UNIVERSITY OF SOUTHAMPTON

# Exact Tests via Complete Enumeration: A Distributed Computing Approach

by

Danius Takis Michaelides

Doctor of Philosophy

in the

Faculty of Social Sciences
Social Statistics

October 1997

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF SOCIAL SCIENCES

SOCIAL STATISTICS

Doctor of Philosophy

Exact Tests via Complete Enumeration:
A Distributed Computing Approach

by Danius Takis Michaelides

The analysis of categorical data often leads to the analysis of a contingency table. For large samples, asymptotic approximations are sufficient when calculating p-values, but for small samples the tests can be unreliable. In these situations an exact test should be considered. This bases the test on the exact distribution of the test statistic. Sampling techniques can be used to estimate the distribution. Alternatively, the distribution can be found by complete enumeration.

This thesis develops a number of new algorithms for complete enumeration of various models. Recursive algorithms are developed to test for independence in $r \times c$ tables. The algorithm is extended for multi-dimensional tables. One algorithm is extended to enumerate tables under the model of quasi-independence, and a rejection stage enables testing of models such as quasi-symmetry and uniform association.

A new algorithm is developed that enables a model to be defined by a model matrix, and all tables that satisfy the model are found. This provides a more efficient enumeration mechanism for complex models and extends the range of models that can be tested. The technique can lead to large calculations and a distributed version of the algorithm is developed that enables a number of machines to work efficiently on the same problem.

# Contents

# List of Tables

# List of Figures

# Acknowledgements

This work was supported by an ESRC Research Studentship (award R0042933423) under the ESRC Analysis of Large and Complex Datasets Programme. The Department of Social Statistics provided additional support for which I am grateful. The use of the SP2 was supported by the IBM SUR programme.

I'd like to thank Dave De Roure for getting me started on the whole PhD thing and for steady supervision and direction throughout the thesis, whilst I disappeared off into the world of statistics. Secondly, thanks to Mac McDonald, as my other half-supervisor, for his guidance on all things statistical and his attention to detail in the final stages of writing up.

I'd also like to thank, for various things, Peter Smith, Toby Prevost, Fiona Steele and Paul Clarke. And Kathy Hooper. Finally, I shall fail to express my appreciation to Pete Tonkin.

To Lauren and Zac.

# Chapter 1

# Introduction

## 1.1 Aims

The aim of this thesis is to investigate and develop new algorithms for the analysis of contingency tables using exact tests. The technique employed is that of complete enumeration to calculate the exact distribution of the test statistic.

The technique of complete enumeration is algorithmically and computationally demanding. This thesis will investigate both these aspects. Algorithmically, the thesis will develop new algorithms for the enumeration of tables. Primarily the aim is to enable the testing of more complex models which no algorithms currently allow. These more complex models allow researchers to test for more complex interactions in their sparse contingency tables. These complex models are often more relevant to the data, e.g., social mobility tables.

The use of complete enumeration can require a large amount of computation. Even for small tables the number of tables in the reference set can be massive. With this issue in mind, the algorithms developed will allow calculations to be performed on a collection of computers, utilising their process power to return an exact p-value quicker than a single computer.

This chapter introduces the background material on the analysis of contingency tables and establishes the current state of algorithms and approaches to performing exact tests.

## 1.2 Contingency Table Analysis

There is a large body of literature on categorical data analysis. There are a number of good surveys on the subject (Agresti, 1990; Agresti, 1996). This chapter will give a general overview of the subject and look closely at the problem of analysing sparse contingency tables, providing a background for the remaining chapters in this thesis.

Categorical data are data for which the measurement scale consists of a number of categories, e.g., gender (male/female); yes/no responses; ratings (never, occasionally, fairly often, very often, almost always). Such categorical data occurs commonly in a wide range of areas. Categories can be ordered, such as ratings, and can also be derived from continuous data, e.g., age divided into categories young and old.

Categorical data are often summarised in the form of contingency tables, which consist of a cross-classification of counts of observations occurring in the possible categories.

### 1.2.1 Anatomy of a Contingency Table

A 2-way contingency table for the cross-classification of two variables $A$ and $B$ consists of $r$ rows, one for each category in $A$ and $c$ columns, one for each category in $B$. This is usually referred to as an $r \times c$ table with $i$ and $j$ used as row and column indices respectively. In addition, the table has a row and a column margin, consisting of the row and column totals respectively. Each cell in table $x$ is denoted as $x_{ij}$. A $+$ is used to indicate summation over all levels of the obscured index, for example, $x_{i+}$ is the $i$th row total and $x_{++}$ is the table total. Table 1.1 shows this structure.

| | $B_1$ | $B_2$ | $\ldots$ | $B_j$ | $\ldots$ | $B_c$ | |
|---|---|---|---|---|---|---|---|
| $A_1$ | $x_{11}$ | $x_{12}$ | $\ldots$ | $x_{1j}$ | $\ldots$ | $x_{1c}$ | $x_{1+}$ |
| $A_2$ | $x_{21}$ | $x_{22}$ | $\ldots$ | $x_{2j}$ | $\ldots$ | $x_{2c}$ | $x_{2+}$ |
| $\vdots$ | $\vdots$ | $\vdots$ | | $\vdots$ | | $\vdots$ | $\vdots$ |
| $A_i$ | $x_{i1}$ | $x_{i2}$ | $\ldots$ | $x_{ij}$ | $\ldots$ | $x_{ic}$ | $x_{i+}$ |
| $\vdots$ | $\vdots$ | $\vdots$ | | $\vdots$ | | $\vdots$ | $\vdots$ |
| $A_r$ | $x_{r1}$ | $x_{r2}$ | $\ldots$ | $x_{rj}$ | $\ldots$ | $x_{rc}$ | $x_{r+}$ |
| | $x_{+1}$ | $x_{+2}$ | $\ldots$ | $x_{+j}$ | $\ldots$ | $x_{+c}$ | $x_{++}$ |

Table 1.1: Anatomy of an $r \times c$ table.

### 1.2.2 Independence

Given two categorical variables $A$ and $B$, the probability that an observation occurs in the $ij$ cell is the joint probability of observing $A_i$ and observing $B_j$,

$$\pi_{ij} = P(A_i \cap B_j), \qquad i = 1, 2, \ldots, r, \quad j = 1, 2, \ldots, c$$

The model of independence specifies that

$$\pi_{ij} = P(A_i)P(B_j) = \pi_{i+}\pi_{+j}$$

for all $i$ and $j$. A test statistic can be used to measure departure from independence across the whole table. This compares the *observed* table cell counts with their *expected* values under the model of independence. The *expected* values $e_{ij}$ under the model of independence are given by:

$$\begin{aligned} e_{ij} &= x_{++}p_{i+}p_{+j} \quad \text{with} \quad p_{i+} = x_{i+}/x_{++} \quad \text{and} \quad p_{+j} = x_{+j}/x_{++} \\ &= \frac{x_{i+}x_{+j}}{x_{++}} \end{aligned} \tag{1.1}$$

where $x_{++}$ is the table total, and $p_{i+} = x_{i+}/x_{++}$ is the observed proportion in the $i$th row and $p_{+j} = x_{+j}/x_{++}$ is the observed proportion in the $j$th column.

One such measure of departure from independence is the Pearson "goodness-of-fit" chi-squared statistic:

$$X^2 = \sum_{i=1}^{r}\sum_{j=1}^{c}\frac{(x_{ij} - e_{ij})^2}{e_{ij}}$$

Here, the larger the value of $X^2$, the larger the departure from the model. Another such test, often used alongside $X^2$ is the likelihood-ratio goodness-of-fit statistic, denoted by $G^2$:

$$G^2 = 2\sum_{i=1}^{r}\sum_{j=1}^{c} x_{ij} \log \frac{x_{ij}}{e_{ij}}$$

Again, the larger values of $G^2$ indicate a greater departure from the model.

The *p-value* for testing the null hypothesis $\mathrm{H}_0$ is the probability of obtaining outcomes at least as extreme as the observed outcome. The larger the p-value the stronger the evidence against the null hypothesis. For large samples, both $X^2$ and $G^2$ are distributed as a $\chi^2$ distribution. The number of degrees of freedom of the $\chi^2$ distribution depends on the model being considered. For the model of independence for an $r \times c$ table, there are $(r-1)(c-1)$ degrees of freedom. Hence, the p-value is given by the tail probability to the right of the observed test statistic. When using this large sample or asymptotic $\chi^2$ distribution the resulting p-values are termed asymptotic p-values. Figure 1.1 shows the $\chi^2$ distribution on 9 degrees of freedom. The shaded region indicates the tail probability for an observed table with test statistic 14.125, giving a p-value of 0.118.

Hence, the procedure to calculate an asymptotic p-value for a goodness-of-fit test of the model of independence for an $r \times c$ table is simple:

1. calculate the fitted values from the row and column margins
2. calculate the test statistic of the observed table
3. calculate the p-value from the $\chi^2$ distribution with the relevant degrees of freedom

Facilities to calculate the asymptotic p-value are available in a number of packages, e.g., Splus, GLIM and StatXact. Tables of the $\chi^2$ distribution exist for manual lookup, for given areas.

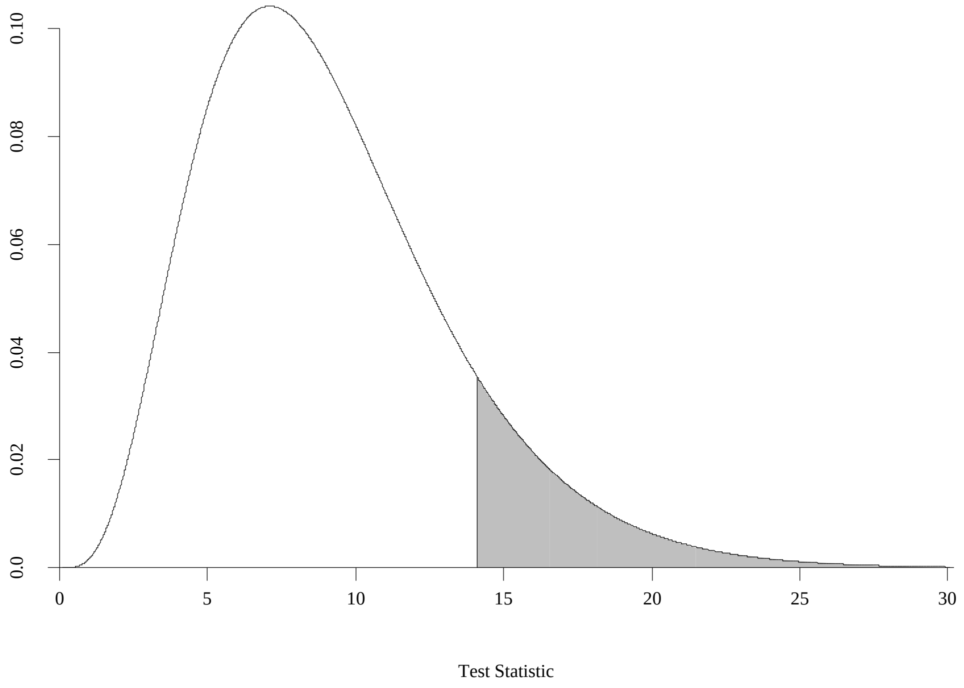


Figure 1.1: $\chi^2$ distribution on 9 degrees of freedom.
shaded region: test statistic $\geq$ 14.125 p-value=0.118

## 1.3 Exact Tests

Using the $\chi^2$ distribution to approximate the exact distribution of the test statistic only holds for large samples. For tables with small cell counts, the exact distribution of the test statistic can sometimes be very different from the asymptotic distribution (Agresti, 1992). Work on when asymptotic approximations become unreliable reveals that there is no hard and fast rule that can be used. In general, the exact distribution of $G^2$ is poorly approximated by the $\chi^2$ distribution when the ratio of number of cells to the table total is less than 5 (Agresti, 1990, pages 246–247). $X^2$ is more robust and expected values of 1 are permissible as long as the expected values for no more than 20% of the cells are no less than 5. A table is defined as sparse if it contains a small count in at least one of its cells.

For sparse tables where the asymptotic approximation is often unreliable, there are two approaches that can be taken. Firstly, adjustments to the test statistics and p-values can be made to cater for the problems of sparseness. The alternative is to work with

| | Guess Poured First | | |
|---|---|---|---|
| Poured First | Milk | Tea | Total |
| Milk | 3 | 1 | 4 |
| Tea | 1 | 3 | 4 |
| Total | 4 | 4 | 8 |

Table 1.2: Fisher's tea tasting experiment.

the exact distribution of the test statistic.

### 1.3.1 Fisher's Tea Tasting Experiment

To introduce the idea of an exact test we describe Fisher's exact test for a $2 \times 2$ table. Fisher described an experiment to test the claim of a woman that she could tell whether a cup of tea was made by putting the milk into the cup first or not. The woman was presented with eight cups of tea, four to which milk was added first and the other four tea first. The woman was told that there were four cups of each type and asked to categorise the eight cups of tea. The results of this experiment are shown in Table 1.2

Under $\mathrm{H}_0$, the probability of a $2 \times 2$ table $x$ with the margins fixed is given by the hypergeometric distribution with density:

$$P(x|x_{++}, x_{1+}, x_{+1}) = \frac{\begin{pmatrix} x_{1+} \\ x_{11} \end{pmatrix}\begin{pmatrix} x_{2+} \\ x_{+1} - x_{11} \end{pmatrix}}{\begin{pmatrix} x_{++} \\ x_{+1} \end{pmatrix}} = \frac{x_{1+}!x_{2+}!x_{+1}!x_{+2}!}{x_{++}!x_{11}!x_{12}!x_{22}!}$$

*Fisher's exact test* for $2 \times 2$ tables is defined as the sum of the table probabilities which are at least as extreme as the observed for all the tables that share the same margin totals. Using the odds ratio ($\theta \geq 1$), this is equivalent to $x_{ij} \geq o_{ij}$, where $o$ is the observed table. The test statistics $X^2$ and $G^2$ have also been mentioned as a method of ordering the tables. For a $2 \times 2$ the ordering given by the table probability, the odds ratio, $X^2$ and $G^2$ are identical (Davis, 1986). As the margins are fixed, as soon as one count in the table is specified, the rest are determined. So we may write the probability of the table as the probability of the count in any single cell, e.g., $x_{11}$. Hence, the probability of the observed table, i.e., $x_{11} = 3$, is 0.229. There is one other table that is more extreme, $x_{11} = 4$, which has a probability of 0.014. Hence, the exact p-value is 0.243, indicating that there is no significant association.

### 1.3.2 Generalising to $r \times c$ Tables

Working with an $r \times c$ table, the exact test can not use the value of the first cell as with the $2 \times 2$ case. Throughout this thesis $X^2$ and $G^2$ are used. Other test statistics could be used to order the tables. Freeman and Halton (1951) used the table probability as the measure of extremeness. The table probability of an $r \times c$ table $x$ under the model of independence is:

$$\frac{\prod x_{i+}! \prod x_{+j}!}{x_{++}! \prod \prod x_{ij}!} \tag{1.2}$$

An exact test for the model of independence for an $r \times c$ table is the sum of the table probabilities of all the tables that have the same marginal totals as the observed table, and which are equally or more extreme than the observed table as measured by some test statistic. Note that under the model of independence both the expected values and the table probability have closed-form expressions (1.1 and 1.2 respectively).

## 1.4 Exact Conditional Inference

Hypothesis tests for log-linear models typically involve a subset of the parameters, which are termed the interest parameters. The remaining parameters are termed the nuisance parameters. To infer information about the parameters of interest, a test statistic is required whose distribution ideally depends on the interest parameters, and is independent of the nuisance parameters. This separation can be achieved by conditioning on the sufficient statistics for the nuisance parameters, in effect eliminating them from the test. Typically one does not need the complete set of observations to describe a parameter. Instead a summary of the data can be used. This summary is referred to as the sufficient statistics.

For example, the model of independence for an $r \times c$ table has interest parameters corresponding to the interaction parameters between the two variables, and the nuisance parameters are the marginal probabilities of the two variables. Hence, to make an inference about the interaction between the two variables, we condition on the margins of the table. This leads to the process of looking at the distribution of the test statistic for all tables with the same marginal totals.

Other more complex models may involve other parameters and sufficient statistics.

## 1.5 Computational Approaches

This section discusses the computational aspects of performing exact tests. The current state of the work in this area falls into two categories, that of the calculation of the exact p-value via complete enumeration of the reference set and the estimation of the exact p-value via simulation methods. The section focuses on the model of independence, but other more complex models are also considered.

### 1.5.1 Complete Enumeration

The method of complete enumeration finds all the tables which have the same row and column margins as the observed table. This set of tables is called the reference set. The test statistic is applied to each enumerated table and the table probability calculated. There are a number of papers on this method, all of which quickly establish the limitations due to the size of the reference set. Even for small tables with modest cell counts, the number of tables in the reference set can be very large. Increasing the table size or sample size increases the size of the reference set at an exponential rate. This section looks at some of the work on complete enumeration and looks at the significant development of the Network Algorithm.

An early paper on the subject entitled "Occupancy of a rectangular array" (Boulton & Wallace, 1973) looked at the simpler problem of finding the size of the reference set. The paper starts by looking at the simple case of a $2 \times 2$ table. Then the $2 \times c$ and $r \times c$ cases are developed by a series of decompositions of the table into a number of

$2 \times 2$ tables. This yields a recursive algorithm to calculate exhaustively the number of tables. Although the algorithm presented (coded in Algol) only counts the number of tables, it can easily be extended to enumerate tables. The disadvantage of this method of calculating the number of tables is that it effectively traverses the search space. Alternatively, the cardinality of the reference set can be estimated (Good, 1976; Gail & Mantel, 1977). Recently, thesis work (Mount, 1995), established two new methods to calculate the number of tables. One method uses a Monte Carlo technique to estimate the number of tables and the other calculates the number of tables exactly. Knowledge of the size of the reference set enables the analyst to determine whether the method of complete enumerations is feasible. For example, enumerating $10^9$ tables is feasible, but enumerating $10^{90}$ tables is not.

There has been one other instance of a recursive technique for enumerating tables. Balmer (1988) adopts a similar method of enumerating tables as Patefield (1981) uses to generate random tables. The recursion operates at a cell-by-cell level, finding the minimum and maximum possible values using the constraints from Patefield. The routine constructs a tree-like representation of the reference set. Whilst this presents some economies in memory usage and maps naturally to the way in which tables are enumerated, the overheads of creating and managing the tree are quite high. The use of recursion seems unpopular in the literature, perhaps pragmatically due to Fortran's poor support for recursion as Verbeek and Kroonenberg (1985) point out.

The remaining algorithms for complete enumeration use some form of looping construct to fill in the tables. Generally, the approach is to cycle each cell from a lower-bound to an upper-bound, and for each iteration cycling the remaining cells in the table. One of the earliest papers to promote this algorithm (March, 1972) used bounds on the looping that lead to over enumeration, in that the set of tables that is generated extends further than the reference set and tables have to be checked to see if they fall in the reference set required. The bounds on a cell can be calculated (Boulton & Wallace, 1973; Patefield, 1981) and this leads to more efficient enumeration (Boulton, 1974).

One of the major problems with implementing the cell looping approach, is that the tables can be of varying size, hence the need for a dynamic number of for-loops. A number of the papers reference an algorithm (Gentleman, 1975) that simulates nested Fortran `DO` loops. In their survey paper, Verbeek and Kroonenberg (1985) present one such algorithm, as well as surveying the state of the work at the time. The paper provides a good source of numerical and algorithmic tricks used by authors from the relevant literature.

One such trick is to avoid enumerating every table in the reference set. Parts of the search space can be avoided if it is known that all the tables in a subspace do not contribute to the p-value. This can yield quite significant reductions in run-time. The technique is used extensively by the network algorithm in the following section as well as in Chapter 2. Pagano and Taylor-Halvorsen (1981) employ this technique. Using the table probability as the measure of extremeness, their algorithm stops generating tables when the probability of a region is known to be greater than the observed. With careful ordering, and a slightly different method of looping the cell counts, the algorithm refines the bounds of the region reducing the amount of enumeration that needs to take place.

Saunders (1984) developed an improvement to enumeration techniques when there are repeated row or column totals. Here, two or more rows that share the same total can be permuted. Hence, the possible rows need only be enumerated once, and appropriate adjustments to the table probability are made, resulting in a considerable saving in enumeration time.

### Network Algorithm

The development of the network algorithm is reported in two papers. Mehta & Patel (1980) looked at the case of $2 \times k$ tables and Mehta & Patel (1983) generalised this to $r \times c$ tables. Their algorithms have been applied to a number of other problems such as $2 \times r \times c$ tables (Mehta *et al.*, 1991) and $r \times c$ tables with ordered categories (Agresti *et al.*, 1990). The algorithm is available in the popular software package StatXact by Cytel Software.

The algorithm firstly constructs a network representation of the tables in the reference set. For a $r \times c$ table the graph consists of $c + 1$ stages, each stage containing a number

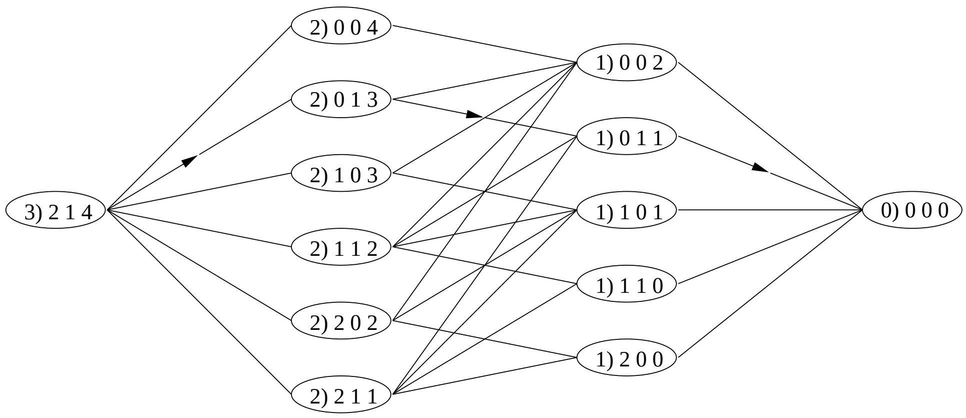


Figure 1.2: Network representation of a $3 \times 3$ contingency table. row margin 2, 1, 4 and column margin 3, 2, 2. The path marked with arrows represents the table with columns 2 0 1, 0 0 2 and 0 1 1.

of nodes. Nodes are labelled with the stage $k$ and a tuple $R_{1k}, \ldots, R_{rk}$. The tuple at stage $k$ represents a possible partial sum of the preceding $c - k$ columns. Arcs connect nodes in the $k$th stage to nodes in the $(k-1)$th stage. The network can be constructed recursively by considering all the possible nodes that precede each of the nodes in the previous stage and connecting the arcs accordingly. Each arc in the network represents a possible column, the cell counts of which are given by the difference of the tuples of the two nodes the arc connects. A path through the network represents a possible table in the reference set. Figure 1.2 shows an example network for tables with a row margin of 2, 1, 4 and a column margin of 3, 2, 2. The path that connects nodes 3) 2 1 4, $\rightarrow$ 2) 0 1 3, $\rightarrow$ 1) 0 1 1, $\rightarrow$ 0) 0 0 0 represents the table with columns 2 0 1, 0 0 2, and 0 1 1.

Considering all the possible paths in turn and calculating the relevant test statistic and probability is equivalent to complete enumeration. The network may be pruned by removing nodes that cannot contribute to the p-value or removing nodes that contribute fully to the p-value. If at a given node the shortest path through the node to the terminal node (length being determined by the test statistic) is greater than the test statistic of the observed table, then all paths that pass through that node are more extreme. Hence, these paths need not be fully enumerated and the node can be removed, making appropriate adjustments to the p-value. Similarly, if the longest path to the terminating node is smaller than the observed, then none of the paths can contribute to the test statistic and the node can be completely removed from the network. If neither of these conditions hold then the pruning process proceeds to the nodes in the following stage.

One difficulty with the network algorithm is computing accurate bounds for the shortest and longest paths to the terminal node. In their papers, Mehta and Patel present the bounds on using the table probability as the test statistic. Here, the structure of the network and the relation to the test statistics results in efficient bounds. Bound estimates for $X^2$ and $G^2$ are not presented and it is not clear whether these exist. In the absence of efficient bounds checking, a backward pass through the network yields the longest and shortest path for each node in the network. In two follow-up papers, Joe (Joe, 1985) presented extensions to the network algorithm that find minimal and maximal tables, to provide the exact values of the longest and shortest paths instead of estimates. This leads to better performance (Joe, 1988), especially in the case of uneven margins when the estimates performed badly.

One major advantage of the network algorithm over other approaches such as complete enumeration is that it significantly extends the range of problems that can be tested. This is especially true for tables with non-uniform margins. The algorithm is ultimately limited by the memory requirement of storing the entire network. Chapter 2 looks at an example where StatXact is unable to compute the p-value due to the size of the problem but methods developed in Chapter 2 are able to completely enumerate the reference set.

### 1.5.2 Simulation

Although the network algorithm significantly extends the range of tables that can be analysed with exact tests, it is still computationally infeasible to use such methods on tables with larger numbers of rows and columns. In these larger tables where there still are a number of small cell counts, the use of asymptotics is questionable. An alternative approach to obtaining an exact p-value is to estimate it using a sampling technique. So called Monte Carlo sampling techniques repeatedly simulate tables from the exact distribution in order to estimate the exact p-value. (Agresti *et al.*, 1979; Boyett, 1979; Patefield, 1981). For the model of independence this involves sampling from the multivariate hypergeometric distribution. A number of algorithms for sampling from the multivariate hypergeometric distribution exist.

Patefield's algorithm is a more efficient algorithm than that used in Agresti *et al.* (1979) and presented by Boyett (1979). The algorithm works by proceeding through the table an cell at a time, in a row-wise fashion. For each cell, a $2 \times 2$ table is formed by collapsing the remaining part of the table, such that the cell in question is the top-left cell. The margins of this table are found by collapsing various quantities in the table taking into account the previously generated cells. These margins are then used to obtain the correct hypergeometric distribution for the cell in question. This approach generates tables from the correct distribution, so the exact p-value can be calculated by taking the ratio of the number of more extreme tables to the number of tables generated in total.

The process of sampling a table takes longer than the generation of a single table by complete enumeration. This is not a significant disadvantage since typically many fewer tables need to be generated. The number of tables that need to be generated is usually dictated by the desired accuracy for the estimated exact p-value result. For example, with 17,000 tables the estimated exact p-value is correct with 0.01 with 99% confidence (Agresti *et al.*, 1979).

Importance sampling may also be used to estimate the exact p-value (Mehta *et al.*, 1988). Instead of generating tables from the correct hypergeometric distribution, tables are sampled from a distribution based on their importance at reducing the variance of the estimated exact p-value. This dramatically reduces the number of tables that need to be generated for a given level of accuracy. Their algorithm utilised the structure from the network algorithm to represent the reference set. This results in the algorithm being memory intensive and inefficient for very large data sets. More recently, Mehta *et al.* have looked at the sensitivity of Monte Carlo techniques to the random number generator being used. In these situations reproducibility of results is important. The guarantee of reproducibility can be achieved by calculating the result to be equal to the exact value to, for example, the first three decimal places. This level of accuracy requires a much larger number of tables to be generated, resulting in much longer computations. The accuracy of the estimated exact p-value can be achieved to a high level with the use of control variates (Senchaudhuri *et al.*, 1995). This results in a significant reduction in the number of tables that need to be generated, in one example, 100,000 tables instead of 74,700,000.

## 1.6 Complex Models

So far this chapter has only looked at 2-way tables and the test of independence. There are a much wider range of models that researchers use.

Firstly, datasets may consist of more than 2 dimensions. For a 2-way table, independence is equivalent to "no 2-way interaction". In a $n$-way table there can be interaction between all the variables, in which case the test is for "no $n$-way interaction," or perhaps interaction between pairs or groups of variables. A number of papers have suggested that their enumeration algorithm generalises to the "no $n$-way interaction" case but there has not been any significant work in this area or on less constrained models. One special case is $2 \times 2 \times K$ stratified data, which often arises in medical statistics. Tests such as the Cochran-Mantel-Haenszel test and Homogeneity of Odds Ratios can be applied. Adaptions to the network algorithm and Monte Carlo simulation have looked at the case of $2 \times J \times K$ tables (Senchaudhuri *et al.*, 1995; Mehta *et al.*, 1991).

In the more general case of $I \times J \times K$ tables, there has been limited work on enumeration. Morgan and Blumenstein (1991) looked at exact tests for hierarchical models. Their algorithm, using a complete enumeration technique, allows the testing of a "full" model against a "reduced" model for multi-dimensional tables.

In many studies, where both cross-classified variables have identical categories, behavioural and social processes cause most of the observations to lie on the main diagonal. This often results in some small off-diagonal counts. In this cases, the model of independence is often implausible and attention is turned to the off-diagonal cells. The model of quasi-independence can be used to analyse tables where there are structural zeros in the table (Mantel, 1970; Goodman, 1968) or when the interest focuses on part of the table, e.g., the off-diagonal cells. Structural zeros occur when a certain cell in the table is impossible. For example, a study looking at cancer and gender may cross classify gender with different types of cancer. In this table, the cell indicating male and ovarian cancer is impossible and would result in a structural zero. Pagano and Taylor-Halvorsen (1981) suggested that enumeration of tables with structural zeros was possible with their algorithm. More recently, changes to Patefield's algorithm allows the exact p-value to be estimated by Monte Carlo simulation (Smith & McDonald, 1995; McDonald & Smith, 1995).

There are a number of other models that can be applied to square contingency tables. An important class of models are those that involve symmetry. Symmetry involves the counts in diagonally opposite pairs of cells, i.e., cell $i, j$ paired with cell $j, i$ $(i \neq j)$. The model of symmetry assumes that the probability of the two cells in each pair are the same, i.e., $\pi_{ij} = \pi_{ji}$ for all $i \neq j$ . This leads to a highly structured model that rarely fits. For example, if the margins are heterogeneous, symmetry cannot fit. A less restrictive model is that of quasi-symmetry. The model of quasi-symmetry involves constraints on the row and column margins, a fixed diagonal and the sum of each pair in the table. In the example of inter-ethnic marriages, discussed in chapter 4, husband's ethnicity is cross-classified against wife's ethnicity. The model of quasi-symmetry assumes that given an inter-ethnic marriage there are no gender differences in the affinity of one ethnic group to another; i.e. a marriage between a man in one ethnic group X to a woman in another ethnic group Y is as equally likely as a marriage between in a woman in group X and a man in group Y. Symmetry models also exist in higher dimensional tables which are still "square". In these cases, the notion of a pair of cells is extended to a group of cells, which have the same permutation of indices.

