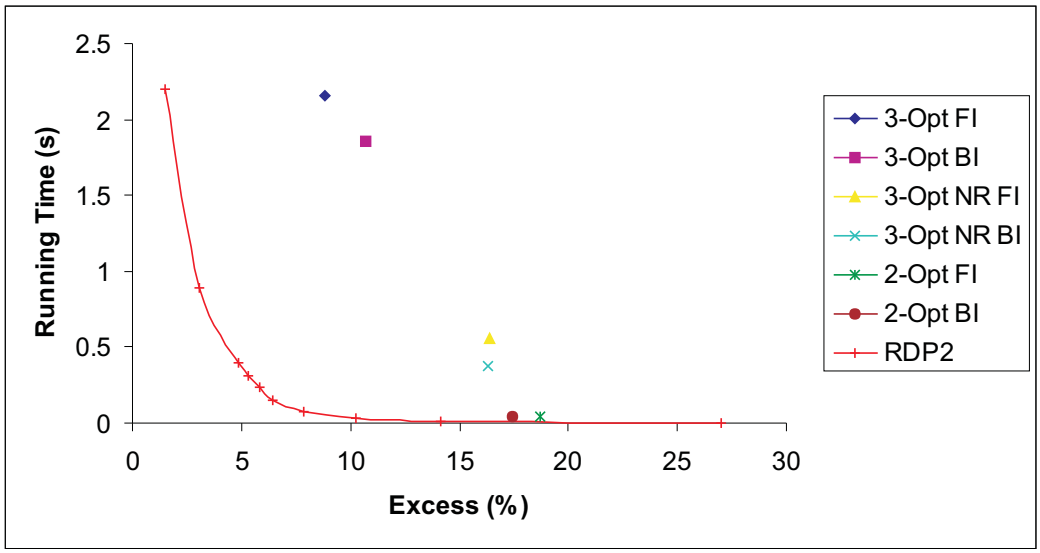

Figure 3.4: Comparison of k-opt approaches for the 20-node problem

	Excess (%)	CPU time (s)
3-opt FI	8.794	2.156
3-opt BI	10.710	1.852
3-opt NR BI	16.344	0.372
3-opt NR FI	16.416	0.562
2-opt BI	17.453	0.045
2-opt FI	18.695	0.043
NN	27.012	0.002

Table 3.4: Results using k-opt for the 20-node problem

Table 3.5 displays results for the 30-node instances. A graphical comparison between the opt methods and RDP2 displayed a picture very similar to Figure 3.4 for the 20-node instances. As we might expect, both Excess values and CPU times have increased, but the relative effectiveness of the approaches remains similar.

	Excess (%)	CPU time (s)
3-opt FI	14.714	21.059
3-opt BI	16.274	11.994
3-opt NR BI	19.477	2.550
3-opt NR FI	20.702	5.270
2-opt BI	21.785	0.202
2-opt FI	22.563	0.274
NN	29.007	0.001

Table 3.5: Results using k-opt for the 30-node problem

3.9.5 3-opt using an improved starting solution

Since the effectiveness of the k-opt algorithms may depend on the quality of the starting tour, the performance of these algorithms may improve if used with a starting tour generated by the RDP2 algorithm. Figure 3.5 shows how applying 3-opt best improvement to tours found using RDP2 with varying H improves the quality of the solution and increases average CPU time (the 3-opt first improvement approach produced almost identical results). We present results averaged across forty 20-node scenarios (the twenty used for RDP testing, plus another twenty that had been generated during construction of the algorithm, thus representing an even broader sample). We can see that the 3-opt algorithm provides a small improvement to the solution value on average, with a small increase in CPU time. The figure shows that only a section of the 3-opt line lies to the left of the RDP2 line, indicating an improvement over RDP2 for these values of H. The 3-opt approach appears to be worthwhile when used to improve on a RDP2 tour with $2000 \leq H \leq 4000$. Otherwise it is better to use RDP2 with a larger value for H to find the solution.

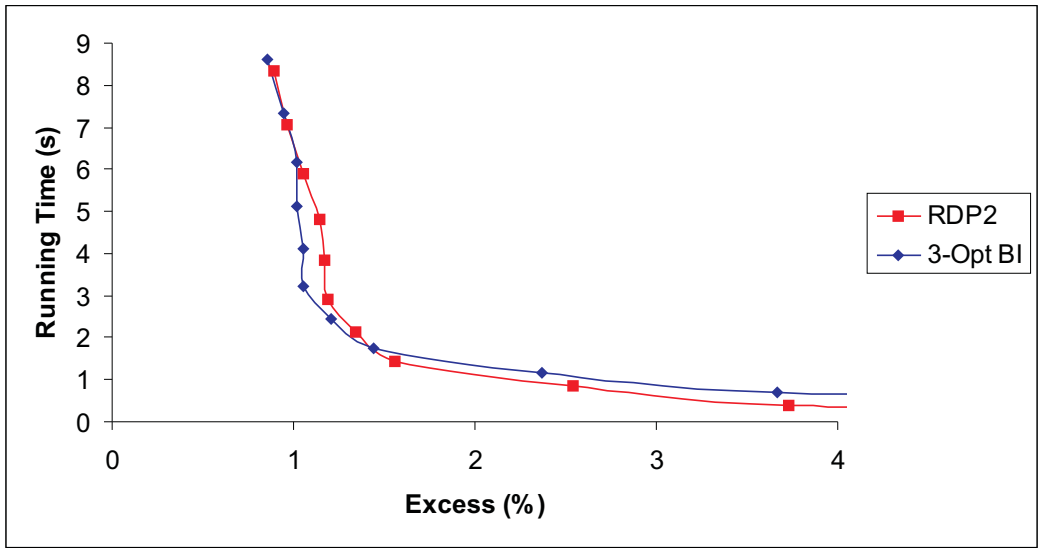

Figure 3.5: Using 3-opt to improve a tour found using RDP2 for the 20-node problem

In many of the instances, the RDP2 solution already had a 0% excess; applying 3-opt to such tours did not produce any tours that were better than our benchmark. Many of the solutions provided by RDP2 were found to be locally optimal with regard to the 3-opt search space, so applying 3-opt merely found that neighbour tours were inferior. This calculation only takes a fraction of a second, but will improve the solution in a few cases. Table 3.6 shows CPU times and solution quality improvement using 3-opt BI on starting tours generated by RDP2 with varying H. For values of H greater than 100, the average improvement in the solution is small, but the additonal CPU time required is also very small. For values of H over 1000, the bulk of the CPU time for the solution is due to the RDP2 starting tour.

H	Excess (%)	Improvement (%)	Total CPU time (s)	3-opt time (s)
1	13.436	18.031	1.663	1.662
100	8.052	1.043	0.498	0.428
500	3.669	0.064	0.688	0.310
1000	2.367	0.172	1.167	0.319
1500	1.444	0.115	1.743	0.313
2000	1.208	0.136	2.440	0.315
2500	1.056	0.130	3.234	0.306
3000	1.056	0.113	4.120	0.295
3500	1.020	0.121	5.115	0.300
4000	1.020	0.037	6.184	0.293
4500	0.947	0.013	7.351	0.287
5000	0.860	0.030	8.619	0.293

Table 3.6: Results using 3-opt BI to improve RDP2 solutions for the 20-node problem

3.10 Extensions to the work presented

We have tested our heuristics for problem sizes of 10, 20 and 30 nodes; these sizes seem reasonable and correspond roughly to the maximum number of ships in naval task groups. Although restricted dynamic programming clearly outperforms simple 3-opt for these instances, we have not tested them on larger scale problems.

Although we may apply some local search procedures that have been shown to be effective for the classical TSP, we should bear in mind that objective values for neighbouring solutions require significantly more computation: any change in node ordering results in different arc costs from the first altered node in the sequence onwards. In addition, since the arc costs may change continuously, techniques used in k-opt for the TSP to speed up the search [61], such as those employing lists of close neighbours, may actually lengthen overall computation. However, algorithms in the k-opt family that restrict the types of allowable move, such as 2.5-opt and Or-opt [99], could provide improved computation time results over 3-opt. The 2.5-opt neighbourhood includes all 2-opt moves, plus those 3-opt moves that delete a single node and reinsert elsewhere in the tour. Or-opt extends 2.5-opt to allow repositioning of segments of up to 3 nodes.

There is also scope for stochastic techniques to be applied to the supply ship TSP. We have only used deterministic approaches.

3.11 Conclusion

This chapter has studied an unusual variation of the travelling salesman problem, an important NP-hard combinatorial optimisation problem. A supply ship must visit and resupply a group of warships while they are in

motion; we have called this problem the supply ship travelling salesman problem. The potential for dynamic programming state-space relaxations to provide lower bounds for this problem was investigated. We found that the bounds provided by this method were probably not strong enough to be useful, on average. We have constructed and applied our variants of the restricted dynamic programming heuristic of Malandraki and Dial to 20-node and 30-node problems, highlighting an approach that seems to provides good solutions in a reasonable time. Finally, we built and applied simple k-opt descent algorithms to solve 20-node problems, discovering that the performance of these approaches is strongly dominated by a restricted dynamic programming approach.

Dynamic programming heuristics are best able to incorporate the strongly 'dynamic' nature of the problem. They are also flexible enough to handle additional concerns that could be added to the problem, such as time windows or precedence constraints.

Chapter 4

The supply ship scheduling problem

4.1 Introduction

Scheduling concerns the allocation of limited resources to required activities. Scheduling problems arise in many real-world environments; these include manufacturing industries, airports and hospitals [80]. Inspired by watercraft allocation issues arising in naval operations, we introduce the *supply ship scheduling problem*, a combinatorial optimisation problem featuring a fusion of flavours from the world of scheduling.

In the supply ship scheduling problem we wish to minimise the number of machines required to process a set of jobs with fixed start times and sequence-dependent set-up times, where any job may be simultaneously processed by multiple machines, obtaining a speed-up in processing time.

We may think of the jobs as being service/cargo tasks that must be completed at a number of ships/docks within an area of water. The machines/supply ships are mobile and travel between these static locations

to complete the tasks; travel times translate to sequence-dependent set-up times between jobs. It is assumed that these machines are identical and travel at the same speed.

The combination of these scheduling components eliminates the applicability of many of the algorithms used to tackle related problems. In this chapter, we introduce the supply ship scheduling problem and develop some heuristic methods for finding good solutions. A problem instance is represented as a directed graph: its arc values determine whether corresponding arcs appear in the transformed network associated with a particular solution. The neighbourhoods utilised in our descent and tabu search procedures are constructed through an analysis of minimum flows in these networks. We also formulate heuristics based on restricted dynamic programming that find feasible allocations of machines to jobs when the total number of machines is limited.

The individual features of the supply ship scheduling problem are listed in Section 4.2. Section 4.3 takes a brief look at the literature for scheduling problems that share some of these features. An introduction to minimum flows and the minflow-maxcut theory is provided in Section 4.4. Section 4.5 contains a description of the supply ship scheduling problem, then goes on to show how threshold values are used to narrow the set of possible solutions and create associated networks. Section 4.6 presents descent and tabu search improvement heuristics whose neighbourhoods are determined through analysis of maximum cuts in solution networks. Section 4.7 describes our initial restricted dynamic programming based heuristic. Our computational experience is presented in Section 4.8. Section 4.9 suggests areas for further study. Section 4.10 includes an improved formulation for a restricted dynamic programming approach that has not yet been coded or tested. Our conclusions are provided in Section 4.11.

4.2 Problem features

1. **Deterministic:** All data are known values.
2. **Non-pre-emptable jobs:** Execution of a job may not be interrupted. Once a machine begins to process a job, it is unavailable to perform any other activity until the job is completed.
3. **Fixed job start times:** The time at which the execution of a job begins is specified as part of the problem data. A schedule must allow for at least one machine to be available to process a job at its fixed start time.
4. **Sequence-dependent set-up times:** Before each job is processed the machine must be prepared. The time required for this preparation depends on the job that was last completed by the machine.
5. **Identical parallel machines:** A number of machines are available to process jobs. Distinct jobs may be processed simultaneously on different machines. Processing times are independent of the machine used.
6. **Moldable processing times:** The processing time of a job may be decreased by increasing the number of machines assigned to execute it. The number of machines is fixed before the job is started – once execution of a job has begun, no machines may quit the job until it is completed and no more may join in its execution.
7. **Objective – minimise the number of machines used:** We wish to find the smallest number of machines that allows us to complete all the jobs, subject to their fixed start times.

This combination of features does not appear to have been studied together in the literature.

4.3 Literature

Scheduling is applied in a variety of situations that require a sensible ordering of activities and allocation of resources. The subject has been widely studied and many approaches have been developed to find optimal or near-optimal solutions. Scheduling problems take many different forms; each has specific goals and conditions on the availability of resources. The basic idea of scheduling is to find an order in which to process *jobs* on one or more *machines*. A machine may only process a single job at a time. Jobs may be subject to a number of conditions (e.g. precedence constraints, release dates, due dates, weights, required machines). The processing order chosen depends on the objective (e.g. minimising the maximum completion time, minimising the number of late jobs). See the text by Pinedo [103] for details on scheduling theory and applications.

The volume of literature on scheduling problems is extensive. We shall narrow our view to three areas that best feature combinations covering some of the key aspects of the supply ship scheduling problem.

4.3.1 Identical parallel machines with sequence-dependent set-up times

This problem arises in production environments where the set-up times are significant. The time required to make a machine ready to process a new job depends on the job the machine last completed. In this parallel machine system, jobs may be partitioned into sets for processing by separate identical machines. This is an extension of the single-machine scheduling problem with sequence-dependent set-up times. (If the objective is to minimise the *makespan* – the maximum completion time – then this single-machine problem is equivalent to the *travelling salesman problem* (TSP), an NP-hard

problem [11]). Features that may appear in these problems include due dates, generally accompanied by an objective to minimise early/tardy costs or maximum tardiness, and *pre-emption* (job splitting). An alternative objective that arises when set-up costs are significant is the minimisation of the total set-up cost. We shall look at some of the heuristics that have been applied to problems that do not allow pre-emption.

Frederickson et al. [51] use heuristic procedures for the TSP to solve the parallel machine scheduling problem with sequence-dependent set-up times, where the objective is to minimise the maximum completion time. One heuristic for the k-TSP (where each city must be visited by one of k salesmen) builds k subtours simultaneously. Another method splits a good tour for one salesperson into k subtours.

Franca et al. [50] propose a heuristic procedure consisting of three phases:
1. An initial solution is constructed by assigning all jobs to the processors.
2. A local search employing a *tabu search* scheme is used to improve the solution's makespan by moving jobs between processors.
3. The sequence on the busiest machine is improved to obtain a better makespan.

Lee and Pinedo [79] consider the problem where jobs have due dates and associated *weights* (values that measure their relative importance). They propose a three phase heuristic to minimise the sum of the weighted tardinesses:
1. Factors associated with the due dates are computed.
2. A sequence is constructed using a *dispatching rule*. This is controlled through two parameters determined by the factors found in the first phase.
3. A *simulated annealing* method is applied to the solution provided by the second phase.

Radhakrishnan and Ventura [105] propose a *simulated annealing* method for solving parallel machine scheduling with earliness-tardiness penalties and sequence dependent set-up times.

Mendes et al. [93] compare two metaheuristic methods to solve the makespan minimisation problem. The first approach is the *tabu search* based heuristic by Franca et al. [50]. The second method is a *memetic algorithm*: a hybrid *genetic algorithm* procedure where a local search is applied to improve individuals. They state that the performance of the algorithms is greatly influenced by the parameters of the instances.

De Paula et al. [37] compare their *variable neighbourhood search* approach to three *greedy randomized adaptive search procedure* (GRASP) algorithms.

4.3.2 Moldable tasks

Due to its application to parallel computing, most of the literature in this area refers to machines as *processors* and jobs as *tasks*. A *parallelizable* task is one that can be "run on an arbitrary number of processors with a running time that depends on the number of processors allotted to it" [118]. A parallelizable task is described as *moldable* when "the number of processors to execute the task is not fixed but determined before the execution ... this number does not change until the completion of the parallelizable task" [42]. If the number of processors may change during execution the task is described as *malleable* [18, 19, 87].

Du and Leung [41] study the complexity of scheduling *parallel task systems*. The parallel tasks presented match the moldable task description. The number of processors in the system is denoted by m. The objective is to minimise the schedule length (makespan) on $m \geq 2$ identical processors. Their analysis includes non-pre-emptive schedules with empty precedence

constraints. They show that the optimal schedule can be found in pseudo-polynomial time for $m = 2$ and 3 and is strongly NP-hard when $m \geq 5$. It is not known whether the problem is strongly NP-hard or solvable in pseudo-polynomial time when $m = 4$.

Belkhale and Banerjee [13] present an approximate algorithm for the *partitionable independent task scheduling problem*. The n independent tasks described fall within the moldable task category. The system consists of m processors. A task i completes in time $\frac{p_i}{\sigma_k(i)}$ when run on k processors, where p_i is the time task i takes to run on a single processor, and $\sigma_k(i)$ is the estimated speed-up that can be obtained by running i on k processors. The objective is to minimise the schedule finish time. For $i \in \{1, \ldots, n\}$ and $1 \leq k \leq m-1$, they assume $\sigma_k(i) \leq \sigma_{k+1}(i)$ and $\frac{\sigma_k(i)}{k} \geq \frac{\sigma_{k+1}(i)}{(k+1)}$ (a result of a convex speed-up curve). The approximate algorithm they present guarantees a solution within $\frac{2}{1+1/m}$ of the optimal solution. The *largest processing time* (LPT) algorithm is a *list scheduling* algorithm in which the tasks are first sorted into a list, ordered by decreasing execution times. The list scheduling algorithm builds up a partial schedule and assigns tasks from the list to processors with the earliest finish times. The main idea of the algorithm of Belkhale and Banerjee is to construct an LPT schedule and iteratively modify it by assigning a task to more processors, provided the assignment results in an immediate decrease in the schedule time. They state that the algorithm can be implemented in $O(n \log n + nm \log m)$ time.

In the system of Turek et al. [118] a task's execution time is given by a non-increasing function of the number of processors allotted to it. They present polynomial time approximate algorithms which provide schedules with length no worse than twice the optimal length. Unlike the algorithm of Belkhale and Banerjee, they do not constrain the running time of the tasks. They provide a family of algorithms that extend techniques used to solve *resource allocation problems* [73]. An algorithm selects candidate numbers

of processors to be allocated to each of the tasks. Once processor numbers are fixed, the problem becomes the easier *multiprocessor scheduling problem* [40].

Parallelizable task systems may be viewed as a generalization of the *orthogonal rectangle packing problem* [31] – rectangles (representing tasks) must be placed within a rectangular 'bin'. The width of a rectangle represents the number of processors used, while the height is the task's completion time when allotted that number of processors. The height of the bin's contents when all rectangles have been placed is the makespan of the corresponding schedule. This packing problem is known to be NP-hard [31]. Since tasks are moldable, each task is represented by a set of rectangles – one rectangle for each possible processor number allocation. Only one rectangle from this set is placed into the bin. Possible heuristics to solve this packing problem include *shelf* algorithms: rectangles are placed into the bottom of the bin, a 'shelf' is then laid over them, resting at the height of the tallest rectangle below. The process continues by placing rectangles on the shelf and laying another shelf on top of them. Turek et al. [117] apply this idea to develop shelf-based algorithms for scheduling parallelizable (moldable) tasks.

Monte and Pattipati [96] develop sub-optimal algorithms for scheduling parallelizable (moldable) tasks to minimise the makespan and weighted sum of the task completion times. The algorithms use *Lagrangian relaxation* [98] – a technique that removes constraints from the problem and adds them into the objective function through the use of *Lagrange multipliers*. Multipliers are found approximately by an iterative method. A feasible solution to the original problem is found based on the solution to the relaxed problem (which itself provides a lower bound).

Burke et al. [22] consider the complexity of scheduling independent malleable tasks. They first study non-pre-emptable task problems where assigned sets of processors do not change until their job is completed. The processing time of a job i is given by $p_i(k) = \frac{p_i}{f(k)}$, a non-increasing function in k for all $i \in \{1, \ldots, n\}$, where k is the number of machines assigned to job i, p_i is a job-specific value and f is a given function. They note that if $p_i(k) \leq \frac{p_i}{k}$, all processors should be assigned to each job in turn and "optimal maximal completion time is the same irrespective of the schedule" [22]. They provide a proof that the problem is NP-hard for the case in which $p_i > p_i(k) > \frac{p_i}{k}$ when $k > 1$.

4.3.3 Tactical fixed job scheduling

In an *interval scheduling* problem there are n jobs available to be processed on m parallel machines. Each job j features a *time window* given by a ready time r_j and deadline d_j between which the job must be completed. If jobs may not be delayed after their ready times the problem is known as a *fixed job scheduling* problem. If the objective is to minimise the total number of machines needed to process all the fixed jobs we call it *tactical fixed job scheduling* (TFJS).

Gertsbakh and Stern [54] introduce the idea of using a step function to solve the TFJS. The value of the function is zero at time zero. Whenever a job starts, the value is increased by one; whenever a job finishes it is decreased by one. The maximum value of the function is the total number of machines required. An optimal assignment of jobs to machines is constructed by forming a string of jobs that are known to be executable by one machine, removing the string from the list of remaining jobs, and repeating. The algorithm is $O(n^2)$ in the worst case. Gupta *et al.* [58] present an optimal algorithm of $O(n \log n)$.

The tactical fixed job scheduling problem is a special case of *Dilworth's problem* [38, 54], which involves the decomposition of a finite set into disjoint sequences that obey an ordering on the elements.

4.3.4 Conclusions

We have provided a very brief overview of the methods that have been applied to three different areas within scheduling. These areas were selected because they share some characteristics with the supply ship scheduling problem. The first two problems are NP-hard, while the third may be solved by an efficient algorithm.

Local search metaheuristics (including tabu search, memetic algorithms and variable neighbourhood search) proved the most popular for scheduling identical parallel machines with sequence-dependent set-up times. Approximate algorithms and shelf-based algorithms were used to schedule parallelizable (moldable) tasks by relating the schedule to bin packing problems. The final problem, tactical fixed job scheduling, may be solved exactly using a polynomial time algorithm.

The combination of the problem characteristics eliminates the direct applicability of their solution methods to the supply ship scheduling problem.

4.4 The minimum flow problem in a directed network

In this section we provide an introduction to determining minimum flows in networks, as some of our approaches to the supply ship scheduling problem rely on this methodology. Networks arise in many applications. Physical networks, such as communication, electronic and transportation systems, are among the most easily recognised.

In a *network flow* problem we wish to determine the *flow* along each arc in the network. In a capacitated network, flow along an arc must be less than or equal to the *capacity* of the arc. We may imagine that the network is a system of pipes carrying water. Each arc is a pipe, with the size of the pipe determining its capacity. Pipes are connected to each other through the nodes at each endpoint. The network may have a *source* node and a *sink* node. The source node can represent water entering the system from outside, while water leaves the system through the sink node. In the nodes in-between, the amount of flow that leaves a node is equal to the amount that enters it (*mass balance*).

4.4.1 The maximum flow problem

An important optimisation problem in network flows is the maximum flow problem, for which several polynomial-time algorithms have been developed.

Maximum flow problem: In a capacitated directed network we wish to send as much flow as possible between two special nodes, a source node s and a sink node t, without exceeding the capacity of any arc [3].

Most algorithms for solving the maximum flow problem fall into two categories:

- *Augmenting path algorithms*: Flow is incrementally augmented along paths from the source node to the sink node. Mass balance constraints are maintained at every node except the source and sink, i.e. total flow into a node equals total flow out.
- *Preflow-push algorithms*: These "flood the network so that some nodes have excesses (or build-up of flow). These algorithms incrementally relieve flow from nodes with excesses by sending flow from the node forward toward the sink node or backward toward the source node" [3].

Algorithms for the maximum flow problem may be used to tackle the minimum flow problem.

Residual networks

Residual networks form an important part of many maximum flow algorithms. "Given a flow x, the residual capacity r_{ij} of any arc (i, j) is the maximum additional flow that can be sent from node i to node j using the arcs (i, j) and (j, i)" [3]. The amount that flow can be increased along arc (i, j) is given by the unused capacity of arc (i, j) plus the current flow on arc (j, i) which may be cancelled. The residual network with respect to the flow x consists of the arcs with positive residual capacities.