Another important class of models is one in which there is some form of ordering of the categories. In these cases, a score can often be associated with each category of the variable. Hence, the score for a particular cell is given by the product of the scores of the corresponding row and column categories. The model of linear-by-linear association adds the constraint that the sum, across all the cells, of the cell count times the score, is the same as the observed. If the scores are equally spaced, then the model is called the model of uniform association.

The current state of performing exact tests on such models is patchy. StatXact performs a number of tests associated with the linear-by-linear models using the Kruskal-Wallis test as an alternative hypothesis. A number of algorithms have been developed that use complete enumeration for specific models (Bedrick & Hill, 1990; Morgan & Blumenstein, 1991). Smith *et al.* (1996b; 1996a) use a Markov chain Monte Carlo method with Metropolis-Hastings and Gibbs sampling techniques to estimate the exact p-values for quasi-independence, quasi-symmetry, and uniform association as well as logistic regression (McDonald *et al.*, 1996).

## 1.7 Resources

Throughout the thesis, repeated reference is made to two types of resources, namely computers and example tables.

### 1.7.1 Computers

Three types of computers are routinely used in this thesis. This section aims to outline the features and performance of each of the computers. These computers all use a Unix variant as their operating system. In most cases, the reference to the type of computer is in the context of speed, and the operating system used is not critical in this application. Notably, equivalent performance is expected for modern PC operating systems such as Windows 95 and Windows NT. The SPEC 95 benchmarks are a useful test to measure overall processor power, based on a number of mathematical and engineering benchmarks. The rating consists of two values, SPECint95 and SPECfp95, for predominantly integer and floating-point applications respectively. Whilst these ratings can give an idea of relative performance, other issues affect the performance, such as what compiler is used. In addition, the nature of the algorithms and implementations discussed in this thesis may favour different aspects of the CPU, such as integer performance.

#### Sun SPARCstation 10 Model 31

The Sun SPARCstation 10 Model 31 contains a 33MHz SuperSPARC processor and runs the Solaris operating system, a Unix variant (System V). This machine is used as the baseline for the SPEC95 ratings and scores 1.0 for both ratings.

#### 150 MHz Pentium PC

A typical desktop PC, with 150 MHz Pentium processor, 16K primary cache and 256K secondary cache. This computer scores SPECint95 and SPECfp95 ratings of 4.35 and 3.11. This machine is representative of the current level of performance of a typical new desktop machine and runs the Linux operating system, which is a free implementation of Unix.

#### IBM SP2

The IBM SP2 is a multiprocessor computer consisting of a number of nodes. Each node consists of what is effectively a workstation. The nodes are connected together by a high-performance switch. The configuration of the SP2 used consists of 22 nodes each with a 66MHz POWER2 processor. 16 of the nodes have a 32K instruction cache and 64K data cache and the remaining 6 have 32K instruction and 128K data caches. The ratings of an individual node are 3.23 and 9.33 for the integer and floating point tests, respectively. These figures are based on a node with the 128K data cache. Hence, the performance for the nodes with smaller data caches maybe lower. Each node runs IBM's Unix implementation AIX.

### 1.7.2 Example Tables

Figure 1.3 shows five tables that re-occur throughout the thesis. Also included in the figure are the number of tables present in the reference set for each table based on the model of independence. These examples are selected for the range of sizes of the reference sets, which are indicated by each table. In addition, these examples are sparse

Example A
728

| | | | |
|---|---|---|---|
| 10 | 1 | 6 | 17 |
| 3 | 5 | 0 | 8 |
| 5 | 0 | 1 | 6 |
| 18 | 6 | 7 | 31 |

Example B
28,019

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 3 | 2 | 7 |
| 2 | 0 | 3 | 1 | 6 |
| 0 | 0 | 0 | 5 | 5 |
| 0 | 2 | 1 | 2 | 5 |
| 3 | 3 | 7 | 10 | 23 |

Example C
3,187,528

| | | | | | |
|---|---|---|---|---|---|
| 2 | 0 | 1 | 2 | 6 | 11 |
| 1 | 3 | 1 | 1 | 1 | 7 |
| 1 | 0 | 3 | 1 | 0 | 5 |
| 1 | 2 | 1 | 2 | 0 | 6 |
| 5 | 5 | 6 | 6 | 7 | 29 |

Example D
12,798,781

| | | | | |
|---|---|---|---|---|
| 1 | 5 | 3 | 3 | 12 |
| 1 | 6 | 6 | 4 | 17 |
| 5 | 7 | 1 | 1 | 14 |
| 1 | 9 | 2 | 1 | 13 |
| 8 | 27 | 12 | 9 | 56 |

Figure 1.3: Example 2-way tables for models of independence.

and prone to the bad asymptotic p-value approximations. As a result, these examples occur frequently in the literature. Examples A and D appeared in Agresti *et al.* (1979), with D appearing originally in Klotz and Teng (1977). Example C appears in Mehta and Patel (1983).

## 1.8 Timings

Throughout this thesis, timings for the duration of computations are quoted. They are often used to demonstrate benefits of optimisations as well as relative performance of machines, and the size of examples. The timings fall into two categories; those performed on single computers and those performed on parallel machines or multiple machines in a distributed system.

In the case of timings on a single computer, the values obtained are the number of CPU seconds that the program took to run. This is different to the time the program actually took to run in real terms. All the machines used run a variant of Unix which is a multi-user and multi-process operating system. This means that a running program may have to compete with other users and programs for the CPU. Hence, using the wall-clock time or real time may not give a true indication of the time a program takes if the system is busy. By quoting the CPU time, these influences are factored out. If the program has total access to the machine, the CPU time quoted is the same as the real time.

For timings of distributed programs the wall clock time is used. Calculating the total CPU usage of a program that is run across a number of machines is not straightforward. The major reason for this is that the delays in the system due to communication are not present in the calculation of CPU time. Hence, the timings used in this thesis are based on the real time that a computation took to complete. Two types of machines were used to run the distributed programs; the SP2 and a group of Desktop PCs. In the case of the SP2, the distributed program is given sole use of the processors that are allocated to it. For the group of desktop PCs, the issues, mentioned above, of how busy a computer are important. Only idle computers were selected for these tests.

## 1.9 Outline of the Thesis

The aim of the thesis is to develop algorithms for the calculation of exact p-values via the complete enumeration. Chapter 2 looks at a basic algorithm for enumerating tables under the model of independence. The chapter identifies a number of weaknesses and inefficiencies in the proposed approach, some of which can be eliminated by restructuring the algorithm. Chapter 3 takes the basic algorithm and extends it to enable the enumeration of 2-way tables under the more complex models of quasi-independence and quasi-symmetry. Chapter 4 looks at a more generic approach to enumerating tables, enabling a model to be specified with a model matrix. The resulting algorithm is able to enumerate a far wider range of models, including models for multi-way tables. The final two chapters address the serious computational requirements of the algorithms developed in the thesis. The notion of distributed computing is introduced, and a system enabling exact p-values to be calculated on a number of machines connected together by a network is developed. Throughout the thesis, a number of example datasets are used to demonstrate the algorithms and provide timings for comparison.

# Chapter 2

# Models of Independence

## 2.1 Introduction

Chapter 1 discussed the current state of the computational aspects of exact tests. This chapter looks at the design and implementation of algorithms to calculate exact p-values for $r \times c$ contingency tables under the model of independence, using complete enumeration.

## 2.2 A Recursive Algorithm

The problem is how to calculate an exact p-value for a given observed $r \times c$ table. This calculation is performed by enumerating all tables that have the same marginal counts as the observed table. For each table, a test statistic is calculated and compared with the observed test statistic. If this indicates that the enumerated table is at least as extreme as the observed table, then the probability of the table is found and added to the running total of the p-value. Typically, Pearson goodness-of-fit and likelihood ratio test statistics are use to measure extremeness.

The main motivation for a recursive solution to the problem is due to the homogeneous nature of the table data and its structure. The process of removing a row from a table, results in another table. Hence, a table can be generated by building up rows, subject to the row and column margin constraints. For independence the constraints on a row are derived from the relevant row and column margins. The row should sum to a given total from the row margins and no element should be greater than its column margin total. Alternatively, the table could equally be built up with columns instead of rows.

The enumeration algorithm should be able to handle arbitrary sized tables, so simply having a fixed number of nested loops depending on the table size is not sufficient. Verbeek and Kroonenberg (1985) cite Gentleman's algorithm (Gentleman, 1975), which simulates dynamic nested loops, as an aid to solving the problem. When enumerating a table, given the margins of the observed table, the constraints that operate on the first row can be extracted. Then, all the rows that satisfy the given constraints can be found. For each row in this list, there is a subtable with adjusted marginal totals. The marginal totals of the subtable are obtained from the rest of the row totals and the column totals minus the generated row counts. The set of tables satisfying the subtable margins can be found by simply calling the table enumeration function recursively. The original row can then be prepended to each subtable in the resulting list, forming a new list of tables satisfying the given marginal totals. Every recursive routine has to have a terminating case, which in this case is when the table to be generated only has one row. Here, the row is given by the remaining column totals.

The basis of the recursive enumeration algorithm can be summarised as: 1) generate all the possible first rows given the current constraints, 2) for each of these rows, adjust the constraints and generate all the subtables under those constraints, 3) for each of the generated subtables, prepend the relevant first row. Figure 2.1 shows this process for a $4 \times 4$ table, where $r_1, r_2, r_3, r_4$ are the row margins, $c_1, c_2, c_3, c_4$ are the column margins and $c'_1, c'_2, c'_3, c'_4$ are the adjusted column margins.

This leaves the problem of generating all the possible sequences of elements, whose total is a set value (from the row margin) and where each individual element has a maximum value (obtained from the column margins). There are a number of solutions to this problem but the basis of all of them is to cycle each cell from zero to the maximum permissible value. This value is given by the minimum of the row margin and the relevant column margin. In a similar way that a row is fixed, and the subtables of that

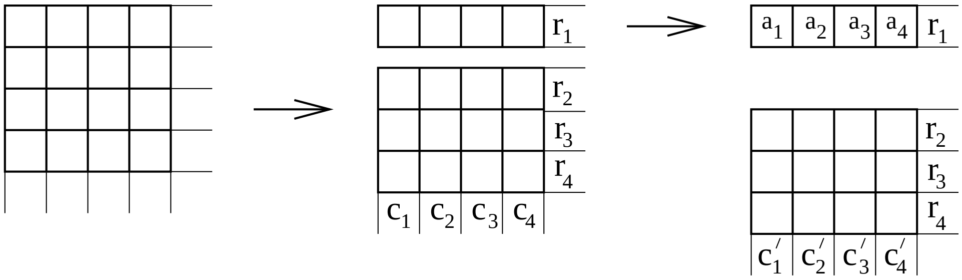


Figure 2.1: The recursive step.

row are generated, a cell can be fixed and the rest of the row generated, giving a list of possible subrows.

This recursive algorithm is similar to that proposed by Boulton and Wallace (1973), which looks at counting the number of tables with fixed margin totals. They decompose the table into $r \times 2$ tables, which are then reduced to $2 \times 3$ and $2 \times 2$ tables. This decomposition allows them to calculate the ranges of elements, and hence the number of tables. This level of decomposition is not needed for enumeration, since it adds extra overheads to row generation. Balmer (1988) used a recursive algorithm at the cell-by-cell level incurring similar problems.

## 2.3 Implementation Issues

Implementing the above algorithm presents a number of problems. With modern programming languages the issues about writing recursive programs raised in Verbeek and Kroonenberg (1985) are no longer problematic. Most modern languages have no problems with recursion and modern compilers and processors run these codes efficiently.

One of the major issues with implementing the recursive algorithm is the sheer number of rows that can exist in the reference set. Hence, an algorithm that generates all possible rows at a given stage, and then for each of these rows, investigates the subtables, will have to store a large number of rows. Given the large number of tables that can be present in the reference set, storing all the tables is not a viable approach.

There are a number of requirements on the language chosen to implement the algorithm. Efficient support for recursion is obviously paramount. In addition, the language should have support for managing structures such as rows and tables. A Lisp language was chosen as the initial prototype language and a subsequent version was written in C. The following two sections discuss the approaches used.

### 2.3.1 Lisp

This section details the implementation of the recursive algorithm in Lisp. Specifically, the implementation was done in Scheme. Scheme (Dybvig, 1987) is a simple dialect of Lisp that has cleaner semantics. Scheme and Lisp are powerful languages which are particularly suited as an interactive development language with good environments. Extensive use was made of a compiler that turns Scheme code into C code, which can then be compiled by a C compiler. This leads to a massive speed increase over interpreted code, but the performance of compiled Scheme code does not reach that of hand coded C code. For speed critical parts of the program, C code can be written and called from the Scheme code using the foreign function interface. In the implementation, all of the statistical routines were written in C. At the time of implementation, the Lisp based statistical package Xlispstat (Tierney, 1990) did not have a compiler or an extensive and easy to use foreign function interface. These facilities have been subsequently developed in Xlispstat.

Given the recursive nature of Scheme, the recursive algorithm can be easily specified. Tables are stored as lists of rows, and a row as a list of elements. This is not a very memory efficient method of storing tables, but it makes handling and construction of rows and tables much easier.

The core of the enumeration algorithm is as follows:

```
(define (enumerate row_margin column_margin)
  (if (final? row_margin))
    (lastrow (head colmargin) rowmargin)
    (flatten
      (map (lambda(x)
             (appendsubtable x (enumerate (tail row_margin)
```

```
                                                (submargin column_margin x))))
          (permute_row (head row_margin) column_margin)))))
```

The function `permute_row` takes a list of column totals and a row total, and returns all the rows with the given total, that are within the bounds imposed by the column totals. The Lisp function `map` takes a function and a list as arguments and applies the function to every element in the list. In this case the function is to call `appendsubtable` with the row given and a list of all the subtables for that row, given by the recursive call to `enumerate`. The result of the `map` function is a list of lists of valid tables. Since the result of the `enumerate` function needs to be a list of tables, the `flatten` function takes the list of lists of tables and appends them into a single list. The recursion is terminated by the `if` test to check for the last row.

The issue outlined earlier in this section concerning storing all the tables is a particular problem, since all the routines generate lists of objects, such as rows and tables. Initially, the algorithm had been written using straightforward list operations that meant that all the tables and rows were stored in memory. This works fine for small tables, but with larger tables, the memory usage became too high. A solution to this problem was to replace the use of lists with streams (see below).

If the actual tables in the reference set are not required by the user, then the passing around of the tables is not required. Instead, the calculation of each generated row's contribution to the test statistic, and table probability can be passed around. This may help to relieve the memory problem, but it would still generate a tuple for each table in the generated set, and hence the use of streams is still required.

### Streams

Streams allow the specification and operation on very large data structures whilst storing only a relatively small amount of data. This demonstrates a very powerful feature of Lisp languages where it is possible to handle code as data.

In Lisp, lists consist of a chain of nodes. Each node, or pair, has a head and a tail, called the `car` and the `cdr`. Pairs can be built up into complex structures, but for a list, the `cdr` of one node points to the next node in the chain. The `car` points to an element of data. A stream similarly has a head and tail, the difference being that the tail consists of a function, which when evaluated gives another stream, i.e., the next node in the chain and the function for the tail. In effect streams store how to make the data as opposed to the data itself.

For example, the definition of an infinite stream of the integer 1 follows:

```
(define ones (cons-stream 1 ones))
```

`Cons-stream`, like the equivalent list operator `cons` is a constructor that creates a stream from its arguments. Note that the second argument is the tail of the stream, which has to be evaluated later. Usually this argument would be evaluated, and the result passed to the function `cons-stream` but `cons-stream` has a special form which delays this process until needed. This delay process can avoid performing redundant computation whose results are never used or needed.

Plugging streams together can create more complex streams. For example, the definition of a function to add two streams of integers together, can be used in conjunction with the streams of ones to create an infinite stream of the positive integers:

```
(define (add-streams s1 s2)
  (cons-stream (+ (head s1)(head s2))
           (add-streams (tail s1)(tail s2))))

(define integers
  (cons-stream 1 (lambda()(add-streams ones integers))))
```

For a good introduction to streams and more examples see the book "Structure and Interpretation of Computer Programs" (Abelson *et al.*, 1985).

### Applying Streams

To use streams in the recursive enumeration code, all the list operations can be replaced with the equivalent stream operations. The result is a stream that can be stepped through, the head of each successive stream returned being the next table generated. Hence, a table can be pulled from the front of the stream and the required statistical tests performed on it. At each step, the routine has a pointer to the current table, and a pointer to the remaining computation. This also means that the algorithm can stop without enumerating all the tables. For example, if the p-value reaches 0.1, then the generation of tables can be terminated since the null hypothesis would not be rejected for p-values larger than 0.1. If the scheme system supports it, the tail of the stream, i.e., the rest of the computation, can be migrated to another machine to distribute the computation.

There is a performance penalty for using streams, compared with using straightforward list operations. Given the restrictions on memory, this is obviously less of an issue. There is a trade off between using the list operations for speed and the stream operations for memory conservation. It was found that it was better to use normal list routines in the generation of rows, since these routines are called so often, and the number of possible rows at a given stage is small enough to be stored in memory.

In this application, the speed of the Scheme implementation is much slower than hand written routines in C. This is due to handling of lists of objects that represent rows and tables. This is an issue for a practical usable system, but for prototyping the algorithm it provides a powerful mechanism.

## 2.3.2 C

This section looks at the approaches used in the hand written re-implementation of the recursive algorithm in C.

In the Lisp version the problem of large memory requirements of storing data was solved by using streams to describe the next step of the computation. These powerful structures are not available to the C programmer, although it is possible to emulate such behaviour. Since this approach had already been taken with the Lisp version, this area was not explored. Also, one of the requirements of the C version was to make it as fast as possible within the bounds of the algorithm. The added overhead of implementing a mechanism to provide stream-like features is too high.

A solution was simply to embed the state of the computation within the nested recursive procedures in the normal way. Hence, the basis of the algorithm is to generate another valid row and make the recursive call to generate the subtables. For each call, the row number to operate on is the only information needed. If the final row has been reached, then the algorithm calls a function to perform the various statistical tests. This function is supplied elsewhere by the caller of the enumeration routine. This function is called at the leaves of the search tree, i.e., each possible table. Conversely, in the Lisp implementation, this function appears at the outermost loop, as the stream of tables is stepped through, since the computation is being "pulled" in a producer-consumer manner. The C approach means that the variables used to accumulate the p-values have to be global, since they are used at the depths of the recursion, for each generated table, as well as when all the tables have been generated and the program reports the various values. Whilst this is not a problem, the untidy global variables hinder modularization. In the final implementation the generation algorithm is in its own module, and the parameters passed to the enumeration function are simply the row and column margins, and a pointer to the function to call for each generated table. The user of the enumeration module then supplies the relevant statistical tests and global state.

### Row Generation

Since at each stage in the computation, a row is generated and the recursive call is made, there is a certain amount of state that needs to be stored. This is so that the next row in the lexicographical order can be generated. A number of strategies could be used to enumerate the rows, each with different characteristics as far as the required state storage is concerned. The row generation algorithm outlined below was chosen because it requires only the previous row generated to generate the next row.

The row generation algorithm is as follows. A variable is maintained, containing the current total "in-hand", representing the count that needs to be deposited on the remaining elements of the row. An additional variable indicates how far along the row the routine is. At each stage, the routine tries to deposit as much of the in-hand total into the current element as the column constraint allows. If this total reaches zero, then the rest of the row can be set to zero. If the end of the row is reached and the in-hand total is not zero, then an invalid row has been generated. On generation of a correct row, the routine makes the recursive call, to enumerate the subtables. When this completes, the routine has to backtrack along the current row, to generate the next row in the lexicographical ordering. The rules for backtracking are to take the value of the last element in the row, and set that to be the in-hand total, then iterate backwards over the row, looking for a nonzero element. Once a nonzero element has been found, its value is decremented and the in-hand total incremented. Decrementing the cell value means that the maximum value constraint is not violated. At this point, the routine returns to the process of proceeding along the row, depositing as much of the in hand total as possible, until the end of the row is reached. The routine terminates generating rows, when the backtracking reaches the beginning of the row.

Figure 2.2 shows the process of generating a row. The row total is 4 and the column constraints are 3, 1 and 3. The arrow on the left indicates which state the algorithm is in, i.e., forward ($\rightarrow$) or backtracking ($\leftarrow$) mode. The circle denotes which is the current cell and the left most value is the in hand total. The ticks and crosses indicate success or failure in generating a row. Note the failure to generate a row, towards the end of the sequence, occurs because the last element can only hold 3 or less. A later section looks at how re-ordering of the columns in a table can help to minimise the amount of failures.

## 2.4 Calculating the Exact P-value

The previous section detailed the algorithm to enumerate tables. This section looks at the issues involved in the computation of the exact p-value. In summary, the exact p-value is the sum of the table probabilities of all the tables in the reference set that

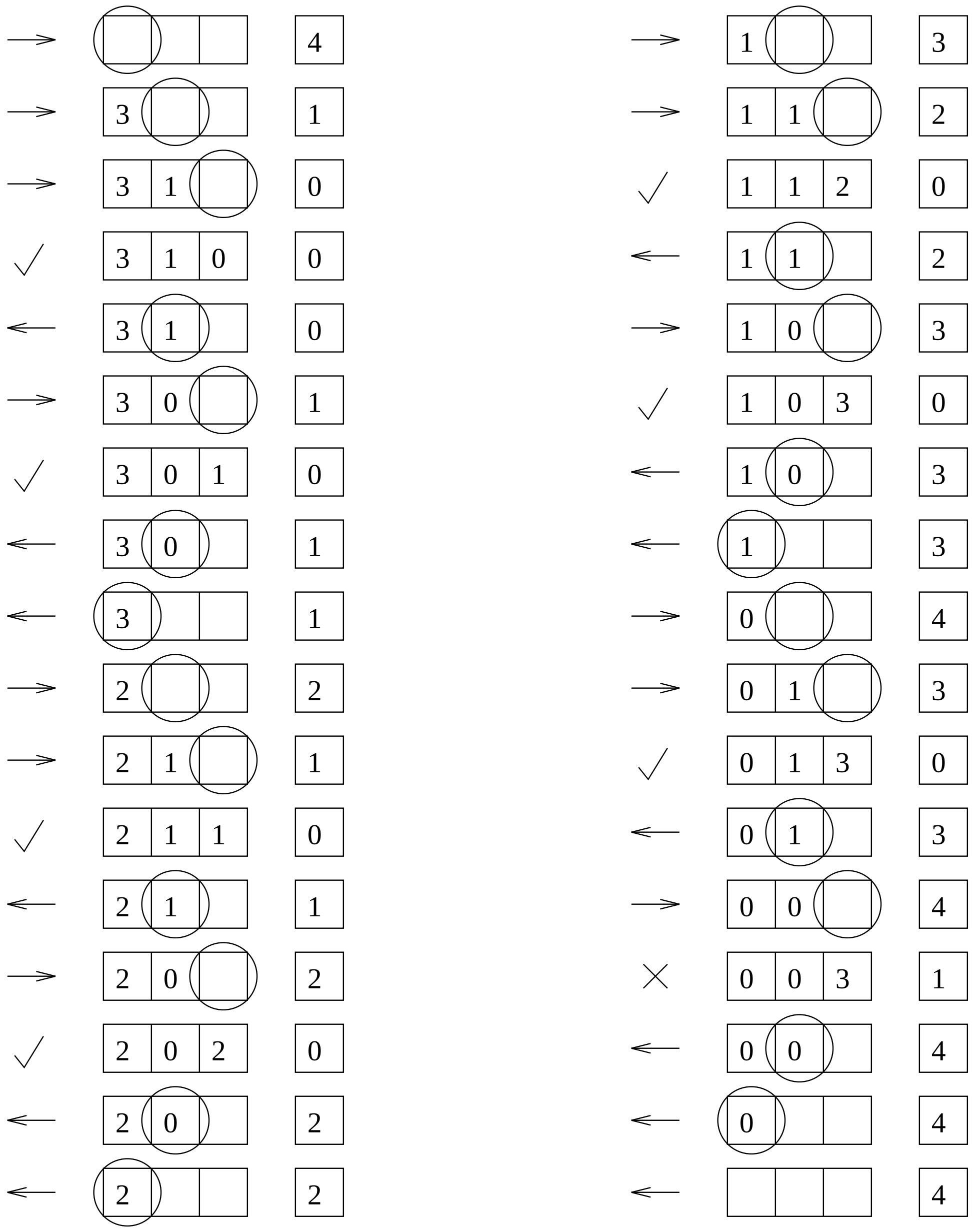


Figure 2.2: Example of row generation.
Row total is 4 and column constraints of 3, 1 and 3.

are equal to or more extreme than the observed table, based on the ordering of the test statistic used. Typically, the Pearson $X^2$ statistic and the log-likelihood-ratio $G^2$ statistic can be used as measures of extremeness.

### 2.4.1 Table Probability

The probability of a table, $x$, under independence is given by:

$$\frac{\prod_i x_{i+}! \prod_j x_{+j}!}{x_{++}! \prod_{ij} x_{ij}!}$$

Due to the factorials, calculating the probability requires the handling of large numbers. A solution to this problem is to calculate the probability by working in logarithms. Hence, the log probability is the sum of the log factorials of the row and column margins, minus the sum of the log factorials of the elements and the table total. The log factorials can easily be stored in a lookup table for speed. Once the log probability of the table is calculated, the table probability is found and added to the relevant p-value accumulators.

### 2.4.2 $X^2$ and $G^2$

The use of the test statistic falls into two areas, as a result presented to the user and as a measure of extremeness. For the user, the full test statistic has to be calculated. For calculating the p-value, the test statistics are used to order the tables, by comparing the test statistic of an enumerated table with that of the observed. Hence, with this in mind, a form of the test statistic that has less numerical operations could be used, as long as it has the same ordering properties.

Note that both $\sum_{ij} x_{ij}$ and $\sum_{ij} e_{ij}$ are $x_{++}$. Using this result, $X^2$ simplifies greatly:

$$\begin{aligned} X^2 &= \sum_{ij} \frac{(x_{ij} - e_{ij})^2}{e_{ij}} \\ &= \sum_{ij} \frac{x_{ij}{}^2}{e_{ij}} - 2x_{ij} + e_{ij} \\ &= (\sum_{ij} \frac{x_{ij}{}^2}{e_{ij}}) - x_{++} \end{aligned}$$

$$X^{2'} = \sum_{ij} \frac{x_{ij}{}^2}{e_{ij}}$$

where $e$ is the table of fitted or expected values and $x$ a valid table.

This reduces the number of operations in the calculation of $X^2$. The calculation of the full test statistic involves the conversion of an integer count to a floating point number, floating point subtraction with the fitted value, a floating point copy and multiplication to square the result, and dividing by the fitted value. The adjusted version requires integer copy and multiplication to square the table count, conversion of an integer to a floating point number, followed by floating point division with the fitted value. Hence, the difference between the old and new forms are a floating point subtraction, copy and multiplication verses an integer copy and multiplication. Although this is not a significant saving on its own, performing this calculation for each element in the table millions of times, the time saved can be appreciable. More importantly, the reduction in the number of operations, especially floating point operations, is important as far as accuracy is concerned.

For $G^2$, the standard form is combined with the closed-form expression for the expected values under the model of independence:

$$\begin{aligned}
G^2 &= 2\sum_{ij} x_{ij}\log(\frac{x_{ij}}{e_{ij}}) \quad \text{where} \quad e_{ij} = \frac{x_{i+}x_{+j}}{x_{++}} \\
G^2 &= 2\sum_{ij} x_{ij}\log(\frac{x_{++}}{x_{i+}x_{+j}}) + 2\sum_{ij} x_{ij}\log x_{ij} \\
&= 2\sum_{ij} x_{ij}\log x_{++} - 2\sum_{ij} x_{ij}\log x_{i+} - 2\sum_{ij} x_{ij}\log x_{+j} + 2\sum_{ij} x_{ij}\log x_{ij} \\
&= 2x_{++}\log x_{++} - 2\sum_{i}\sum_{j} x_{ij}\log x_{i+} - 2\sum_{j}\sum_{i} x_{ij}\log x_{+j} + 2\sum_{ij} x_{ij}\log x_{ij} \\
&= 2x_{++}\log x_{++} - 2\sum_{i} x_{i+}\log x_{i+} - 2\sum_{j} x_{+j}\log x_{+j} + 2\sum_{ij} x_{ij}\log x_{ij} \\
&= k + 2\sum_{ij} x_{ij}\log x_{ij}
\end{aligned}$$

$k$ is derived from the row and column margins and the table total and is, therefore, a known constant. This gives two alternative forms for $G^2$:

$$\begin{aligned}
G^{2'} &= \sum_{ij} x_{ij}\log x_{ij} \quad \text{or} \\
G^{2'} &= \prod_{ij} x_{ij}^{x_{ij}}
\end{aligned}$$

with $0^0 = 1$. Neither form of the test statistics require the fitted values. The second form avoids an expensive log operation, and is entirely integer based, thus avoiding any accuracy problems. The disadvantage of this form is that even for relatively small cell values it leads to large numbers. $23^{23}$ is the largest cell value for a 32-bit integer. For speed, a lookup-table can be used for both forms.

## 2.5 Optimisations

### 2.5.1 Row Generation

One of the significant problems with the row generation algorithm is that invalid rows are generated whilst enumerating. This occurs when the end of the row is reached and the in hand total is nonzero. One of the causes of this is when a cell's value becomes low enough such that the remaining elements in the row are unable to support the row total, due to their column constraints. A solution to this problem is to maintain a vector of maximum values that the remaining cells in the row can support. Hence, if at a given element the in hand total exceeds the maximum value of the remaining elements then the algorithm can immediately backtrack.

### 2.5.2 Target Level

A significant point to note about the recursive algorithm is that under independence every row that is generated forms part of a valid table. This means that for every generated row, a valid subtable exists. The aim of the enumeration is to find the probability of the more extreme tables under the test statistic. Since the test statistics used above are per-element based, a running total of the current test statistic of all the elements can be maintained. If this running total exceeds the target test statistic then all the tables that share the already generated rows will be more extreme (since $(o_{ij} - e_{ij})^2/e_{ij}$ and $o_{ij} \log o_{ij}$ are always greater than or equal to zero). This optimisation appears elsewhere in a number of algorithms (Verbeek & Kroonenberg, 1985) including the shortest path optimisation of the network algorithm (Mehta & Patel, 1983).

Hence, at every point in the enumeration where a row is generated, the contribution to the test statistic of the cells in that row is added to the total test statistic of the cells above the current row. When this running total exceeds the target value, the algorithm need not proceed with generating the relevant subtables. This leaves the problem of finding the probability of a partially generated table.

Consider the following table:

$$
\begin{array}{cccc|c}
x_{11} & x_{12} & \dots & x_{1c} & x_{1+} \\
\vdots & \vdots & & \vdots & \vdots \\
x_{p1} & x_{p2} & \dots & x_{pc} & x_{p+} \\
\vdots & \vdots & & \vdots & \vdots \\
x_{r1} & x_{r2} & \dots & x_{rc} & x_{r+} \\
\hline
x_{+1} & x_{+2} & \dots & x_{+c} & x_{++}
\end{array}
$$

The probability of this table is:

$$
\begin{aligned}
P(x) &= \frac{\prod_{i=1}^{r} x_{i+}! \prod_{j=1}^{c} x_{+j}!}{x_{++}! \prod_{i=1}^{r} \prod_{j=1}^{c} x_{ij}!} \\
&= \frac{\prod_{i=1}^{p} x_{i+}! \prod_{i=p+1}^{r} x_{i+}! \prod_{j=1}^{c} x_{+j}!}{x_{++}! \prod_{i=1}^{p} \prod_{j=1}^{c} x_{ij}! \prod_{i=p+1}^{r} \prod_{j=1}^{c} x_{ij}!} \\
&= \frac{\prod_{i=1}^{p} x_{i+}! \prod_{j=1}^{c} x_{+j}!}{x_{++}! \prod_{i=1}^{p} \prod_{j=1}^{c} x_{ij}!} \frac{\prod_{i=p+1}^{r} x_{i+}!}{\prod_{i=p+1}^{r} \prod_{j=1}^{c} x_{ij}!}
\end{aligned}
$$

The sum of the probabilities of all tables that share the common rows $1, \dots, p$ is:

$$
P(x') = \frac{\prod_{i=1}^{p} x_{i+}! \prod_{j=1}^{c} x_{+j}!}{x_{++}! \prod_{i=1}^{p} \prod_{j=1}^{c} x_{ij}!} \sum \frac{\prod_{i=p+1}^{r} x_{i+}!}{\prod_{i=p+1}^{r} \prod_{j} x_{ij}!} \tag{2.1}
$$

where the summation is over all possible subtables of the rows that are not common (rows $p + 1 \dots r$).

The probability of the subtable of rows $p+1,\ldots,r$ is:

$$P(x_{p+1,\ldots,r}) = \frac{\prod_{i=p+1}^{r} x_{i+}! \prod_{j=1}^{c} \left(x_{+j} - \sum_{i=1}^{p} x_{ij}\right)!}{\left(\sum_{i=p+1}^{r} x_{i+}\right)! \prod_{i=p+1}^{r} \prod_{j=1}^{c} x_{ij}!}$$

The probabilities of all possible subtables sum to 1:

$$\begin{aligned} \frac{\prod_{j=1}^{c} \left(x_{+j} - \sum_{i=1}^{p} x_{ij}\right)!}{\left(\sum_{i=p+1}^{r} x_{i+}\right)!} \sum \frac{\prod_{i=p+1}^{r} x_{i+}!}{\prod_{i=p+1}^{r} \prod_{j=1}^{c} x_{ij}!} &= 1 \\ \sum \frac{\prod_{i=p+1}^{r} x_{i+}!}{\prod_{i=p+1}^{r} \prod_{j=1}^{c} x_{ij}!} &= \frac{\left(\sum_{i=p+1}^{r} x_{i+}\right)!}{\prod_{j=1}^{c} \left(x_{+j} - \sum_{i=1}^{p} x_{ij}\right)!} \qquad (2.2) \end{aligned}$$

Hence, from equations 2.1 and 2.2 the probability of a partially completed table is:

$$P(x') = \frac{\prod_{i=1}^{p} x_{i+}! \prod_{j=1}^{c} x_{+j}!}{x_{++}! \prod_{i=1}^{p} \prod_{j=1}^{c} x_{ij}!} \frac{\left(\sum_{i=p+1}^{r} x_{i+}\right)!}{\prod_{j=1}^{c} \left(x_{+j} - \sum_{i=1}^{p} x_{ij}\right)!} \qquad (2.3)$$

Note that $\sum_{i=p+1}^{r} x_{i+}$ is the sum of the row totals of the ungenerated subtable and $x_{+j} - \sum_{i=1}^{p} x_{ij}$ is the remaining column total of column $j$. Hence, the probability of a partially generated table can be found, by collapsing the rest of the table into one row containing the remaining column totals, with an appropriate row margin, and finding the probability of the table in the usual way.

This means that the p-value can be calculated correctly without having to totally enumerate the reference set. A disadvantage of this optimisation is that the size of the reference set is not found. This is not a significant problem since under independence there are a number of algorithms to calculate the size of the reference set. Indeed, the enumeration of the subtables could proceed, without the calculation of the test statistic, which is generally found to be the most costly part of the p-value calculation.

One of the side-effects of this optimisation is that, since the test statistics are calculated as the enumeration proceeds, there is no need to calculate the test statistic at the end. Previously, calculating the test statistic at the end implied a large amount of repeated

| Optimisation | Example B | | Example C | | Example D | |
|---|---|---|---|---|---|---|
| Off | 0.45 | – | 58.29 | – | 231.41 | – |
| On | 0.44 | 2.22% | 53.85 | 7.61% | 124.05 | 46.39% |
| Set Size | 28,019 | | 3,187,528 | | 12,798,781 | |
| Full | 26,583 | | 2,908,234 | | 6,724,588 | |
| Partial | 185 | | 14,290 | | 64,735 | |

Table 2.1: Example results for target level optimisation.
(Timings in seconds)

calculation, since from one table to the next, there is typically, only a few cell counts that are different. With this optimisation, the repeated work is reduced to just the row level.

Table 2.1 shows the results from turning the target level optimisation on. The *Off* case still has the partial test statistic calculation at row generation time, so the speed-up shows simply the time saved by the reduction in subtable generation. These examples were run on a single SP2 node. The bottom part of the table, shows the number of tables in the reference set, the number of fully completed tables and partially completed tables by the optimisation. This optimisation performs significantly well with Example D where the nearly 50% of the reference set is represented by a relatively small number of partially completed tables. The p-values of this table are 0.1395 and 0.2724 ($X^2$ and $G^2$ respectively). This indicates a higher proportion of more extreme tables than the other two examples, which results in the better performance, due to the target test statistics being exceeded more. Note that with this optimisation, if p-values for multiple test statistics are being calculated, then the target values for all of the test statistics have to be exceeded, before the optimisation takes effect. This requires a reasonable level of agreement between the test statistics for the optimisation to be effective.

### 2.5.3 Repeated Work

Consider the following two partial tables:

| | | | | |
|---|---|---|---|---|
| 1 | 5 | 3 | 3 | 12 |
| 1 | 6 | 6 | 4 | 17 |
| | | | | 14 |
| | | | | 13 |
| 8 | 27 | 12 | 9 | 56 |

| | | | | |
|---|---|---|---|---|
| 2 | 8 | 1 | 1 | 12 |
| 0 | 3 | 8 | 6 | 17 |
| | | | | 14 |
| | | | | 13 |
| 8 | 27 | 12 | 9 | 56 |

The remaining column totals, after the first two rows, are exactly the same for both partially enumerated tables, i.e., (6, 16, 3, 2). This means that the subtables are common, but they will be enumerated for both of these partial tables. This is an example of one of the significant disadvantages of the recursive enumeration algorithm, namely that of repeated work. A previous section looked at repeated work performed in the calculation of the test statistic. This section looks at repeated work due to commonality between tables at the enumeration level.

The problem demonstrated by the above example suggests that there is a better way of splitting up the table. Instead of splitting up the table a row at a time, the whole table could be split into two pieces. This "halving" of the table gives two tables that are approximately equal in size (obviously for an odd number of rows, one table will be larger than the other). Each of the two halves would have a column margin, and these two column margins need to add up to the column margin of the full table. The 2 partial column margins could be thought of as two rows in a $2 \times c$ table. Each possible table, given these margins, will give the column margins for upper and lower subtables. Enumerating these two pairs of margins results in an upper and lower set of tables. Each table in the upper set can be paired with every table in the lower set as shown in Figure 2.3. This means that if all the tables in the lower set can be found and stored, then the repeated work, as shown above, can be eliminated. It should be noted that for tables with larger numbers of rows, there will still be a degree of repeated work. This is because there will still be commonality between subtables at a higher level.

Hence, the algorithm for this method of enumeration would be:

1. if the table only has two rows, then enumerate it,

2. else divide the table into two halves,

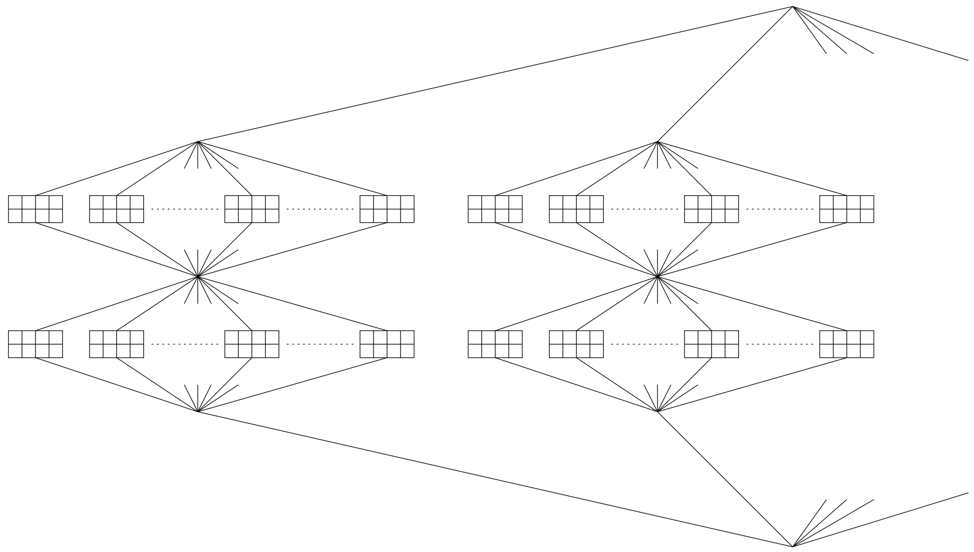

Figure 2.3: An example enumeration tree for table halving algorithm for a $4 \times 4$ table.

3. combine the row margins in the two halves, and enumerate the resulting $2 \times c$ table,

4. for each table, take each of the rows and use as column totals with the relevant row totals,

5. recursively enumerate each half and

6. pair each table in one half with every table in the other half.

A significant problem with this approach is that of memory. At each split in the algorithm, one of the lists of subtables has to be stored. This makes this approach more memory intensive than the previous algorithm.

The idea of grouping rows, such that their effective column totals are the same, leads to the idea of the network algorithm, where nodes in the network are the remaining column totals, and connections between nodes form rows. The network algorithm also has a similar problem of memory usage, yet it is much faster than the complete enumeration approaches. There is a trade off between the amount of memory used and the speed of the algorithm. One of the differences between the network algorithm and the above algorithm is that the network algorithm generates the whole of its search tree, whereas the proposed algorithm can create a part of its search tree at a time, and hence limit

| Method | Example A | Example B | Example C | Example D | Example 2.7 |
|---|---|---|---|---|---|
| Original | 0.13 | 0.45 | 58.29 | 231.41 | 998.88 |
| Half | 0.01 | 0.08 | 7.17 | 27.76 | 241.44 |

Table 2.2: Example results for table halving recursion on an SP2 node. (Timings in seconds)

its memory usage.

If the tables created from the enumeration are only used to calculate the p-value, then it is not necessary to store the actual elements in the table. In a similar way to the target level optimisation, the test statistics and probability contribution of a row can be stored instead of the actual row elements. As well as a reduction in space, this reduces the repeated work in the calculation of the test statistics.

Table 2.2 shows some results on the example tables for this method. The comparison is with the previous version of the algorithm that does not do target level optimisation. The target level optimisation can also be applied to this algorithm and so the main point of interest is the efficiency of the enumeration part of the algorithm compared with the previous version.

As expected, Table 2.2 shows a significant performance gain in using the halving recursion algorithm. Note that Example 2.7 performs less well compared with the other tables. This is likely to be due to the differences in the row totals. The row with total one, and any pair of rows it is in, can only have 5 positions. This unbalances the two halves of the recursion and results in better performance if the row is in the bottom half of the table. The bottom half of the enumeration is the half that is stored and cycled through, for each step of the top half. Since this cycling process requires copying of the stored halves back into the main table, the algorithm benefits when this copying is reduced. In addition, pairing the row of total one with a row that has potentially the most variation, i.e. the row with the highest cell count, reduces the enumeration time further. With these row re-orderings, the enumeration time for Example 2.7 becomes 211.89 seconds; an enumeration rate of approximately 225,000 tables per second. This example table is discussed further in the examples section.

As mentioned earlier, the effectiveness of removing repeated work from the enumeration decreases with larger numbers of rows. This can only be eliminated by increasing the amount of stored intermediate data. This would significantly increase the memory requirement for enumerating large tables, which already use a reasonable amount of memory.

## 2.6 Table Re-ordering

The previous section mentioned a speed improvement from re-ordering an example table. This section looks more closely at this technique.

There are a two types of changes that can be made to the table that do not affect the p-value calculation. Firstly, the transpose of the table can be used. This allows rows to become columns and columns to become rows, which is especially useful for non-square tables. Secondly, the ordering of the rows and columns can be changed, which changes the order of the enumeration process.

### 2.6.1 Row Generation

The generation of rows is fundamental to the enumeration process and the ordering of the columns is vital to the efficiency of the generation process. As a result of the way the row generation routine operates, the best orientation of the column margins is to have them in ascending numerical order. This means that the first element, which is the cell with the least variation, will start at its maximum value. In the alternative situation with descending numerical order, this cell with appear at the end of the row and, hence, there will be a larger amount of backtracking when the algorithm attempts to deposit larger totals on that cell. Hence, having the column margins in ascending order is the most efficient. Experimental results confirm this observation.

### 2.6.2 Row Ordering

Having considered the ordering for row generation, the next decision to make is the order in which rows are enumerated. This chapter has discussed two strategies for complete enumeration, which have slightly different behaviours when changing the row ordering.

For the first strategy of enumeration, the key problem is the amount of repeated work. This means that the best performance is obtained when the amount of repeated work is reduced. The repeated work is due to the commonality of rows between tables. The level of repeated work can be minimised by ordering the rows such that those with the most variation come first in the enumeration.

This decision is also complicated by the optimisations mentioned above. With target-level optimisation, the significant saving is when the algorithm enumerates partial tables which are more extreme. So the most efficient configuration of the rows is to put the rows which are most likely to contain more extreme rows, towards the top of the table. Assessing which rows are good candidates is not always obvious. The aim is to find rows that are more extreme than the observed, based also on the fitted values. For example, if in a row, the fitted values are close to those observed, there is greater potential to have rows which depart significantly from the model of independence. Hence, the ordering needs to look at the contribution to the test statistic of each row in the observed table and also look at the variability of that row.

For the enumeration strategy that splits the table in half, the level of repeated work is no longer an issue. The main inefficiency in the algorithm is the process of copying the data from each of the stored tables into the current table. This process can be minimised by making sure that the lower half of the table has the least amount of possible rows. This suggests that the best orientation is to have the row margins in descending order. Yet the previous section detailed a case where this was not the most efficient configuration. Recall that with this enumeration method the table is reduced into a $2 \times k$ table, and each solution to those margins forms the column margins for the two halves. For a given example, the number of tables in the reference set is fixed. Different orderings of the rows lead to different numbers of half tables. Minimising the number of half tables results in less enumeration and a higher degree of sharing, both of which lead to more

| | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 2 | 3 | 0 | 6 | 12 |
| 1 | 6 | 8 | 6 | 13 | 34 |
| 0 | 7 | 11 | 8 | 21 | 47 |
| 3 | 15 | 22 | 14 | 40 | 94 |

Table 2.3: Example table from Senchaudhuri *et al.* (1995).

efficient enumeration. Calculating this ordering is not a trivial problem. The number of tables in the top and bottom halves is minimised when the row totals in the $2 \times k$ table are as close as possible.

### 2.6.3 Non-square Tables

This section looks at which orientation is best for non-square tables. For near square tables, the difference between enumerating the table in one orientation over another is less pronounced, and the more important factor is likely to be the spread of the row and column margins.

For both methods of enumeration, it would seem that the best orientation of the table is that there are more rows than columns. In this orientation, the level of repeated work is reduced for the normal recursive method. For the table halving method, this orientation allows for greater levels of sharing and hence more efficient enumeration. There is a penalty for orienting the table in this manner, since the increased number of rows results in added overhead. This has to be taken into consideration when ordering near-square tables.

## 2.7 Examples

In Senchaudhuri *et al.* (1995), the authors claim that "in every example the exact $p$ was computationally infeasible, even with StatXact Turbo". Table 2.3 is Table 3 from that paper. This example has been previously referred to as Example 2.7.

The exact p-value based on the $X^2$ statistic is 0.0246681, with a set size of 48,103,355

| | Wife's Response | | | | |
|---|---|---|---|---|---|
| Husband's Response | Never or Occasionally | Fairly Often | Very Often | Almost Always | Total |
| Never or occasionally | 7 | 7 | 2 | 3 | 19 |
| Fairly often | 2 | 8 | 3 | 7 | 20 |
| Very often | 1 | 5 | 4 | 9 | 19 |
| Almost always | 2 | 8 | 9 | 14 | 33 |
| Total | 12 | 28 | 18 | 33 | 91 |

Table 2.4: Rating of sexual fun: husband's response by wife's response.

tables. The exact p-value based on the $G^2$ statistic is 0.212130. This result took 155.20 seconds to calculate on an SP2 node. Senchaudhuri's approach took 190 seconds to estimate a result with a confidence interval of 0.0001. An SP2 node is significantly more powerful that the computer used by Senchaudhuri. Note that in this example, the target level optimisation is less effective since the difference in the p-values meant that there are less subtables where both test statistics exceeded their respective targets. As expected, the asymptotic p-value of 0.0004 based on $X^2$ is unreliable. This is mostly due to the sparse nature of the first row and column.

This table also demonstrates the increased efficiency that can be obtained with the ordering of the table. Using the ordering of the table above, the test takes 245.31 seconds. Re-ordering the table results in row 1 pairing with 4 and 2 with 3, and a column 4 moving to column 2. The test based on this table takes 213.58 seconds. Re-orienting the table and rearranging the rows accordingly, gives the runtime of 155.20 seconds as quoted above.

Table 2.4 shows the husband's and wife's response to their rating of sexual fun. This table is taken from Hout, Duncan and Sobel (1987). Couples answered the question "Sex is fun for me and my partner (a) never, (b) occasionally, (c) fairly often, (d) very often, (e) almost always". The rare (two wives and one husband) "never" responses where combined with "occasionally" responses because the data was sparse.

Under the model of independence there are 947,766,430 possible tables and exact p-values of 0.047117 and 0.113712 for $X^2$ and $G^2$ respectively. This result was calculated in 2620.29 seconds on a 150 MHz Pentium PC. This is an enumeration rate of approximately 360,000 tables per second. Asymptotic results give p-values of 0.045950 and 0.078567 for $X^2$ and $G^2$ respectively.

## 2.8 Conclusions

This chapter has detailed the implementation of two recursive approaches to the problem of complete enumeration under the model of independence for $r \times c$ tables. The inefficiencies of the algorithms were identified and reduced by optimisations, where possible. Although the algorithms are slower than the network algorithm for the majority of tables, some cases are too large for the network algorithm. Here, the algorithms in this chapter sacrifice run-time but have lower memory requirements, and are able to enumerate these problems. The examples demonstrate that whilst complete enumeration involves a large amount of computation, improvements in processors, compilers and algorithms allows enumeration to run in feasible time.

The following chapter looks a number of other models for $r \times c$ tables and extends the algorithms and approaches presented in this chapter to enable enumeration of these models.

# Chapter 3

# Extensions for Complex Models

## 3.1 Introduction

This chapter extends the recursive enumeration algorithm for model of independence examined in the previous chapter. New models such as quasi-independence and quasi-symmetry are introduced and the computational problems explored. Issues of accuracy are also discussed.

## 3.2 Quasi-Independence

The model of quasi-independence (QI) is similar to the model of independence except that it has a number of cells fixed to a certain value. These cells usually contain structural zeros. In many studies, where both cross-classified variables (the rows and columns) have identical categories, behavioural and social processes cause most of the observations to lie on the main diagonal. Typically, the model of independence is implausible in these cases and interest is turned to the off-diagonal cells. Social Mobility tables, are a good example, where the diagonal cells (i.e., no mobility) are of no interest. If the table is sparse then exact tests should be considered.

There are a number of changes that need to be made to the algorithm for independence, in order to calculate p-values under quasi-independence. In fact, all three of the major parts of the routine need to be changed; the enumeration of tables satisfying the constraints implied by the sufficient statistics and the calculation of the fitted values and table probability.

The underlying theme is that the cells that are fixed are marked and then removed from all aspects of the algorithm. When the table is inputted a map of the fixed cells is created and the fixed cell counts changed to zero.

### 3.2.1 Enumeration

Since the fixed cells have been marked and their values considered to be zero, enumerating a table with one or more fixed cells simply involves skipping fixed elements. Forcing the cells to have a fake value of zero means that they do not interfere with the marginal totals. Hence the enumeration can proceed in a row by row basis, skipping any fixed cells.

During row generation involving fixed cells, we must modify the algorithm in two places. Firstly when the algorithm is proceeding forward along the row, depositing the in-hand total. Secondly, when the algorithm is backtracking along the row. For both forward and backward directions, when a fixed cell is found, it is ignored and the algorithm moves on to the next cell.

There is one significant problem with the enumeration. Previously, with independence, any row generated is always part of a valid table. With quasi-independence this is not always the case. For example, consider the following table where the diagonal cells are fixed, which is represented by dashes,

$$\begin{array}{ccc|c} - & 2 & 1^a & 3 \\ & - & {}^b & 1 \\ & & - & 2 \\ \hline 1 & 2 & 3 & 6 \end{array}$$

In this partial table, the first row has been generated. The column with the elements marked $a$ and $b$, has a total of 3. With $a$ already set at 1, and element $b$ having a maximum value of 1, then that leaves a minimum of 1 for the last element in the column, but this element is fixed. Therefore, the first row (–,2,1) is invalid for this table, even though it satisfied the row total and column constraints.

Hence, a check needs to be made when the final row is reached, to see that the remaining column totals are valid, given any fixed cells that may be present. If a cell in the final row is fixed, then the remaining total for that column *must* be zero. If this is not the case, then the table is invalid under quasi-independence and the algorithm backtracks. The process of checking the final row is a form of rejection (a technique that is examined in greater detail later on). Since it is not possible to know whether a row is valid or not, without enumerating the rest of the subtables, it is not possible to perform the target level optimisation. These changes are common to both methods of enumeration as detailed in the previous chapter.

### 3.2.2 Calculation of Fitted Values and Test Statistics

With independence, the fitted values were found by a closed-form expression for each element. With quasi-independence, there is no closed-form expression, instead the values have to be calculated using an iterative procedure. There are a number of routines available to perform the fitting. The routine chosen was iterative proportional fitting, mainly for its simplicity. In this situation, performance is not an issue, since the fitting is only performed once.

The iterative proportional fitting (IPF) routine adjusts the elements, by repeated iteration. Each iteration, involves the row and column totals in turn. For each marginal total, the current total of the fitted estimates is found, the elements are then scaled accordingly to give the margin total, i.e. by the ratio of the observed margin total and the current total. Convergence is achieved when the maximum difference in the fitted values between each iteration is sufficiently small. The starting values for the elements are 1, except the fixed cells which have a value 0. Again, this removes the fixed cells entirely from the calculation, since a value of zero can never change.

For the calculation of the test statistics, care has to be taken setting the fixed cells' expected values to zero, since both calculations contain the division of the expected values. Under normal circumstances, no cell will have an expected value of zero, so when calculating the test statistics, if one is found, it can be ignored. Ignoring a fixed value is not a problem since the true cell value is the same in both the fitted and any generated table, and so their contribution to the test statistics is zero.

### 3.2.3 Table Probability

A similar problem of having no closed-form expression exists for the table probability. Generally, the relative probability of table $x$ is

$$P(x) \propto \prod_{i,j} \frac{1}{x_{ij}!}$$

The calculation of the table probability involves enumerating all tables which have the same sufficient statistics as the observed table, calculating the relative probability for each table and dividing each relative probability by the sum of the relative probabilities over the reference set (Forster *et al.*, 1996). Again, fixed cells are ignored, as they are not part of the distribution.

Having to enumerate all the tables in order to find the normalization constant is another reason why target level optimisation cannot be used. The process of calculating the test statistic contribution as rows are generated can also be applied to the table probability. This can possibly have a negative effect, since rows generated might not yield a valid table. Hence, if the rejection rate is high, the amount of wasted computation will cause the p-value calculation to take longer.

### 3.2.4 Examples

#### Sputum Cytology

Table 3.1 shows the cross-classification of two independent interpretations of sputum cytology slides for lung cancer. This example, originally from Archer *et al.* (1966), was used as an example by Smith and McDonald (1994) in their adaption of Patefield's

| First interpretation | Second interpretation N | A | S | P | T | |
|---|---|---|---|---|---|---|
| Negative | 26 | 19 | 1 | 0 | 7 | 53 |
| Ambiguous cells | 2 | 11 | 5 | 3 | 4 | 25 |
| Suspect | 0 | 1 | 6 | 6 | 0 | 13 |
| Positive | 0 | 0 | 0 | 4 | 1 | 5 |
| Technically unsatisfactory | 1 | 1 | 0 | 0 | 2 | 4 |
| | 29 | 32 | 12 | 13 | 14 | 100 |

Table 3.1: First and second interpretations of sputum cytology slides for lung cancer.

algorithm to generate random tables under the model of QI. In addition the table was used by Becker (1990).

Since most of the interpretations tend to agree, a large proportion of the observations lie on the main diagonal. Clearly the model of independence is implausible for this table. Instead, a test for quasi-independence amongst the off-diagonal cells can be performed, which corresponds to the hypothesis that the interpretations are independent if they differ. The sparseness of the table requires the use of exact tests.

Using the enumeration algorithm detailed in this chapter, the reference set is found to have 133,048 tables, the exact p-values are 0.0000103 and 0.0000164 for $X^2$ and $G^2$ respectively. Hence, the hypothesis of quasi-independence is rejected. This calculation took 92.7 seconds on a SPARCstation 10 Model 30. With reordering of the table, this was reduced to 87.5 seconds.

Smith and McDonald estimated this p-value to be 0.00005 with a 99% confidence interval of (0.00000, 0.00018). This was based on 20,000 tables generated under QI. Their simulate-and-reject routine required the simulation of approximately two million tables of which the 20,000 satisfied the additional constraints of QI.

### Menopause

Table 3.2 shows data collected from two retirement communities, where each subject was asked at what age they reached menopause (Paganini-Hill *et al.*, 1984). The table shows the cross-classification of the reported age with the age from the respondent's

| | Reported at interview | | | | |
|---|---|---|---|---|---|
| In medical record | $\leq$44 | 45–49 | 50–54 | 55+ | |
| $\leq$ 44 | 5 | 4 | 0 | 1 | 10 |
| 45–49 | 4 | 7 | 5 | 2 | 18 |
| 50–54 | 0 | 7 | 17 | 8 | 32 |
| 55+ | 1 | 1 | 2 | 3 | 7 |
| | 10 | 19 | 24 | 14 | 67 |

Table 3.2: Age at natural menopause: interview by medical record data.

medical record. Nearly half of the observations fall on the main diagonal. Hence a test of quasi-independence for the off-diagonal cells is performed. With $X^2$ and $G^2$ values of 11.579 and 14.738 on 6 degrees of freedom, the asymptotic p-values for $X^2$ and $G^2$ are 0.0720 and 0.0224 respectively. Complete enumeration shows that there are 1413 tables in the reference set, with exact p-values for $X^2$ and $G^2$ of 0.0384 and 0.0222 respectively. The results show that the asymptotic p-value based on the $X^2$ statistic was unreliable in this case and the hypothesis of quasi-independence amongst the off diagonal cells is rejected. The complete enumeration took approximately 1 second on a SPARCstation 10 Model 30.

The model of quasi-independence can also be applied to Table 2.4 from the previous chapter, by fixing the diagonal cells, and looking for quasi-independence for the off-diagonal cells. This yields 15,708 tables with exact p-values for $X^2$ and $G^2$ of 0.4002 and 0.5023 respectively. The calculation took 11.23 seconds on a SPARCstation 10 Model 30. The asymptotic p-values for $X^2$ and $G^2$ are 0.4991 and 0.5291 respectively.

**Stroke Data**

Table 3.3 presents data about the recovery of patients after a stroke (Bishop & Fienberg, 1969). As the patients only improve, the final ratings are greater than or equal to the initial rating, where E denotes the worst rating and A the best rating. This triangular table was analysed by McDonald and Smith (1995) using Monte Carlo simulation to estimate the exact p-values. For quasi-independence, there are 3,031,328 tables in the reference set. The exact p-values for $X^2$ and $G^2$ are 0.1267 and 0.1707 respectively. These values are within the 99% confidence intervals for the estimated exact p-values

| Initial rating | Final ratings | | | | | Totals |
|---|---|---|---|---|---|---|
| | A | B | C | D | E | |
| E | 11 | 23 | 12 | 15 | 8 | 69 |
| D | 9 | 10 | 4 | 1 | – | 24 |
| C | 6 | 4 | 4 | – | – | 14 |
| B | 4 | 5 | – | – | – | 9 |
| A | 5 | – | – | – | – | 5 |
| Totals | 35 | 42 | 20 | 16 | 8 | 121 |

Table 3.3: Initial and final ratings on disability of stroke patients.

given by McDonald and Smith (1995). This calculation took 109.12 seconds on an SP2 node. The ordering of the rows and columns is especially important for a triangular matrix. The most efficient ordering is to arrange for rows with more fixed cells to occur towards the top of the table. Ordering the table with the fixed cells towards the bottom results in a runtime over 9 times longer.

## 3.3 Enumerate-and-Reject

This section looks a new approach to enumerating more complex models in 2-way tables. The method, called enumerate-and-reject is similar to the simulate and reject procedure described in Smith and McDonald (1994). For a given model, if a direct algorithm to enumerate all possible tables in the reference set is not available, an alternate method is to enumerate all tables in a reference set which encompasses it, i.e. given $\Gamma_A \subseteq \Gamma_B$, find $\Gamma_B$, and reject tables that do not satisfy the additional constraints which define the difference between $\Gamma_A$ and $\Gamma_B$.

The table probabilities and p-values are calculated in the same way as for quasi-independence.

### 3.3.1 Quasi-Symmetry

The model of quasi-symmetry adds a further constraint to that of quasi-independence. The sum of each element in the table and its diagonally opposite cell are the same as the observed, i.e. $\forall i, j \in 1, .., r, x_{ij} + x_{ji} = o_{ij} + o_{ji}$. For $i = j$, i.e., the diagonal cells, the cell values are fixed. Hence, enumerate under $\Gamma_{QI}$ (diagonal cells fixed) and reject any table that does not satisfy the pairing constraint above.