4.4.2 The minimum flow problem

Following [29], we consider a capacitated network $G = (N, A, l, c, s, t)$. N is the set of nodes while A is the set of arcs in the network. A non-negative capacity $c(i, j)$ and non-negative lower bound $l(i, j)$ is associated with each arc $(i, j) \in A$. The network contains a source node s and a sink node t. A *flow* is a function $f : A \to \mathbb{R}^+$ satisfying the following conditions:

$$f(i, N) - f(N, i) = \begin{cases} v & \text{for } i = s \\ 0 & \text{for } i \neq s, t \\ -v & \text{for } i = t \end{cases} \tag{4.1}$$

$$l(i, j) \leq f(i, j) \leq c(i, j), \quad \text{for all } (i, j) \in A \tag{4.2}$$

for some $v \geq 0$, where

$$f(i, N) = \sum_{j|(i,j)\in A} f(i, j)$$

and

$$f(N, i) = \sum_{j|(j,i)\in A} f(j, i)$$

We call v the *value* of the flow f. We may think of $f(i, N)$ as the flow out of node i, while $f(N, i)$ is the flow into node i. For nodes other than the sink node s and source node t, flow into a node must equal the flow out. Flow along any arc must be at least the value of the lower bound l for that arc while not exceeding the capacity c of the arc. The minimum flow problem is to determine a flow f for which v is minimised. [29]

A *cut* $[S, \bar{S}]$ is a partition of the node set N into two subsets S and $\bar{S} = N - S$. We refer to $[S, \bar{S}]$ as an s - t cut if $s \in S$ and $t \in \bar{S}$. An arc (i, j) is called a *forward arc* of the cut if $i \in S$ and $j \in \bar{S}$, and a *backward arc* if $i \in \bar{S}$ and $j \in S$. Let $(S, \bar{S})$ denote the set of forward arcs in the cut and $(\bar{S}, S)$ the set of backward arcs.

For the minimum flow problem the capacity, $c[S, \bar{S}]$, of an s - t cut, $[S, \bar{S}]$, is defined as the sum of the lower bounds of the forward arcs minus the sum of the capacities of the backward arcs:

$$c[S, \bar{S}] = l(S, \bar{S}) - c(\bar{S}, S)$$

Note that there is a difference between this definition and the more commonly encountered definition of capacity of a cut for a maximum flow problem, where $c[S, \bar{S}] = c(S, \bar{S}) - l(\bar{S}, S)$.

A *maximum cut* is an s - t cut whose capacity is the maximum among all s - t cuts.

Min-flow max-cut theorem [29]: If there exists a feasible flow in the network, the value of the minimum flow from a source node s to a sink node t in a capacitated network with non-negative lower bounds equals the capacity of the maximum s - t cut.

The minmax algorithm

This approach to solving the minimum flow problem is described in [29]. The minimum flow problem can be solved by determining a maximum flow from the *sink* node to the *source* node in the *residual network* given by a *feasible flow*. Any maximum flow algorithm may be used.

1. Let f_f be a feasible flow in network G.
2. Determine the residual network, G_f.
3. Establish a maximum flow, f_r, from t to s in G_f.
4. Combine flows f_f and f_r into the resultant flow f.
5. f is a minimum flow from the source node s to the sink node t.

If a feasible flow is not known, one may be determined by a transformation of the network and application a maximum flow algorithm. "The complexity of the minimum flow problem is equal to the complexity of the maximum flow algorithm used for determining a feasible flow and for establishing a maximum flow from t to s" [29].

4.4.3 The tanker scheduling problem

The following example problem and solution procedure are based on an application of the maximum flow problem demonstrated by Ahuja at al. [3]. The solution network bears some similarities to the transformed network we develop in Section 4.5.2.

A steamship company must deliver goods between several pairs of ports. The customers have specified precise delivery dates when the shipments must reach their destinations: they may not arrive early or late. The steamship company wants to know the minimum number of ships needed to meet the delivery dates.

Shipment	Origin	Destination	Delivery Date
1	Port A	Port C	3
2	Port B	Port D	5
3	Port A	Port D	9
4	Port B	Port C	13

Table 4.1: Example data for the tanker scheduling problem

	C	D
A	3	4
B	4	5

	A	B
C	2	3
D	3	4

Table 4.2: Shipment transit times (left) and return times (right)

This problem may be solved by constructing the network shown in Figure 4.1. Each node corresponds to a shipment. There is an arc from node i to node j if it is possible for a single tanker to deliver shipment j after completing shipment i. Directed paths in the network correspond to feasible sequences of shipments. To solve the tanker scheduling problem we must find the minimum number of directed paths that contain each node in the network on exactly one path.

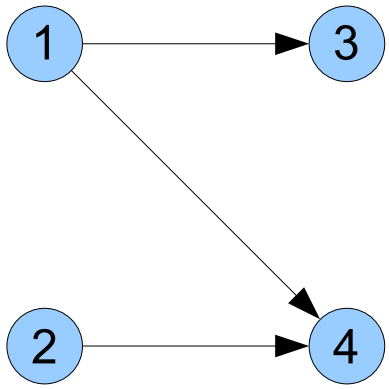

Figure 4.1: Network representing feasible sequences of consecutive shipments

We may transform this network into a minimum flow problem.

1. Split each node i into two nodes: i' and i''.
   - Add an arc (i', i'').
   - Set a lower bound of 1 on the arc.
2. Add a source node s.
3. Connect s to the origin of each shipment; i.e. add arcs (s, i'). (These arcs represent putting ships into service).
4. Add a sink node t.
5. Connect each destination node to t; i.e. add arcs (i'', t). (These arcs represent taking ships out of service).

6. Set the capacity of each arc in the network equal to 1.

Figure 4.2 displays the resulting network. Each directed path from s to t corresponds to a feasible schedule for a single ship: a feasible flow of value x in the network may be decomposed into schedules of x ships. The problem has been reduced to identifying a feasible flow of minimum value.

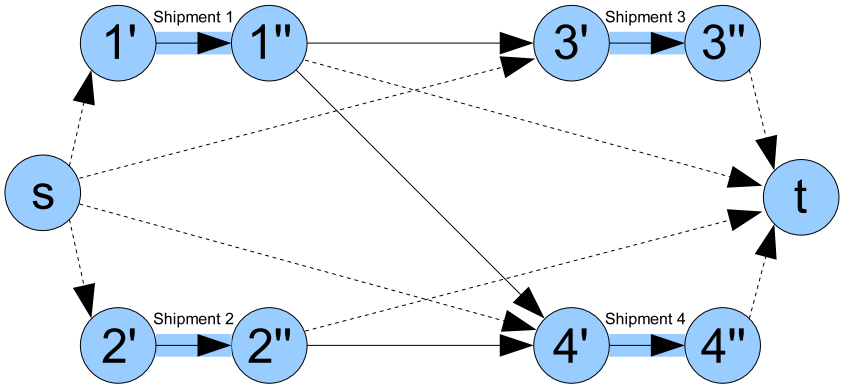

Figure 4.2: Minimum flow model of the tanker scheduling problem

4.5 The supply ship scheduling problem

Let $N = \{1, \ldots, n\}$ be the set of jobs to be processed, numbered by the order of their fixed start times, s_i, $i = 1, \ldots, n$. Jobs may not be interrupted and must commence at their fixed start times: they cannot be early or late. Machines are identical and at least one machine must process each job.

Let t_{ij} be the set-up time for job j when it follows job i on a machine; $i < j$; $i = 1, \ldots, n-1$; $j = 2, \ldots, n$. Set-up times are equivalent to travel times for a machine travelling between job locations. We define these to satisfy:

$$t_{ij} \geq 0 \qquad \text{for} \quad i < j; \tag{4.3}$$

$$t_{ik} \leq t_{ij} + t_{jk} \qquad \text{for} \quad i < j < k. \tag{4.4}$$

We assume that the processing requirements for any job may be divided up equally between the machines assigned to it. Let p_i be the processing time required by a single machine to complete job i. Thus if a solution assigns m_i machines to job i, the processing time function is

$$p_i(m_i) = \frac{p_i}{m_i}.$$

A solution takes the form of a set $M = \{m_1, \ldots, m_n\}$, where each $m_i \in \mathbb{N}$ is the number of machines that are assigned to process job $i \in N$. (Section 4.5.2 covers the decomposition of this into a schedule). Let $\mathbb{M}$ represent the set of all solutions.

Let $Q(M)$ represent the minimum number of machines required for a solution M to be feasible.

The goal is to find $M^* \in \mathbb{M}$, a solution that minimises the total number of machines that are required; that is,

$$M^* = \underset{M \in \mathbb{M}}{\operatorname{argmin}} \{Q(M)\}.$$

Notes:

- Assigning a separate machine to process each job provides an immediate solution, $M = \{1, \ldots, 1\}$, where at most n machines are used in total. We may therefore ignore all solutions with $m_i > n$ for some $i \in N$.
- For a solution $M = \{m_1, \ldots, m_n\}$, a lower bound on $Q(M)$ is given by $\max_{i \in N} \{m_i\}$.
- We may determine the value of $Q(M)$ using minimum flows (more on this in Section 4.5.2).
- Since jobs have been numbered in order of their fixed start times, s_i, set-up times for t_{ij} with $i \geq j$ are not needed.
- Machines originate from a 'depot'. Set-up/travel times from the depot to job locations are not needed: the fixed start times of jobs must allow sufficient time for machines to travel directly to any job from the depot.
- Table 4.3 displays problem data for an instance of the supply ship scheduling problem where four jobs must be completed. We shall use this instance in subsequent sections to illustrate how our networks are constructed.

i	1	2	3	4
s_i	18	20	36	58
p_i	26	24	22	30
t_{1i}		3.89	3.13	1.06
t_{2i}			6.22	3.68
t_{3i}				4.17

Table 4.3: Example data for a 4-job problem

4.5.1 Threshold graph

When looking at this scheduling problem it is important to know which jobs may be scheduled one after another on a particular machine. We calculate how many machines must be assigned to a job so that the machines involved are available to process any subsequent job.

Let $a_{ij} \in \mathbb{N}$ be the minimum number of machines that must be assigned to job i so that at least one of these machines is available to process job j. We shall refer to these a_{ij} as *threshold values*.

- If $p_i > 0$ and $s_i + t_{ij} < s_j$ we have

$$s_i + \frac{p_i}{a_{ij}} + t_{ij} \leq s_j \ . \tag{4.5}$$

  (4.5) represents the condition that job i must start on time at s_i, be processed (taking time equal to p_i/a_{ij}) and then job j must be set up (time t_{ij}) before the fixed start time s_j. We wish to find the smallest integer a_{ij} for which this condition holds. Let $a^*_{ij} \in \mathbb{R}^+$ represent a_{ij} in a situation where jobs may be processed by partial machines. We then have:

$$s_i + \frac{p_i}{a^*_{ij}} + t_{ij} = s_j$$

$$\Rightarrow a^*_{ij} = \frac{p_i}{s_j - s_i - t_{ij}}$$

  We require a_{ij} to be integer, so we let $a_{ij} = \lceil a^*_{ij} \rceil$. This provides $p_i/a_{ij} \leq p_i/a^*_{ij}$. So if $p_i > 0$ and $s_i + t_{ij} < s_j$ then

$$a_{ij} = \left\lceil \frac{p_i}{s_j - s_i - t_{ij}} \right\rceil .$$

- If $p_i > 0$ and $s_i + t_{ij} \geq s_j$ it is not possible for a machine to process both jobs i and j: we denote its value as $a_{ij} = \infty$ to represent infeasibility.

- If $p_i = 0$ then

$$a_{ij} = \begin{cases} 1 & \text{if } s_i + t_{ij} \leq s_j \\ \infty & \text{if } s_i + t_{ij} > s_j \end{cases} \tag{4.6}$$

We will never assign more than n machines to a job, thus we may treat any $a_{ij} > n$ as $a_{ij} = \infty$.

The resulting threshold values for our example instance are displayed in Table 4.4.

		j			
		1	2	3	4
i	1	-	∞	2	1
	2	-	-	3	1
	3	-	-	-	2
	4	-	-	-	-

Table 4.4: Threshold value table for the 4-job example

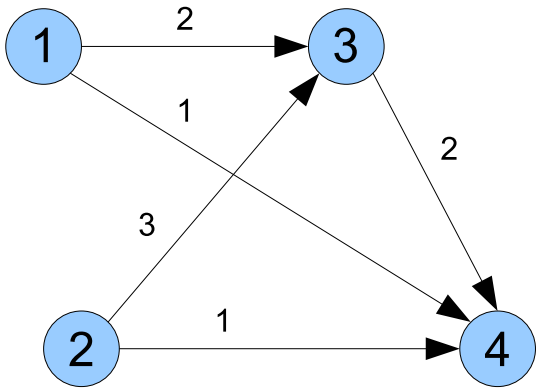

Figure 4.3: Threshold graph for the 4-job example

We may represent the threshold values in a directed graph $G_a = (N, \Gamma)$ (see Figure 4.3). Each job is represented by a node, so the set of nodes is $N = \{1, \ldots, n\}$. We will call these *job-nodes*. There is an arc $(i, j) \in \Gamma$ if

and only if $a_{ij} \in \{1, \ldots, n\}$. We will call these *threshold arcs*. We associate the values a_{ij} with their corresponding arcs. This graph provides a step towards seeing how machines might move from job to job, and which jobs may not directly follow each other on the same machine.

Theorem 4.1 *If there is an arc (i, j) and an arc (j, k) in the threshold graph then there must also be an arc (i, k). The corresponding threshold values satisfy $a_{ik} \leq a_{ij}$.*

Proof: By definition we have

$$(i, j) \in \Gamma \iff s_i + t_{ij} \leq s_j \text{ and } a_{ij} \in \{1, \ldots, n\}.$$

Since $(i, j), (j, k) \in \Gamma$,

$$s_i + t_{ij} \leq s_j \text{ and } s_j + t_{jk} \leq s_k$$

$$\Rightarrow s_i + t_{ij} + s_j + t_{jk} \leq s_j + s_k$$

$$\Rightarrow t_{ij} + t_{jk} \leq s_k - s_i.$$

Using Inequality (4.4):

$$t_{ik} \leq t_{ij} + t_{jk} \leq s_k - s_i$$

$$\Rightarrow s_i + t_{ik} \leq s_k. \tag{4.7}$$

We must now show that $a_{ik} \leq n$.

Case: $p_i > 0$.

$$(i, j) \in \Gamma \iff s_i + t_{ij} < s_j \text{ and } a_{ij} \in \{1, \ldots, n\}. \tag{4.8}$$

By (4.8), $a_{ij} \in \{1, \ldots, n\}$, so $0 < a^*_{ij} \leq n$ satisfies the following equation:

$$s_i + p_i/a^*_{ij} + t_{ij} = s_j. \tag{4.9}$$

For $a^*_{ik} > 0$, we also have

$$s_i + p_i/a^*_{ik} + t_{ik} = s_k. \tag{4.10}$$

Let $c \geq 0$ represent p_j/a^*_{jk} for the case $p_j > 0$, and a slack value for the case $p_j = 0$. In addition to (4.9) and (4.10) we have

$$s_j + c + t_{jk} = s_k. \tag{4.11}$$

(4.9)+(4.11)−(4.10) gives us

$$\begin{aligned} &\frac{p_i}{a^*_{ij}} + c - \frac{p_i}{a^*_{ik}} + t_{ij} + t_{jk} - t_{ik} = 0 \\ &\Rightarrow \frac{p_i}{a^*_{ij}} + c - \frac{p_i}{a^*_{ik}} = t_{ik} - t_{ij} - t_{jk}. \end{aligned} \tag{4.12}$$

Using (4.4) and (4.12) we now know

$$\frac{p_i}{a^*_{ij}} + c - \frac{p_i}{a^*_{ik}} \leq 0.$$

Since $p_i, a^*_{ij}, a^*_{ik} > 0$ and $c \geq 0$ we have

$$\frac{p_i}{a^*_{ij}} \leq \frac{p_i}{a^*_{ik}} \;\Rightarrow\; a^*_{ik} \leq a^*_{ij} \;\Rightarrow\; a_{ik} \leq a_{ij}.$$

So $a_{ik} \leq a_{ij}$, and thus $a_{ik} \in \{1, \ldots, n\}$.

Case: $p_i = 0$.

Minimising $a_{ij}, a_{ik} \in \{1, \ldots, n\}$ while satisfying the fixed start times (following (4.7)) provides:

$$s_i + p_i/a_{ij} + t_{ij} \leq s_j \;\Rightarrow\; a_{ij} = 1;$$

$$s_i + p_i/a_{ik} + t_{ik} \leq s_k \;\Rightarrow\; a_{ik} = 1.$$

□

This means that if it is possible for a machine to process jobs i, j and k in sequence, then it is also possible for that machine to instead process job i and then job k without processing job j. As a consequence, if there is a directed path (of any length) from job x to y then there is an arc from x to y. From this we know that it is not necessary to send surplus machines through intermediate jobs in order for them to access later jobs.

Cutting down the size of the solution space $\mathbb{M}$

Using the upper bound n we restrict each m_i to take values in $\{1, \ldots, n\}$. We may restrict the possible values further by using the threshold values. The objective of the problem is to minimise the total number of machines used to process the jobs in N; in part, we try to minimise the number of machines assigned to any job. Having established the consequence of Theorem 4.1, that 'surplus' machines are not needed, the only remaining reason to increase m_i is to shorten job i's processing time: with a sufficient decrease in processing time, some of the machines already allocated to i are able to process some job j that is unavailable to them at the lower value of m_i. Therefore, the only values for m_i we need consider are the smallest values that allow previously unreachable jobs to be reached by machines.

Let A_i be an ordered set of the distinct threshold values leading from job i: the values at which the set of possible successor jobs expands. So $A_i = \{a|\ a = a_{ij} \text{ for some } a_{ij} \in \{1, a_{i,i+1}, \ldots, a_{in}\}\}$. The elements of A_i are listed in order of increasing magnitude. We may restrict the possible values for the number of machines assigned to job i to belong to A_i. That is

$$m_i \in A_i\ ,\ \forall\ i \in N.$$

A_i must include the value '1': any job i without successors is assigned only one machine in an optimal solution. For the 4-job example shown in Table 4.4 and Figure 4.3 we have:

$$m_1 \in \{1, 2\},\ m_2 \in \{1, 3\},\ m_3 \in \{1, 2\},\ m_4 \in \{1\}.$$

Any new information provided by upper bounds may be used to narrow the useful set of solutions. When a solution M with $Q(M) < n$ is discovered we may eliminate any values in A_i greater than $Q(M)$.

4.5.2 Network flows

For a particular solution M we wish to know $Q(M)$, the minimum number of machines needed to allow the configuration given by M. We may calculate $Q(M)$ by constructing a capacitated network, where the passage of machines between jobs is represented by flow along the arcs. $Q(M)$ is equal to the value of the minimum flow in the network corresponding to M.

Constructing the network

We have the set of jobs, $N = \{1, \ldots, n\}$, the number of machines assigned to each job for a particular solution, $M = \{m_1, \ldots, m_n\}$, and the threshold graph, G_a. We will construct a capacitated network, G_M. Each arc (i, j) has an associated non-negative capacity, $c(i, j)$, and non-negative lower bound, $l(i, j)$. *Inter-job* arcs of G_M represent the opportunity for machines to travel between jobs. *Intra-job* arcs are included to reflect the constraints provided by M. The flow value along an arc from job-node i to j represents the number of machines that process j immediately after processing i. The construction of network G_M begins with G_a as a foundation.

1. Initially, all arcs and nodes in G_a appear in G_M.
2. Each job-node must be assessed to see which arcs emanate from the node. If any of the threshold arcs emanating from job i in G_a have $a_{ij} > m_i$, they are removed from G_M. The remaining arcs represent which jobs may be performed in succession by a machine, given M.
3. A source node s is added, and arcs to each job-node from the source. These arcs represent the ability of as yet unused machines to travel from the machine depot to any job.
4. A sink node t is added, and arcs from each job-node to the sink.

5. All the arcs currently in the network are considered to have unlimited capacity and a lower bound of zero.

6. Each job-node i is split into two nodes: an entry node, i', and an exit node, i''. The entry node now receives all arcs that terminated at the job-node. All arcs that emanated from the job-node now leave the exit node. A single arc connects the entry node and exit node. Flow along this arc is constrained to be equal to the number of machines allocated to the job by M. So the intra-job arc corresponding to job i has both lower bound and capacity equal to m_i.

The resulting network for a solution $M = \{2, 1, 1, 1\}$ of our 4-job example is displayed in Figure 4.4. Arc values represent both lower bounds and capacity of flow.

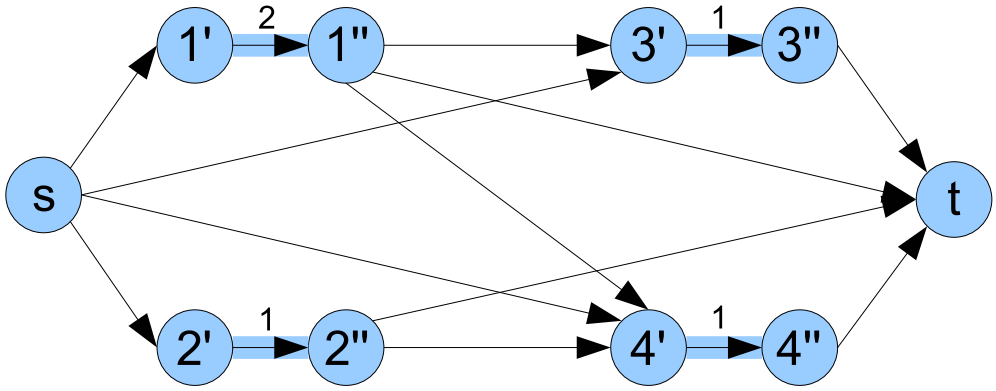

Figure 4.4: Network flow model of the 4-job example problem

Determining the minimum flow

We use the minmax algorithm described in Section 4.4.2. Since the time complexities of most maximum flow algorithms are dependent on the number of nodes and arcs in the network, it may be useful to reduce the size

of the network before applying the algorithm. An example of a reduced network is given in Figure 4.5.

- Any pair of nodes (i', i'') connected *only* to the source node s and sink node t may be removed from the network along with their incident arcs. Any flow along such a path is fixed by the constraints on the intra-job arc and cannot be changed by the maximum flow algorithm.
- Nodes with exactly one terminating arc and one emanating arc may be eliminated; the incident arcs may be replaced with a single arc connecting the nodes adjacent to the eliminated node. The highest lower bound and lowest capacity of the removed arcs apply to the new arc.

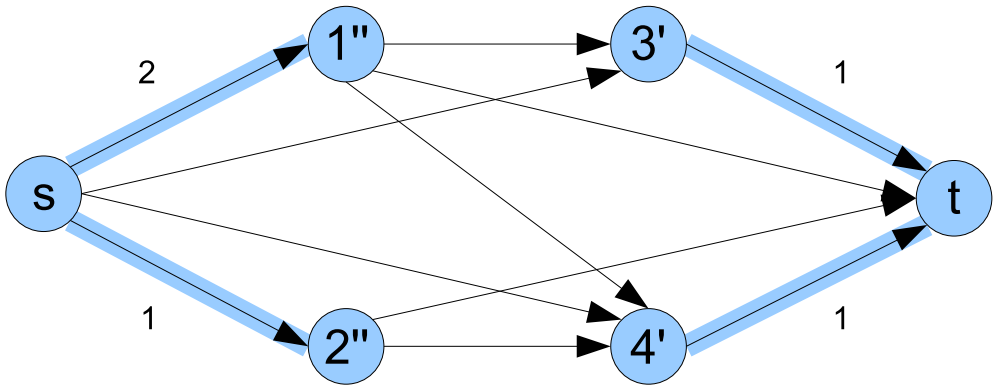

Figure 4.5: Reduced network flow model of the 4-job example problem

The first stage of the minmax algorithm is to find a feasible flow, f_f. Since every job-node was connected to both the source and the sink it is always possible to find a feasible flow. The simplest feasible flow is to send m_i units of flow directly along the arc from the source node to the entry node of job i (arc (s, i') or (s, i'')) then m_i units of flow directly along the arc from the exit node of job i to the sink node (arc (i'', t) or (i', t)). In this way, the flow between nodes i' and i'' meets the lower bound and capacity constraints of that arc. An example of this feasible flow is provided in Figure 4.6.