To calculate the fitted values, the iterative proportional fitting algorithm used for quasi-independence is extended to perform the proportional adjustment on each of the pairs in the model, as well as the marginal totals.

### 3.3.2 Uniform Association

The model of uniform association associates a weight with each cell and a constraint such that the sum of each cell count multiplied by the weight is the same as the observed. The weight for a given cell $y_{ij}$ is $u_i u_j$, where $u$ is an equal-interval scoring. Hence, enumerate under $\Gamma_I$ and reject any table that does not satisfy association sum. A variation of this model is quasi-uniform association (Goodman, 1979), in which the diagonal cells are also fixed. This model can be tested by rejecting from $\Gamma_{QI}$ those tables which do not satisfy the association sum.

### 3.3.3 Examples

The model of quasi-symmetry can be applied to the sexual fun data. As detailed above, the number of tables under quasi-independence is 15,708, which is a reasonable quantity to reject from. The rejection step yields just 161 tables and gives exact p-values of 1.0 for both $X^2$ and $G^2$. This calculation takes just 3.67 seconds on a SPARCstations 10 Model 30.

The model of uniform association is also considered for the stroke data example, Table 3.3, by McDonald and Smith (1995) as an alternative hypothesis. With enumerate-and-reject the model of uniform association can be used as the null hypothesis. Scores of 1,2,3,4 and 5 for A,B,C,D and E were used. For this model, there are 47,629 tables in the reference set. The exact p-values are 0.3510 and 0.3644 for $X^2$ and $G^2$ respectively, indicating a better fit for the model than QI. This calculation took 96.47 seconds to calculate on an SP2 node.

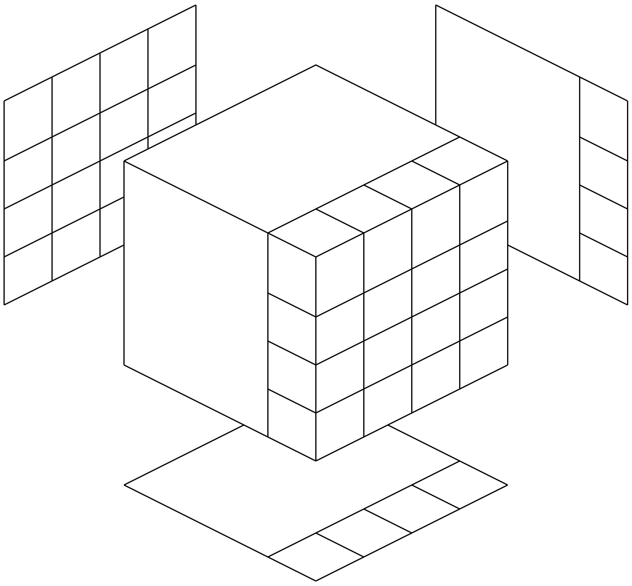

Figure 3.1: Decomposing a 3-dimensional table.

## 3.4 $n$-way Tables

So far, this thesis has examined enumeration for only 2-way contingency tables. A number of papers have looked at multidimensional tables and simple hypotheses, e.g., mutual independence, see Agresti (1992) for a summary. This section modifies the recursive algorithm to work with a table of an arbitrary number of dimensions for the hypothesis of no $n$-way interaction in a log-linear model. For the case of a 2-way table this is equivalent to the model of independence.

The recursive algorithm for 2-way tables, decomposes the table into rows, maintaining the constraints on that row. In a similar way, a $n$-way table consists of a number of $(n-1)$-way tables. Each $(n-1)$-way table can be enumerated recursively. So the enumeration algorithm becomes:

1. find the constraints on the first $(n-1)$th dimensional table
2. enumerate all possible tables given the constraints
3. for each table, update the constraints and enumerate all the possible $n$-way subtables.

When a 2-dimensional table is found, it can be enumerated in the usual way. There is a slight difference from the enumeration for 2-way tables since the $n$-way table has extra constraints. For example, a 3-dimensional table is a series of 2-dimensional tables, with 3 planes as its margins, as shown in Figure 3.1. To enumerate the 2-dimensional table shown requires extracting the margins from two of the planes. The remaining plane is the constraint on the elements perpendicular to the direction of the enumeration. The plane forms a element-wise restriction on the normal enumeration of the 2-way table. This means that when generating a row, there are two restrictions on the maximum value of each element, given by the column total and the element-wise restriction. In the general case, the maximum values of a structure are found by finding the minimum values over a number of constraints.

This leads to the enumeration of $n$-dimensional tables under the model of no $n$-way interaction. The simplicity of the recursive algorithm means that it could be adapted for $n$-dimensional tables.

This extended recursive algorithm was only implemented in Lisp. This was mainly due to the fact that the conversion of the Lisp 2-way algorithm to the $n$-way version was relatively straightforward. The only major change made was to handle the arbitrary number of constraints. The main function of the routine is as follows:

```
 (define (recurse n constraints)
  (if (= n 2)
    (generate-2 constraints) (if (= 1 (car (sizes? (car (cdr (car
    constraints))))))
      (final-element constraints) (flat1
(lazymap (lambda(x)
  (lazymap (lambda (y) (combine x y))
   (recurse n (recurse-constraints x constraints))))
  (generate-element n constraints))))))
```

This has the familiar structure of the previous 2-dimensional version. Firstly, a check is made to see if the table to be generated is 2-dimensional, in which case the appropriate function is called. If the table to be generated is the last one in the sequence, then the special case function `final-element` is called. Otherwise, the familiar procedure of generating all possible $(n-1)$-way tables, and for each generating the subtables, is used. Note the extra work updating the constraints, a lot of which is hidden behind the `generate-element` function.

In the Lisp implementation, a $n$-dimensional table is stored as a number indicating the number of dimensions, a list of the sizes of each dimension, and the data in the table stored in nested lists. The constraints on a table are stored as a list of $n$-dimensional tables in a specific order. The ordering allows for easy manipulation of the constraints such that the correct set of constraints on a given part of the table can be easily found and without storing extra information, such as the hyperplane in which a constraint works in.

The added complexity of handling the constraints and the larger number of cells in a typical table result in the Lisp implementation being impractical for all except small tables. The algorithm could be implemented in C to make it run faster. One of the significant problems with the algorithm is that it only enumerates under the model of no $n$-way interaction. When analysing $n$-way tables, the models used typically involve more complex structures. Hence, the work to implement the C version was not undertaken. Instead a more generic approach to enumerating models was investigated, which is discussed in the following chapter.

### 3.4.1 Example

Table 3.4 is a $2 \times 4 \times 4$ table of data taken from the General Social Survey in the United States in 1991. Agresti (1996) fits various models to this table, including no 3-way interaction model. Applying the above enumeration routine to the table gives 6,597 tables and p-values of 0.6384 and 0.7584 for $X^2$ and $G^2$, respectively. This took 19.35 seconds to calculate with the compiled Scheme version on a SPARCstation 10.

| | | Job Satisfaction | | | |
|---|---|---|---|---|---|
| | | Very | A Little | Moderately | Very |
| Gender | Income | Dissatisfied | Satisfied | Satisfied | Satisfied |
| Female | $< 5{,}000$ | 1 | 3 | 11 | 2 |
| | 5,000–15,000 | 2 | 3 | 17 | 3 |
| | 15,000–25,000 | 0 | 1 | 8 | 5 |
| | $> 25{,}000$ | 0 | 2 | 4 | 2 |
| Male | $< 5{,}000$ | 1 | 1 | 2 | 1 |
| | 5,000–15,000 | 0 | 3 | 5 | 1 |
| | 15,000–25,000 | 0 | 0 | 7 | 3 |
| | $> 25{,}000$ | 0 | 1 | 9 | 6 |

Table 3.4: Job satisfaction and income, controlling for gender.

## 3.5 Arbitrary Precision Calculations

One of the areas often overlooked is that of the loss of precision in the calculation of the exact p-value. Normally, the p-values are calculated using floating point arithmetic, but the floating point operations have a degree of error. Admittedly, the error margin is very small, but it can be of some concern with the nature of the calculation of the exact p-values. This involves the ordering of a large number of tables, via the test statistic and fitted values, followed by the summation of possibly very small table probabilities. The exact p-value is then found by dividing two sums of table probabilities, the numerator often being very small compared to the denominator. Performing all these calculations in arbitrary precision would avoid introducing any inaccuracies.

The following sections takes a brief look at what an arbitrary precision library provides, followed by a discussion of each of the areas of calculation in the algorithm.

### 3.5.1 Arbitrary Precision Arithmetic Libraries

Arbitrary precision arithmetic libraries provide the user with a number of new methods of storing numbers, the main feature of which is that there are no size or accuracy restrictions. Two types of arbitrary precision numbers are usually supplied, an integer-like type with arbitrary size, and a rational number type for storing fractions. Ancillary functions such as fast greatest common denominator (`gcd`) operations for factorising and reducing rational numbers are also available. The internal representation of the numbers are hidden from the user. Rational numbers are usually stored as a numerator and denominator using arbitrary sized integers.

The packaged used in this thesis was the `gmp` library from the Free Software Foundation GNU archive. This library provides both arbitrary sized integers and rational numbers as well as user-specified precision floating point numbers. For user-specified precision floating point numbers, the user can declare the level of precision stored, which can be useful when the user does not require infinite precision, perhaps due to speed or memory restrictions. For calculating exact p-values, the rational arithmetic part of the `gmp` library meets the requirements for the arithmetic side of calculating the p-values.

### 3.5.2 Fitted Values

Except in the case of independence, when the fitted values have a known closed-form expression, fitted values have to obtained by using iterative proportional fitting. This can be implemented in rational arithmetic, the only problem being the stopping rule. Unfortunately, the calculation of the fitted values is the only part of the algorithm that cannot be found exactly. In the same way that a tolerance is used for the floating point versions of IPF, a rational number has to be used.

### 3.5.3 Test Statistics

The test statistic is calculated using the observed and fitted values and is used to order the tables in the reference set. The accurate ordering of the tables in the reference set is vital for calculating the exact p-value. Typically with the floating point version of the code, there is sufficient error introduced, such that the comparison of test statistics has to include a certain tolerance. A tolerance value of $10^{-11}$ was used throughout this thesis.

Given the availability of rational fitted values, the comparison of test statistics can be carried out using rational arithmetic by using reduced versions of both $X^2$ and $L^2$ which have rational forms. Namely,

$$
\begin{aligned}
X^{2'} &= \sum_{ij} \frac{x_{ij}^2}{e_{ij}} \quad \text{and} \\
L^2 &= \sum_{i,j} x_{ij} \log(\frac{x_{ij}}{e_{ij}}) \\
&= \sum_{i,j} \log((\frac{x_{ij}}{e_{ij}})^{x_{ij}}) \quad \text{or equivalently} \\
L^{2'} &= \prod_{i,j} (\frac{x_{ij}}{e_{ij}})^{x_{ij}}
\end{aligned}
$$

### 3.5.4 p-values and Table Probability

The calculation of the table probability is an interesting case since the form of the table probability is regular, i.e., the product of factorials. The table probability could be stored exactly by listing the factorials that make up the value, in effect storing a condensed version of the table. When the table probabilities are divided, the degree of canceling should yield a significant saving in computation. Unfortunately, the operation that is performed the most, namely addition of two table probabilities, is not as simple. The significant problem is that once the greatest common divisor of the two quantities has been factored out, the remaining calculation is the addition of a number of factorials. These factorials will have to be expanded to perform the calculation, the result of which is not going to be in factorial form. Hence the table probability has to be stored as a product of factorials times an integer. In practice, it is found that the level of canceling of common factorials is not very high and the integer factor becomes large, since the expansion of factorial numbers yields large results.

An alternative approach is to store the table probability as primes. The factorials can be decomposed into the product of powers of prime numbers. This form results in a much higher degree of cancellation and factorisation. Again, the problem operation is addition, which results in the addition of two sets of products of primes. Whilst this method works for relatively small tables and element counts, it breaks down with larger problems. This is because, typically, the addition factorises into the addition of two large numbers, constructed from the product of a number of primes. The only way to handle these numbers is to use the integer part of an arbitrary precision maths package. In practise, the overhead of maintaining the list of prime powers and performing the cancellation and then the expansion of prime powers into large integers, is so high that it is quicker to perform all the calculations in the arbitrary sized integer package, which has much better handling of prime numbers and has code which is more highly optimised for different architectures.

## 3.6 Conclusion

The algorithms from Chapter 2 formed the bases of the routines in this chapter for enumerating tables under the model of quasi-independence. The chapter has described how the complete enumeration technique can be applied to cases where there are fixed cells in the table. The simplicity of the recursive enumeration algorithm was key to this extension. The work allows the calculation of an exact p-value, which is beyond the current capability of other published routines. An enumerate-and-reject strategy was proposed for the enumeration of complex models that place additional constraints on the model of quasi-independence. This allows the enumeration of models such as quasi-symmetry and uniform association.

One of the major problems with the enumerate-and-reject algorithm can be the sheer size of the reference set from which the routine has to reject. Typically with the models of quasi-symmetry and uniform association, the number of tables in their reference set is orders of magnitude smaller than the reference set. It was found that for independence and quasi-independence that most of the computation is involved with calculating the test statistics and table probability, i.e. the floating point part of the code. Floating point operations are slower than integer operations, and so its not unrealistic to propose the enumerate-and-reject algorithm which involves more integer operation.

The code for quasi-independence could be changed to enumerate the special case of quasi-symmetry, but this approach is both an uninteresting and limited. There will always be a more complex model to enumerate. The idea of enumerate-and-reject can always be applied in these situations. The computational inefficiencies of the enumerate-and-reject approach are obvious. The following chapter investigates more efficient methods of enumerating such models.

The simplicity of the recursive algorithm allows it to be extended to the enumeration of $n$-way tables, under the model of no $n$-way interaction. Here, the use of the Scheme language for prototyping purposes was key in the implementation.

Chapter 4 looks a more generic solution to the problem of enumerating tables under all of these complex models.

# Chapter 4

# Enumeration for Complex Models

## 4.1 Introduction

Previous chapters investigated algorithms for complete enumeration of tables under the models of independence and quasi-independence. This chapter looks at an alternative method of enumeration that can handle more complex models.

## 4.2 Background

Complete enumeration of more complex models such as quasi-symmetry was achieved by a so called enumerate-and-reject algorithm. Tables were generated under quasi-independence and rejected if the table did not satisfy quasi-symmetry. This is obviously an inefficient way of performing the enumeration since it is likely that a large number of tables will be rejected from the reference set for quasi-independence.

By having a specific algorithm for enumeration under a certain model, a different algorithm and re-coding is required in order to enumerate under another model. An example of this would be for higher dimensional tables. Previously, an algorithm for recursive generation was described, but one problem was that the algorithm was specifically developed for the model of no $n$-way interaction. This was a major limitation since with $n$-dimensional tables the user is typically looking for more interesting structures in the data. A better approach would be to have an algorithm that could enumerate any complex log-linear model.

## 4.3 Evolving

The following is the set of constraints under the model of independence, for a $3 \times 3$ table, where $x$ is the table data and $r_1$, $r_2$, $r_3$ and $c_1$, $c_2$, $c_3$ are the row and column margins.

$$\begin{array}{ccccccc}
x_{11} & + & x_{12} & + & x_{13} & = & r_1 \\
x_{21} & + & x_{22} & + & x_{23} & = & r_2 \\
x_{31} & + & x_{32} & + & x_{33} & = & r_3 \\
\\
x_{11} & + & x_{21} & + & x_{31} & = & c_1 \\
x_{12} & + & x_{22} & + & x_{32} & = & c_2 \\
x_{13} & + & x_{23} & + & x_{33} & = & c_3
\end{array}$$

Serialising the table into a vector of elements gives:

$$\begin{array}{ccccccc}
x_1 & + & x_2 & + & x_3 & = & r_1 \\
x_4 & + & x_5 & + & x_6 & = & r_2 \\
x_7 & + & x_8 & + & x_9 & = & r_3 \\
x_1 & + & x_4 & + & x_7 & = & c_1 \\
x_2 & + & x_5 & + & x_8 & = & c_2 \\
x_3 & + & x_6 & + & x_9 & = & c_3
\end{array}$$

This can be rewritten as a matrix vector product, with $r_+$ denoting the sum of all the row totals, which is the table total:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} r_+ \\ r_1 \\ r_2 \\ r_3 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix}$$

or equivalently $D\mathbf{x} = \mathbf{s}$, where $D$ is the model matrix, and $\mathbf{s}$ is the vector of sufficient statistics. All the solutions of $\mathbf{x}$ are the members of the reference set. Note the introduction of an extra constraint for the table total. Whilst this is not strictly necessary, since it is a combination of the other constraints, it is left in for completeness.

This form of specifying the constraints is the traditional model matrix or model matrix. Later on, the limitations of using a model matrix as a user specification, are discussed, with possible alternatives.

## 4.4 Enumerating from a Model Matrix

This section outlines an algorithm for enumeration of a model specified by a model matrix and its issues of implementation.

In principle, the approach to enumerating all the given tables is the same as before. All the counts are found for each of the elements in the table, such that all of the constraints are simultaneously satisfied. The enumeration is recursive, since it fixes a cell at a certain value, and then focuses on the rest of the vector, returning to the fixed cell once all the subsearch space has been traversed. The process is started at the first cell in the vector $\mathbf{x}$. Note that this is finer recursion than the algorithm for independence and quasi-independence, in that it works on a per-cell basis as opposed to a per-row basis.

Each cell is cycled from 0 to a maximum value. The maximum value is determined by looking at all the constraints which concern the current cell. For each constraint, a maximum value is found by setting all the other unfixed cells to 0, and adjusting the sufficient statistic based on the fixed cells. The minimum of all these values is the value we use for the maximum possible value that the cell can have. This, combined with the minimum value of 0, gives a range that encompasses the real range of possible values. This range estimation is obviously not very good, but it is a quick calculation. Speed is a great factor here since the routine will be called a very large number of times. The danger is that a range estimation routine spends too much time estimating when it is quicker to reject the infeasible values as the algorithm progresses. This is obviously a compromise, since at certain points during the enumeration, i.e., at the start, it would be better to spend more time estimating the ranges, to cut out large parts of the search tree. Conversely, when there are few cells remaining, the extra time spent optimising the range would slow the routine down.

The basis of the enumeration routine is as follows:

1. find free cell
2. find maximum value of free cell
3. for 0 to max value
    (a) fix free cell to current loop value
    (b) check constraints
    (c) if constraints are satisfied, make recursive call
    (d) backtrack fixed cells

The process of checking whether a value is possible for a given cell, involves a recursive search. Firstly, the cell is tentatively marked as fixed. Each of the constraints that include the cell are checked, to make sure they have not been exceeded. If a constraint has been exceeded, then the checking routine returns a failure. If all the constraints are still met, then true is returned.

When fixing a cell to a certain value, there are certain circumstances where this causes other cells to be fixed. In the situation where a constraint has all but one cell fixed, then the remaining cell must be fixed to a certain value by subtraction. If a constraint's target value is met by some fixed cells, the remaining cells in the constraint would have to be zero for the constraint to be satisfied. In all these situations, where cells can be automatically fixed, the checking routine is called for each of these cells, hence the recursive nature of the routine. If the checking routine indicates that these cells cannot be successfully fixed at the given values, then the routine has to fail. On failure, all the cells that were tentatively fixed have to be cleared, and the routine backtracks.

Backtracking is a more complicated procedure than in the algorithm for independence. Previously, backtracking was simply stepping back to the previous element (cell or row), since the elements are visited sequentially, in order. Conversely, since cells are not necessarily fixed in sequential order, backtracking in this algorithm is not straightforward. To aid with the backtracking, each cell that gets fixed has a *rank* associated with it. The *current rank* is incremented every time a cell is fixed. Any cells that are automatically fixed by that cell are marked as having the current rank. Hence, when backtracking, all cells with the relevant rank can be cleared.

The only remaining issue of the enumeration algorithm is how cell values in the model matrix $D$ are treated. These values can be zero or greater. With values greater than 1, special care needs to be taken. Firstly, when evaluating a constraint's current total, cell counts in $\mathbf{x}$ have to be multiplied up accordingly. Secondly, when attempting to find the maximum value of a cell in $\mathbf{x}$, the constraint totals have to be divided by the relevant cell's value in the model matrix. Cell values greater than 1 can be used to weight cells, for example, in the model of uniform association. Negative values in the model matrix are not valid. For the large majority of models used, negative values are not required. With cases such as in uniform association where negative weights might be assigned, the weights can be transformed into positive values. Changing the algorithm to allow negative numbers is possible, but it requires more stringent checking of boundaries, especially when calculating maximum values of cells.

| | Pathologist B | | | | |
|---|---|---|---|---|---|
| Pathologist A | 1 | 2 | 3 | 4 | 5 |
| 1 | 22 | 2 | 2 | 0 | 0 |
| 2 | 5 | 7 | 14 | 0 | 0 |
| 3 | 0 | 2 | 36 | 0 | 0 |
| 4 | 0 | 1 | 14 | 7 | 0 |
| 5 | 0 | 0 | 3 | 0 | 3 |

Table 4.1: An example table for enumerating under QS, with a high degree of simplification.

## 4.5 Model Matrix Simplification

Before a model matrix can be used for enumeration, it has to be checked to remove certain features of the constraints. Typically, these features only slow the algorithm down and do not cause it to fail.

Firstly, if a constraint involves no cells, i.e., a row of zeros, then this can obviously be removed. If a constraint only involves one cell, then the cell has to be fixed at the sufficient statistic. In this case, the cell can be removed from $\mathbf{x}$ as well as the corresponding column in the model matrix $D$. The removal of an cell also requires that the sufficient statistics of all the constraints that involve that cell are updated to reflect the value that the cell is fixed at. If a constraint has a sufficient statistic of zero, then all the cells involved in that constraint must all be zero. Hence, these cells can be marked as fixed and removed from the model matrix and the constraint can be removed.

When a cell has been fixed and removed from the model matrix, it no longer plays a part in the calculation of the exact p-values, since it makes no contribution to the test statistics used. Hence, it can be removed from the enumeration and calculation of the fitted values without causing a problem. Although, in order to reconstruct the original structure of the input data, a list of cells removed is kept. This is mainly for user interface purposes, where the user can see the entire data set, instead of a subset of the cells.

These adjustments to the model matrix have to be repeatedly tested for, until no more changes can be made. This is because one change may adjust the model matrix in such a way that other changes become possible. An example of this can be seen with Table 4.1. Under quasi-symmetry, the final cell in the table (5,5) is fixed, hence its removal. This results in column 5 having a total of zero, so all these cells can be fixed at zero and removed. With QS each of the cells in this column are paired with corresponding cells in row 5. Hence, these cells are fixed and row 5 removed as well. Similarly element (4,4) is fixed. This results in column 4 having a total of zero, so all these cells can be fixed at zero and removed. With QS, each of the cells in column 4 is paired with corresponding cells in row 4. Hence these cells are fixed and row 4 removed. This results in a $3 \times 3$ table with diagonal cells fixed and 3 remaining pairs of cells. The model matrix is as follows:

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

This, in fact, has one degree of freedom and is equivalent to the quasi-independence model. In this case the extra rows in the model matrix are redundant. This exhibits simplification at a higher level, by removing rows based on the rest of the model matrix. The last 4 rows in the above model matrix can be removed. A typical example is with the inclusion of the table total in the model matrix. For most models, the table total is equivalent to summing another group of constraints, such as the row margins, hence it is redundant. Alternatively, instead of removing the table total, the final row margin and column margin could be removed.

For enumeration purposes the tendency is to leave the constraints in the model matrix. This is mainly for efficiency, since if a constraint is removed, in order to assess that constraint, two or more other constraints have to be checked. This checking might lead to a large amount of extra computation. For example, consider the removal of a row margin, and the inclusion of the table total. To assess the removed row constraint, requires the algorithm to look at the table total and the other row constraints. If the cells in these constraints have not already been fixed by the enumeration, then the cells in the remaining row will have much larger maximum values. This can result in a lot of computation where the sum of these cells is in fact larger than the removed constraint allows, but this is not discovered until later in the enumeration when the remaining cells are fixed. A better approach is to remove the table total constraint since this is made up of the row constraints, which are more restrictive on the cells. Again, if all the row margins are kept, then the final column margin can strictly be removed, since the final column margin is the sum of the row margins minus the sum of the other column margins. Yet, this constraint is left in, to aid the enumeration algorithm, as well as the iterative proportional fitting routine which converges faster with the version of the model matrix with more constraints.

## 4.6 Changes to the Basic Algorithm

### 4.6.1 Table Probability

Being able to enumerate highly constrained models means that exact tests are feasible for larger tables. This results in having to deal with table probabilities that have values that are very small. One of the problems encountered was that values became smaller than the range of the exponential function.

Table probabilities are calculated by taking log factorials from a lookup table and summing the values. The exponential of this sum is found to give the table probability and the range underflow of this operation is the cause of the problem. Even if the probability is within range of the exponential function, working with such small numbers can lead to further accuracy problems. Clearly, using arbitrary precision arithmetic would avoid this problem.

A solution to this problem revolves around that fact that the main point of interest is the ratio between the total probabilities of the extreme tables and the whole reference set.

This means that both values can be multiplied by a constant and the ratio preserved. Hence, a constant value can be added to the log probabilities, to bring the value back into range of the exponential function.

All that is required is the choice of the constant value. Typically, the range of the table probabilities is relatively small compared with the minimum allowed value for `exp()`. This means that the constant value can be chosen based on a value that is known to be in the range, as opposed to having to explicitly know the minimum and maximum of the range. The method used to calculate the constant value, takes the fitted values, converts this to integer values and finds the log probability of that table. This is a convenient method, since the fitted values have already been calculated. Converting the fitted values to integer values yields a table, that may not satisfy the constraints. This is not thought to be a problem, since the resulting table probability will be in the required range or sufficiently close to it. Using a table probability of a table close to the fitted value chooses a value for the constant that is at the most likely end of the range. Again, the assumption is that the range is sufficiently tight such that the least likely end would not underflow `exp()`. A better solution would be to use the log probability of the least likely table. Finding the least likely table from a model matrix is non-trivial. There are approaches to finding extreme tables for simple cases such as independence on $r \times c$ tables. For this application, it was considered that this was beyond the scope of the thesis, and could form the basis for some further work.

Another solution would be to have a dynamic updating of the constant. If a log probability fell outside the range of `exp()`, then that value could be used as the constant, and the relevant cumulative totals scaled accordingly. The disadvantage with this approach is that it introduces a lot of extra manipulation to the p-values, which could lead to accuracy problems.

### 4.6.2 Change Tracking

With the enumeration of larger tables the amount of time spent calculating the test statistics becomes large, and yet the amount of changes between the successive tables is generally small.

Instead of needlessly calculating the contribution to the test statistic for each cell, the results from the previous calculation can be saved and reused. To enable this, whilst enumerating, a vector is maintained indicating whether cells have changed since the last time the statistical routines were called. Hence, when it comes to recalculating the test statistics, only the updated cells need be visited.

Again, this optimisation performs much better on larger tables. For smaller tables, the overhead of maintaining the change vector outweighs the benefits. The relationship is also clouded by the fact that the trade off between redoing the full statistical tests and maintaining the change vector is affected by the integer and floating point performance of the processor. In a heterogeneous computing environment this relationship is going to vary. With the use of rational arithmetic for the test statistics, which are much slower than the floating point versions, the benefits of using the change vector are much greater.

One of the other factors is the search space to table production ratio. The change vector has to be updated for every change made to the result vector. This includes changes made whilst testing possible cells. Hence, with an enumeration that spends a lot of time searching for tables, and generates relatively few, the performance benefit of the change vector may be lost.

One attempt to minimise this problem focuses on the start of the computation where we often find a period of searching before the algorithm enumerates the first table. Here, the change vector code could be disabled until a table is found, since until that point it serves no gains. There is also typically a similar search space at the end of the computation, but the true bounds of the reference set space are unknown, and hence it is hard to know when to turn the change vector code off. A drawback of this solution is that there is an overhead in testing when the change vector code is in use. Either a logic test is made just before every piece of vector code, or two versions of the enumeration code are used, with and without the change vector code. In the first case, the test incurs a penalty every time the vector code appears in the algorithm. With the second approach the penalty is just the extra added code, and its maintenance.

### 4.6.3 Matrix Re-ordering

This section discusses re-ordering the model matrix to yield a more efficient enumeration. Re-ordering of the matrix can take place in each dimension, independently, i.e., re-ordering the cells in $\mathbf{x}$ which re-orders the columns of the model matrix and re-ordering the constraints (the rows of the model matrix).

One of the advantages of these optimisations is that it requires no change to the basic algorithm. Indeed the re-ordering of the model matrix can be performed on the model matrix before inputting it into the enumeration program. This means there is no runtime penalty for the optimisation.

#### Element Re-ordering

One of the main problems with the enumeration algorithm is that it requires a large amount of searching. Typically, much of this search tree yields few or no tables. The aim is to reduce this wasted computation.

The enumeration algorithm loops an unfixed element from zero to a calculated *maximum value*. For each value, a further search tree is required. If the range of these loops can be reduced as far as possible, the computation will be more efficient. As discussed in section 4.4, better estimation of the ranges would be an obvious benefit. Another approach that can yield significant benefits revolves around the edges of the search tree.

When a constraint target is reached, all the remaining cells can be fixed. Similarly, when a single cell remains unfixed in a constraint, it can possibly be fixed at the remaining target value. These two operations represent pruning of the search tree, since when a cell is fixed, the range for that cell does not need to be searched. This pruning reduces the computation that is required by the basic algorithm, and it can be exploited in order to reduce further the computation that is required. This can be achieved by arranging for the elements that have large search trees to be naturally pruned by moving them towards the end of the element vector. Hence, as the searching progresses, these cells will be fixed automatically by subtraction from the target values and already fixed cells. The computation required is reduced by biasing the search tree.

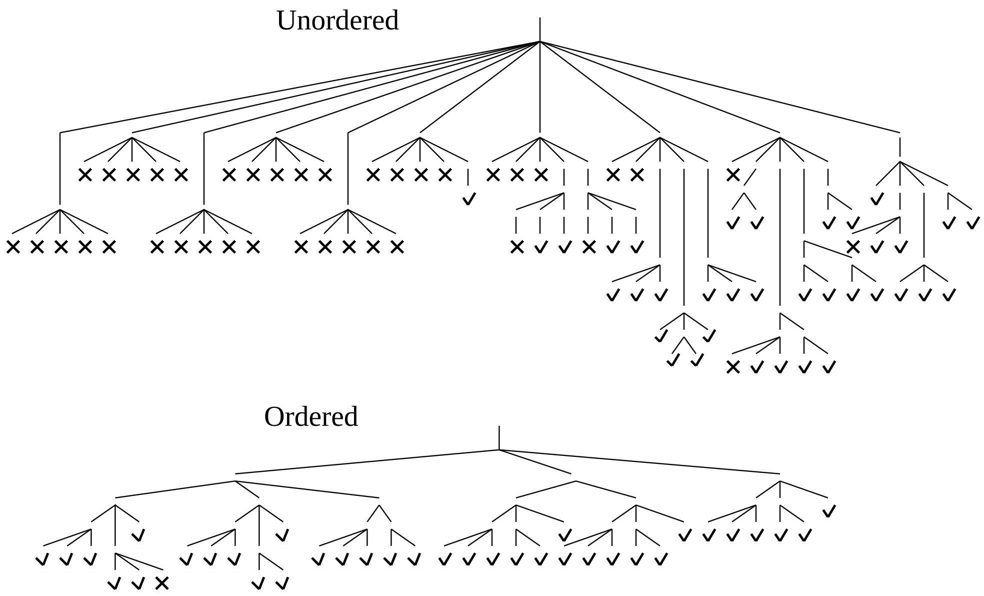


Figure 4.1: Example enumeration trees of unordered and ordered model matrices.

An ordering of elements could be achieved by calculating the maximum values for each cell and sorting on these values, starting with the lowest value as the first element. This would sort the elements without concern for their relationship with other elements in the table. This may lead to the enumeration jumping about the result vector, without ever fixing a significant block of the table. Where previously the algorithm, enumerating in order, would fix blocks (e.g., rows and slices of a multidimensional matrix). Enumerating in a blockwise fashion is only really cosmetic, in that the user is better able to see what part of the table has been enumerated. The idea is that the re-ordering of the table leads to more efficient enumeration.

This can be seen in Figure 4.1. The figure shows two trees representing the enumeration of a simple $3 \times 3$ table for the model of independence on a table with row margins 12,2,2 and column margins 9,4,3. The top tree is the unordered enumeration, and the bottom is the ordered element enumeration. In these trees, the automatic fixing of cells is not shown, the emphasis is on the looping of cells as shown by the branching. A tick indicates successful generation of a table, and a cross means failure, due to one or more constraints being violated. The left-hand side of the unordered tree has more searching yielding no tables. The ordered tree does not exhibit this feature.

Notice that the ordered tree is much deeper, whereas the unordered tree is flatter and wider. By using a tree with deeper branching, the level of shared work, i.e., the number of levels in the tree *above* the node, is increased. The deeper branching is preferable because it indicates less repeated work in the enumeration. The issue of repeated work is one of the major stumbling blocks of complete enumeration algorithms and eliminating it as often as possible is desirable. The depth of the search tree can be examined by looking at the number of cells in a completed table that were fixed by looping, as opposed to the number of cells that were fixed automatically. Hence, a tree with deeper branching will have more cells fixed by looping. There may be concern that this contradicts the idea of automatically fixing as many cells as possible. Yet, for two enumeration trees of the same model, the number of success leaves is going to be the same. What is different is the number of failed leaves and the number of nodes in the tree. The main aim is to reduce the number of failed leaves.

In the $3 \times 3$ example, the number of nodes in the trees are 27 and 18, unordered and ordered respectively. Even if the nodes in the unordered table that do not produce any tables are discounted, it is still 21 and 18. This is due to the unordered tree having to support a larger number of failed leaves. Whilst trying to minimise the number of nodes, the other factor to take into account is the number of branches each node has, i.e., the range of the loop. Smaller ranges naturally mean that more nodes will be required. In effect the re-ordering of the table produces more efficient trees, implying more efficient computation

### Constraint Re-ordering

Similarly, re-ordering of the constraints can also yield some slight performance benefit. Here the focus is on how the algorithm searches through constraints. When testing if a cell is at a valid value, all the constraints that involve that cell are tested. If a constraint is immediately violated, then the loop through the constraints is prematurely terminated. Hence, if the constraints that restrict the elements the most are tested first, then less searching will need to be done. Whilst this is only minor reduction in computation, since this part of the algorithm is called so frequently, an appreciable reduction in runtime can be achieved.

The process of ordering the constraints is complicated somewhat by the fact that the optimisation needs to occur across all the elements. By looking at the ordering of the constraints on an per-element basis, partial orderings of the constraints can be obtained. These partial orderings need to be recombined into the total ordering. If the information used by the sorting function is based on just the information contained in the constraint, then conflicts between partial orderings will not occur. Whereas, if the ordering is based on information that can be different for each cell being examined, then this can lead to partial orderings that conflict. The total value of the constraint could be used as an ordering function. This does not to take into account how many elements form a constraint. For example, a constraint involving two elements would be more constraining than a constraint of 5 elements with the same constraint total, since fixing one of the elements immediately fixes the other cell. So this would imply a sorting favouring constraints with smaller numbers of cells. Conversely, a constraint with a relatively low constraint total would be more constraining on a constraint with 5 elements as opposed to 2 elements, since the number of automatically fixed cells is higher, i.e., this would favour constraints with larger numbers of cells. The common theme seems to be to sort on the probability of automatically fixing cells.

### Dynamic Ordering

The basic algorithm could be changed slightly to attempt to make use of element-wise optimisation. As the enumeration progresses, instead of selecting the next unfixed cell in the vector, the algorithm could select the next cell based on the ordering discussed previously. This would require the extra computation to order the remaining elements and this may prove to take longer than the time saved, given the high frequency that it would be used. An advantage of this is that it takes into account the current status of the computation and is likely to perform better at pruning than the static ordering of the elements before the enumeration starts. This form of dynamic ordering could also be used on the constraints, but again, it is unclear if the trade off is worth it. The constraints could be re-ordered once a cell is successfully fixed. Both of these approaches are likely to be beneficial when there are not very many cells fixed, i.e., when pruning is likely to remove large amounts of computation. The problem occurs towards the end,

Sun SPARCstation 10 Model 31

| Ordering | Table A | | Table B | | Table C | | Table D | |
|---|---|---|---|---|---|---|---|---|
| None | 0.54 | – | 17.48 | – | 2874.38 | – | 11303.97 | – |
| Constraints | 0.52 | 3.7% | 17.22 | 1.5% | 2814.93 | 2.1% | 11242.10 | 0.5% |
| Elements | 0.46 | 15.4% | 14.62 | 16.4% | 2601.65 | 9.4% | 10941.10 | 3.2% |
| Both | 0.49 | 9.3% | 14.67 | 16.1% | 2596.40 | 9.7% | 10165.39 | 10.1% |

SP2 node

| Ordering | Table A | | Table B | | Table C | | Table D | |
|---|---|---|---|---|---|---|---|---|
| None | 0.26 | – | 3.68 | – | 616.21 | – | 2771.61 | – |
| Constraints | 0.09 | 65.4% | 3.00 | 18.4% | 534.38 | 13.7% | 2283.57 | 17.6% |
| Elements | 0.16 | 38.5% | 2.20 | 40.2% | 314.94 | 48.9% | 868.07 | 68.7% |
| Both | 0.12 | 53.8% | 2.08 | 43.5% | 317.24 | 48.5% | 831.91 | 70.0% |

150MHz Pentium

| Ordering | Table A | | Table B | | Table C | | Table D | |
|---|---|---|---|---|---|---|---|---|
| None | 0.06 | – | 1.94 | – | 335.24 | – | 1461.28 | – |
| Constraints | 0.06 | 0% | 1.88 | 3.1% | 327.47 | 2.3% | 1437.05 | 1.7% |
| Elements | 0.07 | -16.7% | 1.29 | 33.5% | 266.96 | 20.4% | 535.86 | 63.3% |
| Both | 0.05 | 16.7% | 1.29 | 33.5% | 270.40 | 19.3% | 536.32 | 63.3% |

Table 4.2: Timings of matrix ordering optimisations, in seconds.

when there are few cells remaining, and it is quicker to complete the table normally instead of adjusting the orderings. The optimisations could be switched on and off, at appropriate points through the computation. This requires knowing when the optimal point occurs, which is likely to depend on the model matrix being used. A reasonable attempt may be to turn the optimisation off when a certain percentage of the table has been fixed.

Optimising the constraints is going to perform better on models where there are large numbers of elements in a constraint, as is typically found with large tables. This is because the amount of time spent calculating the current value of a constraint is higher. Conversely, with the dynamic approach the cost of ordering the constraints is going to be more significant.

### Timings

Table 4.2 presents timings for the calculation of exact p-values for various example tables, with non-ordered, ordered elements and ordered elements plus ordered constraints model matrices. These figures were obtained with static pre-enumeration ordering running on a Sun SPARCstation 10 Model 31, one node of an SP2 and a 150MHz Pentium desktop machine. The table shows that the significant speed-up is obtained from the re-ordering of the elements. Constraint ordering provides little speed-up and in the cases of the small tables, actually runs slower. Notice that for contingency table "D" the improvement due to ordering the constraints is much larger than ordering the elements. This is due to a large proportion of the elements in the table being of similar value and the values of the sufficient statistics being tightly grouped. This results in the search space being more homogeneous, and so less work can be pruned by re-ordering.

Notice the somewhat better speed-up obtained from the SP2 node and Pentium machine, as compared to the Sun machine. This is likely to be due to a larger memory cache enabling more efficient caching of the model matrix. The element ordering is the more important optimisation, with the constraint ordering making little difference. This is due to the small number of constraints that each cell of the table is involved with.

### 4.6.4 Sparse Optimisation

One of the time consuming processes of the algorithm is searching through the model matrix. The search takes two forms. Firstly, given a certain element, all the constraints that involve that element need to be found. Secondly, given a constraint, all the elements involved in that constraint need to be found. The model matrices are generally quite sparse and so these searches through the model matrix are time consuming. This is obviously going to slow down the enumeration of tables with large numbers of cells and constraints.

To avoid repeated searching through the sparse matrix, two indexes are generated. The indexes list for each cell which constraints the cell is involved in and for each constraint which cells it contains. Hence, where previously the algorithm would loop through all

Sun SPARCstation 10 Model 31

| | Table A | | Table B | | Table C | | Table D | |
|---|---|---|---|---|---|---|---|---|
| Original | 0.49 | – | 14.67 | – | 2596.40 | – | 10165.39 | – |
| Sparse | 0.18 | 63.3% | 3.52 | 76.0% | 537.40 | 79.3% | 1442.51 | 85.8% |

SP2 node

| | Table A | | Table B | | Table C | | Table D | |
|---|---|---|---|---|---|---|---|---|
| Original | 0.12 | – | 2.08 | – | 317.24 | – | 831.91 | – |
| Sparse | 0.05 | 58.3% | 1.25 | 39.9% | 171.05 | 46.1% | 528.86 | 36.4% |

150MHz Pentium

| | Table A | | Table B | | Table C | | Table D | |
|---|---|---|---|---|---|---|---|---|
| Original | 0.05 | – | 1.29 | – | 270.40 | – | 536.32 | – |
| Sparse | 0.07 | -40.0% | 0.90 | 30.2% | 136.53 | 49.5% | 394.36 | 26.5% |

Table 4.3: Timings of sparse optimisation.

the cells or constraints, a simple lookup in an array will give the relevant information.

Timings of this optimisation can be see in Table 4.3. This optimisation is effectively converting the model matrix into a sparse array, realising a significant speed-up in the code for all but very simple cases. In the simple case, the added level of look up in the indexes can be less efficient. Additionally, there is an overhead in setting up the indexes and lists of cells. This optimisation performs the best on the Sun workstation than on the other two machines, which have larger memory caches. The less memory intensive nature of this optimisation favours the machine with smaller memory cache, since more of the model matrix can reside in the cache than was the case without the sparse optimisation. For the larger cache processors, it is possible that most, if not all, of the model matrix resided in the cache, for the non-sparse algorithm.

### 4.6.5 Matrix Compilation

The current enumeration algorithm is generic in that it works with an arbitrary model matrix. The algorithm spends a large proportion of time examining the model matrix. These processes are to find element membership in a constraint, as well as operations such as adding up the current total of a constraint. Constructing a cache of the model matrix goes some way to making these operations efficient.

Sun SPARCstation 10 Model 31

| | Table A | | Table B | | Table C | | Table D | |
|---|---|---|---|---|---|---|---|---|
| Generic | 0.18 | – | 3.52 | – | 537.40 | – | 1442.51 | – |
| Compiled | 0.13 | 27.8% | 3.11 | 11.6% | 468.42 | 12.8% | 1230.87 | 14.7% |

150MHz Pentium

| | Table A | | Table B | | Table C | | Table D | |
|---|---|---|---|---|---|---|---|---|
| Generic | 0.07 | – | 0.90 | – | 136.53 | – | 394.36 | – |
| Compiled | 0.05 | 28.6% | 0.94 | -4.4% | 126.08 | 7.7% | 374.08 | 5.1% |

Table 4.4: Timings of matrix compilation optimisation.

Since the model matrix does not change during enumeration, a further method would be to compile the model matrix into the program. This avoids the need for matrix lookups. Although this is never going to be as efficient as a program hand-coded for a specific model, it should lead to a speed improvement. In the standard algorithm, the model matrix is read in, simplified and then ordered. This process is repeated in the matrix compilation program.

The function targeted is `check_constraints`, which checks that setting a certain cell at a certain value does not violate any other constraints. The model matrix is read in by the matrix compiler and the resulting C code output completely replaces the original generic `check_constraints`. The code used is based on the previous version of the matrix enumeration algorithm, but with parts of the code examining the model matrix replaced. The code to add up the current total of a constraint becomes a `switch` statement on the constraint number. Each `case` statement lists each cell in that constraint, checking if it is fixed, and if so, adding its value to the total and incrementing a counter indicating the number of fixed cells. Similarly, the algorithm searches the model matrix to finding out which constraints a given cell is in. This can be replaced with a switch statement, based on the cell number, and each case has the required code for each of the relevant constraints.

Table 4.4 presents the timings of this matrix compilation technique. Both routines utilise the model matrix reorganisation and the sparse optimisation discussed previously. This technique seems to perform less well on the Pentium machine. This could be due to less efficient compilation of the generated C code or as a result of already fairly efficient enumeration in the non-compiled version.

### 4.6.6 State Capture

One of the drawbacks of complete enumeration is that some problems take an extremely long time to run. If the state of the computation could be captured, then a program can be terminated and then resumed by reloading the saved state.

In order to capture state, all the runtime information vital to the program has to be found, this is called *reification*. This runtime information is both the data and variable space used by the program, as well as information relevant to the current position in the code. Later on methods of capturing state are discussed, and the idea of user-level state generation is proposed.

In order to capture state, the program is written in such a way that all the static and scope based information usually found in the program stack, are moved into user data structures. The impact of this is that constructs such as for-loops, which contain their own state, have to be transformed to explicitly use while loops with their own looping variable.

In the situation of our proposed enumeration algorithm, the recursive nature of the routine requires a number of such transforms. Since there is a for-loop from 0 to the maximum value, these values have to be explicitly stored. The problem is compounded by the fact that the routine is recursive and so a single data structure for the loop cannot be used. Instead an array of the relevant information has to be maintained, each level of recursion using a different element in the data structure. In effect, the traditional role of the program stack is being mimicked. It is then relatively straightforward to write out all the state information that is required, including the loop array state and all the variables used.

## 4.7 Fitted Values

Having discussed an enumeration algorithm for a general model matrix, the next thing that is required is the fitted values, in order to calculate the test statistics.

The iterative proportional fitting (IPF) algorithm can be used to find the maximum likelihood estimates of the fitted values in log-linear models. The IPF algorithm is very simple to program and it is very useful for fitting log-linear models with a large number of parameters. While other algorithms have quadratic convergence properties, as compared to IPF's linear properties, for models with a large number of parameters, their memory requirements become quite high. Whereas IPF's modest storage requirements make it reasonable choice. The most general form of the IPF algorithm is applicable to the general model matrices, including continuous covariates.

The IPF algorithm can be described as follows. Starting with initial values of 1 for all the cells, each iteration examines every constraint in the model matrix. For each constraint, the current fitted values are used to calculate a scale factor. All cells that are involved in the constraint are then multiplied by this scale value. The process is repeated until the observed and fitted sufficient statistics are sufficiently close.

The scale factor is determined so as to increase the likelihood. For a model matrix consisting of zeros and ones, the scale factor is given by the ratio of the sufficient statistic of the constraint to the current total of the constraint given the current fitted values. In the case where the model matrix has cell values other than zero and one there is no direct estimate of the scale value. Instead the problem decomposes to a one-dimensional maximisation problem.

Other fitting methods such as Iteratively Reweighted Least Squares or a Newton-Raphson approach could be used but in most cases IPF is sufficient. Alternatively, a package such as GLIM could be used to generate the fitted values.

```
data
  1 1 3 2 2 0 3 1 0 0 0 5 0 2 1 2
vectors
  a 4 1
  b 4 4
fixed 1 6 11 16
model a + b
```

Figure 4.2: An example model specification.

## 4.8 Model Matrix Generation

The input to the enumeration program is the model matrix followed by the observed table. The model matrix object consists of an initial header which indicates the number of constraints (rows) and the number of elements (columns). The model matrix object also contains the sufficient statistic for each constraint. This is simply for compatibility, and these are recalculated from the observed table. The model matrix is not really designed to be used by a user. Except in very simple cases, the model matrix is too large to be easily constructed manually. Hence, methods are needed to specify the model and automatically generate the model matrix. Indeed, any number of different methods and programs can be used, as long as their output is of the form outlined above.

Possibly the most useful and powerful method would be simply to specify the model formula. A simple language can be designed to enable this. Figure 4.2 shows an example of specifying the model of quasi-independence for a $4 \times 4$ table with fixed diagonal cells, in the simple language. Firstly, the data are specified. This implies the number of columns of the model matrix. Following the keyword `vectors` is a list of identifiers. Each identifier has either the number of levels and repetitions of each factor, or a list of the cell numbers in the interaction. In this example, `a` denotes the row term and `b` the column term. There are 4 fixed cells as specified by the `fixed` keyword. Finally, the model formula is specified. All the usual model formulae operators `+`, `-`, `.` and `*` can be used. The language specifies models in a similar way to GLIM.

This input format is relatively straightforward to parse and evaluate. This can be achieved with the standard tools `lex` and `yacc`. `Lex` is a tool that creates a piece of code that will take an input file and *tokenise* it. Here, there will be tokens for the keywords as well as numbers, variable names and the mathematical operators. `Yacc` is typically used in conjunction with `lex`. Again, it creates a piece of code that takes a stream of tokens from `lex` and parses them as specified by a grammar, producing a parse tree. As the input file is parsed, global structures storing the required information are created. The variables specified in the `vectors` block contain the specification of the number of levels, $n$, and the number of repetitions in each factor, $r$. This is translated into a matrix with $n$ rows, with a value of one placed in each column. The row number for placing of the value for the $i$th column is given by $1 + i \bmod r$.

The model formula is parsed, constructing a parse tree. Each node in this tree contains the operator, and pointers to the two operands. An operand can be either a variable name or an expression tree. In order to evaluate the model formula, the expression tree is traversed. At each node, the left-hand operand is evaluated followed by the right-hand operand. The operator is then applied to the two resulting matrices. In the case of the operand being a variable name, the variable's associated matrix is copied from the look-up table. If the operand is another expression tree, then this is evaluated. Operator precedence, as well as parentheses, are defined in the grammar specification. Once the whole expression has been evaluated, the resulting matrix can be printed, with the relevant header, followed by the observed data. This output can be then passed directly to the enumeration program.

This language needs to be extended so that more complex models such as uniform association (i.e., model matrices with some elements greater than 1) can be specified.

## 4.9 Examples

This section looks at a number of example tables and models. The aim is to give a range of different examples illustrating the capabilities of the algorithm discussed in this chapter. One of the most important factors that influences complete enumeration is the "size" of the table and the size of the resulting reference set. Whilst some tables are much larger than others, models with many constraints significantly reduce the size of the problem.

### 4.9.1 Sexual Fun

Re-running the Sexual Fun data described in Chapter 2 and Table 2.4 with the models of quasi-independence and quasi-symmetry, as described in Chapter 3, yields significant reductions in run time. For quasi-independence, the model matrix method took 2.60 seconds, compared with 11.23 seconds for the recursive method using a Sun SPARC-station 10 Model 31. This significant difference is likely to be due to the more efficient ordering of elements and more efficient calculation of the test statistic.

For quasi-symmetry, the enumerate-and-reject algorithm took 3.67 seconds, having enumerated 15,708 tables and rejected 15,547 to give the 161 tables in the reference set. The model matrix method took 0.11 seconds. Note that the model matrix method calculated the fitted values, whereas the enumerate-and-reject algorithm read the values in from a file. This is not a significant problem since the calculation of the fitted values takes just 50ms. The model matrix method performs the test over 30 times as quickly as the enumerate-and-reject algorithm. The exact p-values for both $X^2$ and $G^2$ are 1.000.

### 4.9.2 Religious Mobility

Breen and Hayes (1996) analysed recent data from both Great Britain and Northern Ireland on religious affiliation. The data, shown in Table 4.5, cross-classifies current religion with the religion of the respondent's parents (origin religion). Most of the observations lie on the main diagonal, indicating no mobility. Breen and Hayes proposed a number of models to examine mobility between the categories. The models of QI and Marginal Homogeneity (MH) were rejected. The model of QS provided a much better fit. In addition, a model called QI+PS, which consisted of quasi-independence and symmetry between cells in the last row and column (Partial Symmetry), was also considered. The extra constraints for this model are that the diagonal pairs (as with QS) in the final row and column sum to the same value as in the observed table.

| Origin religion | Current religion | | | | | |
|---|---|---|---|---|---|---|
| | Catholic | Anglican | Protestant | | | None |
| | | | Mainline | Fundam. | Other | |
| (a) Great Britain | | | | | | |
| Catholic | 123 | 2 | 0 | 0 | 1 | 48 |
| Anglican | 10 | 420 | 9 | 1 | 4 | 217 |
| Mainline | 2 | 21 | 102 | 1 | 5 | 54 |
| Fundamentalist | 0 | 8 | 2 | 15 | 0 | 6 |
| Other | 0 | 4 | 0 | 0 | 7 | 5 |
| None | 1 | 3 | 0 | 1 | 1 | 62 |
| (b) Northern Ireland | | | | | | |
| Catholic | 226 | 1 | 1 | 0 | 1 | 13 |
| Anglican | 5 | 137 | 24 | 5 | 3 | 20 |
| Mainline | 2 | 23 | 213 | 9 | 9 | 22 |
| Fundamentalist | 0 | 3 | 2 | 11 | 2 | 4 |
| Other | 0 | 2 | 2 | 1 | 7 | 1 |
| None | 0 | 0 | 1 | 1 | 0 | 7 |

Table 4.5: Breen and Hayes religion data.

| | Results for Great Britain | | | Results for Northern Ireland | | |
|---|---|---|---|---|---|---|
| | Asymptotic | Complete Enumeration | | Asymptotic | Complete Enumeration | |
| Model | p-value | p-value | set size | p-value | p-value | set size |
| QI | 0.0007 | – | – | 0.0243 | – | – |
| QS | 0.0670 | 0.06507 | 15,251 | 0.5987 | 0.80875 | 9,739 |
| QI+PS | 0.1258 | 0.07513 | 39,335,368 | 0.5487 | 0.26500 | 11,472,421 |

Table 4.6: Results from table 4.5

The asymptotic results are shown in Table 4.6 alongside the results from complete enumeration. Although the results show that the asymptotic results gave the correct answer at the 5% level, there are still some differences in the distributions, notably QI+PS in Northern Ireland. Complete enumeration of the QI model is probably infeasible in this case and QI+PS was obtained by using the distributed enumeration algorithm, detailed in Chapter 6.

### 4.9.3 Stroke Data

Re-running the stroke example from the previous chapter, the model matrix method took 97.75 seconds compared to the 109.12 seconds for the recursive quasi-independence method. The timings emphasise the more efficient enumeration of the model matrix method. This is due to the ordering mechanism and cell by cell enumeration which allows the more flexible selection of cells compared to the relatively ill-suited row by row approach of the enumeration algorithm for quasi-inpendence from Chapter 3, to this problem.

The model of uniform association took 76.57 seconds on an SP2 node using the model matrix method, compared to the 96.47 seconds of the enumerate-and-reject method. The rate of table generation for the model matrix enumeration is much lower than for the model of quasi-independence. This is due to the uniform-association constraint that involves every element in the table. This is effectively a rejection step occurring every time a cell is fixed, resulting in a significant overhead. This suggests that a more efficient enumeration, for this type of model, maybe to use the enumerate-and-reject approach of enumerating under QI, using the matrix model method, and only apply the uniform-association constraint to reject tables at the end.

### 4.9.4 Intermarriage Table

Table 4.7 shows the distribution of marriages by ethnicity of each spouse among first-generation US immigrants for eight ethnic groups. The table shows the three blocks: British and Irish, Old Northern European and Jewish, referred to as BI, O and J. Smith *et al.* (1996b) discuss a number of models for this table, including QI, QI+UA, QI+J, QI+J+BI, QI+J+BI+O and QS [1]. They use Monte Carlo methods to estimate the p-values. Due to the size of the table, only the QS model can be completely enumerated, all the other models are sufficiently large to make them infeasible to calculate in reasonable time. The model of QS gives 327,766 tables in the reference set, and an exact p-value of 0.062558 based on $G^2$. This is close to the estimated exact p-value of 0.0602 that

[1] Note that the published value of 0.5774 for the QS asymptotic p-value was calculated on the incorrect degrees of freedom (21 instead of 13).

| Husband's Ethnicity | Wife's ethnicity: British | Irish | Scandi-navian | German | Italian | Polish | European Jewish: Central | European Jewish: Eastern |
|---|---|---|---|---|---|---|---|---|
| British | 314 | 63 | 10 | 15 | 0 | 1 | 1 | 0 |
| Irish | 27 | 625 | 2 | 5 | 0 | 0 | 0 | 0 |
| Scandinavian | 4 | 9 | 835 | 20 | 1 | 0 | 0 | 0 |
| German | 26 | 26 | 10 | 1096 | 0 | 4 | 0 | 0 |
| Italian | 3 | 6 | 0 | 4 | 477 | 1 | 0 | 0 |
| Polish | 1 | 0 | 0 | 7 | 0 | 421 | 0 | 0 |
| Central European Jewish | 1 | 0 | 0 | 1 | 0 | 1 | 112 | 11 |
| Eastern European Jewish | 1 | 0 | 0 | 1 | 0 | 1 | 30 | 347 |

Table 4.7: Husband's by wife's ethnicity for all immigrants married in the USA. 1910 public use sample: source, Pagnini and Morgan (1990).

Smith *et al.* calculated, and is within their 99% confidence interval.

Figure 4.3 shows the distribution of the likelihood ratio statistic under the model of QS. This highlights the major discrepancy between the asymptotic p-value and the exact p-value.

### 4.9.5 Job Satisfaction

Figure 4.4 shows the model specification, in the model language detailed earlier in this chapter, for the model of no 3-way interaction on the job satisfaction data detailed earlier in this thesis (Table 3.4). The enumeration of this example took 2.00 seconds, compared with 19.35 seconds for the Scheme implementation of the recursive $n$-way algorithm. There are 6,597 tables in the reference set. Agresti (1996) also looks at the model of $G.I + G.S$, which removes the $I.S$ constraint. This test is a stratified test, which checks for independence in the two slices of the table. The number of tables in each slice is 1,500,930 and 50,617 for the female and male slices respectively. Hence, the model $G.I + G.S$ contains 75,972,573,810 tables in the reference set, which are obtained by combining every table in the first slice to every table in the second. The exact p-value can be calculated in a similar multiplicative method. A list of all the possible tables and its values of $X^2$, $G^2$ and probability are maintained for each slice.

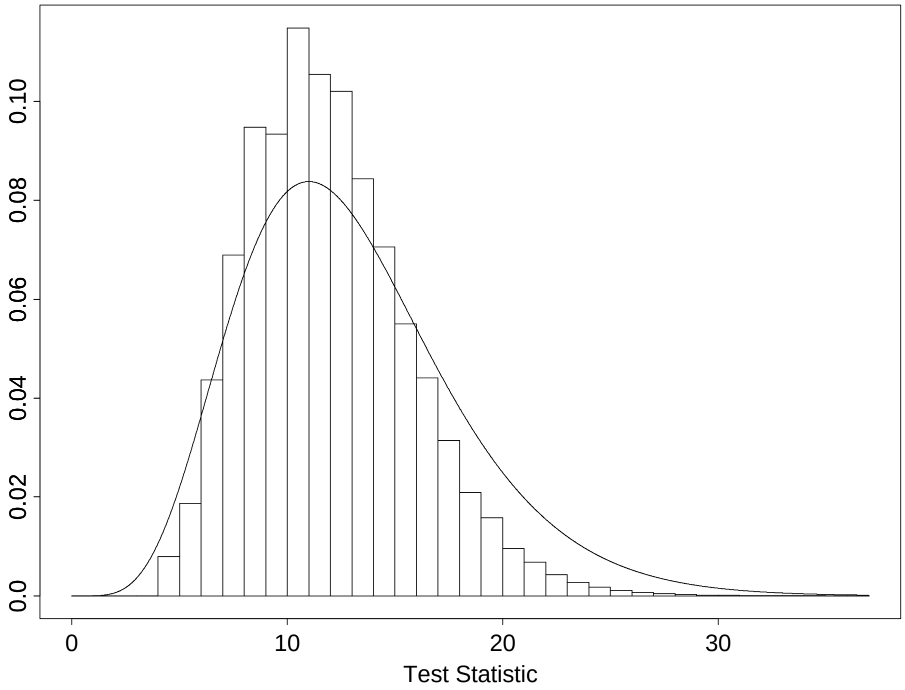


Figure 4.3: Asymptotic and exact distributions for $G^2$ for model of QS for Table 4.7.

```
Data
  1 3 11 2   2 3 17 3   0 1 8 5   0 2 4 2
  1 1  2 1   0 3  5 1   0 0 7 3   0 1 9 6

Vectors
  S 4 1
  I 4 4
  G 2 16

Model
  G.I + I.S + G.S
```

Figure 4.4: Model specification for no 3-way interaction on job satisfaction data.

### 4.9.6 Hierarchical Models in Multidimensional Tables

Morgan and Blumenstein (1991) presented an exact conditional test (ECT) approach to the analysis of multidimensional tables using hierarchical models. Their algorithm allows the testing of a "full" model against a "reduced" model. The ECT algorithm examines the full model by conditioning on the reduced model.

In practice, the test involves generating tables under the reduced model. For each generated table, the sufficient statistics of the constraints present in the full model, but not the reduced model, are examined. Tables are grouped based on their sufficient statistics, since a number of tables may have the same sufficient statistics. The probability of each resulting sufficient statistic is found by summing the probabilities of the tables that fall into that group. The final p-value is calculating by summing the probabilities of groups which are less than or equal to that group in which the observed table lies. This test can be performed using the model matrix algorithm by using the routine to generate a list of all the possible tables and their table probabilities. The list can then be processed by a series of filters which calculates the sufficient statistics of the full model and accumulates the probability as required. Hence, this calculation can be performed in a separate process to the enumeration (in this case, written as a series of Perl and Awk scripts).