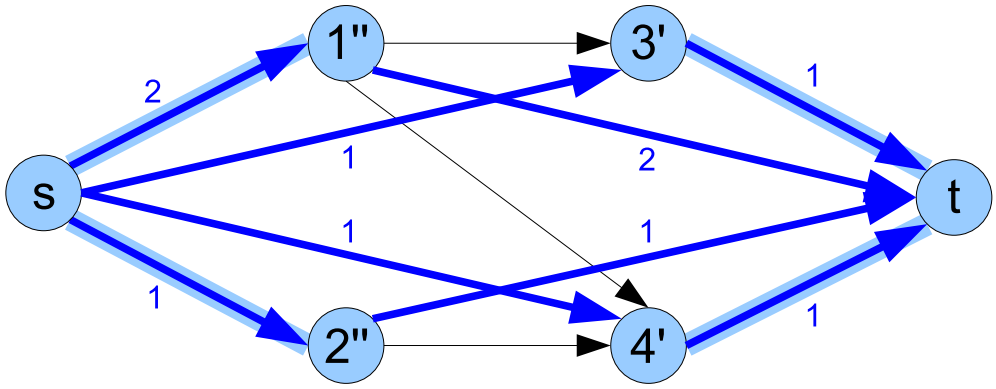

Figure 4.6: A feasible flow for the 4-job example problem. Blue arc values represent flow.

Now that a feasible flow, f_f, has been established we must create its residual network, G_f. An example of such a residual network is shown in Figure 4.7.

- Since the flow on intra-job arcs is fixed (i.e. lower bound = capacity), the residual capacity for such arcs must be zero and so these arcs do not appear in the residual network.
- We treat inter-job arcs as having unlimited residual capacity. In practice we let the residual capacity of such arcs be $1 + \sum_{i \in N} m_i$. In this way these arcs will always have a positive residual capacity.
- The arcs from the source and arcs to the sink are those that carry the flow in the feasible flow. The flow along such arcs may be cancelled to increase the flow in the reverse direction.

We may now treat the residual network, G_f, as a network in which we wish to find a maximum flow, f_r, from t to s. The residual capacity of an arc may be treated as the capacity of the arc. Figure 4.8 provides an example of a maximum flow.

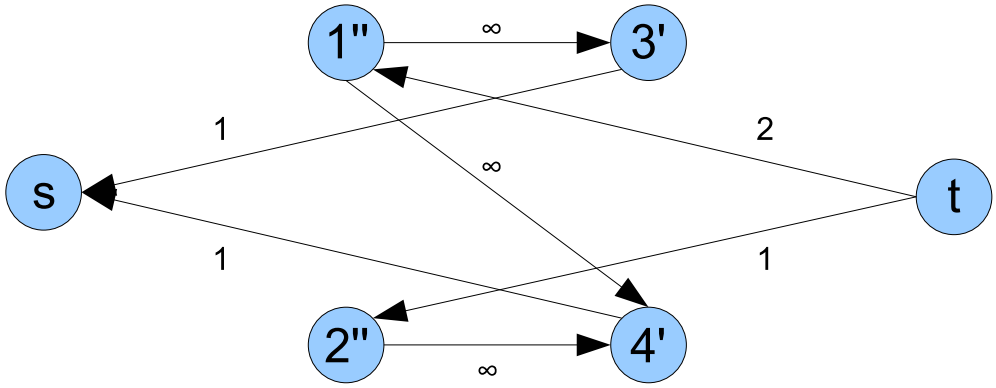

Figure 4.7: Residual graph with respect to the feasible flow for the 4-job example problem

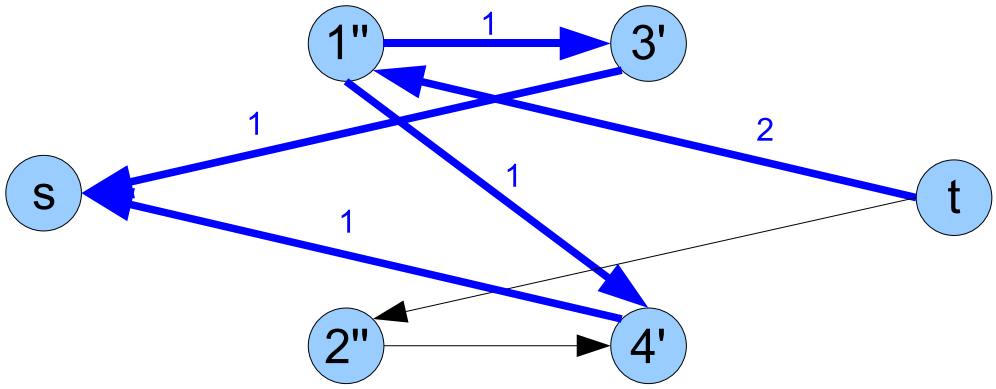

Figure 4.8: A maximum flow from t to s in the residual network for the 4-job example problem

The feasible flow, f_f, and maximum flow, f_r, from t to s in the residual network may be combined to form the minimum flow, f, in our reduced network. Figure 4.9 displays a minimum flow for our example: the combination of flows in Figures 4.6 and 4.8.

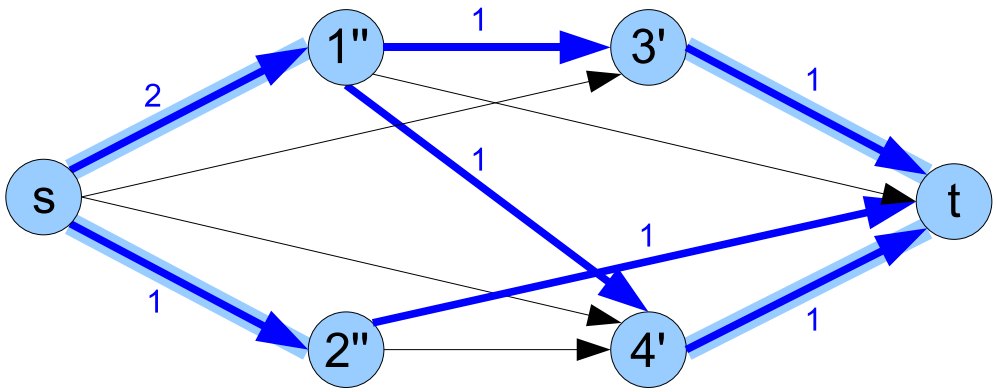

Figure 4.9: A minimum flow for the 4-job example problem

The maximum flow algorithm we use in our computational testing is an efficient implementation of the *push-relabel* method. This runs in $O(\eta\alpha \log(\eta^2/\alpha))$ time on an η-node, α-edge graph [56, 25]. The C source code for the algorithm is from hi_pr version 3.6 by IG Systems [120], a more robust version of the h_prf code implemented in [25].

An upper bound on the optimal value

The most immediate upper bound on the minimum number of machines needed is n. This is the number of machines used when a distinct machine is assigned to each individual job. An improved bound may be found by creating the network to represent a solution M with $m_i = 1$ for $i = 1, \ldots, n$. The minimum flow, $Q(M)$, in this network provides a better upper bound on the optimal value, as individual machines can process a number of jobs in sequence.

A lower bound on the optimal value

At least one machine is needed if all the jobs are to be processed. It is simple to check whether all jobs may be completed by a single machine:

$$\min_{M\in\mathbb{M}}\{Q(M)\} = 1 \iff a_{i,\,i+1} = 1,\ \forall\ i \in \{1,\ldots,n-1\}.$$

If any $a_{i,\,i+1} \neq 1$, then it is not possible to complete all the jobs using the same machine. This gives us an initial lower bound of two machines for such a problem; a very weak bound in most cases.

We suggest that by relaxing the problem we may determine a slightly better lower bound. The set of jobs j that may immediately follow job i on a machine is determined by the threshold values, a_{ij}, and machine allocation, m_i, to job i; we propose to expand this set by introducing relaxed threshold values, a^r_{ij}. Let $u \leq n$ be an upper bound on $\min_{M\in\mathbb{M}}\{Q(M)\}$. We let

$$a^r_{ij} = \begin{cases} 1 & \text{if } a_{ij} \leq u \\ a_{ij} & \text{otherwise} \end{cases} \qquad \text{for all } i,j \in N.$$

We assign each job the minimum allowed number of machines, i.e. $m_i = 1$, for all $i \in N$. This relaxed problem is then represented as a network G_l, constructed in the same way as the network G_M described above, but using the relaxed threshold values, a^r_{ij}, in place of a_{ij}. Calculating the minimum flow in G_l provides a lower bound on $\min_{M\in\mathbb{M}}\{Q(M)\}$. Better values for the upper bound, u, may tighten the relaxation.

Figure 4.10 shows a network G_l for our example instance, where $u = 3$; the key difference between this network and a network G_M for $M = \{1,\ldots,1\}$ is the inclusion of previously restricted inter-job arcs. Inter-job arcs are represented in green.

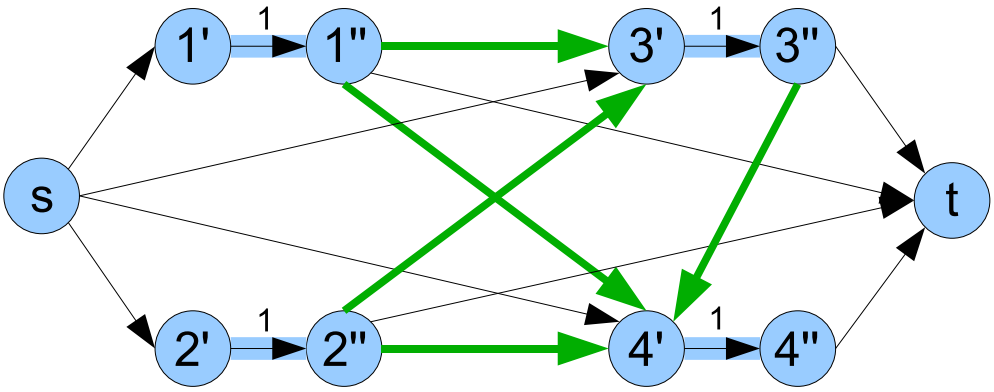

Figure 4.10: A network to find a lower bound for the 4-job example problem

Determining a schedule for individual machines

A solution to the supply ship scheduling problem is a set of values, M, where each element represents the number of machines allocated to an associated job. We can take this solution a step further by identifying individual machines and determining the set of jobs to be completed by each machine. A schedule for each machine is given by the decomposition of the minimum flow into unit-flow directed paths from the source to the sink. The decomposition may be accomplished by applying a decreasing path algorithm [29], iteratively reducing flow along a path from the source to the sink by one unit and recording the nodes the path traverses. Each unit path represents an assignment of jobs to a machine.

4.6 Local search heuristics for the supply ship scheduling problem

4.6.1 A selective neighbourhood structure

The network G_M is dependent on the values in $M = \{m_1, \ldots, m_n\}$. We wish to find the solution, M^*, that creates a G_M with the smallest minimum flow. Recall that we may restrict the possible values for m_i to belong to the set A_i, for all $i \in N$ (Section 4.5.1). Notice that the values in M have two direct effects on G_M:

1. The intra-job arcs of G_M have lower bounds and capacities equal to the values in M.
2. The values in M determine the inter-job arcs leaving each job-node for subsequent job-nodes. A greater value for m_i indicates a larger number of arcs leaving job-node i.

We will use *maximum cuts* in G_M and the *min-flow max-cut theorem* to inform changes to the values in M and influence the minimum flow. Let $[S, \bar{S}]$ be an s - t cut, so $s \in S$. Recall that the capacity of a cut $[S, \bar{S}]$ is given by

$$c[S, \bar{S}] = l(S, \bar{S}) - c(\bar{S}, S),$$

the sum of the lower bounds on the forward arcs minus the sum of the capacities of the backward arcs.

Each job i may be represented by two nodes in the network: the entry node i' and the exit node i''. There is a single arc (i', i'') connecting node i' and i'' that we refer to as the intra-job arc for job i. Since these intra-job arcs have associated lower bounds and capacites, i.e. $l(i', i'') = c(i', i'') = m_i$ for all $i \in N$, they will play an important part in any maximum cut.

Inter-job arcs (i'', j'), where j' is the entry node for job $j \in N$, $i < j$, represent the ability of machines to process successive jobs; they have a lower bound of zero and are considered to have unlimited capacity, i.e. $l(i'', j') = 0$, $c(i'', j') = \infty$ for all $i, j \in N$, $i < j$.

Since the minimum flow value is equal to the maximum cut value by the min-flow max-cut theorem, if we can reduce the value of the maximum cut we will reduce the value of the minimum flow. If we know the arcs belonging to a maximum cut for G_M we may attempt to reduce the value of the maximum cut by manipulating M and generating a new G_M. Two methods to potentially reduce the value of the maximum cut are:

1. **Decrease the lower bounds on forward arcs in the maximum cut**. Let i be a job whose intra-job arc (i', i'') appears in a maximum cut. (i', i'') has an associated lower bound equal to $m_i \in A_i$: decreasing m_i must decrease the value of the cut. (Decreasing m_i will also reduce the number of arcs in the network as at least one arc emanating from i'' will disappear). If there are additional maximum cuts that do not include (i', i''), the minimum flow will not be smaller in the new network. (It would be useful to know all maximum cuts in G_M, to inform the choice of i).

2. **Introduce a backward arc in the cut which has a large capacity**. Let i be a job whose corresponding exit node i'' lies on the sink side of a maximum cut, i.e. $i'' \in \bar{S}$. By increasing $m_i \in A_i$, we increase the number of arcs emanating from i''. If any of these new arcs terminate at a node $j' \in S$, the capacity of this new backward arc will be subtracted from the capacity of the cut. The new arc has unlimited capacity, so the capacity of the cut becomes $-\infty$. This eliminates this particular cut from being a maximum cut in the new network.

Given a solution $M = \{m_1, \ldots, m_n\}$, we may generate a neighbour of M by increasing or decreasing the value of $m_i \in A_i$ for a particular $i \in N$. To improve our chances of finding a better solution, our selection for i and its modification are restricted to jobs and modifications highlighted by the maximum cut information. Let $[S, \bar{S}]$ be a maximum cut in the network, G_M. The selective neighbourhood structure we propose features the following two components:

1. Let
$$R(M) = \{r|\ r \in N, (r', r'') \in [S, \bar{S}], m_r > 1\}.$$
$R(M)$ is the set of jobs with intra-job arcs that appear in the maximum cut and have a machine allocation that may be reduced. For each job r in $R(M)$ there is a neighbourhood move in which m_r is decreased to the preceding value in the ordered set A_r.

2. Let
$$W(M) = \{w|\ w \in N, w'' \in \bar{S}, \exists\ j' \in S \text{ with } m_w < a_{wj} \leq n\}.$$
$W(M)$ is the set of jobs that introduce a backward arc into the cut when their machine allocation is increased. Each job w in $W(M)$ corresponds to a neighbourhood move in which m_w is increased by the minimum amount necessary to allow a backward arc in the cut $[S, \bar{S}]$ to emanate from w''.

The neighbourhood contains at most $2n$ solutions, but the selective structure typically provides many fewer than this.

A good starting solution could be $m_i = 1$ for all $i \in N$. This solution provides us with a simple upper bound on the optimal value. It may also be close to optimal for instances with widely spread fixed start times or short processing times, since these may feature a large number of 'critical' threshold values that equal one.

4.6.2 Descent methods

We now describe how we use the selective neighbourhood structure within descent heuristics to improve our solutions.

First improvement

1. Starting solution M: $m_i = 1$ for all $i \in N$.
2. Determine $R(M)$ and $W(M)$.
3. By applying a modification given by $R(M)$ or $W(M)$ find a neighbour M' with $Q(M') < Q(M)$. When M' is found let $M = M'$ and go to step 2. If no such neighbour is found go to step 4.
4. Stop. (M is a local minimum).

The order in which neighbours are generated from $R(M)$ and $W(M)$ may be important. A sensible method would be to first select neighbours that we believe could lead to the greatest improvement in $Q(M)$. We shall use the following simple heuristic rules for the order in which we generate neighbours:

- *Work backwards through* $R(M)$. A reduction of the allocation to a job will reduce the set of subsequent jobs available to those machines. Later jobs have fewer following jobs, so the effect of cutting off sequences is minimised.
- *Work forwards through* $W(M)$. Machines newly assigned to earlier jobs may process a longer sequence of jobs and may allow other machines to process longer sequences.

In the first improvement descent method we shall call *FI-RW*, we generate neighbours from $R(M)$ before neighbours from $W(M)$. This gives priority to removing any over-assignment of machines.

In the alternative first improvement descent method *FI-WR*, we generate neighbours from $W(M)$ before neighbours from $R(M)$. We hope to increase the length of the paths travelled by machines by increasing the number of arcs in the network.

First improvement allowing neutral moves

How will performance of the algorithm change if we accept a move to the first neighbour M' with $Q(M') \leq Q(M)$? Allowing neutral moves may result in cycling, but opens up the possibility of exploring more of the search space. Since we do not accept deteriorating moves, solutions in a cycle will all have the same value for Q. To cycle back to a previous solution, all moves equating to an increase in machine allocation to a job (i.e. those in W) must be countered by moves that decrease the allocation (i.e. those in R). To avoid cycling, we apply the following acceptance conditions:

Accept a move generated from $R(M)$ *if* $Q(M') < Q(M)$.

Accept a move generated from $W(M)$ *if* $Q(M') \leq Q(M)$.

In this way, a move from W can only be undone by a move from R when it improves the value of Q.

In addition to the prevention of cycling we hope these conditions will lead to better solutions from subsequent neighbours. When we decrease the machine allotment to job i we wish to see a decrease in $Q(M)$. Such an improvement is a direct result of the reduction of the machines to job i; we have improved on an over-assignment. If we reduce the machine assignment

and $Q(M)$ remains the same, it means that one or more arcs emanating from job i have been removed but we have seen no improvement in the minimum flow. In a possible subsequent neighbour where machine allocation to another job has increased, those arcs could have allowed the machines of job i to process longer sequences of jobs. It seems beneficial to maintain these arcs. On the other hand, if $Q(M)$ remains unchanged for a decrease in machine allotment for a job in R, it indicates there is another maximum cut in the network of the same value. At least one other job must have its allotment decreased before we will see an improvement in the minimum flow $Q(M)$. The drawback of applying the anti-cycling condition is that the discovery of such possibilities is prevented.

We will test two variants of this approach: *FIN-RW* and *FIN-WR*.

Best improvement

In the best improvement (or steepest descent) algorithm, we look for the best improving move among all neighbours of the current solution (neutral and deteriorating moves are not accepted).

1. Starting solution M: $m_i = 1$ for all $i \in N$.
2. Determine $R(M)$ and $W(M)$.
3. By applying the modifications given by $R(M)$ and $W(M)$ generate all neighbours of M.
4. - If there is a neighbour M'' with $Q(M'') < Q(M)$, select the neighbour M' with the minimum value for $Q(M')$.
   - Otherwise stop. (M is a local minimum).
5. Let $M = M'$. Go to step 2.

Since the entire neighbourhood of a solution is searched, the order in which the neighbours are generated may seem less important; however, there may be multiple neighbours with the same value for $Q(M')$ that could provide the best improving move. Of these equally good moves, it will be the first to be generated that is selected. We shall call these best improvement methods *BI-RW* and *BI-WR*.

The modified methods, where we accept neutral moves when no improving move can be found, shall be called *BIN-RW* and *BIN-WR*. These neutral move methods use the anti-cycling acceptance rule employed in the first improvement methods described previously.

4.6.3 Tabu search

Since the acceptance rule applied within *FIN-RW* and *FIN-WR* may prevent good solutions being discovered, it may be worth implementing a *tabu list* to tackle the threat of cycling instead. This technique has the added bonus of driving the search into new areas. We broaden the acceptance of a move to be:

Move to the best neighbour that is not restricted by the tabu list.

A simple scheme for implementation of a tabu list is to store the recent modifications made when moving between solutions and forbid moves that undo these modifications. By the use of this form of tabu list we still risk cutting off access to good solutions. In an attempt to counteract this we use a simple *aspiration criterion*:

Accept a move to a neighbour with objective value better than that of the current best solution, even if the move is restricted by the tabu list.

To prevent the search from continuing indefinitely, we apply the following

termination criterion:

Stop the search after a number of iterations without an improvement in the best objective function value.

Let T be the tabu list, $|T|$ be the current length of the tabu list and L be the maximum length of the tabu list. Let Q_b be the best objective value found so far by the search, h be the current number of iterations since Q_b improved and I be the maximum number of such iterations.

Tabu search algorithm

1. Starting solution M: $m_i = 1$ for all $i \in N$.
   $Q_b = Q(M)$. $T = \emptyset$. $|T| = 0$. $h = 0$.
2. Determine $R(M)$ and $W(M)$.
3. By applying the modifications given by $R(M)$ and $W(M)$ generate all neighbours of M.
4. - If $Q(M'') < Q_b$ for any neighbour M'': select the neighbour M' with the minimum value for $Q(M')$; let $Q_b = Q(M')$ and store M' as the best solution found so far; set $h = 0$.
   - Otherwise: set $h = h + 1$ and select the neighbour M' with the minimum value for $Q(M')$ from among the neighbours not generated by a tabu modification within T. If the only neighbours are on the tabu list or there are no neighbours, stop.
5. Record in T the modification made to M to obtain M'. If $|T| > L$ remove the oldest modification from T.
6. Let $M = M'$.
7. If $h < I$ go to step 2. Otherwise stop and return the best solution found.

4.7 Restricted dynamic programming based approaches

In our scheduling problem the processing time of a job depends on the number of machines assigned to it; this affects the availability of those machines for subsequent jobs. Approaches inspired by *dynamic programming* (DP) may be useful in deciding how many machines should be allocated.

Let x be the total number of machines available to process the jobs $N = \{1, \ldots, n\}$. For a fixed value of x we must determine whether there is a feasible assignment, $M = \{m_1, \ldots, m_n\}$, of machines to jobs. If there is no feasible M for the given x, we know that the minimum number of machines required to process the jobs of N is at least $x + 1$; if there is a feasible assignment, the minimum required number of machines is at most x.

In the DP based algorithm we present for this problem, each successive *stage* corresponds to a job in $N = \{1, \ldots, n\}$. At stage i, the algorithm determines feasible allocations to job i by assigning available machines from the depot and jobs $1, \ldots, i-1$. A *state* (d, C_i, M_i) in stage i contains the following information:

- d, the number of machines currently left unused at the depot.
- $C_i = \{c_1, \ldots, c_i\}$, the number of machines that have completed their job and are available from jobs $1, \ldots, i$.
- $M_i = \{m_1, \ldots, m_i\}$, the number of machines assigned to jobs $1, \ldots, i$.

In the initial stage there is only one state: $(x, \emptyset, \emptyset)$, all x machines are at the depot and no jobs have yet been allocated any machines. Let job 0 refer to the depot, while stage 0 is this initial stage. States in stage i are generated from each state in stage $i - 1$. All possibilities for 'movement' of numbers of

machines from jobs in $\{0, 1, \ldots, i-1\}$ to job i are computed to create new states.

Example: State $(2, \{1\}, \{1\})$ in stage 1 generates the following states in stage 2 (assuming $a_{12} = 1$):

- $(2, \{0, 1\}, \{1, 1\})$: 1 machine moves from job 1 to job 2.
- $(1, \{1, 1\}, \{1, 1\})$: 1 machine moves from the depot to job 2.
- $(1, \{0, 2\}, \{1, 2\})$: 1 machine moves from job 1 to job 2, and another comes from the depot.
- $(0, \{1, 2\}, \{1, 2\})$: 2 machines move from the depot to job 2.
- $(0, \{0, 3\}, \{1, 3\})$: 1 machine moves from job 1 to job 2, and 2 more come from the depot.

For any state in stage i it holds that $c_i = m_i$. Machine allocation values for job i are restricted to be members of A_i (see Section 4.5.1). In valid states, at least one machine must be assigned to job i. If a state generated for stage i cannot lead to states in stage $i+1$ it is discarded from stage i: this is checked by calculating whether any machines will be able to move to job $i+1$.

Stage n is the final stage. If there is at least one state in stage n then it is feasible to process the jobs of N using at most x machines. If we are not restricting the number of states retained at each stage, then stage n will contain states corresponding to an optimal solution: a feasible assignment of machines to jobs that requires the smallest number of machines. Let $Q_{min} = \min_{M \in \mathbb{M}} Q(M)$, the optimal value for the scheduling problem. Optimal states of stage n have the greatest value for d amongst all states of stage n.

Let d_{max} be this maximum value of d. We have $Q_{min} = x - d_{max}$, the total machines available minus the number of machines left unused at the depot.