As an example, Morgan and Blumenstein present a $2^4$ table, with variables denoted by $H$, $I$, $R$ and $V$. The reduced model of $HRV + HI + IR + IV$ was tested against the full model of $HRV + HI + IR + IV$. The reduced model produces 951 tables, which fall into 7 groups of sufficient statistics, of which 3 are more or equally extreme as the observed. This gives an exact p-value of 0.030, which indicates that the $HRI$ interaction is important. The model matrix enumeration method is in agreement with these figures, giving a p-value of 0.02953. The reduced model of $HRV + HRI$ was also tested. This yields 3,780 tables, with 41 groups, 33 of which are more or equally extreme as the observed. Again, the resulting p-values are in agreement (0.126 and 0.12585). Estimates of these p-values have also been obtained by Markov chain Monte Carlo methods (Forster *et al.*, 1996).

Two other examples looked at two $3^3$ tables. Notably, the second example was much larger with 1,130,500 tables for the reduced model, and 3,984 groups of which 3,083 were more extreme. Again, the two algorithms were in agreement. This example demonstrates that the enumeration algorithm agrees with an existing independent algorithm for complex models. It also demonstrates that the enumeration routine can be used effectively in other complex tests, in a pipeline fashion.

### 4.9.7 Logistic Regression

Recent work (Bedrick & Hill, 1990; Agresti, 1992; McDonald *et al.*, 1996) has looked at applying exact techniques to logistic regression modelling. Logistic regression is used to model the probability of a binary response variable against as a linear function of a set of predictor variables. When the resulting data are sparse, the use of asymptotic approximations is questionable, and the exact approach should be used.

Bedrick and Hill (1990) looked at explicit enumeration of all the possible responses consistent with the sufficient statistics for the observed data. Their specialised enumeration algorithm, also recursive, traverses the search space by developing a set of moves that can be used, and repeatedly applies these moves. The first example was an experiment in which 53 prostrate cancer patients underwent surgery to determine whether they had cancer in their lymph nodes. The data consists of 5 binary predictor variables based on preoperative tests, and a variable indicating the presence of cancer. The model proposed was a main effects model, with 17 degrees of freedom and observed test statistics $X^2 = 15.46$ and $G^2 = 18.07$. With 6,034 possible response vectors, p-values of 0.803104 and 0.791733 for $X^2$ and $G^2$ were obtained, which are in agreement with the Bedrick and Hill's results. This calculation took 7.7 seconds on a SPARCstation 10 Model 30, which compares to 19 seconds for Bedrick and Hill's method, run on a slightly slower machine.

Their second example presented was based on a dose-response experiment of a drug administered to mice. The log of the dosage level was used as a predictor. The levels were approximately equally spaced. The enumeration finds all the possible response vectors with the same number of responses and the same sum of the response counts times the log dose. There are 1,637 possible response vectors and p-values of 0.013231 and 0.006415 for $X^2$ and $G^2$ respectively were obtained, again agreeing with Bedrick and Hill's results. This example ran significantly slower than the time quoted by Bedrick and Hill. This is due to the large range of the maximum value of the cells, as a result of relatively few constraints. This requires a much larger search than is expected with the small number of tables in the reference set.

McDonald *et al.* (1996) use a Metropolis-Hastings algorithm to estimate the p-values of second example.

### 4.9.8 Symmetry Models

Von Eye and Spiel (1996) examined how to specify symmetry models. The model of axial symmetry, or complete symmetry, is similar to that of quasi-symmetry except the row and column totals are not maintained. The only constraints maintained are the sums of the symmetrically opposite cells in the table. The paper defines the required interactions using vectors in a model matrix, in the same way as the models are specified in this chapter. Their resulting model matrix has fewer vectors than the equivalent used in this thesis. This is due to the full specification required by the enumeration algorithm.

A number of simple examples are presented, the p-values of which can all be calculated by complete enumeration. The first example consists of a $3 \times 3$ with a total of 176 observations. With just 3 pairings, there are only 1,820 possible tables. The larger cell counts result in the asymptotic p-value being close to the true exact p-value.

The second example consists of a $2 \times 3 \times 3$ table showing the popularity of children, assessed at two points in time, for two different groups. The model of symmetry is tested within the two $3 \times 3$ tables, giving six pairings. Due to the sparseness of the data, the use of asymptotics is questionable. The authors quote an asymptotic p-value of 0.107 based on 6 degrees of freedom, whereas complete enumeration gives a p-value of 0.043932, with 50,048 tables in the reference set. Closer examination of the table reveals that for one of the six pairs, both cell counts are zero. This results in there only being 5 degrees of freedom, giving an asymptotic p-value of 0.063564. This highlights one of the significant problems with using asymptotics, in that for complex models the calculation of the number of degrees of freedom is not straight forward. Two other models are fitted to this data, based on extending the models to incorporate a trend hypothesis. For both these models, the number of degrees of freedom are wrong. Using the correct degrees of freedom, asymptotic p-values of 0.162730 and 0.237977 are obtained. Complete enumeration of the models gives p-values of 0.257615 and 0.314282 respectively.

The final example looks at a quasi-symmetry model on a $3 \times 3$ table. Complete enumeration gives an exact p-value of 0.146334, which suggests a slightly better fit than the asymptotic p-value of 0.085.

The important feature of the symmetry models is that there is no other interaction between the cells outside of the pairings. This means that the reference set consists of combinations of each of the possible values for the pairings. Currently the enumeration algorithm does not exploit such structure in the model, and performs a lot of repeated work. For small examples, such as those in this section, this is not a significant problem, but for larger problems the naive enumeration algorithm becomes infeasible. By exploiting the structure of the model, such larger models may become feasible.

## 4.10 Further Work

#### Linear Diophantine Equations

The class of problem that this chapter has being trying to solve are systems of non-homogeneous Diophantine equations. Much work is present in the literature about solving these problems. Contegean and Devie (1994) looks at a promising algorithm to find all solutions to the more general case of linear Diophantine equations. This algorithm searches for minimal solutions to the system of equations. The entire solution set can be found by combining minimal solutions. The timings from this paper suggest that the speed of their algorithm does not compare well to the speed of the model matrix enumeration algorithm. Further work could look more closely at the algorithms and techniques used in the area and apply them to the special case of enumerating all possible tables subject to linear constraints.

### Upper and Lower Bounds

Currently the model matrix enumeration algorithm assumes that the lower bound of a given cell is zero. This is not always the case, and in these cases the algorithm will spend time searching in part of the space that yields no solutions. The upper bound is given by a crude function, the minimum of all the sufficient statistics of constraints to which the cell belongs. The advantage of using these two bounds is that they are quick to calculate. This performs well for small or medium sized problems, but for larger problems, extra time could be spent to limit the search space further. Better estimations of the ranges will typically take longer, so there is a trade off between the time that the range estimation function takes, and the amount of search time the improved ranges save.

Further work could look at investigating better performing range estimation routines to optimise the search space.

### Repeated Work

Chapter 2 looked at a basic algorithm for enumerating under independence and identified the problem of repeated work. The problem was significantly reduced by an alternative method of enumeration, relying on storing of portions of tables.

Further work could be done in applying this technique to the more general case of enumerating from a model matrix.

### Exploiting the Structure of the Model

Consider the model of conditional independence in a $r \times c \times k$ table where the row and column variables are conditionally independent in each of the $k$ strata. This model, as well as the model of symmetry in a $r \times r$ table, do not enumerate efficiently with the algorithm. The structure of these models is such that parts of the table can be enumerated independently and the test statistics and table probability recombined. Enumeration of these models under the basic algorithm would result in the independent parts being enumerated repeatedly.

Spotting this structure from the model matrix is not always straightforward and it is maybe easier to examine the model formula or graphical model (Whittaker, 1990). Further work could look more closely at this problem and produce an algorithm for more efficient enumeration.

### Simultaneous Models

When analysing a table, the user may want to test a number of models. These models often share a number of constraints, for example, quasi-independence and quasi-symmetry. The intermarriage example contained a number of nested models, which share a number of constraints. An enumerate-and-reject algorithm could certainly be used in this situation, but this may not be the case for all sets of models that the user may want to test.

Further work could look at the issues surrounding this problem and the enumerate-and-reject method.

### Extreme Tables

Joe (1988) finds the extreme tables in the reference set under models of independence, where extremeness is based on the test statistic. The ability of being able to find these tables is a particularly useful one, for example, in checking whether a partially generated table will yield any tables that are more extreme than the observed table. Finding extreme tables under an arbitrary model is a complex problem and further work could be done in this area.

### Counting Tables

Algorithms are available to calculate (Mount, 1995) or provide an estimate (Good, 1976; Gail & Mantel, 1977; Agresti *et al.*, 1979) of the number of tables that exist for a the model of independence. This enables a decision to be made about the size of the computation required and the feasibility of complete enumeration. The ability to determine or estimate the size of the reference set would be most useful for arbitrary models.

#### Network Algorithm

The network algorithm (Mehta & Patel, 1983) is a well known approach to calculating an exact p-value for a table under independence. The approach used is to transform the problem of finding all tables into finding all possible paths through a directed acyclic graph of a given length. Further work could establish if this approach could be applied to the general problem of a model specified by a model matrix.

#### Monte Carlo Simulation

Significant research (Smith & McDonald, 1995; Smith *et al.*, 1996b; Forster *et al.*, 1996; McDonald & Smith, 1995; Smith *et al.*, 1996a; McDonald *et al.*, 1996) has looked at estimating exact p-values by Monte Carlo techniques. These techniques can currently be used on a limited number of models. Simulating from more complex models proves to be a significant challenge. Further work could examine the problem more closely, and look to see if some aspects of the algorithm for complete enumeration presented here, can be used to extend the range of models for simulation.

## 4.11 Conclusions

This chapter has addressed the problem of calculating exact p-values by complete enumeration for complex models. The algorithm is a significantly more efficient than that used in the previous chapter. The algorithm is generic in that it takes a model specification as input. This allows the calculation of the exact p-value for a general log-linear model, extending the range of models and data sets that can be analysed by complete enumeration.

The basic algorithm was optimised in various ways to address specific areas of inefficiency. These optimisation have made a significant improvement in the speed of enumeration. This extends the size of models that are feasible. As with all complete enumeration approaches, the number of unfixed cells and their counts have to be generally small, otherwise the search space becomes too large. The examples tested give an idea as to the size of problems that are feasible. The size of the search space also depends on the model, and some models such as quasi-symmetry place a large number of constraints on the cells and significantly restrict the search space.

The logistic regression example identified that the algorithm developed is less efficient on certain types of models found in this class of analysis. Whilst logistic regression is an important area, it is not the main focus of the problems tackled by this thesis.

# Chapter 5

# Distributed Computing

## 5.1 Introduction

This chapter serves as an introduction to distributed computing. It motivates why distributed computing may be necessary and how it compares with parallel processing. The different types of machines that can be found on the network are discussed, as well as the issues raised by using a heterogeneous environment. A section on programmer support for distribution details what languages and libraries can be used to construct a distributed system.

## 5.2 Motivation

In the modern computing environment the network plays a vital roll. It allows computers to be connected together creating rich environments for the user. Yet, the technology to exploit this environment to its full potential is still relatively young in computer science terms.

The main theme of distributed computing is to allow the user to utilise the resources found on the network. These resources may include processing power, storage capacity, specific data, and other hardware such as input and output devices. This thesis mainly deals with processing power as the vital resource, but other applications may focus on other resources, such as information sources for multimedia.

The motivation for harnessing the available processing power on the network is simply to increase performance and obtain results faster. Consider a typical local area network (LAN), where many machines on the network will be idle for significant portions of time. If these wasted processor cycles could be utilised, they could represent a significant processing resource.

## 5.3 Distributed Computing vs Parallel Processing

This section compares distributed computing to parallel processing, which seem to have similar ambitions as far as processing and performance are concerned.

The main difference is that parallel processing is tightly focused on specific architectures and machines. Distributed computing takes the wider view of the whole network and large parallel machines become just another available resource. There is, however, some common ground between the two areas in aspects such as programming libraries and problem decomposition.

The distinction between the two areas is becoming more blurred. Historically, parallel processing was involved in large vector-based machines. More recently, the trend is towards processors connected together on a high performance switch, which looks, from a software point of view, similar to a group of machines on a network. Meanwhile, workstations are getting faster as are the networks connecting them. The technologies appear to be converging on some notion of a common platform or architecture. A "cluster" of workstations connected on a network is often more immediately available to a user than parallel machines. For example, a group of desktop computers in an office can be used as a cluster for distributed computing overnight.

## 5.4 Types of Machines

Classifying types of machines found in the parallel world is traditionally achieved with Flynn's taxonomy (Flynn, 1972). This defines 4 categories:

- SISD – Single Instruction Single Data
- SIMD – Single Instruction Multiple Data
- MISD – Multiple Instruction Single Data
- MIMD – Multiple Instruction Multiple Data

SISD is a conventional sequential computer, where every instruction initiates an operation on a single piece of data. With SIMD, each instruction from the program initiates operations on multiple pieces of data. Typical examples of this form of parallel machine are array processors and vector processors. MISD presents the rather unusual situation of multiple instructions operating on a single piece of data. Finally, with MIMD, multiple instruction streams operate on their own pieces of data. The bulk of the parallel machines fall in this category.

The MIMD category is large and it needs to be examined more closely to get a better idea of the types of machines available on the network and how their features are different. All these machines are multiprocessor machines. The key distinction is how the processors and memory are organised.

*Shared memory* machines have a single bank of memory, to which a number of processors are connected. This is a relatively straightforward method of producing a multiprocessor machine. Particular attention has to be made to the caches present on each processor, which need to be aware of all the other caches in the system to avoid cache and memory inconsistencies. Although this method works well for a reasonable number of processors, typically eight or less, it does not scale to a large number of processors. This is mainly due to the memory bus, shared by all the processors, becoming saturated and causing a bottleneck. Faster and more complex bus architectures are constantly being developed. This causes larger machines to be disproportionally expensive. Multiprocessor shared memory machines are becoming increasingly popular in desktop workstations and file-servers with smaller numbers of processors, providing useful performance increases.

The other form of multiprocessor machines have local memory for each processor in the system. Some form of communication hardware allows nodes in the system to pass messages to one and another. Topologies of the links between processors vary greatly in these machines. The topology of the interconnects becomes critical to the performance of the machine. These machines are becoming increasingly popular in the high-end parallel processing world, mainly due to their architectures scaling better to large numbers of processors, e.g., a Cray T3D with 512 processors. One of the immediate benefits of having local memory for each processor is that the processor has sole use of the bandwidth to its memory, which can be a vital factor given that improvements in bus and memory performance are consistently out paced by that of processor design. Additionally, until very recently, the maximum memory size was restrictive, but for this class of machines each processing node can be equipped with that maximum memory, allowing much larger problems to be run on these machines than shared memory machines. In addition, the shared memory paradigm can be used on these machines by using distributed shared memory (DSM), providing there is support for it. These multiprocessor machines are very similar to a collection of workstations on an extremely fast network.

A further classification that is often used is single program multiple data (SPMD). This is not quite the same as SIMD, where the same instruction executes on all processing nodes. With SPMD, the same program executes on each processing node, operating on different data. This is suited to data parallel problems where the data decomposes into a number of pieces. The same executable code is loaded onto each node and assigned a section of the dataset.

## 5.5 Heterogeneity

Programming a distributed system raises an important issue not found when writing sequential applications. In a distributed system, there are a number of different types of processors, architectures of machines and operating system. Writing an application for a distributed system requires writing code for a heterogeneous environment.

The previous section outlined a number of different machine architectures. Writing an application to run effectively on such a diverse environment is far removed from the single target of a sequential application. The following sections look at the software support available to the programmer to aid with writing applications for a system of multiple machines. Firstly, the problems of writing applications for multiple types of machine are examined.

The issue of heterogeneity fall into two categories, code and data. With different types of processors on the network, there is no unified standard for code and data. Different processors have different byte orderings for data. A byte, being a 8 bit number can only represent 256 combinations. Sequences of bytes are used to encode integers and floating point numbers. In most cases, the difference between data representations is the ordering of these bytes. Some processors are "little endian", the least significant byte first, others are "big endian", with the most significant byte first. The ordering means that data cannot be exchanged between processors by just sending the sequence of bytes used by the processor to encode the data type, since they may not have the same encodings across processors.

In a similar way that there is no guarantee of data compatibility, there is no guarantee of code compatibility. The very nature of having a wide range of processors with differing features and capabilities, results in different and incompatible machine code. Code compiled for one type of processor will not run on another. Indeed, code compiled for one operating system typically will not run on another operating system, even if both operating systems run on the same type of processor.

In a parallel machine, the processing elements are (usually) identical. They run the same code, and have the same data encoding. Data can be passed between processors without encoding. Once a program is compiled, it can run on any of the processors. A heterogeneous distributed system has to be able address the problems of a code and data incompatibilities.

## 5.6 Support for Distribution

This section looks at some aspects of writing an application for a distributed system. Any distributed system has to have concurrency and communication between its components. This is usually achieved in one of two ways. Either a language is used that has been designed with writing distributed applications in mind, i.e., a language that has the syntax and semantics for distributed activities. Alternatively, libraries are used to provide functions to perform various distributed activities. Languages and libraries that provide support for distributed programming fall into two categories; those that make the communications explicit and those that hide the communication.

At the operating system level, on top of which all distributed systems have to be built, the user is provided with a number of low-level mechanisms. In a typical Unix environment, the operating system provides two communication mechanisms and a number of mechanisms for providing concurrency within the bounds of the machine. Some operating systems such as Amoeba (Tanenbaum & Mullender, 1981) and Chorus (Rozier *et al.*, 1991) have been designed from the bottom up to support powerful mechanisms to support distributed activities, such as distributed virtual shared memory and process migration. Unfortunately, these operating systems are not widespread in today's computing environment. This is possibly due to the systems not being portable across a wide range of architectures. Instead, this chapter focuses on systems and languages that can run on a wide range of operating systems and architectures since this increases the range of systems that can be incorporated in a distributed environment. Languages and systems that are able to run on a diverse number of machines need to rely on the operating system as little as possible, typically requiring some form of communication mechanism and a notion of a process. The disadvantage of this approach is that it means that more of the distributed system has to be written in a library or the language runtime. This is not a significant problem, unless it means that the system has to be implemented in a less efficient fashion.

Programming a distributed system at the low-level mechanisms provided by the various operating systems prove to be hard and unsuitable for application programming. Both libraries and languages aim to shield the programmer from the complexities of the distributed system by providing suitable abstractions.

The following sections look at the various areas that allow the programming of distributed systems. Firstly, a brief look at some of the types of languages that can be used. This is followed by a look at the technique of message passing and libraries that support it. Finally, other styles of distributed programming are considered such as remote procedure call (RPC), threads and distributed shared memory.

### 5.6.1 Languages

The argument for using a specific distributed language for writing distributed applications is that the syntax and semantics are much cleaner. The disadvantages are that it typically requires learning a new language and often the languages are not common across a large number of architectures and operating systems. This section looks at only a selection of languages as there are many languages that exist in this area.

Distributed languages typically fall in two categories, languages that are designed from the beginning for writing distributed applications, and languages that have distributed features extending existing sequential languages. All have a programming paradigm that encapsulates the concurrency and communication of the distributed system. Many object oriented languages use remote objects and remote method invocation, decomposing the application into a number of distributed objects, e.g., the Mentat system (Grimshaw, 1993). For procedural languages the system is decomposed into a number of concurrent streams of execution. The names and features of these units of execution vary, but are usually based on some notion of a process or thread. In these situations, the languages need to solve the problems of referring to processes, communication between processes and synchronisation as well as control across the processes.

#### Distributed Lisps

Lisp and its many dialects historically have a number of properties that make it particularly well suited to being used as a distributed language, most importantly the property of being able to treat code as data. This unusual property for a language means that code can be created and changed by a program. Perhaps more important in a distributed system is that converting code to data means that it can be transmitted to another machine and invoked; a form of migration. Another vital aspect in a distributed system is the issue of running programs on a number of different architectures. Since programs in Lisp languages are usually either interpreted or compiled to bytecode, it is independent of the underlying architecture of the machine. Lisp programs running on top of the runtime system and can be migrated from one runtime system running on one machine, to another runtime system running on another machine. The runtime system of the Lisp implementation is the only part of the Lisp system that needs to be ported to a new architecture. Code and data transparency is vital for a language to run well on a distributed system.

Distributed Lisp languages, typically try and solve a number of areas. Distributed garbage collection and side-effects are particular problems that these languages and systems seek to address. These systems are constructed usually on top of an existing language or library that provides concurrent and distributed facilities. For example, the aim of the Icsla language (Queinnec & De Roure, 1992) was to add a core set of features to the Scheme dialect of Lisp. These features were simple and easy to understand whilst not restrictive. Upon these features, more usable abstractions such as `future` (Halstead, Jr, 1984) can be developed. De Roure and Michaelides (1994) added communication primitives into of the Xlispstat, ontop of which similar abstractions as those used in Icsla could be built, with the added benefit of the statistical nature of the language.

### Orca

Orca (Bal, 1990) is an example of a language that was designed from the outset to be used for distributed programming. Designed as an applications language, it incorporates only high level language features for distribution. Low-level features that are only useful for systems programming are not available in the language. Sequential and distributed constructs have been designed to integrate well with each other. This contrasts well with languages which are based on sequential languages, and have distributed constructs added awkwardly on the side.

Orca is a strongly typed procedural language, somewhat like Modula-2. Concurrency in Orca is based on the explicit creation of sequential processes. A process in Orca is a procedure that runs in parallel. Processes can hence vary in size, from relatively small pieces of code right up to large components. Processes can be explicitly placed on a certain processor, by using a numbering system which is assigned at run-time. Communication between processes is achieved by the passing of shared data as arguments to process calls. Hence the data is shared between just the two processes, as opposed to other strategies such as the global shared memory of Linda (see following section). This simple communication mechanism makes systems written in Orca easier to write, read and understand.

#### Linda

Linda (Gelernter, 1985) provides a concept of a *tuple space*, which is globally accessible from all the programs running in the application. This effectively acts as a global blackboard with tuples read and written from it. The tuple space can be searched for matches, i.e. to look for specific messages that have been posted. This relatively simple message system proves to be a powerful mechanism for expressing communication within a parallel or distributed system. The tuple space effectively forms a global shared memory, allowing all processes running to access all the globally shared memory, a feature that can be both an advantage, in its simplicity, and a disadvantage, due to the care that needs to be taken by the programmer. Note that Linda is not bound to a specific language, so the basic concept can be used in a number of different languages. Typically only procedural languages are used, e.g., C and Fortran.

### 5.6.2 Message Passing

In a distributed environment where processors do not share memory, an effective and efficient method needs to be used to communicate between parts of the system. A widespread technique used is that of message passing. The necessary information is packaged up by a process into a message and passed to some form of communication layer. The message packet is then transferred to the required destination, where it is passed onto the receiving process. In this example, the message was passed from point to point; this is not always the case and other communication modes such as broadcast (one to many) are usually supported.

There are a number of issues that need to be examined to give a clearer understanding of message passing libraries. Firstly, how is data encoded into a message and decoded from a message? When an application chooses to send a piece of data, the data needs to be packed into a message packet body. Messages typically have a header, containing source and destination address, and a body, containing the data. The message header may be private to the message passing library and not available to the user. The data needs to be coded so that it can be reassembled at the remote end. The data needs to be decomposed into a stream of basic data types that can be transferred by the message passing library. The basic types are usually equivalent to those found in C, i.e., character, integer and float. For the encoding step, data is serialised into a sequence of the basic data types. Decoding is achieved by deserialising the sequence, based on equivalent rules to those used in the encoding. The writing of serialising and deserialising code is a mechanical task and a code generator is often supplied.

The transportation of the bytes of the serialised data is the task of the message passing library. The protocols, topologies and physical transport mediums used are hidden from the user except where there is a choice or a efficiency issue. Message passing libraries useful in a distributed system are those that can use a variety of protocols and transports, such as high performance switches, ATM, FibreChannel and Ethernet.

In order for a point to point network to operate, processes need to have a unique identifier so that other processes can refer to it. A consistent naming scheme needs to be used so that processes on processors on machines with different operating systems on various networks can all be named. It is significantly more useful if the names can be passed around to other processes.

When a process sends a message to another process, it can wait for an answer, or it can proceed having initiated the message. These two modes are called synchronous and asynchronous messaging. During the sending of a synchronous message the sender pauses until the result comes back. When the message arrives at the remote process, the message has to wait until the receiving process is ready to complete the transaction. Depending on the messaging layer, the receiving process may perform some computation before returning a result back to the sender. As can be seen, synchronous messages can entail a lot of time delay for the sender before it is able to continue processing. Also note that whilst it is waiting, often called blocking, the sending process cannot do any processing, and send or receive other messages. This leads to the possibility of deadlock in the distributed system. In these situations, extra complex code to avoid getting into deadlock situations is required. With asynchronous messaging, no blocking occurs, and the sending process continues with its computation once the message has been assembled and handed over to the message passing library for delivery. Asynchronous messaging is much more resilient to deadlocking. One disadvantage is that the code necessary in a distributed system is often slightly more complex. This is due to the fact that a process has to react to the problem of messages arriving in any order. So for example a sender may send a message to a remote machine, which asynchronously performs some calculation and sends a reply. If the sender is performing a number of these transactions, it has to be able to differentiate one reply message from another. This tends to encourage a more event driven style of programming. Asynchronous communications typically require the use of queues to buffer messages.

**Parallel Virtual Machine**

Parallel virtual machine (PVM)(Geist *et al.*, 1994; Sunderam, 1990) is a popular package aiming to provide a useful system for programming heterogeneous networks of machines. The package consists of a library for C and Fortran programmers, and a set of system programs that support the virtual machine.

In PVM, a virtual machine consists of a number of machines linked together by a piece of support software (a “daemon” under Unix). The user is able to specify a list of machines to include in the virtual machine. This is not a static configuration, as machines can be added and removed at any time. One of the particularly useful features of PVM is that the configuration is entirely user based. This is especially useful in a multi-user environment, allowing different users to have different configurations, as well as avoiding the problem of requiring a privileged user to configure the virtual machine. The user can also associate a speed rating to each machine in the network. This allows PVM to bias placing processes towards faster machines. This is a fairly rudimentary measure and does not take into account other factors such as load and network performance.

PVM supplies the programmer with a simple message passing library and a simple set of process control routines. Processes can be started by simply specifying the program name of the new process. If required, the user can specify a certain machine, or certain architecture. PVM then selects a machine, and instructs the daemon on that machine to run the required program. PVM programs are normal processes, except that they make calls to the PVM library, and the PVM support daemon is aware of their presence. This allows the user to perform process housekeeping easily across the virtual machine. Every process that is running in the PVM system is given a unique id, called a `tid`. Processes can discover their own `tid` and that of their parent, i.e., the process that spawned them.

Messages in PVM consist of a message tag and the message body. PVM supplies a set of routines to pack and unpack data to and from the message body. Once packed, a message can be sent to a specific process by using its `tid` and the message is given a tag. The tag is a user-defined number that can be used to give some notion of the type of the message. Messages are sent asynchronously and are queued up at the receiving end. The receiving process can either take the next message in the queue, or use a message tag to take a particular type of message. If there are no messages waiting, the process will block when it performs the receive operation. A probe function allows the process to test if there are any messages waiting. This can be used to avoid performing a blocking receive operation. Messages from one machine to another are guaranteed to arrive and to be delivered in order.

The packing and unpacking functions allow PVM to encode and decode the data into suitable format for transmission in the heterogeneous environment. This is entirely transparent to the user, apart from a couple of message options that allow the user to specify the encoding method. These options mean that the user can turn off encoding of the data, which is instead sent in its native format. This is only useful if the user knows that the program will be run in a homogeneous environment where all the native data formats are identical and a slight performance benefit can be achieved by avoiding data encoding and decoding.

PVM supplies a number of ancillary functions used for communicating with groups of processes. A process can enroll in a specific named group and operations can be performed on the group as a whole, such as a broadcast to the group or a barrier operation that waits for all the processes to synchronise. A process can also perform a broadcast operation by specifying a list of `tids` to send a specific message to.

In use, PVM supplies a relatively low-level library for writing distributed systems, but provides a solid base on which to write message passing applications, allowing the user to avoid writing many messy low level communications routines. One of the strengths of PVM is that it runs on many different architectures and across a variety of networks. Primarily, these machines are Unix based, but the software is being converted to run on many different operating systems, notably Windows 95 and NT machines. Also, PVM and its source code are freely available, which allows programmers to extend easily and write utilities for the environment in which they are working.

### Message Passing Interface

The Message Passing Interface (MPI) library was the product of a committee of more than 40 members from both research and industrial communities in 1993 and 1994 (MPI Forum, 1996). The mission of the committee was to develop a standard for message passing. The standard was aimed to stop the increasing prevalence of vendor specific and proprietary message passing libraries with incompatible interfaces, hence requiring users to learn and program a number of different libraries. The main focus was to deliver high performance. The style of programming and approach required by the user is the same as for PVM, i.e., interfaces to message creation, sending, receiving and unpacking functions.

MPI-1, the first version of the specification, focuses mainly on a large set of point-to-point communication routines. In addition, the library provides support for collective communication among groups of processes. Topologies of communication networks between process can be specified, enabling the implementation to fine tune the placement of processes on the physical processors and the communications channels on the underlying physical interconnection network. The richer set of communications modes was designed to encompass all of the message passing features of various parallel machines and networked clusters. This allows the implementation to exploit special communications modes that the hardware may support.

MPI-1 is not designed to be used in a heterogeneous environment, and different vendor's implementations are not required to inter-operate with each other. Applications written using MPI are portable across the various implementations, but when the application is executing there is no defined inter-operability between applications running on different implementations. Hence, MPI implementations tend to use architecture specific encodings. In addition, MPI implementations are tuned by the vendors to extract the maximum performance from the hardware. This means that MPI programs typically run faster than equivalent programs written in PVM, especially when the MPI can take advantage of special communications modes of the hardware.

MPI-1 provides a static environment. When a MPI program is loaded onto a machine (and there is no standard interface to this process) the system runs until all the processes have terminated. Resources, such as nodes and interconnects are not typically released until the system terminates. This is in total contrast to the dynamic facilities provided by PVM. In use, MPI is only really viable when the application will only run on one type of machine and typically on one MPI implementation.

The MPI-2 draft specification is scheduled to be finished in the summer of 1997. The draft expands the number of functions and communication methods between processes, language bindings, and parallel input and output. In addition, the draft also includes facilities for process control and management. This allows for dynamic creation of processes instead of the static environment of MPI-1. It provides similar functions to those found in PVM, but does not feature facilities for dynamic resource management, e.g., the adding and removing of nodes from the virtual machine.

### Sockets

Sockets are a communication mechanism originating from the Unix operating system. They provide a low-level set of routines that can be used to communicate between two machines, creating a layer of independence from the particular network technologies. They form the basis of many protocols and libraries found in the Unix network environment and have become the standard for the Internet.

A socket is a bi-directional communications pipe between two processes. To establish the pipe, one process must first create a connection point. This process typically sits in a form of server-mode, waiting for a connection. Another process connects to the connection point by specifying a machine name and port. Once the connection is established, the two processes can communicate freely. The communication maybe shut down immediately after an initial data transfer, or it may persist for a number of exchanges of data.

Sockets are usually used in conjunction with IP and TCP or UDP. Internet Protocol (IP) is a protocol that allows packets of information to be sent to and received from other IP-speaking machines, typically over the Internet. TCP and UDP are two protocols that form a layer above IP. Transmission control protocol (TCP) is a connection based protocol, that guarantees reliable, ordered and two-way communication between its endpoints. Universal datagram protocol (UDP) is a connectionless unreliable protocol. UDP is more lightweight than TCP since it does not have the added overhead of creating a connection and of checking for errors. UDP is known as a send-and-pray protocol. A socket can be created as either using TCP or UDP.

Sending data over a socket is done at the level of sending bytes. The user simply specifies a block of data to be sent and the transfer happens asynchronously. At the receiving end, the programmer has to reconstruct the data, as required. There are a number of options that can be applied to a socket, such as blocking and how message boundaries are defined.

Sockets are only useful as a low-level communications layer on top of which more useful libraries and systems can be constructed. The PVM library discussed earlier uses sockets.

### 5.6.3 Remote Procedure Call

Remote Procedure Call (RPC) is a programming style used in client-server applications (Birell & Nelson, 1984). As the name suggests, the style revolves around invoking a procedure call on a remote process or machine. This is a simple concept to understand, since its based on the familiar function or procedure call used in sequential languages. Its likeness to the sequential procedure call means that it is easy to port existing sequential software to a distributed system employing RPC.

The client assembles the parameters to the procedure call and transfers them to a server process. The server process then unpacks the parameters and makes the procedure call. Results are passed back in the same way as parameters. The client has to block whilst communication and computation occurs. The call also acts as a synchronisation between the two processes. One of the significant problems with RPC is that in the absence of shared memory, pointers to data are meaningless on the remote machine. This makes parameters which are pointers troublesome for use over RPC. In simple uses of the pointer, the data can be copied across to the remote machine. If the remote process makes changes to this data, then it needs to be sent back. This is particularly cumbersome and may prove to be difficult with large and complex data structures, especially if they are cyclic.

A widely used implementation is Sun Microsystem's RPC standard. Sun's RPC forms the basis of a number of remote services, notably NFS. Built upon sockets and TCP/IP or UDP/IP, or more recently TLI, a layer called XDR or eXternal Data Representation, provides a standard encoding for fundamental types and a basis for encoding more complex structures. XDR is not limited to use with RPC, and can operate over any communication method provided primitives for reading and writing bytes exist, e.g., over sockets. The server registers with a local daemon its program number. Any client wishing to use the server specifies the program number and hostname of the server machine. The client process connects to the daemon on the server machine, specifying the program number. The daemon acts as a predefined and known contact point on the machine for all RPC services. It handles the registering of server processes and connects clients to servers. Each program can have any number of procedures, each designated with a number. The client call supplies this number along with the parameters for the procedure call. Hence, this requires that the program and procedure numbers are known across the distributed system. Sun's RPC also has a notion of a version number, allowing a number of different sets of procedures to share the same program number. This aids with expanding and improving the server whilst maintaining backwards compatibility with clients.

The writing of the XDR and server routines to handle the communication is a mechanical task and Sun provide a `rpcgen` program that handles much, if not most, of this work. The user supplies a C header-like protocol specification file, which is translated into a number of C files. Firstly, XDR routines for all the data structures are created. C files are automatically created to construct the server. The user need simply write the RPC procedures and the rest is automatically generated. Similarly, the client side of system can be created. This provides a simple and painless mechanism for writing a distributed client/server application. For more complex servers and clients, the lower level functions of the RPC library have to be used. Such features need to be used for asynchronous calls, where the client sends the data and does not wait for a reply.

RPC provides a simple mechanism for distributed processing in Client/Server situations with the use of uncomplicated data structures.

### 5.6.4 Threads

Decomposing a distributed system into a number of programs or processes can often be problematic and inefficient. Processes typically cause a lot of status information such as open files, child processes and communication channels to be stored. Creation and deletion of a process are expensive operations. Processes represent a heavyweight unit of concurrency and for a large number of tasks a lighter-weight mechanism is sufficient.

Threads or light weight processes are smaller unit of concurrency than a process. Threads typically share the same address space as the process that created them.

Threads within a process can communicate via the process address space, with careful consideration to concurrent reads and writes to shared data. Threads can run concurrently in the usual time-sharing fashion on a single processor or can run on separate processors, if the operating system supports it. Some multi-processor operating systems allow threads to be bound to a particular processor, which is important for real-time system performance.

Threads libraries consist of a number of routines to handle process creation and deletion. Access to shared data is controlled by semaphores and mutexes, with appropriate function calls. Signals provide a mechanism to notify threads of events.

Programming with threads is effective when the level of concurrency in the system decomposes into small pieces of computation. Threads are often used to handle events, such as communications with other processes, where response is critical. A thread can be created to handle the transmission of data allowing the thread of computation to continue. Similarly, a thread can handle the receiving of data, and respond, if required. This is especially useful for messages that require a simple data look up, or independent computation. This is similar to active messages and the handlers in Nexus (see below). Decomposing the system into threads in this manner can often lead to more concise and manageable components. There are pitfalls, especially with sharing of data and deadlock. Debugging threaded applications is often harder than debugging equivalent sequential programs.

Thread libraries and implementations vary across operating systems. POSIX define a common standard pthreads, which most of the popular Unix systems support. Support for threads is available in the Windows 95 and Windows NT operating systems.

Threads are an important paradigm for concurrent programming.

### 5.6.5 Active Messages

Active messages combine the idea of remote procedure calls and message passing approaches. The idea is that often in message passing systems, messages are sent between processes that require a small amount of computation and a subsequent reply. An active message is a message with a specific handler identifier, which when received initiates the required handler routine which performs computation and returns a result. In this simple form, it is very similar to a remote procedure call. Active messages extend this architecture into a more generic form creating both request handlers and reply handlers. An active message is sent to the request handler, which will reply by sending an active message to the reply handler.

The active message interface provides a generic interface to an active message system, which has support for multi-threaded applications. In addition it defines a powerful naming mechanism, which allows the construction of complex message systems.

The main benefit of active message systems are its high performance, derived from the lower overhead of the handler mechanism. This avoids costly polling operations, and simplifies messaging events to sequential and parallel applications.

### 5.6.6 Nexus

Nexus (Foster *et al.*, 1994) couples global pointers with RPC/active message-like Remote Service Requests, in a multithreaded environment. An RSR specifies a global pointer, a procedure to be invoked and the arguments for the procedure. The arguments are transferred to the location that the global pointer points to, and the required procedure invoked. The Nexus RSR is similar to an active method, except that the Nexus model tackles the limitations of active messages. Specifically, active methods are restricted in the computation that the handler can perform, due to the interrupt mechanism used. Nexus removes this restriction with the use of threads.

Nexus is effective at constructing communication layers that can operate over a number of communications methods. This "multimethod" communication (Foster *et al.*, to appear) is obtained by associating a communication handler table with a global pointer. This details different methods, corresponding to different communication layers, that can be used to access the global pointer. When a global pointer is transferred to a remote machine, the communication handler table is also sent. Hence, the receiver of the global pointer has access to all the methods that it can use to access the pointer.

These methods, in combination with the capabilities of the receiver, allow the receiver to select the most effective communication layer.

Nexus is primarily designed as a runtime system for task-parallel languages. Ease of use was sacrificed for efficiency and so Nexus is less suited as a general purpose library for use by the application programmer, like PVM for example.

### 5.6.7 Distributed Shared Memory

Shared memory is a mechanism where multiple processes on a machine are able to access the same piece of memory. This is similar to threads and their common address space. Blocks of shared memory are created by a process, yielding a handle to the memory. This handle can then be passed to other processes which also open the shared memory block. Support for shared memory is a fundamental service provided by the operating system.

Distributed shared memory, as its name suggests, is the equivalent mechanism mapped onto a distributed environment. Distributed shared memory tries to bring the benefits of programming a shared memory system to a distributed system. In a distributed system, providing this level of support is a complex procedure.

The main benefit of a distributed shared memory system is that it provides the programmer with a much simpler programming paradigm. In addition, programs written for a shared memory machine can easily be ported to the distributed memory system. Complex data structures, such as linked lists, trees and graphs, can be used easily between different processes. This contrasts with message passing systems where serialisation of these objects is complex and often inefficient. Communication occurs simply by passing pointers to blocks of memory in the shared memory pool, with the memory containing the required data. As with threads, care has to be taken to avoid problems with concurrent reads and writes to the same memory location.

Implementation of distributed shared memory varies. One approach is to provide support in the operating system. Here, distributed shared memory is similar to the approaches used to implement virtual memory and shared memory. A page in the virtual memory address space is designated as distributed shared memory and the relevant source location is stored. When the application tries to access the page, a memory trap occurs, as with normal virtual memory. The page is fetched from the remote machine, and execution continues as normal. This is similar to virtual memory where the page comes from disk. The complications with this approach lie in writing to the page. The handling of writing to a page has to cause all other copies of the page to be invalidated, so effectively there is always one copy of the page that has the privilege to write. The performance of such a system is poor, especially if a large number of processes need to write to the page. Page sizes on a modern Unix workstation are 4 or 8 kbytes, resulting in course-grained memory operations. This impacts the performance due the amount of data that needs to be transferred and that there maybe a large number of write operations to different locations in a page, causing page thrashing around the distributed system.

A different approach to this problem is solved by languages which provide native language support for distributed shared memory. The language has a conceptual address space that extends over a number of machines. The programmer is unaware that a variable or object is on a remote machine. This is an advantage for the programmer, but can lead to serious implementation and performance difficulties. For example, the access patterns and location of a shared object may result in a lot of remote access. Languages such as Orca expose the programmer to the notion of location and use shared data objects between procedures and locations.

Other languages promote the use of a global pointer, which represents a handle to a possibly remote object. Instead of hiding the creation of shared variables or objects from the programmer, global pointers expose the distributed system. Hence, the programmer is more aware of issues of location and efficiency. Some global pointer systems allow the physical location of objects to change, allowing the data to migrate to the location that is using it the most frequently. Global pointers may also be cached raising the usual issues of concurrent writing and maintaining the correct values across all the cached versions of the data. Global pointers are useful in strongly typed or object oriented languages. A global pointer object can be constructed that acts as a form of proxy for the real object that may reside at a remote location. Access to the slots of the object result in relevant remote calls to the real object to fetch the slot value. Object coherency and location can all be hidden behind the proxy object. Issues of garbage collection, if the language supports or requires it, also map well onto the proxy object system. The proxy objects and global pointers incur a performance penalty since all accesses have to go through the extra level of indirection and possibly communication.

Language support for distributed shared memory provides a less coarse-grained approach to the problem than low-level operating system support.

## 5.7 Distribution for Speed-up

The main focus of the distributed aspects of this thesis are on obtaining results faster. The distributed system is used to give speed-up in the computation by using a number of machines concurrently. Speed-up is the main aim of parallel processing. Crossover of these techniques into the distributed world is a natural progression. Parallel machines are large and expensive whereas a distributed system can contain any number of cheaper workstations. This is a more flexible and cost effective system. Workstations may be in use as desktop machines, but become part of a distributed computation resource when not in use, for example, at night.

There are significant differences between the two resources that effect the performance and range of applications that can be run. A distributed system will have a slower network in terms of latency and bandwidth. This gap is closing, with the widespread use of faster network technologies such as 100baseT Ethernet, ATM and Myrinet. Workstations typically have less memory than processing nodes in a parallel machine. The performance of the processor is often not as great as the processing elements in a parallel machine. With the rapid development and need for ever faster processing on the desktop, this relationship is ever changing. Modern parallel machines use the faster and more expensive variants of the processors designed for desktop machines. The pace of development means that these faster processor reach desktop machines in appreciable numbers only months behind parallel machines.

The category of applications that run well in a distributed environment are those where the problem decomposes into coarse grained computation. The communications requirement of the application has to be lower. Such applications are often referred to as "embarrassingly parallel", due to the ease of mapping the computation onto a concurrent system. Distributed systems are going to be less capable of performing the "Grand Challenges" favoured by the parallel community. These large and complex applications are always going to require the facilities of parallel machines. The significant point about distributed systems is that a number of parallel machines may be present on the network and the mechanisms to harness their collective power are no different to those used with normal workstations.

### 5.7.1 Measures of Speed-up

Speed-up is often used as an indication to the performance of a parallel algorithm, implementation or system. It is defined as the ratio of the execution times of the sequential version to the parallel version. The sequential version is used as a best effort version on a single processor. Speed-up is often plotted against number of processors to give a speed-up curve. The ideal speed-up is where the speed-up is equal to number of processors. Super-linear speed-up is where speed-up is greater than the number of processors.

Amdahl's law looks at the maximum attainable speed-up for a given algorithm. It assumes that a parallel program can be reduced to a sequential portion and a parallel portion. If $s$ is the proportion of the sequential portion and $r$ the parallel, then the speed-up $S_p$ for $p$ processors is

$$S_p \leq \frac{1}{s + (r/p)}$$

This suggests that there is an upper bound on the speed-up that any given algorithm can have. It also suggests that for larger values of $r$ larger values of $p$ are more effective.

### 5.7.2 Farms

A farm or processor pool is an effective way of managing the task of concurrent computation. A master process controls a number of slave processes. The master distributes data to the pool of slaves, which perform the calculations. The results are then passed back to the master process which performs some form of calculation to return the final result to the user. Problems that decompose into a number of course-grained and independent tasks are particularly suited to this farming of tasks.

The most efficient computation occurs when none of the slave processes are idle. Idle time occurs in two places. Firstly when a slave process is communicating with the master to either receive more data to work on, or when it returns results. The time delay between returning results and receiving more data in order to resume computation can be avoided by a buffer which queues up a number of data packets at the slave.

Idle time also occurs when a slave finishes its computation and there is no remaining data left. Here, the slave sits idle until all the other slave processes have finished. This is due to uneven distribution of work amongst the slaves. Obtaining an even distribution of work can be complex, especially if the computation time for a given piece of data is hard to estimate or the machines in the system experience fluctuating load due to other users or services. Reducing the grain size can result in more even load balancing, but this results in a larger number of data or work packets and also larger amounts of communication between master and slave. If an effective estimate of the computation time is available, then the decomposition and scheduling of the work can result in quite efficient computation, providing it takes into account the performance and load on the individual machines and communication times.

The use of farms provides a generic method of distributing computation for those applications that decompose suitably. For more complicated algorithms, different strategies for distribution are required. For example, if an algorithm consists of a series of different routines, with streams of results from one routine being passed as inputs to the next, then a pipeline topology could be used with each routine running on a different processor.

Parallel processing typically employ topologies such as rings and meshes. Rings are especially useful when the mode of communication is such that at the end of an iteration each node needs to broadcast a piece of data to every other node. Meshes are effective on problems where there is some form of geometric decomposition and nodes need only communicate with local neighbours. These topologies exploit the underlying network topologies of parallel machines, by performing communication concurrently between numbers of nodes over numbers of links. In a distributed system, this level of concurrent communication is often not present. Ethernet, the most widely used local area network, only allows one node on a segment to broadcast at a time. A network can be divided up into a number of segments with a bridge, separating the traffic on individual segments. Ethernet switches seek to increase the bandwidth on a network by creating temporary circuits between pairs of communicating nodes. As these and technologies such as ATM become more widespread and local area network topologies more complex, the potential for concurrent communication increases.

## 5.8 Fault Tolerance

In a distributed system the potential for failure is much higher. In a parallel system, failure of a node or part of the system typically effects the whole of the machine. A failure of the system is usually not handled by parallel programs. In a distributed system, large parts can survive when there is a failure, for example, when a single machine or part of the network fails. Hence, being able to recover from a fault is vital. Faults can be temporary in nature, such as a minor network problem.

The subject of fault tolerance is often quoted as one of the main features of distributed systems, but most libraries and distributed languages do not provide sufficient support for fault tolerance. This is due to its complex nature.

In order to recover from a partial failure, firstly the system has to detect that a failure has occurred. Depending on the failure, the node experiencing the failure may or may not be able to inform the rest of the system. For failures such as a total machine crash or network failure, the rest of the system cannot be informed directly of the failure. Instead availability probes and timeouts on communications have to be used. Once a failure has been detected, the system has to recover in such a manner that the result or behaviour of the system is exactly the same as if no error occurred. Recovering from a fault requires that the system has state information about the failed part of the system.

This impacts the computation lost, any messages in transit, and any side-effects that a process or part of the system was performing.

Responsibility for recovering from a failure can be left entirely to the application. Here, the operating system notifies the application of an error, for example, when it tries to communicate with a failed node. The application then has to recover from the failure. For example, if a slave in a farm fails, then the master can reallocate the work that the failed slave was performing to another slave. In the example case of a slave failing, the necessary state to recover from the failure is known by the master, but this can be inefficient since the computational effort expended by the slave on the piece of work is entirely lost. In the case of long periods of computation, more efficient recovery from slave failures can be achieved by the slaves periodically sending their current state back to the master.

The example of slave node failure is relatively straightforward to recover from, mainly due to the structure of the system. A failure of the master node would be a situation that would be impossible to recover from. In this situation, and many other more complex structures of the distributed system, a significant architectural redesign has to take place in order for the application to recover effectively from a fault. This places a significant burden on the application programmer and can lead to complex programs.

The alternative to placing the burden of recovering from a fatal failure on the application, is to perform the recovery at the language or operating system level. Here, the application and the programmer does not have to worry about handling a failure and leaves it to the distributed operating system to "auto-magically" recover. Typically this can only be effectively achieved with co-operation between the language runtime system and the operating system. The operating system is unaware of the behaviour and side-effects of the application, and can only possibly be aware about the state of the processes in the system and any communications that have been and are currently being performed. The language runtime system has more detailed knowledge of the application behaviour. This coupled with the information captured at the operating system level is potentially enough to recover from a failure, provided this information is available to the system after the failure.

Some approaches to transparent fault tolerance require the runtime system to write its state to a stable storage medium (typically hard disk). This can happen at specified checkpoints decided by the programmer or by the language runtime system. After a failure, the collective checkpoint state of the whole system can be examined and the application recovered with the right behaviour.

Very few distributed languages or systems currently provide sufficient support for transparent fault tolerance.

## 5.9 Statistics and Distributed Computing

This section looks at what aspects of statistics are particularly amenable to distributed computing.

Distributed computing performs well on applications with coarse-grained computations, where the amount of communication between processes is kept low. Hence, applications that run well are those that can be partitioned into separate coarse units. Partitioning can occur by dividing up the computation, or by dividing up the data, depending on the structure of the problem.

Adams *et al.* (1996) looks at parallel processing and its use in statistics. The authors identify that many statistical algorithms are constructed from building blocks of numerical algorithms. Many of these building blocks exhibit parallelism and, hence, scope for speed-up. Some of the building blocks have fine-grained parallelism and, hence, are unsuitable for distributed computing. The level of partitioning of the problem needs to be much higher for the application to run efficiently in a distributed environment. The fine-grained approach is a bottom-up approach to decomposing the problem. For distributed computing, the top-down approach of decomposing the application is often more effective.

An example of a statistical technique whose computation can easily be divided up is that of Monte Carlo simulation. Here, a computation is repeated a large number of times. In cases where each repetition is independent of the previous repetition, the individual computation can be performed on separate machines. Typically the total number of iterations required is divided into a number of smaller pieces which can then be distributed onto a number of machines. Some Monte Carlo problems require information from the previous iteration, such as Markov chain processes. In these cases, the total computation can not be divided into independent chunks. For certain classes of these problems, it can be meaningful to run multiple chains, the results of which can be combined at the end of the computation. Hence, the multiple chains can be distributed. Other sampling based techniques such as bootstrapping can similarly be partitioned.

The process of model selection involves fitting a range of models to a data set. Each of the models are fitted independently and hence the models can be fitted concurrently. The only communication required is the collation of the results at the end of the computation.

Some areas of statistics handle large datasets, which can be spread across the distributed system. For example, the analysis of stratified data requires operations to be performed on each strata of the data set. These strata can be spread across the distributed system. Applications such as data mining can exploit the resources of the network. The notion of resource can be extended beyond computing power, since large datasets may have to be physically stored across a number of machines. In these situations, the program has to migrate to the data, instead of the more typical case of data migrating to the program.

The key to running statistical applications on a distributed system is to find processes that can be performed concurrently that require little or no communication between them. With the constant increase in network speed, communication latency and bandwidth becomes less critical to distributed applications, enabling distributed system to perform well on more fine grained applications.

## 5.10 Conclusions

With the increasing deployment of faster networks, connecting large numbers of machines, the use of distributed computing is vital to use the resources effectively. This chapter has outlined important aspects of distributed computing, focusing on using the distributed environment as a computational resource. There are number of common issues between distributed computation and the more established parallel processing world, and there is a convergence of technologies, both hardware and software. This chapter has given an overview of a number of different programming paradigms, software libraries and software architectures that can be used in a distributed system.

The following chapter brings together work earlier on in the thesis on exact tests and the distributed computing, to develop a distributed complete enumeration application.

# Chapter 6

# Distributed Exact Tests

## 6.1 Introduction

This chapter combines the work on exact p-value calculation with the introductory material from the previous chapter on distributed computation. The aim is to create an application capable of calculating exact p-values on a number of machines, with reasonable efficiency.

Firstly, the requirements of such a system are examined, followed by a prototype implementation for 2-way tables. Finally, an implementation of the model matrix enumeration on a distributed system is discussed, highlighting the important points that enable efficient computation of the exact p-value.

## 6.2 Requirements

As the results from Chapters 2, 3 and 4 show, the search space for some models is very large, making it infeasible to do the calculation in reasonable time on a single machine. The aim is to extend the reach of the algorithms presented in this thesis, by creating distributed versions.

The application should be able to run on a wide-range of machines, connected by widely available network technology, to form a distributed computing environment. The aim is to maximise the range of machines that an end-user can use to run the distributed application. The application should perform well in the distributed environment, scaling well as more processes and machines are added.

Any underlying systems architecture should be generic and be usable in other applications, not just of a statistical nature, that exhibit a similar decomposition as the distributed exact tests application. Hence, the system should provide a straightforward and easy environment in which a user can program.

## 6.3 Prototype

This section discusses the work on the prototype implementation of the distributed application. The first section looks at the decisions as to what environment and architecture were used for the prototype implementation.

### 6.3.1 Architectural Decisions

#### Language

The chosen language to implement the bulk of the prototype system was Scheme. The interactive nature of the language is helpful when developing and debugging a distributed system. Again, extensive use of the foreign function interface in the Scheme implementation was made, enabling the use C network library functions to be used.

Debugging of a distributed system can be often complex and time consuming. The interactive nature of Lisp languages lends itself particularly well to this problem. Whilst the system is running, variables and functions can be invoked interactively to investigate the state of the system. In addition, functions can be rewritten without having to reload the application. This dramatically reduces the “edit compile run” cycle time, especially when finding intermittent bugs in the system which are often prevalent in concurrent programs.

The Scheme system used was available on most popular Unix systems, but not, at the time of implementation, available in a stable form on Windows PC architectures. For a prototype this is not a significant problem, but a final implementation would have to take steps to penetrate the widespread Windows environments.

### Network Protocols

With the easy binding of C library functions into the Scheme language, many of the available libraries for distribution could be used. At the time of implementation, many of the libraries discussed in the previous chapter were not available or sufficiently mature enough. MPI was only available on a limited number of parallel machines and was expensive. Even so, the limitations of the MPI environment meant that it was never a viable option. PVM is a natural candidate, providing many of the features required for the implementation.

For partly historical reasons, a binding of the Sockets library in Scheme was used. This library had already be implemented and tested, whereas the PVM library would require an interface and binding layer to be implemented in Scheme. The use of the lower level Sockets library allows for a more extensive look at what features are important in a distributed system when implementing the applications in this thesis.

### Target Architecture

The main architecture that the prototype system would be run on was a network of Sun SPARC workstations. The cluster of machines, connected by a standard Ethernet network, provides a suitable environment to investigate the feasibility of cluster computing. In addition, the multi-user Unix environment is especially suited to the idea of using idle processor time on the network. This environment can provide a significant computation resource, especially as these machines are idle and left switched on overnight.

### 6.3.2 Application

The first statistical application used for the prototype was an implementation of Patefield's algorithm (1981) which enables 2-way tables to be randomly generated under the model of independence. The Fortran code included in the paper was converted into C code, in order that it can be called from a Scheme program, as well as avoiding the problem of lack of availability of a Fortran compiler.

In contrast to the complete enumeration methods already presented in this thesis, Patefield's algorithm is used to estimate the exact p-value by simulation. The algorithm is run repeatedly, each time generating a random table. As before the test statistics are calculated for each generated table and the table is checked to see if it is as least as extreme as the observed. A running total of the number of tables which are at least as extreme is maintained. The exact p-value is obtained by dividing the number of extreme tables by the number of tables sampled. No calculations involving the individual table probability are required since Patefield's algorithm selects tables from the correct distribution.

The generation of the tables can be performed independently, apart from issues surrounding the random number generator. Hence, the application is an embarrassingly parallel example. This means that performance on the distributed system is expected to be relatively high, since there is little demand for communication. This example, although perhaps trivial, is effective in that it facilitates the investigation of other important factors of the distributed system such as load balancing, communications, and the general performance of the prototype system.

The second application written to run on the prototype system was a distributed version of the algorithm for complete enumeration of 2-way tables under independence, detailed in Chapter 2. The distribution of the computation in this example can be easily achieved by enumerating the set of first rows of the table, and enumerating independently the subtables of each those rows. The second, more efficient, algorithm detailed in Chapter 2 was not implemented. The distribution of the computation of this algorithm could be achieved by sending each node a $2 \times k$ table of the margins for the top and bottom half of the table.

These two applications in combination can be quite effective at analysing a wide range of 2-way tables. In a number of situations, the number of tables that need to be sampled, before the accuracy of the exact p-value estimate is high enough, is so large that the complete enumeration routine can evaluate the whole search space and return the exact p-value in a shorter space of time. Hence, it is useful to run the two applications in tandem on a given problem. If the complete enumeration method returns a result, then the simulation method can be terminated prematurely. Conversely, if the simulation method indicates that the exact p-value is high enough to be out of the 5%-level area, or an estimate to a suitable level of accuracy is achieved, then the complete enumeration processes can be terminated.

### 6.3.3 Software Architecture

The obvious architecture or topology to use in this application is the processor farm. Both prototype applications can be easily decomposed into independent tasks, requiring very little communication.

Chapter 5 details a processor farm as consisting of a master process and a number of slave processes that perform the computation required. The master handles all the communications with the slave processes, passing "work packets" and waiting for results of the computation to be returned. A typical implementation of this architecture would create both the master process, and the required slave processes that are able to perform the required computation, as well as any of the required systems software to enable communication amongst the master and slaves. The computation would be co-ordinated by the master process. The results of each of the slave processes would be processed by the master process. This means that the master process is specific to that application and cannot be reused. In addition, the computation required to recombine the results from the slave is performed on the master processor or machine. This is a fairly "application-centric" view of a farm.

The farm prototype that is proposed is a more system oriented structure. The power and flexibility of a distributed system comes from using the resources on the network. Resources can take the form of services on the network. The farm outlined in the

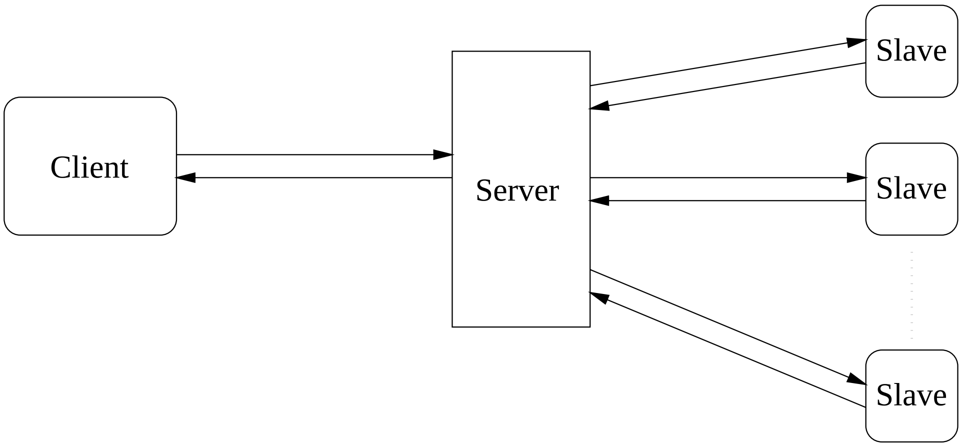


Figure 6.1: Improved structure of the farm system.

previous paragraph can be thought of as two services. The farm topology consists of the necessary components to allow slaves and masters to communicate, and for the master to manage the computation on the slave processes. Hence, this can be thought of as supplying a service to the application running on the farm topology. At the application level, a given application running on the farm topology could be thought of as supplying a service on the distributed system to clients or users on the network. Hence, with a two layer farm system, the master process is split into two programs, the *server* handling the communication and computation of the slaves, and the other, called the *client*, handling the work packets and results, both specific to the application. The slave processes also have two conceptual halves, but the communication side of the slave is straightforward and can be contained within a library. Figure 6.1 shows the structure of the system.

One of the main motivations for separating the traditional role of the master, into the client and server is that the server part is now invariant of the application. This has a number of benefits. The server can be recoded or re-implemented in a different language, without having to re-implement any other part of the system. This adds a degree of future-proofing to the software, for example, alternative communications protocols may become available at a later date. Secondly, since the server is common, it can be used by a number of different applications and users, providing the low-level service in the distributed system outlined earlier. This means that the protocol used to conduct the communication between parts of the system needs to be multi-application. The system can also be multi-user, the security aspects of which are not addressed in this prototype.