Since the DP algorithm results in an exponential explosion in the number of states as the problem size n increases, we attempt to find good solutions using a *restricted* DP heuristic (RDP) [92]. The RDP heuristic keeps the H best states at each stage and discards the rest. This means that the number of states in stage i, from which the states of stage $i + 1$ are generated, is limited. Fewer generating states will often result in fewer possibilities for states in the next stage and a large decrease in computational resources. The heuristic must judge which states are most likely to lead to feasible states at stage n. It should also endeavour to maximise the number of machines remaining at the depot, thus minimising the number of machines used. We shall refer to the restricted dynamic programming algorithm as $RDP(x, H)$, where up to H states at each stage are retained with the objective of finding feasible solutions when there are x machines available.

4.7.1 Restricting and sorting the list of retained states

To judge which to keep and which to reject we associate a statistic with each state: we shall call this its *opportunity* value. The opportunity value should reflect a state's potential to lead to a feasible solution; this potential is suggested by its ability to lead to states in all subsequent stages. Since states in subsequent stages are generated by 'moving' machines from completed jobs to jobs awaiting machines, the number of opportunities for such movements from completed jobs will impact the number of states that can arise.

The formula we use results from the idea that each machine provides one opportunity 'point' for each unallocated job accessible from its current job. A limit, b_k, is placed on the maximum number of machines that may move

from any individual job to job k, where b_k is the largest element of A_k, the greatest allocation of machines allowed for job k. For a state (d, C_i, M_i) in stage i, we use

$$\text{opp}(d, C_i, M_i) = \sum_{k=i+1}^{n} \min\{d, b_k\} + \sum_{j=1}^{i} \sum_{k=i+1}^{n} \min\{g_{jk}, b_k\}$$

where

$$g_{jk} = \begin{cases} c_j & \text{if } m_j \geq a_{jk} \\ 0 & \text{otherwise.} \end{cases}$$

As states are generated, they are sorted by their opportunity value. We wish to keep the H states with highest opportunity values, since these may lead to a greater number of feasible solutions. Among states with the same opportunity value, we sort by the value d. We prefer states with a larger value of d as these may lead to states at stage n with the greatest value of d. Our sorting process uses ordered 'buckets' corresponding to each opportunity value, and 'sub-buckets' for each value of d.

An alternative is to sort by d alone, ignoring opportunity. This places less emphasis on retaining states that could lead to feasible solutions and more emphasis on using as few machines as possible.

Sometimes a state from stage $i-1$ will generate a state for stage i that is identical to one already retained in the list. We discard the duplicate state. Even if two states of the same stage are not quite identical, they may be similar enough that we need keep only one of them. Let $K_j = \{k|\ k \in N, k > i, m_j \geq a_{jk}\}$ be the set of jobs after job i that are reachable by machines at job j, given m_j. If there are no machines at job j, i.e. $c_j = 0$, we set $K_j = \emptyset$. Assume we have two states (d', C'_i, M'_i) and (d'', C''_i, M''_i), with corresponding sets K'_j and K''_j for each $j \leq i$. If $d' = d''$, $C'_i = C''_i$ and K'_j=K''_j for all $j \leq i$, then the two states are *similar* and we may discard one of them. Note that when $c'_j \neq 0$ and $c''_j \neq 0$, we have K'_j=K''_j if $m'_j, m''_j \geq \max_{k>i}\{a_{jk}\}$.

4.7.2 Upper bounds on the optimal value

Recall that d_{max} is the value of d that is the greatest among all states of stage n. The machine assignments in M_n for states with $d = d_{max}$ reveal the best solutions provided by the RDP heuristic for a particular H and x.

If there is a stage $i \leq n$ for which no states can be generated, the $RDP(x, H)$ heuristic cannot find any feasible solutions for the problem of completing the set of jobs using at most x machines, given the restriction H. Applying the heuristic with a larger value of H may reveal feasible solutions. To be certain of infeasibility, or to find the optimal solution, H must be an unrestrictive value.

Let $y(x, H) = x - d_{max}$ be the number of machines actually used in the best feasible solution produced by $RDP(x, H)$. If $x \geq Q_{min}$ and $RDP(x, H)$ produces at least one feasible solution, then $x \geq y(x, H) \geq Q_{min}$.

$RDP(x, H)$ may still provide an upper bound on Q_{min} even if no feasible solutions are found. If stage i is the last stage at which states were found then we know that x machines are able to complete jobs $\{1, \ldots, i\}$. If $i > x$ we may improve on our simplest upper bound, n. Starting from a partial solution in stage i, we allocate an additional machine for each of the $n - i$ remaining jobs. This provides a simple feasible solution to the scheduling problem that uses $x + (n - i)$ machines.

Note that due to the way that states are selected to be retained we *cannot* guarantee that $y(x, H_1) \leq y(x, H_2)$ for $H_1 > H_2$, or that $y(x_1, H) \leq y(x_2, H)$ for $x_1 < x_2$. It is possible for a critical state to be accepted for some values of the parameters and rejected in favour of others when other parameter values are used. In general, we expect lower values for $y(x, H)$ as H increases, as fewer critical states are rejected. We might also expect $y(x_1, H) \leq y(x_2, H)$ when $x_1 < x_2$, since lower values for x result in a smaller

number of states generated from each state at every stage: with fewer states generated there is less competition between states to be retained.

4.7.3 Time complexity

The number of steps required by $RDP(x, H)$ in the worst case is $O(n^2H^2 2^x)$, where x is the number of machines, n is the number of jobs in N, H is the maximum number of states retained at each stage, and we assume $H \geq n$.

- The exponential term, 2^x, arises from the generation of states for a stage i: each state contains a total of x machines waiting at jobs $\{0, 1, \ldots, i-1\}$ and each machine either moves to job i or it does not.
- The sorting and checking of each state is $O(nH)$. (If $n > H$ and opportunity values are calculated, this becomes $O(n^2)$).
- Each stage contains at most H generating states.
- There are at most n stages.
- The algorithm requires $O(nH)$ space. Only two stages are retained at a time.

This shows that the running time of $RDP(x, H)$ is most dependent on the magnitude of x. The heuristic may be very inefficient for instances where x must be a large value.

4.7.4 Improving the upper bound

Let u represent the current best upper bound on Q_{min}, while l is the best lower bound. We may obtain initial values for u and l using the networks presented in Section 4.5.2. A better upper bound is provided by any feasible solution M with $Q(M) < u$. We may use $RDP(x, H)$ to discover such feasible solutions, improving the upper bound to $u = y(x, H)$. For a fixed value of the parameter H, two search methods we might use are:

1. **Bisection search:**
   (a) Find an initial lower bound, l, and an upper bound, u_0, on Q_{min}.
   (b) Let $u = u_0$ and $l_u = l$.
   (c) While $(l_u \neq u)$: { Apply $RDP(x, H)$ using $x = \left\lfloor \frac{l_u+u}{2} \right\rfloor$.
      - If $RDP(x, H)$ finds a feasible solution, let $u = y(x, H)$.
      - Otherwise, let $l_u = x + 1$ (the next lowest value that may produce feasible solutions). }
   (d) Return u.
2. **Decremental search:**
   (a) Find an initial upper bound, u_0, on Q_{min}.
   (b) Let $u = u_0$.
   (c) Loop: { Apply $RDP(x, H)$ using $x = u - 1$.
      - If $RDP(x, H)$ finds a feasible solution, let $u = y(x, H)$.
      - Otherwise, go to step (d). }
   (d) Return u.

In the bisection search algorithm, l_u indicates a lower bound on the best upper bound achievable by RDP(x, H), not a lower bound on Q_{min}.

Let x_H represent the smallest value of x for which RDP(x, H) provides a feasible solution. The relative efficiency of the two search methods depends on the number of calls to RDP(x, H), and the values of x that must be explored:

- Increases in the value of x lead to significantly larger running times for each RDP(x, H).
- RDP(x, H) requires fewer computations when no feasible solutions are found, as it terminates at the last feasible stage.
- If $x_H = u_0$, the decremental search needs only one iteration of RDP(x, H): a common occurrence if the initial bound is already near-optimal.
- The worst case occurs when $x_H = l$. The bisection search requires $O(\log(u_0 - l))$ iterations, while the decremental search requires $O(u_0 - l)$ iterations.

4.8 Computational experience

The algorithms were coded in the C programming language and compiled using Microsoft Visual Studio .NET 2003. The computer used to test the algorithms used a Pentium 4 processor (2.4 GHz with 504 MB of RAM).

By considering both the excess value of resulting solutions over the best solutions we have discovered, and the number of instances for which an approach produces the best objective value, we may compare the effectiveness of different approaches. If U is the objective value given by our heuristic method, and B is the best objective value we have found for the instance (by any method), then we let

$$\text{Excess} = 100 \times \frac{(U - B)}{B}.$$

4.8.1 Generating instances of the supply ship scheduling problem

Any n-job instance of the problem may be specified by its threshold values: data for such an instance consists of $\frac{1}{2}n(n-1)$ elements corresponding to a_{ij} for $i = 1, \ldots, n-1$; $j = i+1, \ldots, n$. Our scheduling problem requires that set-up times satisfy the triangle inequality, so these a_{ij} cannot be randomly generated directly. To ensure that all set-up times obey this restriction we generate instances by randomly locating n jobs on a 100×100 unit grid. Each instance is controlled by the random job locations and three parameters. *Set-up times* are calculated by dividing the distance between jobs by a parameter representing a machines speed: the greater the speed, the smaller the set-up times. An upper bound on the possible single-machine *processing time* for any job is given by a processing parameter. A lower bound on the single-machine processing time is set to

be 20% of the upper bound. Generation of *fixed start times* is controlled by a density parameter: the higher the density, the smaller the average time between consecutive jobs. The latest possible start time for any job is given by n divided by the density parameter. The random values for start times are sorted into ascending order before being assigned to the current job ordering. For each parameter we allow three possible values:

- Start time density: $\{0.1, 0.2, 0.3\}$.
- Processing time: $\{25, 50, 100\}$.
- Set-up speed: $\{5, 10, 20\}$.

Using every combination of these parameter values, we form a group of twenty-seven n-job instances. Each member shares the same set of job locations, as well as some subset of their parameters, with others in the group. Use of these groups ensures our instances cover a wide range of start time, processing time and set-up time combinations. The resulting threshold values can be significantly different between instances even for a seemingly small change in parameter.

For each $n \in \{10, 20, 30, 40, 50, 60, 70, 80, 90\}$ we generate five groups. This gives us 1215 instances on which to test our heuristics.

To simplify our analysis, and presentation of results, we divide the set of instances into subsets:

- *Small* represents $n \in \{10, 20, 30\}$ (405 instances).
- *Medium* represents $n \in \{40, 50, 60\}$ (405 instances).
- *Large* represents $n \in \{70, 80, 90\}$ (405 instances).

4.8.2 Simple upper and lower bounds

The upper bound derived from solving the minimum flow in a one-machine-per-job network (see Section 4.5.2) provides surprisingly good solutions for many of the instances considered (see Table 4.5). The best solution for several of our instances is this basic one-machine-per-job solution, since each machine may process a number of jobs. This simple upper bound is obtainable in almost zero time and should form a good starting solution for the local search methods.

	Small	**Medium**	**Large**
Excess (%)	16.61	16.83	17.59
Best (%)	23	15	10
CPU time	0.000	0.001	0.003

Table 4.5: Average excess (%) of the simple upper bound over the best solution; percentage of instances for which the simple upper bound gave the best solution; and run time (seconds)

The lower bound (as described in Section 4.5.2) is 46.55% below the best solution found on average across all instances. This is a fairly weak lower bound, but because it takes practically zero time to compute in most instances (maximum of 0.047 seconds for 100-task instances) it may be useful to narrow the search area for a good feasible solution. (A small modification to the bounding calculation would allow it to be used for sub-problems in a branch-and-bound scheme. Fixing numbers of machines for a set of jobs, thus eliminating some arcs between jobs, should improve the effectiveness of the bound, but this has not been tested).

4.8.3 Local search

Descent

Results for the eight varieties of descent method tested are shown in Tables 4.6, 4.7 and 4.8. Each method uses the selective neighbourhood prescribed by a maximum cut in the network corresponding to the current solution. The neighbourhood is formed from two parts: R (containing decreases in machine allocation) and W (containing increases in machine allocation).

First improvement:

The results for FI-RW and FI-WR reveal that there was no difference between the methods in practice: both lead to equivalent local minimums in every instance without a significant difference in the CPU times required. Indeed, the first few accepted moves for each method should be identical from a starting solution of $m_i = 1$ for all $i \in N$. Moves that reduce machine assignment, i.e. those in R, only apply to jobs i with $m_i > 1$. The two neighbourhood order variations are equivalent if R is empty.

First improvement with neutral move acceptance:

Acceptance of neutral moves provides much better solutions in practice. There is some difference between FIN-RW and FIN-WR however. Generating neighbours from R before W results in a slightly longer average running time and allows the approach to settle in a better local minimum for some instances. FIN-RW produces good solutions in very little time.

Best improvement (steepest descent):

Checking the entire neighbourhood before selecting a move results in a doubling of the CPU time over the FI methods, but no improvement in solution quality. The order in which neighbourhood fragments R and W

were checked (and thus the selection of a move from among equals) did not affect the results. BI-RW and BI-WR behaved almost identically.

Best improvement with neutral move acceptance:

BIN-RW and BIN-WR provide the best solutions among the descent variants we have tested. There is not a significant difference between the performance of BIN-RW and BIN-WR. Average CPU time remains very small, but is the greatest among the descent methods.

It seems well worth applying BIN to our problem to achieve good solutions in a fraction of a second.

Tabu search

Results for five tabu search approaches using different parameters are given in Tables 4.9, 4.10 and 4.11. The tabu search algorithm was applied with list lengths from 1 to 4 (small values seem appropriate since the neighbourhoods are relatively small). The search was set to continue for 100 iterations after the last improvement to the current best solution. In addition, since the list length of 3 seemed to produce the best results, an experiment to extend the termination criterion to 1000 iterations was also included.

The solutions obtained by the tabu search approaches appear excellent, achieving the best solution we have found for about 90% of the instances. Average CPU times are also very short; this may be due to many instances terminating before the 100 iteration (post improvement) limit is reached. If the neighbourhood of a solution is very small (perhaps indicating it is a strong solution) and the tabu list forbids those moves that are available, the algorithm terminates. Average CPU times for the 1000 iteration limit are reasonable and even larger iteration limits (or a fixed time limit) may produce better results for some instances.

Method	Small	Medium	Large
FI-RW	6.35	9.27	9.36
FI-WR	6.35	9.27	9.36
FIN-RW	2.39	2.77	2.40
FIN-WR	2.45	2.89	2.63
BI-RW	6.35	9.27	9.36
BI-WR	6.35	9.27	9.36
BIN-RW	1.57	2.24	1.83
BIN-WR	1.58	2.24	1.82

Table 4.6: Average excess (%) for the descent methods.

Method	Small	Medium	Large
FI-RW	52	29	26
FI-WR	52	29	26
FIN-RW	79	66	70
FIN-WR	78	64	67
BI-RW	52	29	26
BI-WR	52	29	26
BIN-RW	85	72	77
BIN-WR	85	72	77

Table 4.7: Percentage of instances for which the descent methods found the best solution.

Method	Small	Medium	Large
FI-RW	0.001	0.011	0.042
FI-WR	0.001	0.012	0.041
FIN-RW	0.003	0.060	0.271
FIN-WR	0.001	0.023	0.089
BI-RW	0.002	0.022	0.085
BI-WR	0.002	0.022	0.086
BIN-RW	0.008	0.163	0.770
BIN-WR	0.008	0.162	0.768

Table 4.8: Average CPU time (seconds) for the descent methods.

Method	Small	Medium	Large
TS L1 It100	0.81	1.16	1.18
TS L2 It100	0.69	1.03	1.01
TS L3 It100	0.71	0.87	0.89
TS L4 It100	0.78	0.89	0.96
TS L3 It1000	0.71	0.83	0.81

Table 4.9: Average excess (%) for the tabu search methods.

Method	Small	Medium	Large
TS L1 It100	92	85	85
TS L2 It100	93	87	86
TS L3 It100	94	90	88
TS L4 It100	93	90	88
TS L3 It1000	94	90	90

Table 4.10: Percentage of instances for which the tabu search found the best solution.

Method	Small	Medium	Large
TS L1 It100	0.059	0.806	10.031
TS L2 It100	0.051	0.748	9.514
TS L3 It100	0.045	0.709	8.676
TS L4 It100	0.038	0.682	8.039
TS L3 It1000	0.356	5.203	24.570

Table 4.11: Average CPU time (seconds) for the tabu search methods.

4.8.4 Restricted dynamic programming heuristics

Results for four approaches using restricted dynamic programming (RDP) are given in Tables 4.12, 4.13 and 4.14. Four different values for the parameter H, the maximum number of states retained at each stage, are displayed for each approach. Two search methods are used: bisection search and decremental search. The bisection search uses the simple upper and lower bounds given in Section 4.8.2 to narrow the search, while the decremental search begins at the simple upper bound. Results for two different sorting criteria are provided. OD represents utilisation of an approach that sorts states first by opportunity value, and then (among states with equal opportunity) by the depot number, d. Sorting of states by d alone is represented by D.

In general, as H increases the solutions obtained improve and CPU times increase. The performances of the RDP approaches are much better for small instances than for medium or large instances; the best solution is found for 66% of the small instances in very little time.

As problem size and H increase, the difference in average CPU time between RDP methods using different sorting criteria becomes more marked. The two OD methods, which employ both the opportunity value and depot number to sort and retain states, require significantly more time

than the two D methods, that sort only by the depot number. Three factors contributing to this large difference are:

1. Computation of each opportunity value is $O(n^2)$. In the OD methods, an opportunity value must be determined for every state that is generated .
2. States with a high opportunity value tend to generate more states (and thus even more computation) in subsequent stages than those with lower opportunity.
3. At each stage, the D algorithm implementations use the worst retained value of d to break early from the generation loop, cancelling the generation of worse states. Many fewer states must be considered.

The use of the opportunity statistic provides better solutions for the small instances when restricted to the smallest values of H. Overall however, opportunity values are not useful: by selecting states to retain using the depot number alone, we can provide better solutions in a shorter time, increasing H if necessary.

We now compare the performance of the bisection search procedure, D-Bisection, versus the decremental search, D-Decrement. The decremental search is slightly faster on average, but the bisection search provides a better solution in a few cases. We might expect the bound improvement from each of these methods to be identical, but results from RDP(x, H) can occasionally appear peculiar; for example, for a particular 20-job instance, $RDP(x = 15, H = 20)$ provided a valid upper bound of 14, with an associated solution; but $RDP(x = 14, H = 20)$ did *not* find a feasible solution. In this rare situation, $RDP(x = 14, H = 20)$ must have discarded some critical state that would have lead to the feasible solution.

Since the most promising approach of those presented is D-Bisection we

present further results for increasing H in Table 4.15. We see that the heuristic performs very well for small instances, but cannot match the performance of the tabu search heuristics. Large CPU time requirements for some medium and large instances mean that the average CPU time of the RDP method cannot compete with the very quick FIN and BIN descent methods or the tabu search method.

The results given are for search procedures using a fixed value of H. A scheme that increases H as the search progresses may improve CPU times.

Small	$H = 5$	$H = 10$	$H = 20$	$H = 40$
OD-Bisection	7.19	6.34	5.49	4.71
OD-Decrement	7.26	6.40	5.56	4.75
D-Bisection	10.48	8.83	7.01	5.10
D-Decrement	10.48	8.83	7.01	5.10
Medium	$H = 5$	$H = 10$	$H = 20$	$H = 40$
OD-Bisection	12.82	12.08	11.57	10.34
OD-Decrement	12.86	12.17	11.57	10.37
D-Bisection	13.89	12.79	11.00	10.11
D-Decrement	13.89	12.79	11.00	10.11
Large	$H = 5$	$H = 10$	$H = 20$	$H = 40$
OD-Bisection	13.94	13.53	12.81	12.09
OD-Decrement	14.00	13.59	12.88	12.14
D-Bisection	14.29	13.11	11.89	10.37
D-Decrement	14.29	13.15	11.89	10.37

Table 4.12: Average excess (%) for the RDP approaches.

Small	$H = 5$	$H = 10$	$H = 20$	$H = 40$
OD-Bisection	53	57	62	67
OD-Decrement	53	57	61	67
D-Bisection	44	51	57	66
D-Decrement	44	51	57	66
Medium	$H = 5$	$H = 10$	$H = 20$	$H = 40$
OD-Bisection	21	24	26	30
OD-Decrement	21	24	26	30
D-Bisection	24	27	34	36
D-Decrement	24	27	34	36
Large	$H = 5$	$H = 10$	$H = 20$	$H = 40$
OD-Bisection	15	17	20	23
OD-Decrement	15	16	20	23
D-Bisection	18	21	26	32
D-Decrement	18	21	26	32

Table 4.13: % instances for which the RDP approach finds the best solution.

Small	$H = 5$	$H = 10$	$H = 20$	$H = 40$
OD-Bisection	0.013	0.022	0.040	0.074
OD-Decrement	0.015	0.028	0.058	0.118
D-Bisection	0.001	0.003	0.005	0.010
D-Decrement	0.001	0.001	0.004	0.008
Medium	$H = 5$	$H = 10$	$H = 20$	$H = 40$
OD-Bisection	1.740	3.375	6.541	13.037
OD-Decrement	1.898	3.928	7.841	15.655
D-Bisection	0.011	0.022	0.045	0.103
D-Decrement	0.005	0.015	0.035	0.084
Large	$H = 5$	$H = 10$	$H = 20$	$H = 40$
OD-Bisection	11.862	22.559	44.588	78.127
OD-Decrement	17.032	35.450	94.735	228.494
D-Bisection	0.053	0.079	0.137	0.385
D-Decrement	0.031	0.054	0.107	0.339

Table 4.14: Average CPU times (seconds) for the RDP approaches.

Small	$H = 80$	$H = 160$	$H = 320$	$H = 640$	$H = 1280$
Excess (%)	3.88	3.19	2.43	1.69	1.13
Best (%)	71	74	78	83	88
CPU time	0.022	0.054	0.136	0.352	0.981
Medium	$H = 80$	$H = 160$	$H = 320$	$H = 640$	$H = 1280$
Excess (%)	8.31	7.06	6.17	5.37	4.58
Best (%)	42	49	54	58	63
CPU time	0.294	0.847	2.700	8.814	37.288
Large	$H = 80$	$H = 160$	$H = 320$	$H = 640$	$H = 1280$
Excess (%)	9.15	8.17	6.85	5.75	4.87
Best (%)	37	42	48	54	59
CPU time	1.144	3.380	12.254	41.519	187.295

Table 4.15: Further results for the D-Bisection RDP approach.

4.9 Extensions to the work presented

We have concentrated on entirely deterministic methods for finding good solutions. Almost all of our approaches could be modified to accept random choices; for example, the best improvement descent methods could randomly select between equally good neighbours; the probability of retaining a state in the RDP could be based on a function of its elements, or entirely random. It would be interesting to discover the change in performance these ideas might allow.

We could apply our selective neighbourhood structure within alternative local search metaheuristics; for example, simulated annealing. With the development of an appropriate kick modification, an iterated descent might enhance the ability of our simplest procedures. We decided to use a one-machine-per-job solution as the starting point for all our heuristics because it often represented a fair solution. An alternative heuristic for

generating starting points might allow access to new areas of the search space and allow the use of multi-start descent algorithms: we could use random selection of machine assignments, or the feasible solutions provided by our RDP for small H.

The tabu list in our tabu search approach was created from forbidden solution modifications. This allowed us to use a short list but meant that there was a possibility of cutting off access to better solution possibilities. We could instead store all the recently explored solutions, allowing a more thorough investigation of the search space.

We have tested instances consisting of 10 to 90 jobs. Our descent methods provided good solutions within a fraction of a second in most cases. These procedures might also be successful for considerably larger problems.

4.10 An improved formulation for the restricted dynamic programming approach

As we saw in Section 4.7.3, the worst-case time performance of the RDP technique presented is very poor. We noted that each state may 'generate' up to 2^x states in the next stage, where the parameter x is the number of machines allowable in the problem. By the following reformulation we may reduce this significantly: each state generates at most n states in the next stage.

The previous formulation employed perhaps the simplest and most intuitive approach for movements of machines: each state provided the number of machines currently at each job and the number of machines allocated to

each job; thus we could determine the availability of machines for remaining jobs. New states were generated by moving a subset of those machines to the current job. The idea behind our improved approach is that a state simply provides the number of machines available for each remaining job, together with the number of machines that remain unused at the depot. When we allocate a number of machines to a job, it is important to know whether those machines come directly from the depot or are already 'in the system' of jobs. Remember that we wish to maximise the number of jobs remaining at the depot in order to minimise the total number of machines required to complete the jobs.

Let a state (d_i, X_i) at stage i contain the following information:

- d_i, the number of machines currently left unused at the depot.
- $X_i = \{x_{i,\,i+1}, x_{i,\,i+2}, \ldots, x_{i,\,n-1}, x_{in}\}$, where each x_{ij} is the number of machines available to process job j for this state in stage i.

Assuming there are x machines allowed in the problem, the initial stage contains only the state $(d_0 = x, X_0 = \{x_{0j} = x, \forall\ j = 1, \ldots, n\})$, i.e. all x machines are at the depot and are available to process any job.

New states for stage i are generated from those in stage $i-1$ by fixing m_i to be each member of A_i in turn (see Section 4.5.1) with the condition that $m_i \leq x_{i-1,\,i}$; we cannot assign more machines to job i than are available to it. The assignment of only m_i machines to job i may restrict those machines from processing some of the subsequent jobs. We determine these jobs by comparing m_i to the threshold values a_{ij}.

We calculate the values of the elements in X_i for the new state by modifying those of X_{i-1} for the generating state. We use

$$x_{ij} = x_{i-1,\,j} - m_i \delta_{ij} \quad \text{for } j = i+1, \ldots, n$$

where

$$\delta_{ij} = \begin{cases} 0 & \text{if } m_i \geq a_{ij} \\ 1 & \text{otherwise.} \end{cases}$$

We must also calculate d_i for the new state. Since we wish to maximise the number of machines remaining at the depot, we re-use machines whenever possible. Let us call a machine that has already completed some previous job a *used* machine. By subtracting the value of d_{i-1} from $x_{i-1,\,i}$ we can determine the availability of used machines for job i. If we assign more machines to job i than the available used machines, we must take the additional machines from the depot: if $m_i > x_{i-1,\,i} - d_{i-1}$, we have

$$d_i = d_{i-1} - (m_i - (x_{i-1,\,i} - d_{i-1})).$$

We may therefore use

$$d_i = \begin{cases} d_{i-1} & \text{if } m_i \leq x_{i-1,\,i} - d_{i-1} \\ x_{i-1,\,i} - m_i & \text{otherwise.} \end{cases}$$

Application of the above process for generation of states is likely to produce multiple instances of the same state; we maintain only one entry for the state in our list. We will also discover states that are *dominated* by other states. We say a state (d'_i, X'_i) in stage i is dominated by a non-identical state (d''_i, X''_i) if $d''_i \geq d'_i$ and $x''_{ij} \geq x'_{ij}$ for all $j = i+1, \ldots, n$. We may discard any state that is dominated.

A state is *infeasible* if any $x_{ij} = 0$, $j = i+1, \ldots, n$. A value of zero for x_{ij} means that there are no machines available to process job j; we may discard the state.

Formulating the states in this way allows us to find the minimum number of machines required to complete the jobs. To find a solution, M, corresponding to this value we must also associate a set $M_i = \{m_1, \ldots, m_i\}$ with each state in stage i. (These M_i need not be considered when

determining if two states are identical, or if one dominates the other; they merely provide an example partial solution). In this way, any state consists of $n + 1$ elements.

If we maintain a list of H states at stage i, then we only need consider $|A_i| \times H$ states in the next stage (where $|A_i| \leq n$). Some potential statistics to use for the ordering and retaining of our H states include:

1. d_i. Retain those states in which the most machines remain unused.
2. $x_{i,\,i+1}$. Retain those states that allow the greatest number of machines to process the next job (this leads to the largest number of states at the next stage).
3. $\sum_{j=i+1}^{n} x_{ij}$. Retain those states with the greatest sum of available machines across the remaining jobs.

An algorithm using this formulation would have a time complexity of $O(n^3H^2)$ and space complexity of $O(nH)$.

4.11 Conclusion

We have introduced and studied a combinatorial optimisation problem we have named the supply ship scheduling problem. By calculating statistics we called threshold values, we may represent any instance as a directed graph and reduce the field of possible solutions. Objective values for solutions to this problem may be calculated by constructing an appropriate network and deducing its minimum flow. This network was used to find upper and lower bounds on the optimal value for the problem.

We proposed a selective neighbourhood structure for local search procedures based on finding a maximum cut in the network representing the solution.

The neighbourhood was applied within several descent algorithms and a tabu search procedure.

We have also presented a heuristic inspired by restricted dynamic programming, suggesting statistics for use in selection of states to retain. Two simple search procedures were outlined. These methods were used to improve previous upper bounds and find good feasible solutions.

The local search heuristics seem to perform very well on the instances tested. Good solutions may be found in a very short amount of time. We highlight the BIN (best improvement with acceptance of neutral moves) and tabu search methods as the best approaches.

The best RDP approach we tested requires a significant amount of time in order to match the best solutions given by the local search. It is most effective for fairly small problems.

It remains an open problem whether the supply ship scheduling problem is NP-hard.

Chapter 5

Solving task allocation problems using semidefinite programming

5.1 Introduction

In a *task allocation problem* (TAP) a set of *tasks* must be assigned to a set of *processors* so that the overall cost is minimized. Costs arise from processors, task assignment and inter-processor communication. Processors may have a limited capacity which must be shared by any tasks assigned to them.

The TAP arises in distributed computing systems [114], an area that has become increasingly important with the development of micro-processor systems, multi-processor computers, the proliferation of networked computers and demand for solutions to complex modular programs. It is useful to know how components of such programs should be spread amongst the available processors to obtain the fastest results. We may consider costs to be the amount of time required by a processor to execute a particular

task, plus the time required for the communication of task data between processors.

In industrial applications, we want to know which processors and data links should be installed in a system from a number of different options. When costing a proposed system of micro-processors the aim is to minimise the installation cost of both the processors and inter-processor communication bandwidth, ensuring that all tasks are executable within a fixed time cycle. The prime example of such an application is within the car manufacturing industry [106], where the monitoring processes of chassis, suspension and fuel injection are performed by a sub-system of microcomputers.

In this chapter we apply *semidefinite programming* (SDP) relaxation techniques to three variants of the TAP known as the UTAP, CTAP and CMAP. The UTAP features uncapacitated processors. The CTAP does not include execution costs, while the CMAP excludes the costs of processors. We employ a partial higher lifting approach to improve the relaxations, proposing a number of heuristics for selection of indices for matrix variable extension. We also suggest heuristics for selection of branching variables and use a *branch-and-bound* search tree to find lower bounds and solutions for the CMAP.

Sections 5.2 and 5.3 give a brief introduction to semidefinite programming and positive semidefinite matrices. Standard linear programming formulations for variants of the task allocation problem are provided in Section 5.4. Section 5.5 is a short review of the key literature pertaining to the TAP. Our semidefinite programming relaxations, and heuristics to aid the effectiveness of partial higher lifting improvements, are presented in Section 5.6. In Section 5.7 we present our branch-and-bound algorithm and a number of strategies for selection of the branching variable. Sections 5.8, 5.9 and 5.10 contain our computational experience of the application of the

SDP relaxations to three variants of the TAP. We suggest an area for further study in Section 5.11, then conclude the chapter in Section 5.12.

5.2 Semidefinite programming

Semidefinite programming [124] is a mathematical programming technique that involves optimisation over matrices. If all the matrices are diagonal, a semidefinite program becomes a linear program.

SDP problems can be solved efficiently in practice by *interior-point algorithms* [4]. The approach has been successfully applied in combinatorial optimisation, including quadratic 0-1 programming problems.

The standard SDP problem [5]:

An $n \times n$ matrix variable, X, is used to formulate a problem as:

$$\begin{aligned} \max \quad & C \bullet X \\ \text{s.t.} \quad & A_k \bullet X = b_k, \quad k = 1, \ldots, m \\ & X \succeq 0 \end{aligned}$$

where C and A_k are $n \times n$ matrices of real numbers, each b_k is a real number and

$$C \bullet X = \sum_{i=1}^{n} \sum_{j=1}^{n} C_{i,j} X_{i,j}.$$

Each equation $A_k \bullet X = b_k$ represents a linear constraint on the elements of X.

$X \succeq 0$ denotes that X is symmetric and positive semidefinite.

5.3 Positive semidefinite matrices

An $n \times n$ symmetric matrix, $X \in \mathbb{R}^{n \times n}$, is said to be positive semidefinite if

$$y^T X y = \sum_{i=1}^{n} \sum_{j=1}^{n} X_{i,j} y_i y_j \geq 0 \quad \forall\, y \in \mathbb{R}^n.$$

Positive semidefinite matrices have a number of important properties:

- All *eigenvalues* of a positive semidefinite matrix are non-negative.
- All its diagonal entries are non-negative.
- The *trace* is non-negative, since this is the sum of the diagonal entries.
- The *determinant* is non-negative, since this is equal to the product of the eigenvalues.
- Let $S \subseteq \{1, 2, \ldots, n\}$. A principal submatrix, $X(S)$, is the matrix that results from the deletion of indexed rows and columns of X that are complementary to S. Any *principal submatrix* of a positive semidefinite matrix is positive semidefinite.
- The *principal minors* are non-negative. These are the determinants of the corresponding principal submatrices.

Two further properties, that we shall make use of in Section 5.7.1, are:

Fact 5.1 *If a diagonal entry of a positive semidefinite matrix is zero, any other entries in the same row or column must also be zero.*

$$\begin{pmatrix} a & c \\ c & b \end{pmatrix} \succeq 0 \text{ and } b = 0 \quad \Longrightarrow \quad c = 0$$

Proof:

$$\begin{vmatrix} a & c \\ c & b \end{vmatrix} \geq 0 \quad \Longrightarrow \quad ab - c^2 \geq 0 \quad \Longrightarrow \quad c^2 \leq 0 \quad \Longrightarrow \quad c = 0.$$

Fact 5.2 *If the first entry of a positive semidefinite matrix is one and the diagonal entry and the first entry in a column/row are one, then the column/row is identical to the first column/row of the matrix.*

$$\begin{pmatrix} 1 & 1 & a \\ 1 & 1 & b \\ a & b & a \end{pmatrix} \succeq 0 \quad \Longrightarrow \quad a = b$$

Proof:

$$\begin{vmatrix} 1 & 1 & a \\ 1 & 1 & b \\ a & b & a \end{vmatrix} = \begin{vmatrix} 0 & 0 & a-b \\ 1 & 1 & b \\ a & b & a \end{vmatrix} = (a-b) \begin{vmatrix} 1 & 1 \\ a & b \end{vmatrix} = -(a-b)^2 \geq 0 \Rightarrow a = b.$$

5.4 The task allocation problem

We have n tasks to be assigned to m processors. Each task must be assigned to exactly one processor. Communication links between processors are identical. Let c_{ij} be the cost of communication between tasks i and j; we assume that $c_{ij} = c_{ji}$ and $c_{ii} = 0$. The communication cost, c_{ij}, is incurred if and only if the tasks i and j are assigned to different processors. A task may require different amounts of running time if assigned to different processors. Let e_{ik} denote the execution cost of task i if it is assigned to processor k. If used, processor k incurs a fixed cost of f_k and has a total resource capacity of b_k. Let a_i denote the amount of resource required to execute task i.

To formulate the general TAP problem, we introduce two sets of 0-1 decision variables:

- $x_{ik} = 1$ if and only if task i is assigned to processor k;
- $y_k = 1$ if and only if processor k is assigned at least one task.

The problem formulation is as follows [47]:

$$\begin{array}{lll}
\min & \displaystyle\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} c_{ij}\left(1-\sum_{k=1}^{m} x_{ik}x_{jk}\right) + \sum_{i=1}^{n}\sum_{k=1}^{m} e_{ik}x_{ik} + \sum_{k=1}^{m} f_k y_k & \\
\text{s.t.} & \displaystyle\sum_{k=1}^{m} x_{ik} = 1 & i = 1,\ldots,n \\
& \displaystyle\sum_{i=1}^{n} a_i x_{ik} \le b_k y_k & k = 1,\ldots,m \\
& x_{ik} \le y_k & i = 1,\ldots,n; \quad k = 1,\ldots,m \\
& x_{ik} \in \{0,1\} & i = 1,\ldots,n; \quad k = 1,\ldots,m \\
& y_k \in \{0,1\} & k = 1,\ldots,m
\end{array}$$

The objective function is minimizing the total cost, which is the sum of the communication costs, the execution costs and the processor costs. The constraints ensure that each task is assigned to exactly one processor, that the total resource usage by the tasks assigned to a processor does not exceed its capacity, and that no task is assigned to a processor that is not used. Notice that if $a_i > 0$ for all i, then the last set of inequalities is redundant, since it can be deduced from the other constraints.

To simplify the formulation of the objective function we multiply out the brackets, remove the constant value of the sum of communication costs and let $d_{ij} = -c_{ij}$.

$$\begin{array}{lll}
& \min & \displaystyle\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} c_{ij}\left(1-\sum_{k=1}^{m} x_{ik}x_{jk}\right) + \sum_{i=1}^{n}\sum_{k=1}^{m} e_{ik}x_{ik} + \sum_{k=1}^{m} f_k y_k \\
= & \min & \displaystyle\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} c_{ij} - \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{m} c_{ij}x_{ik}x_{jk} + \sum_{i=1}^{n}\sum_{k=1}^{m} e_{ik}x_{ik} + \sum_{k=1}^{m} f_k y_k \\
\equiv & \min & \displaystyle\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\sum_{k=1}^{m} (-c_{ij})x_{ik}x_{jk} + \sum_{i=1}^{n}\sum_{k=1}^{m} e_{ik}x_{ik} + \sum_{k=1}^{m} f_k y_k \\
= & \min & \displaystyle\sum_{k=1}^{m}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} d_{ij}x_{ik}x_{jk} + \sum_{k=1}^{m}\sum_{i=1}^{n} e_{ik}x_{ik} + \sum_{k=1}^{m} f_k y_k
\end{array}$$

A number of special cases of the TAP appear in the literature.

5.4.1 The capacitated problem (CTAP)

In this version of the problem, costs are usually associated with the installation of communication links and processors. Execution costs are not a factor.

$$\begin{array}{lll}
\min & \displaystyle\sum_{k=1}^{m}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} d_{ij}x_{ik}x_{jk} \;+\; \sum_{k=1}^{m} f_k y_k & \\
\text{s.t.} & \displaystyle\sum_{k=1}^{m} x_{ik} = 1, & i = 1,\ldots,n \\
& \displaystyle\sum_{i=1}^{n} a_i x_{ik} \le b_k y_k & k = 1,\ldots,m \\
& x_{ik} \le y_k & i = 1,\ldots,n; \quad k = 1,\ldots,m \\
& x_{ik} \in \{0,1\} & i = 1,\ldots,n; \quad k = 1,\ldots,m \\
& y_k \in \{0,1\} & k = 1,\ldots,m
\end{array}$$

This case also arises if execution costs are a constant value. If $e_{ik} = e$ for all i, k, then

$$\sum_{i=1}^{n}\sum_{k=1}^{m} e_{ik}x_{ik} = e\sum_{i=1}^{n}\sum_{k=1}^{m} x_{ik} = e\sum_{i=1}^{n}\left(\sum_{k=1}^{m} x_{ik}\right) = e\sum_{i=1}^{n}(1) = en.$$

This is a constant and may be removed from the objective function.

5.4.2 The constrained module allocation problem (CMAP)

Costs usually represent time in this case. Fixed costs to use processors are not included, eliminating the need for y_k variables.

$$\begin{array}{lll}
\min & \displaystyle\sum_{k=1}^{m}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} d_{ij}x_{ik}x_{jk} \;+\; \sum_{k=1}^{m}\sum_{i=1}^{n} e_{ik}x_{ik} & \\
\text{s.t.} & \displaystyle\sum_{k=1}^{m} x_{ik} = 1, & i = 1,\ldots,n \\
& \displaystyle\sum_{i=1}^{n} a_i x_{ik} \le b_k, & k = 1,\ldots,m \\
& x_{ik} \in \{0,1\} & i = 1,\ldots,n; \quad k = 1,\ldots,m
\end{array}$$

5.4.3 The uncapacitated problem (UTAP)

Processors are assumed to have unlimited capacities. Fixed costs to use processors are not included. The y_k variables and capacity constraints no longer appear.

$$\begin{array}{lll} \min & \sum_{k=1}^{m}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} d_{ij}x_{ik}x_{jk} \;+\; \sum_{k=1}^{m}\sum_{i=1}^{n} e_{ik}x_{ik} & \\ \text{s.t.} & \sum_{k=1}^{m} x_{ik} = 1 & i = 1,\ldots,n \\ & x_{ik} \in \{0,1\} & i = 1,\ldots,n; \quad k = 1,\ldots,m \end{array}$$

5.5 Literature

Our interest in task allocation problems arose from the report by Ernst et al. [47] in which they present several integer linear programs and column generation formulations for the UTAP and CTAP. Their approaches to the UTAP proved very successful: two of their linear relaxations provided the optimal solution for many of their instances, while another formulation provided a good lower bound in a short amount of time. Their column generation approach was even more successful in most cases. Their experiments for the CTAP indicated that it was a much harder problem. Their branch-and-bound approaches for the CTAP were unable to converge quickly, even after a number of cutting techniques were applied in CPLEX.

The success of the linear programming approaches for the UTAP inspired us to apply semidefinite programming to these classes of problems. Subsequent to construction of our SDP formulations for the UTAP and CTAP we came across a report by Elloumi et al. [45] in which they compare seven different lower bounds for the constrained module allocation problem (CMAP): another special case of the TAP that seems to fall between the difficulty of the UTAP and CTAP. The bounds they consider come from three families

of optimisation techniques: linearisation, semidefinite programming and Lagrangian decomposition. Their SDP relaxations are created by applying a set of rules (from [108]) that convert any quadratic or linear program with bivalent variables into an SDP. (Their most basic SDP is similar to our basic SDP when applied to the CMAP). They show that two of their SDP relaxations are tighter than the linear and Lagrangian decomposition methods. They also note the structural difficulty of the problem which is "much more difficult to solve when there are no execution costs and easier to solve when these execution costs are more important" [45].

We now take a brief look at some of the other approaches that have been used for the UTAP, CMAP, CTAP and related problems: these include graph theory, heuristics and integer programming.

Stone [114] models the 2-processor TAP as a graph and uses a *maximum flow* algorithm to find the optimal solutions given by a minimum cut in the transformed network. The problem of finding an optimal assignment of tasks to 3 or more processors is known to be NP-hard [59] (except for specially restricted cases [36]); exact approaches are useful only for small instances, while heuristics may be used to find good solutions to larger problems.

A *branch-and-bound* technique is used by Ma et al. [89] to solve a TAP with applications in distributed computing for air defence. Utilisation of each processor is balanced while satisfying several engineering constraints.

The *graph matching* approach for task allocation with non-identical communication links is proposed by Shen and Tsai [113].

Lo [85] proposes a family of greedy heuristic algorithms to find good solutions for the UTAP based on the approach of Stone [114], extending the problem to include *interference costs*. These additional costs are incurred by assigning tasks to the same processor and are designed to aid in processor

load balancing.

Sarje and Sagar [109] propose a heuristic for the UTAP with load balancing. The technique forms task clusters by analysing the communication costs associated with each task, restricting the cluster size to the average load across all processors.

Kopidakis et al. [76] transform the UTAP into a maximisation problem in which they try to determine and avoid large communication costs. Applying a graph transformation, they present two fast heuristics to find good solutions: graph matching and greedy edge selection.

A branch-and-bound algorithm of Billionnet et al. [16] uses *Lagrangian relaxation* to solve (or find tight lower bounds for) the UTAP. Their approach is effective when a small percentage of communication costs are non-zero.

Lewis et al. [81] re-cast the UTAP as an *unconstrained quadratic binary program* which they then solve by a tabu search. They state their approach is competitive with other methods and outperforms CPLEX (a mathematical programming optimisation software package) for larger instances.

The CMAP was formulated as a quadratic program with 0-1 variables in [17]. Roupin shows that "unless $P = NP$, no polynomial-time algorithm can guarantee to find a feasible solution within c percent of the optimal value, where c is any fixed positive constant" [107].

Hamam and Hindi [64] apply *simulated annealing* to find good solutions to the CMAP, while Elsadek and Wells [46] use a greedy heuristic to cluster tasks (similar to Sarje and Sagar [109] for the UTAP), improving their solution using simulated annealing.

The *cross entropy* method is applied to the CMAP with alternative objectives by Widell and Nyberg [122]. They state that cross entropy efficiently generates high quality solutions for the CMAP. A cross entropy method uses "a distribution with parameter v to generate sample allocation. The generated samples are then used to update v according to sample quality. This process continues until the distribution converges to a possibly optimal solution" [122].

Hadj-Alouane et al. [63] propose a hybrid of Lagrangian relaxation and *genetic algorithms* to tackle the CTAP arising from a car assembly line [106]. The method requires a significant amount of time to obtain optimal solutions for larger problems.

Chen and Lin [24] present another hybrid search technique: *tabu search* and *noising method*. Their hybrid approach is shown to be much more effective at obtaining good solutions for the CTAP than the method in [63].

A general variable neighbourhood search algorithm is developed for the CTAP and TAP by Lusa and Potts [88]. This technique outperforms the hybrid algorithm of Chen and Lin [24] for 72% of the instances.

We have seen that there are several varieties of task allocation problem and the techniques for solving them are equally diverse. Mathematical programming approaches to find lower bounds have demonstrated encouraging results, but there is still scope for improvement in this area.

5.6 Structured SDP relaxations for variants of the TAP

We build SDP relaxations for the variants of the TAP in order of increasing difficulty. We attempt to take advantage of the TAP's structure to construct our matrix variable.

Note that 0-1 constraints may be reformulated as quadratic constraints: the only solutions to $x_{ik}^2 = x_{ik}, 0 \leq x_{ik} \leq 1$ are $x_{ik} \in \{0, 1\}$.

5.6.1 UTAP

Let us begin with the UTAP in the form:

$$\begin{array}{lll} \min & \sum_{k=1}^{m} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_{ik} x_{jk} \quad + \quad \sum_{k=1}^{m} \sum_{i=1}^{n} e_{ik} x_{ik} & \\ \text{s.t.} & \sum_{k=1}^{m} x_{ik} = 1 & i = 1, \ldots, n \\ & x_{ik}^2 = x_{ik} & i = 1, \ldots, n; \quad k = 1, \ldots, m \\ & 0 \leq x_{ik} \leq 1 & i = 1, \ldots, n; \quad k = 1, \ldots, m \end{array}$$

We first set up column vectors v_k containing a "1" as the first entry, then each of the variables $x_{ik}, i = 1 \ldots n$.

$$v_k = \begin{pmatrix} 1 \\ x_{1k} \\ x_{2k} \\ \vdots \\ x_{nk} \end{pmatrix}$$

A symmetric matrix $X^{(k)}$ is formed by multiplying the vector v_k and its transpose [5].

$$X^{(k)} = v_k v_k^T = \begin{pmatrix} 1 \\ x_{1k} \\ x_{2k} \\ \vdots \\ x_{nk} \end{pmatrix} \begin{pmatrix} 1 & x_{1k} & x_{2k} & \dots & x_{nk} \end{pmatrix}$$

$$= \begin{pmatrix} 1 & x_{1k} & x_{2k} & \cdots & x_{nk} \\ x_{1k} & x_{1k}^2 & x_{1k}x_{2k} & \cdots & x_{1k}x_{nk} \\ x_{2k} & x_{1k}x_{2k} & x_{2k}^2 & \cdots & x_{2k}x_{nk} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{nk} & x_{1k}x_{nk} & x_{2k}x_{nk} & \cdots & x_{nk}^2 \end{pmatrix}$$

Since all its rows are a multiple of one vector, the *rank* of $X^{(k)}$ is one. We can show that $X^{(k)}$ is positive semidefinite, since

$$y^T X^{(k)} y = y^T (v_k v_k^T) y = (y^T v_k)(v_k^T y) = (y^T v_k)^2 \geq 0, \qquad \forall\ y \in \mathbb{R}^n.$$

By assuming the symmetry of $X^{(k)}$, and implementing the quadratic constraint $x_{ik}^2 = x_{ik}$ that equates each diagonal entry to the first entry in its column/row, we may simplify our representation:

$$X^{(k)} = \begin{pmatrix} 1 & x_{1k} & x_{2k} & \cdots & x_{nk} \\ & x_{1k} & x_{1k}x_{2k} & \cdots & x_{1k}x_{nk} \\ & & x_{2k} & \cdots & x_{2k}x_{nk} \\ & & & \ddots & \vdots \\ & & & & x_{nk} \end{pmatrix}$$

We index the rows and columns of $X^{(k)}$ using $0, 1, 2, \dots, n$. Let $X_{ij}^{(k)}$ represent the entry in row i and column j of $X^{(k)}$.

We propose the use of a matrix variable X for the UTAP with the following *block diagonal* structure:

$$X = \begin{pmatrix} X^{(1)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & X^{(2)} & & \mathbf{0} & \mathbf{0} \\ \vdots & & \ddots & & \vdots \\ \mathbf{0} & \mathbf{0} & & X^{(m-1)} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & X^{(m)} \end{pmatrix} \tag{5.1}$$

A standard SDP approach for this problem would employ an $m(n+1) \times m(n+1)$ matrix variable with rows and columns indexed by all pairs (i,k) for $i = 1,\ldots,n; k = 1,\ldots,m$. The block diagonal structure is equivalent to using only m matrices of dimension $(n+1) \times (n+1)$: a significant reduction in the number of variables. Using the proposed matrix variable, $x_{ik}x_{jk} = X^{(k)}_{ij}$ and $x_{ik} = X^{(k)}_{0i} = X^{(k)}_{ii} = x^2_{ik}$. We can formulate the UTAP as follows:

$$\begin{array}{lll} \min & \sum_{k=1}^{m}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} d_{ij}X^{(k)}_{ij} \;+\; \sum_{k=1}^{m}\sum_{i=1}^{n} e_{ik}X^{(k)}_{ii} & \\ \text{s.t.} & \sum_{k=1}^{m} X^{(k)}_{ii} = 1 & i = 1,\ldots,n \\ & X^{(k)}_{00} = 1 & k = 1,\ldots,m \\ & X^{(k)}_{ii} = X^{(k)}_{0i} & i = 1,\ldots,n; \quad k = 1,\ldots,m \\ & X \succeq 0 & \\ & \operatorname{rank}(X^{(k)}) = 1 & k = 1,\ldots,m \end{array}$$

All diagonal entries of a positive semidefinite matrix must be non-negative, so $x_{ik} = X^{(k)}_{ii} \geq 0$; this, together with the constraint $\sum_{k=1}^{m} X^{(k)}_{ii} = 1$, ensures that $0 \leq x_{ik} \leq 1$.

The form of the block in (5.1) is enforced by the constraint on each block's rank, together with the symmetry of the positive semidefinite matrix. By removing the condition that the rank must be one, we obtain an SDP relaxation; the objective value provides a lower bound on the optimal value of the UTAP. The entries in the matrix will have the precise structure in (5.1) only if a 0-1 solution is found.

5.6.2 CMAP

The formulation is identical to that for the UTAP, but we now include the resource constraint:

$$\sum_{i=1}^{n} a_i x_{ik} \leq b_k, \qquad k = 1, \ldots, m.$$

In terms of our matrix variable, this constraint becomes

$$\sum_{i=1}^{n} a_i X_{ii}^{(k)} \leq b_k, \qquad k = 1, \ldots, m.$$

We add this constraint to our SDP formulation of the UTAP to obtain the CMAP.

5.6.3 CTAP

The CTAP reintroduces the processor usage variables, y_k. We must make modifications to our matrix variable $X^{(k)}$ to incorporate this aspect. In the UTAP formulation the entry $X_{00}^{(k)}$ is fixed to be '1'; we shall now set this entry of our new matrix block to be equal to y_k. Let

$$Y^{(k)} = \begin{pmatrix} y_k & x_{1k} & x_{2k} & \cdots & x_{nk} \\ & x_{1k} & x_{1k}x_{2k} & \cdots & x_{1k}x_{nk} \\ & & x_{2k} & \cdots & x_{2k}x_{nk} \\ & & & \ddots & \vdots \\ & & & & x_{nk} \end{pmatrix}$$

The corresponding matrix variable is

$$Y = \begin{pmatrix} Y^{(1)} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & Y^{(2)} & & \mathbf{0} & \mathbf{0} \\ \vdots & & \ddots & & \vdots \\ \mathbf{0} & \mathbf{0} & & Y^{(m-1)} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & Y^{(m)} \end{pmatrix} \tag{5.2}$$

We begin with the following CTAP formulation:

$$\begin{array}{lll}
\min & \sum_{k=1}^{m}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} d_{ij}x_{ik}x_{jk} + \sum_{k=1}^{m} f_k y_k & \\
\text{s.t.} & \sum_{k=1}^{m} x_{ik} = 1, & i = 1,\ldots,n \\
& \sum_{i=1}^{n} a_i x_{ik} \le b_k y_k & k = 1,\ldots,m \\
& x_{ik} \le y_k & i = 1,\ldots,n; \quad k = 1,\ldots,m \\
& x_{ik}^2 = x_{ik} & i = 1,\ldots,n; \quad k = 1,\ldots,m \\
& 0 \le x_{ik} \le 1 & i = 1,\ldots,n; \quad k = 1,\ldots,m \\
& y_k \in \{0,1\} & k = 1,\ldots,m
\end{array}$$

In a similar fashion to the UTAP and CMAP cases, our SDP formulation becomes

$$\begin{array}{lll}
\min & \sum_{k=1}^{m}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} d_{ij}Y_{ij}^{(k)} + \sum_{k=1}^{m} f_k Y_{00}^{(k)} & \\
\text{s.t.} & \sum_{k=1}^{m} Y_{ii}^{(k)} = 1, & i = 1,\ldots,n \\
& \sum_{i=1}^{n} a_i Y_{ii}^{(k)} \le b_k Y_{00}^{(k)} & k = 1,\ldots,m \\
& Y_{ii}^{(k)} = Y_{0i}^{(k)} & i = 1,\ldots,n; \quad k = 1,\ldots,m \\
& Y \succeq 0 & \\
& Y_{00}^{(k)} \in \{0,1\} & k = 1,\ldots,m \\
& \operatorname{rank}(Y^{(k)}) \in \{0,1\} & k = 1,\ldots,m
\end{array}$$

The $x_{ik} \le y_k$ constraint is enforced by Y being positive semidefinite. We may show this by looking at the principal submatrix of the positive semidefinite matrix $Y^{(k)}$ that includes rows/columns zero and i:

$$\begin{pmatrix} y_k & x_{ik} \\ x_{ik} & x_{ik} \end{pmatrix} \succeq 0$$

We know that that diagonal elements are non-negative: $y_k \ge 0; x_{ik} \ge 0$. We also know that the determinant of this positive semidefinite matrix is non-negative:

$$\begin{vmatrix} y_k & x_{ik} \\ x_{ik} & x_{ik} \end{vmatrix} = y_k x_{ik} - x_{ik}^2$$

Thus

$$y_k \geq 0, x_{ik} \geq 0 \text{ and } y_k x_{ik} - x_{ik}^2 \geq 0 \quad \Longrightarrow \quad x_{ik} \leq y_k.$$

The rank constraint has been modified to allow $Y^{(k)}$ to have either rank zero or rank one.

- When $y_k = Y_{00}^{(k)} = 1$, then $Y^{(k)} = X^{(k)}$, which we have seen has rank one.
- If $y_k = Y_{00}^{(k)} = 0$, then all the entries of $Y^{(k)}$ are zero: if a diagonal entry of a positive semidefinite matrix is zero then entries in the same row or column are also zero. By setting $y_k = 0$ we cause all entries in the first row to be zero. By the constraint $x_{ik}^2 = x_{ik}$ this causes all diagonal entries to be zero, which in turn means that all entries in all rows and columns are zero. The zero matrix has rank zero.

To obtain our SDP relaxation we remove the rank constraint and replace the 0-1 constraint on $Y_{00}^{(k)}$ with

$$Y_{00}^{(k)} \leq 1 \qquad k = 1, \ldots, m.$$

Due to Y being positive semidefinite we know that all diagonal entries are non-negative and so $Y_{00}^{(k)} \geq 0,\ k = 1, \ldots, m.$

5.6.4 TAP

We generalise our SDP formulation for the CTAP by including the total execution cost in the objective function.

$$\begin{array}{lll}
\min & \sum_{k=1}^{m}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} d_{ij}Y_{ij}^{(k)} \;+\; \sum_{k=1}^{m}\sum_{i=1}^{n} e_{ik}Y_{ii}^{(k)} \;+\; \sum_{k=1}^{m} f_k Y_{00}^{(k)} & \\
\text{s.t.} & \sum_{k=1}^{m} Y_{ii}^{(k)} = 1, & i = 1,\dots,n \\
& \sum_{i=1}^{n} a_i Y_{ii}^{(k)} \le b_k Y_{00}^{(k)} & k = 1,\dots,m \\
& Y_{ii}^{(k)} = Y_{0i}^{(k)} & i = 1,\dots,n; \quad k = 1,\dots,m \\
& Y \succeq 0 & \\
& Y_{00}^{(k)} \in \{0,1\} & k = 1,\dots,m \\
& \operatorname{rank}(Y^{(k)}) \in \{0,1\} & k = 1,\dots,m
\end{array}$$

Our SDP relaxation is then

$$\begin{array}{lll}
\min & \sum_{k=1}^{m}\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} d_{ij}Y_{ij}^{(k)} \;+\; \sum_{k=1}^{m}\sum_{i=1}^{n} e_{ik}Y_{ii}^{(k)} \;+\; \sum_{k=1}^{m} f_k Y_{00}^{(k)} & \\
\text{s.t.} & \sum_{k=1}^{m} Y_{ii}^{(k)} = 1, & i = 1,\dots,n \\
& \sum_{i=1}^{n} a_i Y_{ii}^{(k)} \le b_k Y_{00}^{(k)} & k = 1,\dots,m \\
& Y_{ii}^{(k)} = Y_{0i}^{(k)} & i = 1,\dots,n; \quad k = 1,\dots,m \\
& Y_{00}^{(k)} \le 1 & k = 1,\dots,m \\
& Y \succeq 0 &
\end{array}$$

The objective value of the SDP relaxation provides a lower bound on the optimal value for the TAP.

5.6.5 Strengthening the SDP relaxations

We can tighten the SDP relaxation and improve the quality of the bounds using a *partial higher lifting* approach [6]. Extra rows and columns are added to each block in the matrix variable; each additional row/column corresponds to some subset of the tasks. The newly available entries are included in constraints that involve the entries in the original matrix, tightening the relaxation. It is hoped that the improvements in the bound will provide a worthwhile trade-off for extra computational effort associated with solving the relaxation for a larger matrix variable.

We pick a subset, $S \subseteq \{1, \ldots, n\}$, of tasks in some way. We shall call S the *task set*. Using this set S, new rows/columns are added to a block of the matrix variable by selecting pairs of tasks: if $i \in S$ and $j \in S$, then an extra row/column with first entry $x_{ik}x_{jk}$ is added to block $Y^{(k)}$. The remaining entries are multiples of the first term with the variables $x_{1k}, \ldots, x_{nk}$, as described for the standard rows/columns.

If there are $|S|$ tasks selected for a block, then $\binom{|S|}{2}$ extra rows/columns (corresponding to each pair of tasks in S) will be added. We shall call the set containing all pairs of tasks from S the *extension set*, E: every member of E corresponds to a new row/column to augment $Y^{(k)}$. An alternative to selecting S is to choose the members of E directly, that is, choose *pairs* of tasks with which to form rows/columns.

Further constraints relating the entries in each block can be added to the augmented matrix. Using the quadratic constraint, $x_{ik}^2 = x_{ik}$, we have

$$x_{ik}x_{jk} = x_{ik}^2 x_{jk} = x_{ik}x_{jk}^2 = x_{ik}^2 x_{jk}^2.$$

For each row/column that is added to the block, four more constraints are included in the SDP relaxation to constrain the appropriate elements of the block to be equal. If the new row/column representing the combination of tasks i and j is placed in row/column position h ($> n$), these constraints are as follows:

$$\begin{aligned} Y_{ij}^{(k)} &= Y_{0h}^{(k)} \\ Y_{ij}^{(k)} &= Y_{ih}^{(k)} \\ Y_{ij}^{(k)} &= Y_{jh}^{(k)} \\ Y_{ij}^{(k)} &= Y_{hh}^{(k)} \end{aligned}$$

For example, if $S = \{1, 2\}$ is chosen to augment block k, then the structure becomes:

$$Y^{(k)} = \begin{pmatrix} y_k & x_{1k} & x_{2k} & x_{3k} & \cdots & x_{nk} & \boxed{x_{1k}x_{2k}} \\ & x_{1k} & \boxed{x_{1k}x_{2k}} & x_{1k}x_{3k} & \cdots & x_{1k}x_{nk} & \boxed{x_{1k}x_{2k}} \\ & & x_{2k} & x_{2k}x_{3k} & \cdots & x_{2k}x_{nk} & \boxed{x_{1k}x_{2k}} \\ & & & x_{3k} & \cdots & x_{3k}x_{nk} & x_{1k}x_{2k}x_{3k} \\ & & & & \ddots & \vdots & \vdots \\ & & & & & x_{nk} & x_{1k}x_{2k}x_{nk} \\ & & & & & & \boxed{x_{1k}x_{2k}} \end{pmatrix}$$

The additional constraints should tighten the relaxation and lead to a better lower bound. We shall refer to this approach as an *SDP relaxation with extension*.

By including even more rows/columns based on the combination of three or more tasks, we might tighten the relaxation even further. Initial testing indicated that this is likely to increase running times without providing significant improvements.

We now suggest a number of heuristics for the selection of tasks for the task set, S:

Close to 0.5

We wish to tighten constraints on variables furthest from having a 0-1 value. A sensible proposal is to select for S those tasks with variable values closest to 0.5. This method requires an initial run of the SDP relaxation without extension, so that the values of the variables can be analysed. We then choose the $|S|$ tasks with a variable closest to 0.5 from among the variables x_{ik}; $i = 1, \ldots, n$; $k = 1, \ldots, m$. The resulting extension set is applied to all blocks in the matrix variable.

'Improved' close to 0.5

A possible improvement to the previous method involves choosing a task set for each processor block separately. This could result in different task sets for each block in the matrix, with each extension being specific to its block.

Resource requirement

Selects tasks based on the size of their processor resource requirement a_i. It is not necessary to run an initial SDP relaxation before the extended relaxation. The same extension set is applied to all blocks in the matrix variable. Two possible approaches are to choose tasks i corresponding to either the largest or smallest values of a_i.

Further improvements may be gained by choosing pairs of tasks for the extension set, E, directly. We propose the following heuristics:

Communication cost

This heuristic chooses pairs of tasks to be entered as elements of the extension set based on the communication cost between those tasks. The same extension set is applied to all blocks in the matrix variable. With this method, a pair of tasks (i, j) is selected based on its associated coefficient in the objective function (we return to the original $-c_{ij}$ terms in place of the simplifying d_{ij} terms). Since this information is available as part of the problem instance, it is not necessary to run an initial SDP relaxation before the relaxation with extension. Two possible approaches are:

- *Maximum communication:* choose (i, j) with the largest values of c_{ij} first.
- *Minimum communication:* choose (i, j) with the smallest values of c_{ij} first.

Paired value difference

This method requires an initial run of the SDP relaxation without extension. We then choose pairs of tasks (i, j) based on the difference between the values of $x_{ik} \times x_{jk}$ and the $x_{ik}x_{jk}$ term; i.e. $\left|Y_{ii}^{(k)}Y_{jj}^{(k)} - Y_{ij}^{(k)}\right|$. The larger the difference between these two values, the further the solution matrix block is from having rank one, and the greater the benefit of tightening the constraint by adding the paired tasks to the extension set. There may be a different extension set for each block, k.

5.7 Branch-and-bound

We may a obtain an improvement in our bound by selecting a variable and *branching* on it. We can compute a new bound by the following approach:

1. Solve the SDP relaxation.
2. Select a variable x_{ik}.
3. Solve the SDP relaxation with $x_{ik} = 0$. Let us call the corresponding objective value $F(x_{ik} = 0)$.
4. Solve the SDP relaxation with $x_{ik} = 1$. Call the corresponding objective value $F(x_{ik} = 1)$.
5. A new lower bound on the optimal value is given by

$$\min\{F(x_{ik} = 0), F(x_{ik} = 1)\}.$$

We may explore deeper by branching from each of these sub-problems.

Note that due to the constraint $\sum_{k=1}^{m} x_{ik} = 1$, by fixing $x_{il} = 1$, we are also fixing $x_{ik} = 0$ for all processors $k \neq l$.

5.7.1 Reducing the dimension of the UTAP/CMAP SDP relaxation

Once a variable has been fixed by a branching process, the SDP relaxation for the resulting instance can be reduced in size by exploiting the structure of the matrix variable and its positive semidefinite properties. This has a positive impact both on the dimension of the matrix variable and the number of constraints in the SDP relaxation. Consider the structure of the

k^{th} block of the matrix variable:

$$X^{(k)} = \begin{pmatrix} 1 & x_{1k} & x_{2k} & \cdots & x_{nk} \\ & x_{1k} & x_{1k}x_{2k} & \cdots & x_{1k}x_{nk} \\ & & x_{2k} & \cdots & x_{2k}x_{nk} \\ & & & \ddots & \vdots \\ & & & & x_{nk} \end{pmatrix}$$

If the variable x_{ik} is set to zero then the $(i+1)^{\text{th}}$ diagonal entry becomes zero. Because $X^{(k)}$ is positive semidefinite, all the entries in the same row or column as this diagonal entry then become zero (see Fact 5.1). Since the whole column/row is zero its variables play no further part in the constraints or objective function and may be removed from this block of the matrix variable.

Similarly, if the variable x_{ik} is set to one then the $(i+1)^{\text{th}}$ entry in the first row, first column and along the diagonal become '1'. All the entries in the $(i+1)^{\text{th}}$ column/row become identical to the first column/row of the matrix. Once again, this can be seen by the use of the properties of a positive semidefinite matrix with '1' as the first entry (see Fact 5.2). Since the whole column/row is a repeat of a previous one it is redundant and may be removed from this block of the matrix variable. By fixing $x_{ik} = 1$, we are also setting $x_{il} = 0$ for all $l \neq k$: the equivalent column/row may be removed from every block of the matrix variable.

The appropriate coefficients of a variable fixed to '1' must be included in the objective value and resource constraints. If a task i is assigned to processor k in the CMAP, we must reduce the remaining capacity of processor k by its resource requirement, a_i. If any a_j, $j \neq i$, are greater than the remaining capacity of processor k, then j may not be assigned to k; therefore $x_{jk} = 0$.

The tasks corresponding to fixed variables are not eligible to be members of S or E in an SDP relaxation with extension (Section 5.6.5).

5.7.2 Branching heuristics

We now discuss the selection of variables on which to branch. The following ideas may be used to construct heuristics for this purpose:

- Choose the variable that is furthest from being integer; i.e. a variable x_{ik} with value closest to 0.5. All entries are '0' or '1' in solution matrices having rank one, so entries nearest 0.5 are the least desirable.
- Choose the variable from among the tasks that contribute most to the objective value:
  - Communication costs: choose a variable x_{ik} for a task i with the maximum communication cost c_{ij}.
  - Summed communication costs: choose a variable x_{ik} for a task i with the maximum summed communication costs, $\sum_{j=1}^{n} c_{ij}$.
  - Execution costs: choose a variable x_{ik} corresponding to the largest e_{ik}.

Under the simplest scheme to apply these ideas, many equally 'good' variables would become candidates for branching. To narrow down the list of candidate variables we apply an ordered subset of these ideas to select a branching variable.

Some of our heuristics are based on the analysis of coefficients in the objective function. We consider the symmetric variant of the objective function (i.e. both $c_{ij}x_{ik}x_{jk}$ and $c_{ji}x_{jk}x_{ik}$ appear) where c_{ij} represents half of the total communication cost between i and j.

$$\sum_{k=1}^{m}\sum_{i=1}^{n} e_{ik}x_{ik} - \sum_{k=1}^{m}\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ik}x_{jk} \ .$$

Heuristic 1: Closest to 0.5

1. Choose the variable with value closest to 0.5.
2. If there is more than one candidate, select those with maximum summed communication costs for their associated task (see Heuristic 4).
3. If there is still more than one candidate, choose the variable with maximum execution cost (see Heuristic 2).

Heuristic 2: Maximum execution costs

1. Choose the variable with the maximum coefficient in the linear term of the objective function (this is usually its execution cost; see below).
2. If there is more than one candidate, select from these the variable with value closest to 0.5.

If a previous branch has fixed some variable to '1', then the corresponding task has been fixed to its processor and communication costs to this task move to the linear term in the objective function. If $x_{hk} = 1$, the objective function for the UTAP and CMAP is modified thus:

$$\begin{aligned} & \sum_{k=1}^{m}\sum_{i=1}^{n} e_{ik}x_{ik} - \sum_{k=1}^{m}\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ik}x_{jk} \\ = & \sum_{k=1}^{m}\sum_{i=1}^{n} e_{ik}x_{ik} - 2\sum_{k=1}^{m}\sum_{i=1}^{n} c_{ih}x_{ik}x_{hk} - \sum_{k=1}^{m}\sum_{i\neq h}\sum_{j\neq h} c_{ij}x_{ik}x_{jk} \ . \\ = & \sum_{k=1}^{m}\sum_{i=1}^{n}(e_{ik} - 2c_{ih})x_{ik} - \sum_{k=1}^{m}\sum_{i\neq h}\sum_{j\neq h} c_{ij}x_{ik}x_{jk} \end{aligned}$$

Heuristic 3: Maximum communication cost

1. Calculate the set of tasks $J = \{\underset{i}{\text{argmax}}\{c_{ij}\}\}$. Form a candidate set of variables $B = \{x_{ik} : i \in J\}$.
2. Select from B the candidate variable with value closest to 0.5.

Heuristic 3.1: 'Improved' maximum communication cost

This method improves on Heuristic 3 by modifying the c_{ij} for any processor with fixed variables before selecting candidates. Let $c_{ij}^{(k)}$ represent a communication statistic between tasks i and j on processor k. Initially, $c_{ij}^{(k)} = c_{ij}$ for all k, i, j.

If any $x_{hl} = 0$, we let $c_{ih}^{(l)} = c_{hi}^{(l)} = 0$ for $i = 1, \ldots, n$. (Any objective function terms involving x_{hl} will be equal to zero; since this includes $c_{ih}x_{il}x_{hl}$ we consider $c_{ih}x_{il}$ unimportant for $i = 1, \ldots, n$). We now use the modified $c_{ij}^{(k)}$ to pick candidate variables.

1. Calculate the candidate set of variables
$$B = \left\{ x_{ik} : \{(i,k)\} = \underset{(i,k)}{\operatorname{argmax}} \{c_{ij}^{(k)}\} \right\}.$$
2. Select from B the candidate variable with value closest to 0.5.

Heuristic 4: Maximum summed communication costs

1. Calculate the set of tasks
$$J = \left\{ \underset{i}{\operatorname{argmax}} \left\{ \sum_{j=1}^{n} c_{ij} \right\} \right\}.$$
Form a candidate set of variables $B = \{x_{ik} : i \in J\}$.
2. Select from B the candidate variable with value closest to 0.5.

Heuristic 4.1: 'Improved' maximum summed communication costs

This method improves on Heuristic 4 by altering the values of communication costs in the same way as the improved maximum communication costs heuristic.

1. Calculate the candidate set of variables

$$B = \left\{ x_{ik} : \{(i,k)\} = \underset{(i,k)}{\operatorname{argmax}} \left\{ \sum_{j=1}^{n} c_{ij}^{(k)} \right\} \right\}.$$

2. Select from B the candidate variable with value closest to 0.5.

5.7.3 A branch-and-bound algorithm for the UTAP and CMAP

The algorithm calculates a tree of solutions. At each node:

1. Solve the SDP relaxation with the branch constraints that apply at that node.
2. Either
   - pick a set of tasks with which to extend the matrix (see Section 5.6.5),
   - or **prune**. Move to the next node.
3. Solve the SDP relaxation again with the branch constraints and extended matrix.
4. Either
   - pick a new branching variable (see Section 5.10.2): two more nodes are generated,
   - or **prune**.
5. Move to the next node.

An SDP relaxation provides a lower bound on the objective value of the sub-problem given by the branch constraints. If this lower bound shows that

the optimal solution for the sub-problem must be worse than the best solution to the main problem we have so far discovered, we may prune the corresponding node. Effective pruning of the tree depends on the tightness of the bounds provided by the relaxation and the speed of discovery of solutions to the main problem.

The SDP relaxations become more effective as variables are fixed; this is due to the reduction in the dimension of the matrix variable with each branch (see Section 5.7.1).

Search strategy for optimal solutions

For the purpose of finding an optimal solution to the problem we apply a *depth-first* search strategy: we look for quick solutions to the root problem by following branches to the bottom of the tree. We first follow the branches where the variable is fixed to be '1' (fixing a variable to '1' also fixes another $m-1$ variables to be '0'): this provides a greater strengthening of the SDP relaxation and we expect such branches to lead to an integer solution more quickly, thus achieving more efficient pruning.

Search strategy for global lower bounds

We may also use a partial branch-and-bound tree to find *global lower bounds*. Such bounds may be used to analyse the quality of solutions produced by a heuristic. The objective value of any node forms a *local* lower bound on the optimal value for the associated sub-problem. Let G be a set of nodes with the following property: each feasible solution for the root problem is equivalent to a feasible solution for some sub-problem in G. The minimum objective value corresponding to a node in G forms a global lower bound for the instance.

Each complete level of the partial tree qualifies as a set G. To obtain these sets speedily, we proceed in a *breadth-first* search fashion. With each completed level we obtain a better global lower bound.

On the possible application of our branch-and-bound approach to the CTAP

In addition to x_{ik} variables that would be fixed by branching, the CTAP also has y_k variables representing which processors are used. One possible solution method is to calculate a tree of nodes by branching on the y_k alone. Once a feasible solution is found it may be used to prune nodes.

At the bottom level of the tree, the nodes represent sub-problems where only a subset of the processors are being used. These sub-problems take the form of the CMAP. We use each of these nodes as the root node for the branch-and-bound tree approach for the appropriate CMAP.

The fastest way to find a feasible solution may be to branch on those y_k that lead to the smallest feasible CMAP: order the processors in non-increasing order of capacity, b_k. By including each successive processor from the start of the list, determine P, the smallest subset of processors so that

$$\sum_{i=1}^{n} a_i \leq \sum_{k \in P} b_k.$$

Another alternative to find a good trial solution is to solve the most flexible (but largest) CMAP sub-problem: let $y_k = 1$ for $k = 1, \ldots, m$.

5.8 Computational experience for the UTAP

The procedures were coded in MATLAB and tested on an Intel Pentium 4, CPU 2.40 GHz, 504 MB RAM computer. The MATLAB code uses SDPT3 [115] to solve each semidefinite program relaxation. SDPT3 employs a predictor-corrector primal-dual path-following method and is able to exploit a block diagonal structure.

Table 5.1 displays the results for twenty problems of different sizes. The problems are those generated by Ernst et al. [47]. For each problem the table provides the number of processors (m), the number of tasks (n), the optimal value, the lower bound obtained by the SDP relaxation of the UTAP and the CPU time required to obtain the bound. The relative error of the bound from the optimal value is displayed. The average error across these instances is 10.5%, requiring 29.1 seconds on average.

Results for three LP relaxations were presented in [47]: LP2, LP3 and LP4. Their computer used a 500MHz alpha processor. LP2 provides slightly better bounds than ours on average, with an error of 8.3%. LP2 also provides its bounds significantly faster than ours, requiring 3.9 seconds on average. LP3 is able to obtain the optimal value in many cases, but requires significantly longer CPU times (494.5 seconds on average). Thus our SDP bounds do not compare very favourably to those found by the LP relaxations in [47].

m	n	Optimal	Bound	CPU time	Error %
5	30	710.59	691.11	1.285	2.74
5	40	1001.62	951.76	1.783	4.98
5	50	1389.05	1284.80	2.595	7.51
5	70	2368.20	2009.90	5.127	15.13
5	100	3784.95	3233.70	9.550	14.56
10	30	532.16	520.23	3.233	2.24
10	40	853.33	804.29	4.763	5.75
10	50	1226.46	1110.60	6.874	9.45
10	70	2178.08	1797.00	12.186	17.50
10	100	3439.86	2966.90	22.128	13.75
20	30	443.75	436.04	10.266	1.74
20	40	729.23	697.26	18.580	4.38
20	50	1073.58	983.36	23.265	8.40
20	70	2090.35	1690.40	41.794	19.13
20	100	3591.56	2861.90	72.266	20.32
30	30	394.62	379.50	24.373	3.83
30	40	662.81	619.27	34.030	6.57
30	50	1020.57	914.47	48.058	10.40
30	70	1983.18	1585.10	85.125	20.07
30	100	3458.34	2729.40	154.081	21.08

Table 5.1: Lower bounds using SDP relaxation for twenty instances of the UTAP.

5.9 Computational experience for the CTAP

Table 5.2 displays the results for six instances of the CTAP. These test problems have previously been used by [47] and [63]. For each problem the table provides the number of processors (m), the number of tasks (n), the best solution found in [63], the lower bound obtained by the SDP relaxation of the CTAP and the CPU time taken to obtain the bound. We also display the relative error of the bound from the best solution found in [63].

Problem	**m**	**n**	**Best solution**	**Bound**	**CPU time**	**Error %**
A	6	20	13804	8666.7	3.030	37.22
B	6	20	11946	6900.0	2.095	42.24
C	6	20	11120	5934.8	1.655	46.63
D	12	40	39680	17333	7.282	56.32
E	12	40	36575	13800	8.188	62.27
F	12	40	35821	11870	8.236	66.86

Table 5.2: Lower bounds using SDP relaxation for six instances of the CTAP.

The SDP bound values are identical in most cases to those provided by the LP relaxations in [47]. SDP relaxation CPU times are roughly 2 seconds longer than the LP relaxation for A, B and C; and 4 seconds longer for D, E and F. The SDP provided a slightly better bound for problem D.

5.10 Computational experience for the CMAP

The sixteen example problems are sampled from those described by Elloumi et al. [45], available at [44]. These include a representative from each of eight configurations of communication and execution costs. There are two problem sizes: (10 tasks, 3 processors) and (20 tasks, 5 processors), we represent these by the name components 1003 and 2005 respectively. The ranges of values corresponding to each instance code name are as follows:

- **A**: $e_{ik} \in [1, 100]$, $c_{ij} \in [0, 100]$, for all i, j, k.
- **B**: $e_{ik} \in [1, 10]$, $c_{ij} \in [0, 100]$, for all i, j, k.
- **C**: $e_{ik} \in [1, 100]$, $c_{ij} \in [0, 10]$, for all i, j, k.
- **D**: $e_{ik} = 0$, $c_{ij} \in [0, 100]$, for all i, j, k.

Half of the instances have complete communication graphs (instance names are preceded by a **d** for dense) while the others have 50% communication cost density. (The final lower case letter in the code name represents the particular instance within the configuration it represents).

Some of the bounding methods in [45] produce fairly good results. One particular linear relaxation provided an average error of 30% across all (10 task, 3 processor) instances and required only 1 second. This method seems to outperform those we present below. One of the semidefinite programming relaxations in [45] gives average bounding errors of 7%, but requires over 600 seconds for full convergence to a solution, on average.

5.10.1 Applying extensions to the matrix variable

The *extension set* contains pairs of tasks (i, j) that correspond to the first entry $x_{ik}x_{jk}$ in new columns/rows used to extend the matrix variable. The choice and size of the extension set will affect the bound obtained and the CPU running time for computation of the SDP relaxation with extension. It is desirable to find the smallest extension set that provides a bound value close to the value we shall call the *best bound by extension*. The best bound by extension is the bound obtained if all possible pairs of tasks are used in the extension set.

The sixteen CMAP example problems were solved first by the SDP relaxations without extension to provide our simplest SDP bound. We then solved the SDPs using the maximum possible size of extension set to discover the attainable improvement on the basic bound. The results are shown in Table 5.3.

The initial bounds found are extremely weak in general. Only the bounds for 'C' configuration instances were within 35% of the optimal; the lower bound for instance '1003Cc' is within 1.5% of optimal. (In a 'C' instance execution costs are up to 10 times as large as communication costs). Average running times were 0.66 seconds and 1.28 seconds for (10 task, 3 processor) and (20 task, 5 processor) instances respectively, with little variation between instances.

Instance	Optimal	Initial bound	Error (%)	Extension: Error reduction
1003Aa	731	420.20	42.52	5.30
d1003Aa	1616	480.38	70.27	0.76
1003Bb	528	58.20	88.98	0.08
d1003Bb	865	56.64	93.45	0.01
1003Cc	347	341.93	1.46	1.46
d1003Cc	475	388.83	18.14	10.59
1003Dd	445	0.00	100.00	0.00
d1003Dd	956	0.00	100.00	0.00
2005Aa	3059	896.72	70.69	1.58
d2005Aa	6412	911.49	85.78	0.24
2005Bb	2088	100.21	95.20	0.02
d2005Bb	5371	100.56	98.13	0.00
2005Cc	772	667.91	13.48	11.39
d2005Cc	1197	788.79	34.10	11.10
2005Dd	2211	0.00	100.00	0.00
d2005Dd	5594	0.00	100.00	0.00

Table 5.3: Initial SDP bounds for the 16 instances of the CMAP. The final column displays the reduction in the bound error when maximum matrix extension is used.

Instances	SES	CPU time
1003Aa	21	2.17
1003Cc	3	0.95
d1003Cc	28	3.14
2005Cc	66	31.77
d2005Cc	55	25.45

Table 5.4: Size of extension set (SES) used and CPU times required to obtain a bound within 0.5% of the best bound by extension.

The method of improving the bound by extending the matrix variable did not prove effective for most instances. The worst performance was for the 'D' configuration instances (in which execution costs are zero) as they received a lower bound of zero even with a fully extended matrix. However, extension did prove more effective for the 'C' instances: the error of the bound from the optimal value may be reduced by up to 11 percentage points. Note that the optimal solution for instance '1003Cc' is found when an extension is applied.

In Section 5.6.5 we proposed a number of heuristics for the selection of tasks, or pairs of tasks, to form the extension set. Experimentation has revealed that methods based on analysis of communication costs often provide the best improvement in the bounds for the smallest extension set. The *maximum communication costs* heuristic seems to be the best way to choose the most efficient extension set in general. Table 5.4 displays the running time required for the augmented SDP to obtain a bound within 0.5% of the best bound by extension; it displays only those instances who received a significant benefit. The run times are significantly larger than the times required for the basic SDP relaxation.

5.10.2 Comparison of branching heuristics

We implemented the branch-and-bound algorithms in MATLAB. Breadth-first search trees for each of the different branching heuristics (see Section 5.10.2) were generated for the CMAP example data. Global lower bounds on the optimal value were calculated at each level of the tree. A comparison of the increase in global lower bounds provided by each heuristic is given in Figure 5.1. A higher bound at a lower level indicates a superior heuristic.

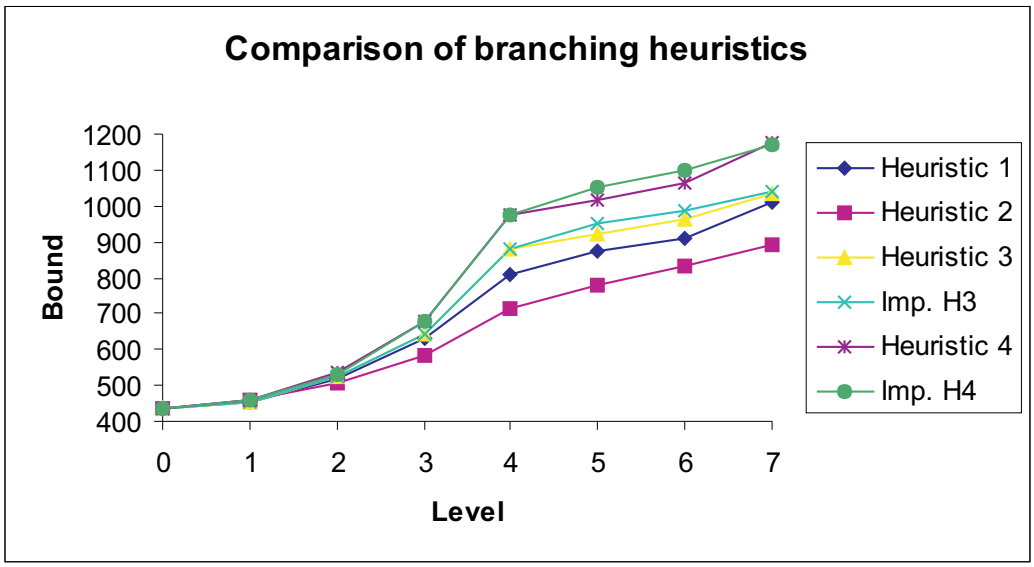

Figure 5.1: Bounds found at each level using different branching heuristics. Bound statistic is the average result of the bounds from eight (20 task, 5 processor) instances.

The results of the experiment revealed that the heuristics based on analysis of communication costs provided the best bounds. Heuristic 4.1: *'improved' maximum summed communication costs* yielded the best bounds on average. This produced slightly better bounds than Heuristic 4 at some levels of the tree. The next most effective were Heuristic 3.1 and Heuristic 3, which also utilise communication costs. Heuristic 1 was next; this selects

branching variables based on their value. Heuristic 2, which compares execution costs, was the least effective overall.

5.10.3 Global lower bounds for the CMAP

Implementing Heuristic 4.1 for selection of branching variables, we wish to determine the number of levels that should be calculated to achieve a good lower bound in a reasonable run time.

We first study (10 task, 3 processor) problems. Table 5.5 shows how the global lower bound improves as more levels of the tree are calculated; no extension to the matrix variable is used. In [45] the results suggested that the '1003D-' configuration instances (in which execution costs are zero) are generally the most difficult, while '1003C-' instances (in which execution costs are up to 10 times larger than communication costs) are the easiest to solve. The pattern is similar here. The bounds displayed in the tables compare favourably with those in [45] for the (10 task, 3 processor) instances. The algorithm provides the optimal solution within 10 levels of the tree in several instances. (Instance 1003Cc is solved at the first level, requiring only 1.94 seconds).

We next look at the (20 task, 5 processor) instances. Table 5.6 displays the performance of the search tree for eight instances after nine levels have been calculated. The CPU time for these instances is significant while the bound remains extremely weak. These bounds do not compete with the best of those found in [45], where errors were reported as between 1%-30%.

Application of extension sets did prove useful for two 'C' instances. The most efficient bound found for these is given in Table 5.7. We next take a closer look at the application of the extension sets to one of these examples: d2005Cc.

Level	**0**	**2**	**4**	**6**	**8**	**10**
1003Aa						
Error %	42.52	20.21	8.57	8.56	0.00	—
CPU time	0.55	3.11	12.94	19.04	20.30	—
1003Bb						
Error %	88.98	43.55	24.84	5.91	0.00	—
CPU time	0.51	2.92	12.37	32.56	36.38	—
1003Dd						
Error %	100.00	47.53	28.93	8.61	0.00	—
CPU time	0.54	3.09	12.70	48.84	93.09	—
d1003Aa						
Error %	70.27	49.47	35.40	20.27	11.54	5.09
CPU time	0.70	3.20	11.90	45.50	154.00	200.60
d1003Bb						
Error %	93.45	68.79	33.35	19.00	1.77	0.00
CPU time	0.54	2.85	11.55	42.91	63.70	64.34
d1003Cc						
Error %	18.14	11.28	9.07	5.26	3.21	1.66
CPU time	0.73	3.28	13.42	44.41	81.84	91.93
d1003Dd						
Error %	100.00	63.03	43.20	16.17	5.31	0.00
CPU time	0.67	2.95	11.72	44.70	60.99	63.64

Table 5.5: Error and CPU times (seconds) for (10 task, 3 processor) instances.

Level	**0**	**1**	**3**	**5**	**7**	**9**
2005Aa						
Error %	70.69	68.16	62.33	51.50	46.83	37.08
CPU time	1.50	3.60	15.90	65.60	262.70	1058.00
d2005Aa						
Error %	85.78	84.55	79.69	70.19	67.03	57.92
CPU time	1.30	3.20	16.20	64.70	257.90	1014.70
2005Bb						
Error %	95.20	95.00	84.96	72.50	62.01	48.00
CPU time	1.30	3.80	16.90	67.30	269.10	1095.10
d2005Bb						
Error %	98.13	98.09	91.93	78.70	75.86	70.86
CPU time	1.20	3.30	16.20	66.40	261.60	1018.70
2005Cc						
Error %	13.48	11.61	9.56	5.13	4.79	4.29
CPU time	1.30	3.70	17.50	71.40	288.20	1162.70
d2005Cc						
Error %	34.10	31.63	27.54	24.09	19.94	17.88
CPU time	1.20	3.40	16.50	68.30	275.80	1110.20
2005Dd						
Error %	100.00	100.00	89.76	73.35	70.91	49.68
CPU time	1.20	3.20	15.80	65.60	262.90	1046.70
d2005Dd						
Error %	100.00	99.82	92.01	81.17	78.32	65.65
CPU time	1.20	3.30	16.00	66.00	262.40	1034.60

Table 5.6: Error and CPU times (seconds) for (20 task, 5 processor) instances.

Instance	Error %	CPU Time	SES	Level
2005Cc	1.55	640	50	4
d2005Cc	11.41	2277	30	7

Table 5.7: Error and CPU times for the given size of extension set (SES) and level of the tree for two (20 task, 5 processor) instances.

Analysis of the size of the extension set

In Section 5.10.1 we decided that the best heuristic we had proposed for selection of an extension set was the *maximum communication costs* method. We selected this heuristic based on its ability to improve our lower bound at the root node for a reasonable increase in the size of the matrix variable. However, extension was only effective for 'C' configuration instances.

Instance d2005Cc

Figure 5.2 shows a plot of bound obtained against CPU time needed for calculation. Each series of points shows the progressive increase in the bound and run time as more levels of the tree are calculated (each point represents a level). The points lying closest to the bottom right of the graph are the best, indicating a higher bound calculated in a shorter run time. The slope of the lines between points measures the trade off between bound and run time at each level. A steep slope represents a poor trade off, so any 'elbow' points in the plot could indicate the number of levels of the tree to calculate for similar examples.

Figure 5.2 demonstrates that it is better to use a larger extension set when calculating bounds for this instance. The lines representing extension sets of size 30 and 40 lie furthest to the bottom right (SES 30 is better than 40 for the later levels). The ideal SES probably lies between 30 and 40 for this

example. Using SES greater than 40 does not produce large enough increases in the bound to compensate for the increase in run time.

Fewer levels need to be calculated for the larger sizes of extension set. The graph indicates that for SES 0, the benefit of calculating more levels diminishes after level 7. For SES 30 and 40, the benefit diminishes after level 4.

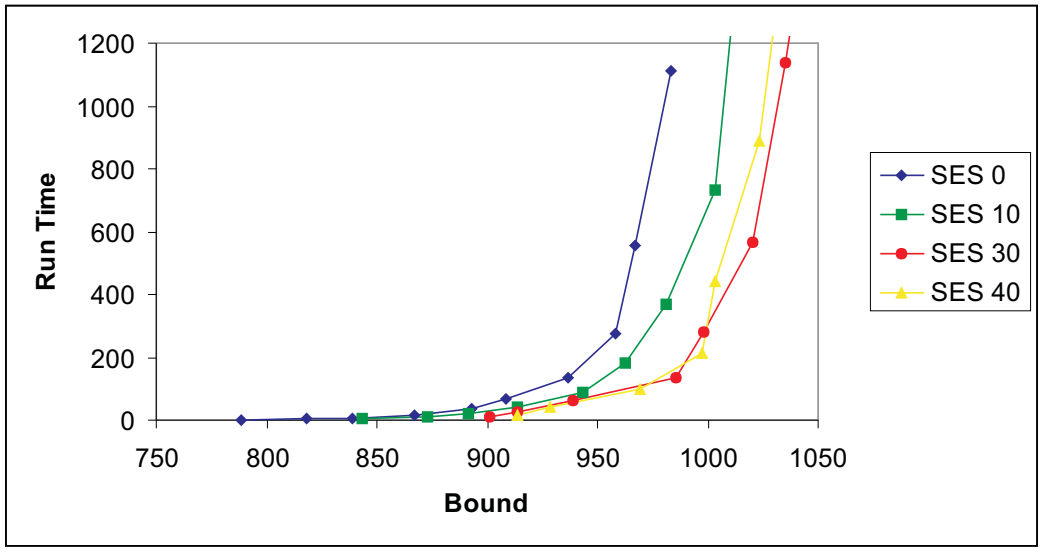

Figure 5.2: Comparison of bounds against CPU times for different sized extension sets (Instance d2005Cc: optimal value 1197).

5.10.4 Finding optimal solutions to the CMAP

In Section 5.10.3 we used a branch-and-bound tree to find global lower bounds for the optimal value of the CMAP. This involved searching the tree in a breadth-first fashion. The branch-and-bound tree may also be used to find the optimal solution to the CMAP. This is best achieved using a depth first search of the tree. Use of this search strategy should yield an integer solution faster than the breadth-first search. This can be used to prune nodes with an objective value higher than that of the integer solution.

Instances	CPU time (s)
1003Aa	17.98
d1003Aa	140.93
1003Bb	31.83
d1003Bb	53.01
1003Cc	1.51
d1003Cc	65.11
1003Dd	38.26
d1003Dd	51.10

Table 5.8: CPU times required to find the optimal solution

Table 5.8 shows the CPU time required to find the optimal solution to eight (10 task, 3 processor) instances.

We also performed experiments applying extensions to the matrix variable. We found that a larger extension set decreased the number of nodes in the tree, but resulted in a longer running time for six of the '1003' instances. Again the two 'C' instances benefited from matrix augmentation: with an extension set containing six pairs of tasks, the CPU time for 1003Cc reduced to 0.77 seconds; with one pair of tasks, d1003Cc reduced to 60.45 seconds.

The depth-first search of the tree did not prove effective for the larger (20 task, 5 processor) problems (CPU times exceeded 24 hours).

5.11 Further work: redundant constraints

We can tighten the SDP relaxation and improve the quality of the bounds by including redundant constraints [7].

The following approach may be applied to the CMAP. We start with the resource capacity constraint:

$$\sum_{i=1}^{n} a_i x_{ik} \leq b_k, \qquad k = 1, \ldots, m.$$

By dividing through by b_k, then including a slack variable w_k, this becomes

$$\sum_{i=1}^{n} \frac{a_i}{b_k} x_{ik} + w_k = 1, \qquad k = 1, \ldots, m. \tag{5.3}$$

We may generate redundant constraints by multiplying Equation (5.3) through by x_{jk} for a particular $j \in \{1, \ldots, n\}$.

$$\sum_{i=1}^{n} \frac{a_i}{b_k} x_{ik} x_{jk} + x_{jk} w_k = x_{jk}, \qquad k = 1, \ldots, m. \tag{5.4}$$

To include these in our SDP relaxation, an extra row/column with first entry w_k is added to each block $X^{(k)}$.

$$\begin{pmatrix} 1 & x_{1k} & x_{2k} & x_{3k} & \cdots & x_{nk} & w_k \\ & x_{1k} & x_{1k}x_{2k} & x_{1k}x_{3k} & \cdots & x_{1k}x_{nk} & x_{1k}w_k \\ & & x_{2k} & x_{2k}x_{3k} & \cdots & x_{2k}x_{nk} & x_{2k}w_k \\ & & & x_{3k} & \cdots & x_{3k}x_{nk} & x_{3k}w_k \\ & & & & \ddots & \vdots & \vdots \\ & & & & & x_{nk} & x_{nk}w_k \\ & & & & & & w_k^2 \end{pmatrix}$$

By relating the appropriate entries of the augmented matrix using the redundant constraints we may tighten the relaxation.

Further redundant constraints can be found by squaring Equation (5.3), then simplifying. We find

$$\sum_{i=1}^{n} \frac{a_i}{b_k} x_{ik} w_k + w_k^2 = w_k. \tag{5.5}$$

These may be included in the relaxation in addition to the constraints in (5.4) without increasing the size of the matrix any further.

If this approach is combined with the partial higher lifting approach (extending the matrix using pairs of tasks), more redundant constraints can be generated by multiplying the capacity constraints through by $x_{jk}x_{lk}, j, l \in v$.

We implemented the inclusion of the redundant constraints given above and achieved significant increases in the objective values for the resulting relaxations. Unfortunately, in a few cases the objective values were slightly higher than the optimal solution for the problem, indicating some kind of error. We were not able to find anything wrong with the coding for our implementation. For a further study of the benefits of these redundant constraints we might investigate whether the problem persisted when using a different SDP solver.

5.12 Conclusion

We have looked at applying semidefinite programming relaxation methods to task allocation problems. This method provides reasonable bounds for the UTAP but performs poorly for the CTAP and for many instances of the CMAP.

Methods for improving the bounds were investigated: extension to the matrix variable and creation of a branch-and-bound search tree for the CMAP. The branch-and-bound method was also used to find optimal solutions. We proposed strategies for choosing variables on which to branch and found that the heuristics based on analysis of communication costs produced the best results.

Extension to the matrix variable produced a small improvement in the bound. When combined with the branch-and-bound search tree, extension was only useful for a specific category of problems, where execution costs are generally larger than communication costs.

Our SDP relaxations were only useful for solving small instances of the CMAP; larger examples required unreasonable running time. Further investigation of redundant constraints may lead to significant improvements in the SDP bounds.

Chapter 6

Conclusion

6.1 Summary of contributions

This thesis looked at finding bounds and good solutions to two NP-hard combinatorial optimisation problems, and another combinatorial optimisation problem whose computational complexity has not yet been established.

6.1.1 The supply ship travelling salesman problem

We applied dynamic programming state-space relaxation approaches to determine lower bounds, and suggested that state-space modifiers should take the value of the corresponding node's replenishment time. This provided a significant improvement over the standard relaxation techniques.

Our variant of the cost function for a restricted dynamic programming heuristic provided good solutions in a reasonable time, outperforming simple 2-opt and 3-opt approaches for the instances we tested. Another of our cost measures, one that predicted final tour cost using the nearest neighbour

heuristic, also provided good solutions but required longer computation times.

6.1.2 The supply ship scheduling problem

We introduced the supply ship scheduling problem and showed how problem instances could be represented as directed graphs. Objective values for solutions to this problem were calculated by constructing an appropriate network and determining its minimum flow. Quick upper and lower bounds could be found using this network structure, but the lower bounds were found to be weak.

We proposed a neighbourhood structure based on a maximum cut in the solution network. The aim of this structure was to restrict possible moves to those likely to yield an improvement in the objective value. Several effective descent algorithms and a tabu search procedure were implemented using this neighbourhood, each providing quick solutions.

We developed a heuristic inspired by restricted dynamic programming. It is only competitive with the tabu search for fairly small problems, but allows us to find solutions corresponding to a specified objective value. We have indicated, but not implemented, a formulation that may slightly improve its performance.

6.1.3 The task allocation problem

We considered the application of semidefinite programming relaxation methods to variants of the task allocation problem. This method provides reasonable bounds for the UTAP and also for some small instances of the CMAP. Application of semidefinite programming relaxation to the CTAP

produced almost identical results to those from linear relaxations.

A partial higher lifting approach for improving the relaxations was investigated, involving the proposal of a number of heuristics for selection of task sets. The approach did not provide significant improvements, but communication costs were highlighted empirically as important indicators for the selection of tasks.

A branch-and-bound approach, utilising the semidefinite programming relaxation, was only useful for solving small instances of the CMAP, but could be used to find reasonable global lower bounds. We proposed a number of heuristics for selection of the branching variable. Empirical testing showed that strategies based on communication costs provided the best global lower bounds.

6.2 Suggestions for further work

6.2.1 The supply ship travelling salesman problem

The bounds derived from dynamic programming state-space relaxation were never tested within a branch-and-bound scheme. Such a scheme would need to determine appropriate sub-problems and branching strategies. A straightforward approach would be as follows: each branch adds a single warship to the next position in the partial tour for that sub-problem.

Only the most basic, and deterministic, search procedures have so far been applied; stochastic approaches may yield improved results. Remember that a seemingly simple change alters all the arc costs in the tour after the first affected node. An efficient local search procedure must adequately deal with this property.

A variation of the restricted dynamic program could be tested on the supply ship TSP with additional concerns, such as time windows and precedence constraints. It seems likely that this sort of approach could deal fairly well with these issues.

6.2.2 The supply ship scheduling problem

Local search approaches that employ a random element may provide improved solutions. More sophisticated search procedures (e.g variable neighbourhood search) may also do well.

We may formulate the problem as a linear integer program. Recall the definitions of a_{ij} and A_i from Section 4.5. The IP variables may be defined as follows. Let m_i be the number of machines allocated to job i. Define x_{ij} to be the number of machines travelling from job i to job j. Let y_{ij} be a 0-1 variable representing whether machines may process job j after job i. We have

$$
\begin{aligned}
\min \quad & \sum_{j=1}^{n} x_{0j} & & \\
\text{s.t.} \quad & \sum_{i=0}^{j-1} x_{ij} = m_j, & & j = 1, \ldots, n \\
& m_i \geq 1, & & i = 1, \ldots, n \\
& m_i \geq \sum_{j=i+1}^{n} x_{ij}, & & i = 1, \ldots, n \\
& m_i \geq a_{ij} y_{ij}, & & i = 1, \ldots, n-1; j = i+1, \ldots, n \\
& x_{ij} \leq n y_{ij}, & & i = 1, \ldots, n-1; j = i+1, \ldots, n \\
& m_i \in A_i & & i = 1, \ldots, n \\
& y_{ij} \in \{0, 1\}, & & i = 1, \ldots, n-1; j = i+1, \ldots, n \\
& x_{ij} \in \{0, 1, 2, \ldots, n\}, & & i = 0, 1, \ldots, n-1; j = i+1, \ldots, n \\
& x_{ij} = 0 \text{ and } y_{ij} = 0 & & \text{if } a_{ij} > n \text{ or } a_{ij} = \infty.
\end{aligned}
$$

The objective is to minimise the number of machines required, which is equivalent to the sum of the machines leaving the depot. The constraints

impose conditions on the total number of machines entering a job. At least one machine must process each job and total machines leaving a job cannot exceed the number allocated to it. The remaining two inequalities ensure that machines may only travel between jobs i and j if enough machines were allocated to job i to allow it; thus x_{ij} may only be non-zero if m_i is at least a_{ij}. This integer programming formulation was not implemented or explored any further in this thesis. Perhaps small instances could be solved (or lower bounds discovered) through the development of this method.

We have not yet shown that the supply ship scheduling problem is NP-hard, although it appears to be a difficult problem.

In our variant of the supply ship scheduling problem the objective was to minimise the number of machines required. An alternative problem is to constrain the number of machines available and instead minimise the number of missed jobs. A restricted dynamic programming approach may be better suited to solving this variant than the one studied in this thesis.

6.2.3 The task allocation problem

Although the SDP relaxations we have presented did not produce impressive bounds, there is still scope for SDP to be a useful tool for these problems. A further investigation into the benefits of adding capacity based redundant constraints to the CMAP is warranted.

References

[1] E. Aarts and J.K. Lenstra (Eds). *Local Search in Combinatorial Optimization*. John Wiley & Sons, Chichester, 1997.

[2] T.S. Abdul-Razaq and C.N. Potts. Dynamic programming state-space relaxation for single-machine scheduling. *Journal of the Operational Research Society*, 39(2):141–152, 1988.

[3] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows: Theory, Algorithms, and Applications*, chapter 6, pages 166–206. Prentice Hall, New Jersey, 1993.

[4] F. Alizadeh. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM Journal on Optimization*, 5:13–51, 1995.

[5] M.F. Anjos. Mathematical programming. Lecture notes, 2002.

[6] M.F. Anjos. An improved semidefinite programming relaxation for the satisfiability problem. *Mathematical Programming*, 102(3):589–608, 2004.

[7] M.F. Anjos and H. Wolkowicz. Semidefinite programming for discrete optimization and matrix completion problems. *Discrete Applied Mathematics*, 123(1–2):513–577, 2002.

[8] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. On the solution of traveling salesman problems. *Documenta Mathematica*, Extra Volume ICM 1998(III):645–656, 1998.

[9] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[10] D. Applegate, W. Cook, and A. Rohe. Chained lin-kernighan for large traveling salesman problems. *INFORMS Journal on Computing*, 15(1):82–92, 2003.

[11] K.R. Baker. *Introduction to Sequencing and Scheduling*. Wiley, New York, 1974.

[12] E.B. Baum. Iterated descent: A better algorithm for local search in combinatorial optimization problems. Technical report, Caltech, Pasadena, CA, 1986.

[13] K.P. Belkhale and P. Banerjee. An approximate algorithm for the partitionable independent task scheduling problem. In *International Conference on Parallel Processing*, August 1990.

[14] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[15] R. Bellman. Dynamic programming treatment of the traveling salesman problem. *Journal of the ACM*, 9(1):61–63, 1962.

[16] A. Billionet, M. Costa, and A. Sutter. An efficient algorithm for a task allocation problem. *Journal of the Association for Computing Machinery*, 39:502–518, 1992.

[17] A. Billionnet and S. Elloumi. Placement de tâches dans un système distribué et dualité lagrangienne. *RAIRO Recherche Opérationelle*, 26(1):83–97, 1992.

[18] J. Blazewicz, M.Y. Kovalyov, M. Machowiak, D. Trystram, and J. Weglarz. Scheduling malleable tasks on parallel processors to minimize the makespan. *Annals of Operations Research*, 129:6580, 2004.

[19] J. Blazewicz, M. Machowiak, G. Mounie, and D. Trystram. Suboptimal approaches to scheduling malleable tasks. *Computational Methods in Science and Technology*, 6:25–40, 2000.

[20] F. Bock. An algorithm for solving travelling-salesman and related network optimization problems. Manuscript associated with talk presented at the 14th National Meeting of ORSA, St. Louis, MO, 1958.

[21] 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 et al., editor, *EvoWorkshop 2001, LNCS 2037*, pages 203–212, 2001.

[22] E.K. Burke, M. Dror, and J.B. Orlin. Scheduling malleable tasks with interdependent processing rates: Comments and observations. *Discrete Applied Mathematics*, 156(5):620–626, March 2008.

[23] G. Carpaneto, M. Dell'Amico, and P. Toth. Exact solutions of large-scale, asymmetric traveling salesman problems. *ACM Transactions on Mathematical Software*, 21(4):394–409, 1995.

[24] W. Chen and C. Lin. A hybrid heuristic to solve a task allocation problem. *European Journal of Operational Research*, 27:287–303, 2000.

[25] B.V. Cherkassky and A.V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997.

[26] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. Technical Report CS-93-13, Carnegie Mellon University, 1976.

[27] N. Christofides, A. Mingozzi, and P. Toth. State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11:145–164, 1981.

[28] J. Cirasella, D.S. Johnson, L.A. McGeoch, and W. Zhang. The asymmetric traveling salesman problem: algorithms, instance generators, and tests. In *ALENEX 2001 Proceedings*, pages 32–59, 2001.

[29] E. Ciurea and L. Ciupala. Sequential and parallel algorithms for minimum flows. *Journal of Applied Mathematics & Computing*, 15(1-2):53–75, 2004.

[30] G. Clarke and J.W. Wright. Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12:568–581, 1964.

[31] E. Coffman, M. Garey, D. Johnson, and R. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826, November 1980.

[32] G.A. Croes. A method for solving traveling salesman problems. *Operations Research*, 6:791–812, 1958.

[33] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling salesman problem. *Journal of the Operational Research Society of America*, 2(4):393–410, 1954.

[34] G. Dantzig, R. Fulkerson, and S. Johnson. On a linear-programming combinatorial approach to the traveling-salesman problem. *Operations Research*, 7(1):58–66, 1959.

[35] L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[36] W. Fernandez de la Vega and M. Lamari. The task allocation problem with constant communication. *Discrete Applied Mathematics*, 131(1):169–177, 2003.

[37] M.R. de Paula, M.G. Ravetti, G.R. Mateus, and P.M. Pardalos. Solving parallel machines scheduling problems with sequence-dependent setup times using variable neighbourhood search. *IMA Journal of Management Mathematics*, 18(2):101–115, 2007.

[38] R.P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51:161–166, 1950.

[39] M. Dorigo and T. Stützle. *Ant Colony Optimization*. The MIT Press, 2004.

[40] M. Drozdowski. Scheduling multiprocessor tasks – an overview. *European Journal of Operational Research*, 94:215–230, 1996.

[41] J. Du and J.Y-T. Leung. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics*, 2(4):473–487, November 1989.

[42] P-F. Dutot, G. Mouni, and D. Trystram. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter Scheduling parallel tasks – approximation algorithms. CRC Press, Boca Raton, 2004.

[43] W.L. Eastman. *Linear programming with pattern constraints*. Ph.d thesis, The Computation Laboratory, Harvard University, 1958.

[44] S. Elloumi. Website: Task assignment problem instances. http://cedric.cnam.fr/oc/TAP/TAP.html, accessed 10/11/03.

[45] S. Elloumi, F. Roupin, and E. Soutif. Comparison of different lower bounds for the constrained module allocation problem. Technical Report 473, CEDRIC-CNAM, 2003.

[46] A.A. Elsadek and B.E. Wells. A heuristic model for task allocation in heterogeneous distributed computing systems. *The International Journal of Computers and Their Applications*, 6(1), 1999.

[47] A. Ernst, H. Jiang, and M. Krishnamoorthy. Exact solutions to task allocation problems. Technical report, CSIRO Mathematical Information and Sciences, 2002.

[48] T. Feo and M. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[49] K.R. Fox, B. Gavish, and S.C. Graves. An n-constraint formulation of the (time-dependent) traveling salesman problem. *Operations Research*, 28(4):1018–1021, 1980.

[50] P.M. Franca, M. Gendreau, G. Laporte, and F.M. Muller. A tabu search heuristic for the multiprocessor scheduling problem with sequence dependent setup times. *International Journal of Production Economics*, 43:79–89, 1996.

[51] G. Frederickson, M.S. Hecht, and C.E. Kim. Approximation algorithms for some routing problems. *SIAM Journal on Computing*, 7(2):178–193, May 1978.

[52] M.L. Fredman, D.S. Johnson, L.A. McGeoch, and G. Ostheimer. Data structures for traveling salesman. In *Proceedings of the fourth annual ACM-SIAM symposium on discrete algorithms*, pages 145–154, 1993.

[53] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[54] I. Gertsbakh and H.I. Stern. Minimal resources for fixed and variable job schedules. *Operations Research*, 26(1):68–85, 1978.

[55] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[56] A.V. Goldberg and R.E. Tarjan. A new approach to the maximum flow problem. *Journal of the Association for Computing Machinery*, 35(4):921–940, 1988.

[57] L. Gouveia and S. Voß. A classification of formulations for the time-dependent traveling salesman problem. *European Journal of Operational Research*, 83:69–82, 1995.

[58] U.I. Gupta, D.T. Lee, and J.Y-T. Leung. An optimal solution for the channel-assignment problem. *IEEE Transactions on Computers*, 28(11):807–810, November 1979.

[59] M. Gursky. Some complexity results for a multi-processor scheduling problem. private communication from H.S. Stone, 1981.

[60] G. Gutin and A.P. Punnen (Eds). *The Traveling Salesman Problem and Its Variations*. Kluwer Academic Publishers, Dordrecht, 2002.

[61] G. Gutin and A.P. Punnen (Eds). *The Traveling Salesman Problem and Its Variations*, chapter 9. Experimental analysis of heuristics for the STSP. Kluwer Academic Publishers, Dordrecht, 2002.

[62] G. Gutin and A.P. Punnen (Eds). *The Traveling Salesman Problem and Its Variations*, chapter 10. Experimental analysis of heuristics for the ATSP. Kluwer Academic Publishers, Dordrecht, 2002.

[63] A. Hadj-Alouane, J. Bean, and K. Murty. A hybrid genetic/optimization algorithm for a task allocation problem. *Journal of Scheduling*, 2:189–201, 1999.

[64] Y. Hamam and K. Hindi. Assignment of program modules to processors: A simulated annealing approach. *European Journal of Operational Research*, 122:509–513, 2000.

[65] M. Hammar and B.J. Nilsson. Approximation results for kinetic variants of tsp. *Discrete & Computational Geometry*, 27:635–651, 2002.

[66] P. Hansen and N. Mladenović. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449–467, May 2001.

[67] P. Hansen and N. Mladenović. A tutorial on variable neighborhood search. Technical report, Les Cahiers du GERAD, HEC Montreal and GERAD, 2003.

[68] M. Held and R.M. Karp. A dynamic programming approach to sequencing problems. *Journal of SIAM*, 10:196–210, 1962.

[69] M. Held and R.M. Karp. The traveling-salesman problem and minimum spanning trees: Part ii. *Mathematical Programming*, 1(1):6–25, 1971.

[70] K. Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126:106–130, 2000.

[71] C.S. Helvig, G. Robins, and A. Zelikovsky. The moving-target traveling salesman problem. *Journal of Algorithms*, 49:153–174, 2003.

[72] S. Hewitt. Support ship routing in a deployed task group. Master's thesis, School of Mathematics, University of Southampton, 2004.

[73] T. Ibaraki and N. Katoh. *Resource Allocation Problems: Algorithmic Approaches*. The MIT Press, Cambridge, Massachusetts, 1988.

[74] Q. Jiang, R. Sarker, and H. Abbass. Tracking moving targets and the non-stationary traveling salesman problem. *Complexity International*, 11:171–179, 2005.

[75] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[76] Y. Kopidakis, M. Laman, and V. Zissimopoulos. On the task assignment problem: Two new efficient heuristic algorithms. *Journal of Parallel and Distributed Computing*, 42:21–29, 1997.

[77] A.H. Land and A.G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[78] E.L. Lawler. The quadratic assignment problem. *Management Science*, 9(4):586–599, 1963.

[79] Y.H. Lee and M. Pinedo. Scheduling jobs on parallel machines with sequence-dependent setup times. *European Journal of Operational Research*, 100:464–474, 1997.

[80] J.Y-T. Leung. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman & Hall, 2004.

[81] M. Lewis, B. Alidaee, and G. Kochenberger. Modeling and solving the task allocation problem as an unconstrained quadratic binary program. *Operations Research Letters*, 2004.

[82] S. Lin. Computer solutions of the traveling salesman problem. *Bell Systems Technical Journal*, 1965.

[83] S. Lin and B.W. Kernighan. An effective algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–416, 1973.

[84] J.D.C. Little, K.G. Murty, D.W. Sweeney, and C. Karel. An algorithm for the traveling-salesman problem. *Operations Research*, 11:972–989, 1963.

[85] V. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Transactions on Computers*, 37:1384–1397, 1988.

[86] A. Lucena. Time-dependent travelling salesman problem - the deliveryman case. *Networks*, 20:753–763, 1990.

[87] W.T. Ludwig. *Algorithms for scheduling malleable and nonmalleable parallel tasks*. Ph.d thesis, University of Wisconsin - Madison, Department of Computer Sciences, 1995.

[88] A. Lusa and C.N. Potts. A variable neighbourhood search algorithm for the constrained task allocation problem. Technical Report IOC-DT-P-2006-5, EOLI - Institut d'Organitzaci i Control de Sistemes Industrials, 2006.

[89] P. Ma, E. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, C-31:41–47, 1982.

[90] C. Malandraki. *Time dependent vehicle routing problems: Formulations, solution algorithms and computational experiments*. Ph.d dissertation, Northwestern University, Evanston, IL, 1989.

[91] C. Malandraki and M.S. Daskin. Time dependent vehicle routing problems: Formulations, properties and heuristic algorithms. *Transportation Science*, 26:185–200, 1992.

[92] C. Malandraki and R.B. Dial. A restricted dynamic programming heuristic algorithm for the time dependent traveling salesman problem. *European Journal of Operational Research*, 90:45–55, 1994.

[93] A.S. Mendes, F.M. Muller, P.M. Franca, and P. Moscato. Comparing meta-heuristic approaches for parallel machine scheduling problems. *Production Planning & Control*, 13(2):143–154, 2002.

[94] J.E. Mitchell. *Integer programming: Branch and cut algorithms, In "Encyclopedia of Optimization" (C.A. Floudas and P.M. Pardalos, Eds)*, volume 2, pages 519–525. Kluwer Academic Press, 2001.

[95] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):10971100, 1997.

[96] J.D. Monte and K.R. Pattipati. Scheduling parallelizable tasks to minimize make-span and weighted response time. *IEEE Transactions on Systems, Man and Cybernetics - Part A*, 32(3):335–345, May 2002.

[97] T.E. Morton and D.W. Pentico. *Heuristic Scheduling Systems.* John Wiley & Sons, New York, 1993.

[98] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* Wiley, New York, 1988.

[99] I. Or. *Traveling Salesman-Type Combinatorial Problems and their Relation to the Logistics of Regional Blood Banking.* Ph.d thesis, Northwestern University, Evanston, IL, 1976.

[100] M. Padberg and G. Rinaldi. Facet identification for the symmetric traveling salesman polytope. *Mathematical Programming*, 47(2):219–257, 1990.

[101] C.H. Papadimitriou. *Computational Complexity.* Addison Wesley, 1993.

[102] J.C. Picard and M. Queyranne. The time-dependent traveling salesman problem and its application to the tardiness problem in one-machine scheduling. *Operations Research*, 26(1):86–110, 1978.

[103] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Prentice Hall, 2nd edition edition, 2001.

[104] C.N. Potts. Deterministic operational research techniques: Heuristics. Lecture notes, 2002.

[105] S. Radhakrishnan and J.A. Ventura. Simulated annealing for parallel machine scheduling with earliness-tardiness penalties and sequence-dependent set-up times. *International Journal of Production Research*, 38(10):2233–2252, 2000.

[106] K.N. Rao. Optimal synthesis of microcomputers for gm vehicles. Technical report, 1992.

[107] F. Roupin. On approximating the memory-constrained module allocation problem. *Information Processing Letters*, 61(4):205–208, 1997.

[108] F. Roupin. From linear to semidefinite programming: an algorithm to obtain semidefinite relaxations for bivalent quadratic problems. Technical Report 388, CEDRIC-CNAM, 2003.

[109] A. Sarje and G. Sagar. Heuristic model for task allocation in distributed computer systems. *IEE Proceedings*, E-138:313–318, 1991.

[110] M.W.P. Savelsbergh. *Branch and price: Integer programming with column generation, In "Encyclopedia of Optimization" (C.A. Floudas and P.M. Pardalos, Eds)*, volume 1, pages 218–221. Kluwer Academic Press, 2001.

[111] J. Schneider. The time-dependent traveling salesman problem. *Physica A*, 314:151–155, 2002.

[112] A. Schrijver. On the history of combinatorial optimization (till 1960). Technical report, Department of Mathematics, University of Amsterdam, The Netherlands, 1996.

[113] C.C. Shen and W.H. Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transactions on Computers*, 34(3):197–203, 1985.

[114] H. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE-3:85–93, 1977.

[115] K.C. Toh, R.H. Tutuncu, and M.J. Todd. Sdpt3 version 3.02 – a matlab software for semidefinite-quadratic-linear programming. http://www.math.nus.edu.sg/ mattohkc/sdpt3.html, accessed 15/05/03.

[116] M.A. Trick. A tutorial on dynamic programming. Website: http://mat.gsia.cmu.edu/classes/dynamic/dynamic.html, 1998.

[117] J. Turek, J.L. Wolf, K.R. Pattipati, and P.S. Yu. Scheduling parallelizable tasks: putting it all on the shelf. In *SIGMETRICS '92/PERFORMANCE '92: Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, 1992.

[118] J. Turek, J.L. Wolf, and P.S. Yu. Approximate algorithms scheduling parallelizable tasks. In *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, 1992.

[119] 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.

[120] Website. *Andrew Goldberg's Network Optimization Library.* www.avglab.com/andrew/soft.html, accessed 06/07/07.

[121] Website. *Concorde Home.* www.tsp.gatech.edu/concorde, accessed 7/11/08.

[122] N. Widell and C. Nyberg. Cross entropy based module allocation for distributed systems. In *Proceedings of the 16th IASTED International Conference on Parallel and Distributed Computing and Systems*, 2004.

[123] R.J. Vander Wiel and N.V. Sahanidis. Heuristic bounds and test problem generation for the time-dependent traveling salesman problem. *Transportation Science*, 29(2):167–183, 1995.

[124] H. Wolkowicz, R. Saigal, and L. Vandenberghe. *Handbook of semidefinite programming: theory, algorithms and applications*. Kluwer Academic Publishers, 2000.