To enable the multi-application support in the system, each application has a unique tag. When a slave connects into the system, it identifies what application it is part of by supplying the tag to the server. Similarly, when the client joins it supplies a tag. This then allows the master application to make a matching between clients and slaves in the same application. If a slave of the required application is free, then server passes to the slave any parameters that the client supplied. The slave then performs the required computation, returning to the server the results, which are passed on to the client. Note that the server is transparent to the data that is passed through it, and can effectively perform a byte-wise copy between the incoming and outgoing message bodies. From the client point of view, the whole process is very similar to that of RPC, which uses a program number, version number and procedure number tuple to identify an "application", the difference being that the client will most likely want to initiate a number of remote computations on the farm of slaves. The server acts as a form of proxy for the slave in this RPC-like mechanism.

In this system, both the slave and client have to connect to a known location, typically specified by the machine name and port number. The problem of finding the server location could be solved by some form of broadcast mechanism, directory service, or other resource discovery solutions. Similarly, given multiple servers, a client may wish to know which server it should connect to in order to contact a given type of slave. A symbolic name for an application could be used, with a name-service creating a mapping between name and tag number. This would allow for greater flexibility and ease of use. A symbolic name could be used directly in the protocol, but this would make the messages passed around the system less concise and require extra processing each time a message is received. A number representation of the tag is more efficient in these often performed operations.

### 6.3.4 Implementation

Having identified the major architecture and structure of the system, the next step is design and implement the individual components of the system.

### Communicating Scheme Processes

Firstly, communication channels between Scheme processes have to be constructed. The decision to use Sockets as the transport mechanism for sending bytes around the system, has already been made. A binding of the Socket library functions, such as `socket()`, `bind()`, `select()`, `read()` and `write()`, allows the Scheme processes to establish communication channels between two processes and send bytes back and forth. Whilst this is sufficient to enable the rest of the system to be constructed, it is not a very easy to use this set of primitives from within Scheme. Specifically the use of `read()` and `write()` to send blocks of bytes is not very useful. Instead, a layer on top of these two routines is constructed to enable the communication of Scheme types and structures. This builds on the work on communicating Xlispstat processes (De Roure & Michaelides, 1994). The layered structure of the system means that the communication method can be easily changed at a later date.

To enable the use in a heterogeneous environment, XDR routines are used to encode basic types in the system. Scheme objects are encoded into streams of these basic types. The encoding consists of a integer identifier that indicates the type of the following object, followed by the data for that object. The simple Scheme objects such as integers, floating point numbers, characters, strings, true, false and nil encode in a straightforward manner. The more complicated objects such as `cons` pairs and vectors, consist of an identifier followed by a depth-first representation of the elements in the object. For example, with a `cons` pair, the identifier is followed by the encoding of the Lisp object in the `car` of the pair, and then the `cdr`. These complex structures can be cyclic, and so the encoding has to handle these cases. Also related to this problem are two Scheme operators `equal?` and `eq?`. Both these operators are used to test for equality between two Lisp objects. The difference being that `equal?` looks at the data in the Lisp object, whereas `eq?` looks if the objects are physically the same, usually by looking at the memory addresses.

Obviously `equal`-ness is preserved with the above encoding. To preserve `eq`-ness and the cyclic structure of the Scheme objects, the encoding needs to be able to refer to objects previously sent. This can be achieved by maintaining a cache of objects sent. Two identical caches of objects are maintained at each end of the communication channel. When an object is sent, the cache is checked to see if it has already been sent, in which case a reference to which element in the cache is sent instead. If the object is not in the cache, it is sent in the usual manner, and added to the local cache. The remote end, on receiving the object also adds the object to its cache. This encoding represents `eq`-ness, and `equal`-ness can also be used in the communications. An `equal` code can be used when the Lisp object in question is not the same as a previously sent object (`eq?`), but the data it contains are the same (`equal?`). This can usefully compress the amount of data sent across the slow communication links when there is a large amount of equal data, for example sparse data.

The object cache can be efficiently implemented by using a hash-table to speed-up lookups in the cache. The hash key of an object can be obtained from the data in the object. For the simple list of objects, the value of the object can be used, but for the complex types, the address of the object can be used. This results in `eq?` objects hashing to the same value.

The only remaining consideration is the lifetime of the cache. If the lifetime is too long, then the cache will become large, and searches less efficient, impacting communication performance. In this application and in most cases, it is sufficient to preserve equality and the cyclic structure of data only within the bounds of a message. Hence, the cache at both ends of the socket are reset every time a message is sent.

### Message Formats

Having established a communication layer that allows Scheme objects to be transferred between Scheme processes, the next step is to identify what messages need to be passed around the system.

1. a slave registers itself with the server, supplying its application tag.
2. a client connects to the server requesting a slave with the required application tag, and supplying the initial parameters.
3. the server indicates success or failure at allocating a slave to the client

4. the server passes on the initial parameters to the slave
5. the slave completes the computation and passes the result back to the server
6. the server passes on the results to the client
7. the client closes the connection to the server
8. the slave closes the connection to the server

This details the required communications to enable the farm system. The server process has to maintain state about what slaves are registered, if they are free, and to which client they are allocated, if they are not free. A slave can only talk to one client, but the client can talk to many slaves. Therefore, the server has a unique identifier for each transaction between client and slave. This identifier is passed back to the client when it is informed of a successful request for a slave. The identifier is then used when passing the results to the client.

As the system stands, the level of communication between the client and slave is limited to just passing parameters and returning results. This restriction can be lifted to allow the client and slave to communicate freely with each other during the period of the transaction. Hence, the server becomes a message router between the client and slave. To aid with routing, the server also passes the slave a unique identifier that it must use when sending a message to the client. This change to the communication protocol allows for more flexibility in the application. For example, the slave may periodically send intermediate results to the client process, which could be used to give feedback to the user, or allow the client to assess whether the computation should continue.

Table 6.1 shows the types of messages and the bodies sent between client, server and slave. Note that messages sent in the system are asynchronous. Clients can only have one pending request for a slave at a time. Messages sent between client and slave are always sent via the server which inspects the *id* to pass the message onto the relevant process.

| From | To | Type | Body | Comments |
|---|---|---|---|---|
| Slave | Server | `Register` | *app.tag* | |
| Client | Server | `Request` | *app.tag* | |
| Server | Slave | `Work` | *id* | *slave* allocated, use *id* in messages |
| Server | Client | `Work` | *id* | *id* of -1 indicates failure |
| Client | Slave | `Msg` | *id data* | |
| Slave | Client | `Msg` | *id data* | |
| Client | Slave | `Quit` | *id* | |
| Slave | Client | `Quit` | *id* | |

Table 6.1: Messages sent between processes in the prototype.

### Client and Slave Processes

The implementation of client and slave processes is relatively straightforward. Both consist of an application specific portion and the communications portion.

In the basic form, the client consists of splitting up the computation into a number of "work packets", each of which is then passed onto a slave via the server. The process continues until all the work packets have been sent and the results returned. The results are then accumulated in some application specific way and the result returned to the user. All of the communication with the server and slaves can be hidden from the application programmer. A function `farm` can be defined, which takes the application tag number and a list of work packets. Typically a work packet is a list of parameters. Within the `farm` function, a loop waits for communication events from the server. Initially, the function requests slaves until it receives a failure from the server, or until it exhausts the work list. For each requested slave it sends a work packet. When a result is obtained, the result is appended to a results list and another request for a slave is sent and subsequently another element from the work list. This process continues until the work list is empty, at which point the `farm` function waits for the pending slaves to return their results. The results list is then returned to the caller.

The slave process has a much simpler structure. The process firstly registers its application tag with the server, and then enters a loop. Within the loop, the slave waits for a message from the server indicating it has been requested by a client. At this point, in the basic situation, the slave reads a message from the client which are the arguments to the application. The application specific computation then occurs, generating a result. The result is passed back to the client via the server, and the slave returns to the beginning of the loop. The loop may be terminated at this point, for example, if the slave decides that the machine is in use by a user.

For more complicated requirements from the application, the user will have to extend the basic structure of both the client and slave.

The only difficulty in this implementation of the client process is due to the asynchronous nature of the communication. When the client sends a message to the server, for example to request a slave, it cannot be guaranteed that the next message back will be a response to that message. In order to facilitate easier programming, an extra layer can be created above the message layer. This layer buffers incoming messages into queues, based on the type of the message. Hence, when the programmer needs to receive a specific type of message, the layer checks the relevant buffer to see if a message exists in that queue. If no message is available, the layer repeatedly receives messages and appends them to the relevant queues, until a message of the correct type is received, which is then returned. Care has to be taken with the use of these queues due to the semantics of the protocol and the ordering of messages.

### Server Process

The server process runs a daemon, waiting for connections from processes and responding to messages on existing connections. When a process connects, the initial message that it sends identifies it as a client or slave. The server has to maintain a list of slaves that are connected. Associated with each slave is the socket that they are communicating on, their application tag and whether it is free or not. A list of clients does not need to be maintained, since they are not maintained by the server. The server maintains a list of "switches" which represent a links between the clients and slaves. Each switch has a unique identifier, and the client and slave sockets. The unique identifier is the same as the identifier given to both the client and slave.

When a message is received by the server, it acts accordingly, adjusting the state of the system. In the simplest case, a client is sending a message to a slave or vice versa. The message quotes the unique identifier of the switch. The server can then pass the message onto the other process in the pair, by comparing the incoming socket number with the two listed in the switch. A quit message from either pair in a switch, results in the message being passed onto the other member and the switch deleted. The slave in the pair is then marked as free. A successful request by a client results in the creation of a new switch with the relevant information, the slave is marked as busy, and messages sent to both the processes to indicate the start of the transaction.

The server consists of a loop. At the start of the loop a `select()` library call is made. This library function inspects a set of file descriptors and socket handles, and marks which sockets have incoming data. The server then receives a message from each socket in turn and acts accordingly. The round robin method of reading from sockets maintains a level of fairness in the system. The server also has a special socket that waits for new connections. When this socket is active, a new connection is being established. This results in a new socket, which is included in the set of sockets for inspection by the `select()` call. Hence, in a subsequent time around the loop, the server will receive the initial message from the new socket, and act accordingly.

In the prototype there was little or no error checking. In a more robust system, the server would have to check the validity of operations that are being performed. For example, when passing on a message, it should check whether the source process has a valid connection to the other pairing in the switch. This also avoids problems if the switch identifiers are reused.

### 6.3.5 Examples

To evaluate the performance and viability of using a cluster of machines, the distributed Patefield example was run on a group of 10 Unix Solbourne SPARCstations. The total work for each run consists of generating 17,000 tables. The total work was split evenly among the number of available workers. Figure 6.2 shows the results from 5 runs, the points indicating the average speed-up. Note that the tests did not have sole use of the cluster, resulting in the slave processes having to compete for processor time. This

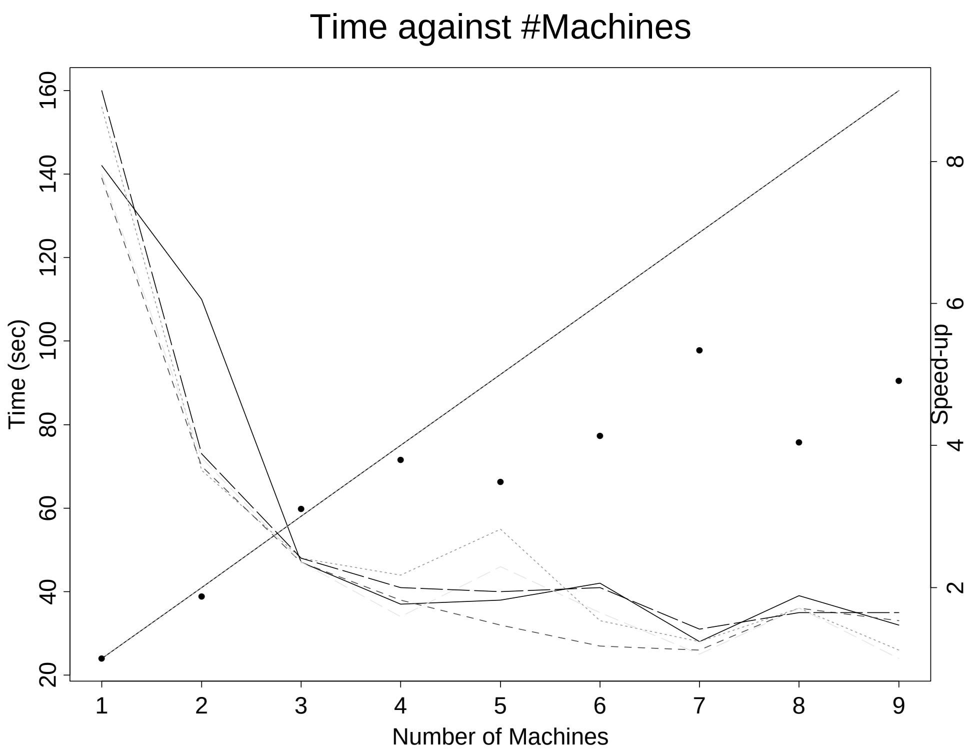


Figure 6.2: Graph of runtime against number of machines for 5 runs.

leads to the rather erratic timings and subsequent speed-up. This can be seen in the speed-up value for 3 machines, which exceeds the line indicating linear speed-up.

A major source of the poor timings are from inefficient load-balancing. Since the total work was divided evenly among the processors, machines that have a higher loading due to other processes will take longer than less loaded machines, resulting in longer runtimes, and processors idle during the computation. This problem can be addressed by dividing the total amount of work into a larger number of pieces. Machines that are less loaded will consume more pieces of work than busier machines. Although this method requires more communication, the resulting load-balancing makes the computation more efficient. The results of this can be seen in Figure 6.3, which shows two runs, with associated average speed-up. The results are more consistent than the previous results. The tests were made in a similar environment of fluctuating load on the clusters. The results also exhibit a larger speed-up than the previous tests, indicating that the smaller work size led to better load-balancing of the computation. Empirically, the results show a reasonably speed-up, for example 5 times on 6 processors. As is typical with most

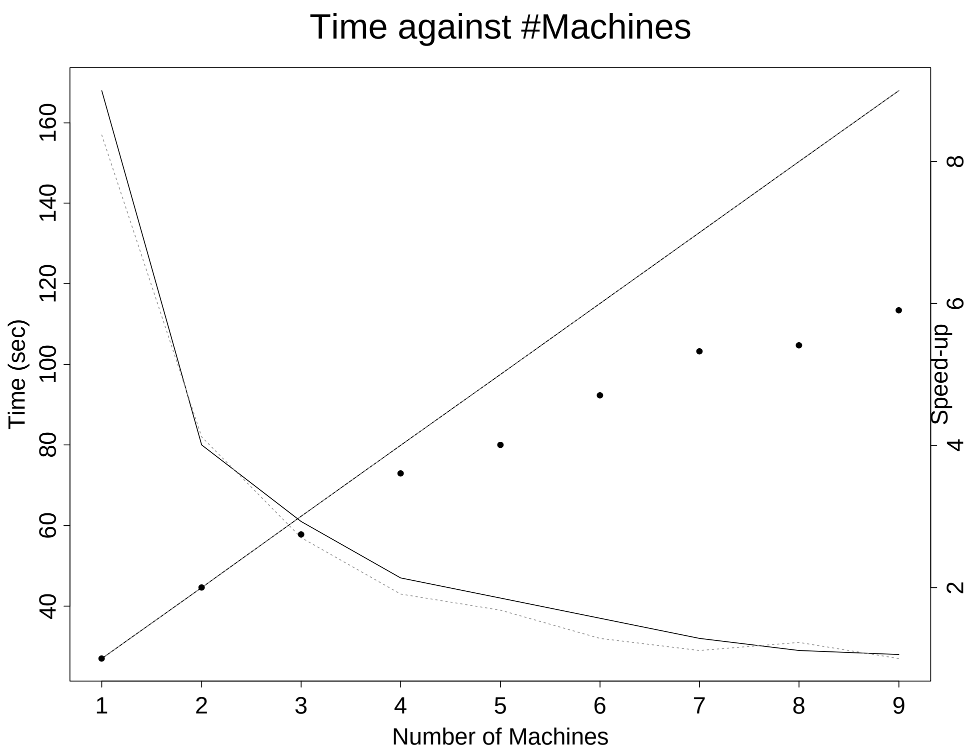


Figure 6.3: Graph of runtime against number of machines with smaller work packets.

parallel and distributed applications the speed-up tails off as the number of processors increases, due to the increased amount of communication in the system.

## 6.4 Distributed Model Matrix Enumeration

The rest of this chapter examines the design and implementation of a distributed version of the model matrix enumeration algorithm described in Chapter 4.

PVM was chosen to implement this system as opposed to the prototype system described earlier in this chapter. The main reason for taking this decision is that it allows a better comparison of the features of the prototype and a widely available message passing library. In addition, one of the target architectures for the implementation was an IBM SP2 system, consisting of 22 nodes connected by a high performance switch. At the time of implementation the Scheme system used in the prototype was not available on the SP2. Re-implementation of the prototype entirely in C, would allow the prototype to be used, but this would not facilitate any useful comparisons of the prototype.

The SP2 system also has an implementation of the PVM library that uses the native communications protocols of the high performance switch, which hence provides higher performance than the use of sockets, as used in the prototype.

Much of the structure and internals of the server process in the prototype were used in the new implementation. A move back to the standard master and slave farm configuration was made, partly due to restrictions imposed by the PVM library. These restrictions and ways to circumvent them are examined later in the chapter when a comparison between the two systems is made. The master and slave configuration is both more natural in the PVM environment and offers higher performance.

### 6.4.1 Distributed Enumeration

Recall that the algorithm from Chapter 4 loops a cell from zero to a maximum value. At each iteration of the loop, the algorithm recursively enumerates all tables that share the already fixed cells. The recursive enumeration is independent between iterations of the loops. This provides a reasonable computational structure that can be performed concurrently. Hence, in a similar way to that in which the previous example of enumeration for independence distributed rows to each slave, the enumeration under a arbitrary model matrix distributes at the cell level. This is a more fine grained decomposition of the problem than the row distribution, but with an arbitrary model matrix, large structures that can be distributed may either not exist, or may be hard to identify.

Hence, the distributed enumeration is identical to the single processor enumeration, except that when a cell is successfully fixed at a given value, the partially generated vector of elements is returned to the master, for enumeration concurrently. The algorithm then continues to increment the cell, instead of making the usual recursive call to search the subspace, since this space has been submitted for enumeration elsewhere. At some stage, the algorithm has to stop distributing work back to the master, usually when there are a reasonable number of fixed cells and the enumeration can traverse the rest of the search space in reasonable time.

Figure 6.4 shows an example enumeration tree and how the computation is decomposed

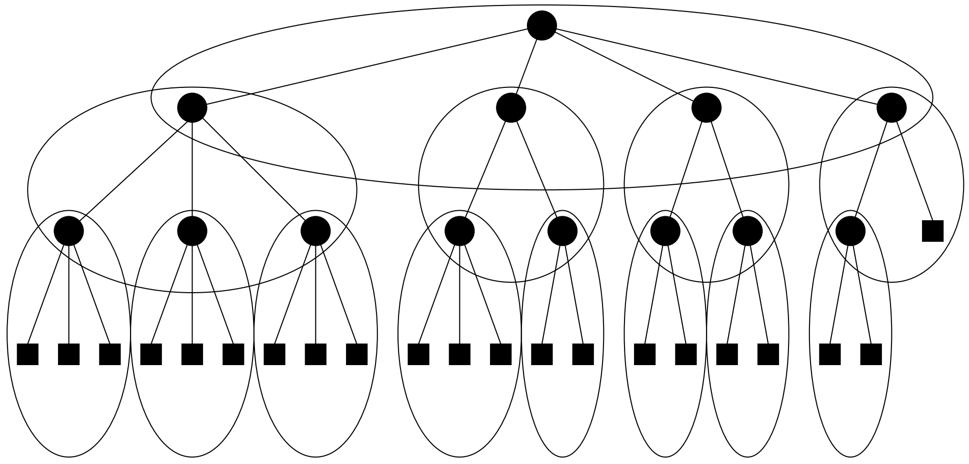

Figure 6.4: An example enumeration tree showing distributed structures.

and distributed. Ellipses in the diagram show the enumeration that is performed in each work packet. A square indicates where the slave searched the whole remaining subspace.

### 6.4.2 Implementation

The implementation of the distributed system involves dividing the existing program into master and slave. The pre- and post-amble of the original program becomes part of the master. These sections read in the model matrix and observed table, and report the final results to the user. The master has to create the slave processes, using the PVM `pvm_spawn()` call. Once the slave processes have been created, the master then sends the initial data. This consists of the model matrix and observed table. The slaves proceed to simplify the matrix and calculate the fitted values. This process could be performed on the master and the matrix and fitted values passed to the slave processes, but this has the disadvantage that it increases the size of the initial message, as well as introducing any inaccuracies due to truncation in the communication process. In addition, even though the computation is repeated across the slaves, the total time for a given example will be the same or less than performing the calculation on the master, unless the master node is significantly more powerful than the slave nodes.

Once the slaves have been initialised, the system enters the usual loop of providing work to slaves and gathering the results. The structures used to maintain lists of slaves, free

| From | To | Type | Body | Comments |
|---|---|---|---|---|
| Master | Slave | `Init` | *initial data* | |
| Master | Slave | `Work` | *data* | |
| Slave | Master | `Work` | *data* | |
| Slave | Master | `Result` | *results* | |
| Master | Slave | `Quit` | | computation completed |
| Slave | Master | `Quit` | | leaving system |

Table 6.2: Messages types in the distributed model matrix enumeration system.

slaves and pending work are the same as used in the prototype. Slaves in the pending work list also have the work packet that they are currently working on.

Since slaves also send the master work packets as well as results, data in the system has to be sent in a number of different types of messages. Table 6.2 shows these message types. Note that the `Work` message from both master and slave have identical format.

The slave process has to handle three types of message. Hence, the main task of the slave is to sit in a loop, waiting for a message. An `Init` message causes the slave to read in a model matrix and observed table, and perform the initialization outlined above. A `Quit` message indicates that the computation is over, and the slave can shutdown. The slave responds with a `Quit` message and exits. This final response from the slave is not absolutely necessary, but it allows clean shutdown of the system to occur. A `Work` message results in the slave unpacking the work packet and starting enumeration.

The work packet has to encode enough state so that the slave resumes the computation in exactly the same way as if the originating slave had continued enumerating. At any point in the enumeration, each cell in the vector of cells has either been fixed by the enumeration or is unfixed. The value of a fixed cell is greater or equal to zero. Hence, fixed cells can be encoded using their value, and unfixed values can carry a negative value. The slave is then able to unpack the message, update the relevant data structures used by the enumeration algorithm and resume enumeration. This process is made possible by the restructuring of the code to enable state saving, as detailed in Chapter 4, resulting in the algorithm being re-startable from defined places in the enumeration algorithm. One of these defined places is the point at which the algorithm has fixed a cell at a given value, and is ready to proceed in enumerating the subspace by making the recursive call. Effectively, the transaction between slave and master and then master and a new slave results in this recursive call being made on another machine or at a later point in time. The work packet is not the same as the state of the enumeration that was discussed in the Chapter 4. The state of the distributed system is discussed later in this chapter.

As each successful table is generated, a number of variables are updated. Firstly, the number of tables is incremented. Secondly, the variables used to calculate the p-values are updated. The table probability of every table is accumulated, and for each test statistic used, the table probability is added to the current total if the table is at least as extreme as the observed. Since all these variables are added together, they can be accumulated in groups, i.e., as each subspace is searched. When a slave completes the enumeration of the region specified by the work packet it returns the four accumulated variables back to the master.

The main body of the master consists of a loop, receiving and responding to messages from the slaves. A request for work from a slave either results in a work packet being sent to the slave or the slave is moved to the free list if there is no work available. Each slave can be in one of three states, waiting for work, busy working, or stuck. A message from the slave containing results, cause the slave to move from the busy state to the stuck state. The variables in the result message are accumulated with the equivalent global message, and the work packet that the slave was working on is finally deleted. The stuck state provides a intermediate state for the slave between being in the busy state and either moving to the busy state again, or to the free state. The stuck state is not the same as the free state, because the slave has not yet requested more work. If a slave went directly to the free state after returning a result, a work message could be sent to it, but in the meantime, the slave could have sent a quit message and exited the system.

When a work message from a slave is received, the master checks if any slaves are currently free to handle the new work packet, in which case the master passes on the work message and updates the state of the newly awakened slave accordingly. If no slaves are currently free, the work packet has to be added to the free list. Note that the work packet has to be added to the *front* of the free list. This mimics the stack approach to a depth first traversal of the search tree. Adding the work to the end of the work list would result in a breadth first traversal, causing the master to have to store a possibly very large set of work packets.

As the distributed searching nears the end, the work list will deplete and become empty and the free slave list will grow. Slaves cannot be instructed to quit, because the currently running slaves may generate more work packets. Termination of the slave processes can only occur once all the slaves are free and the work list is empty. At this point, the master issues a `Quit` message to each of the slaves and waits for the corresponding quit echo from the slave, to ensure clean tidying of the system. In the situation where a slave issues a `Quit` message before the computation has finished, the slave is removed from the system. If the slave was in the stuck or wait states, then the slave can be removed immediately. In the case of the slave being in the busy state, the master has to restart the work packet that the slave was working on elsewhere. This is not as straightforward as it seems. If the slave, whilst working on the work packet issued more work, then simply restarting the work packet would result in the issued work being repeated. The following section solves this problem.

Note that a slave has to issue two messages before computation continues after a work packet has been completed. This inefficiency can be addressed by including a new message, `ResultRequest`, combining the action of returning results and requesting a new work packet. On the master side, the slave avoids the "stuck" state. The slave is also restricted to committing itself to performing more computation.

### 6.4.3 Distributed State

There are a number of reasons why being able to capture the state of the system is useful. It enables the recovery of the computation in the event of a failure. It also enables the computation to be frozen and the state captured and stored, for later resumption. For example, a number of workstations may only be available for use at night, or the queuing policy on a parallel machine might restrict the amount of CPU time a parallel job can consume. The state can be captured and resumed, in these cases either the next night or in a new job.

Most of the work in enabling distributed state capture has already been done. The sequential algorithm has already been written in such a way that the state can be captured. The difference in the sequential and distributed version of the algorithm is simply that the sequential version is enumerating the entire search space, whereas the distributed version is searching a region of the space, delimited by the already fixed cells given by the work packet.

Recall that the enumeration algorithm maintains a list of cells indicating whether they are currently fixed or not. Fixed cells are marked with the *rank* at the time of fixing. The *rank* effectively orders the cells so that when the algorithm backtracks it unfixes the relevant cells. In the distributed version when a work packet is started by the slave, the already enumerated cells are fixed with a *rank* of one. The distributed algorithm is changed slightly to limit the *rank* to values greater than one, causing the algorithm to terminate when it exhausts the subspace and would normally backtrack and start changing the fixed cells given by the work packet.

The state of the entire system is simply the state of all the slaves, the remaining work packets and the current values of the variables used to calculate the exact p-value. Hence, the master process needs to maintain a structure containing the state of the system. This raises the issue from the previous section where care has to be taken to avoid the repeated calculation of work packets. If a slave, during the process of enumerating its search space, has not sent any new work packets to the master then its computation can be restarted from the work packet it was sent. In the case of the slave having issued more work, then correct recovery of the system requires that either the new work packets are removed, or when resuming, the slave continues enumerating at a point *after* the last work packet was sent. The former approach is hard to achieve without maintaining a representation of the enumeration tree in the master detailing where the work packets and their results occur. The latter approach, can be easily achieved by sending the current state of the slave whenever a new work packet is sent. Note that the sending of a new work packet and sending of the state has to be *atomic*.

The master maintains a list of slaves and the work packet each is currently working on. If a slave issues a new work packet then the state it also supplies replaces the work packet associated with it. Hence if the master needs the entire state of the system, to write it to disk, for example, then it simply needs to write out the work packet or state held for each slave, the list of remaining work to be done and the global variables. To make the handling of the distributed state easier, a work packet can either consist of a list of already fixed elements, or the state of a slave. Hence the state is a list of work packets to complete the enumeration. This can be read in by a new master directly into the work list, and the computation resumes in a seamless fashion.

This method of capturing the state of the system, or “checkpointing”, is the minimum required to guarantee correct resumption. It may not be the most efficient, since a slave may spend a large amount of time traversing its search space without communicating to the master. This computation would be lost if there were some form of failure. The system could be easily extended to allow the slave to periodically send its state to the master, reducing the amount of lost computation when a checkpoint is resumed. Significantly, this increases the level of communication in the system, and requires the slaves to pause computation and communicate. The benefit of finer checkpointing can be outweighed by the penalty of the extra communication. In addition, an acceptable period between the slaves sending their state has to be decided. This is related to the size of a work packet and effective load balancing of the system.

### 6.4.4 Load Balancing

The efficiency of performing a computation on the distributed system depends on good load balancing. The aim is to minimise the time that an application takes. This is achieved by maximising the percentage of available CPU power spent on the application computation. Hence, the aim is to minimise the idle time of the slaves and the time spent communicating. The equation is complicated by the heterogeneous and dynamic nature of the distributed system. Processors in the system may be of differing power, and the load, which affects the percentage of CPU time that a slave is given, varies across the system and varies over time. In addition the communications links between slaves and the master may differ radically in performance. If communication was instantaneous and the overhead of starting, finishing and creating a work packet was zero, then the best load balancing is achieved with the smallest possible granularity. Here the grain size becomes a function of the processing power each slave commands. For example, given two nodes, one of which is ten times faster than another, the best configuration is to have eleven equal-sized work packets. Conversely if the communication penalty is massive, then the best load balancing requires a large grain size. In a realistic system, the best grain size is a trade off between these two situations.

In the application of distributed enumeration, the calculation decomposes into irregular sized pieces of computation. This problem is compounded by the fact that there is no estimate as to how long a work packet will take to calculate. If an estimate were available, the scheduling of work packets becomes easier, based on the performance of the slaves. In the absence of dynamic information about the current performance of the system, the best approach is to specify a size of reasonable computation. At each stage in the enumeration, a cell is fixed at a given value. The slave can either distribute the search space rooted at that value, or proceed with the enumeration itself. If the search space is less than a reasonable level of computation, then the slave should proceed, otherwise it should be distributed. Since there is no estimate as to how long a computation will take, the only indication that can be used is how many cells are left unfixed. Hence, if the number of unfixed cells becomes lower than a specified level, then the slave will stop distributing work. The computational effort required to fix the remaining cells not only depends on the model, but on the region of the search space in question. Whilst this method is not ideal, one of its major benefits is that it requires no extra communication or changes to the basic algorithm.

An alternative approach to the problem, would have the master solicit work from busy slaves when there are idle slaves. Here, the busy slaves would have to periodically check for messages from the master requesting work. This imposes an added overhead on the slaves, which could render the approach inefficient. The periodic check by the slaves could be eliminated by the master including an indication of how busy the rest of the slaves are, with each work packet. This would be less dynamic than the polling approach, but if the work packets were sufficiently small, i.e., communication was often, then this would not be a major problem. The slave could then make a decision about

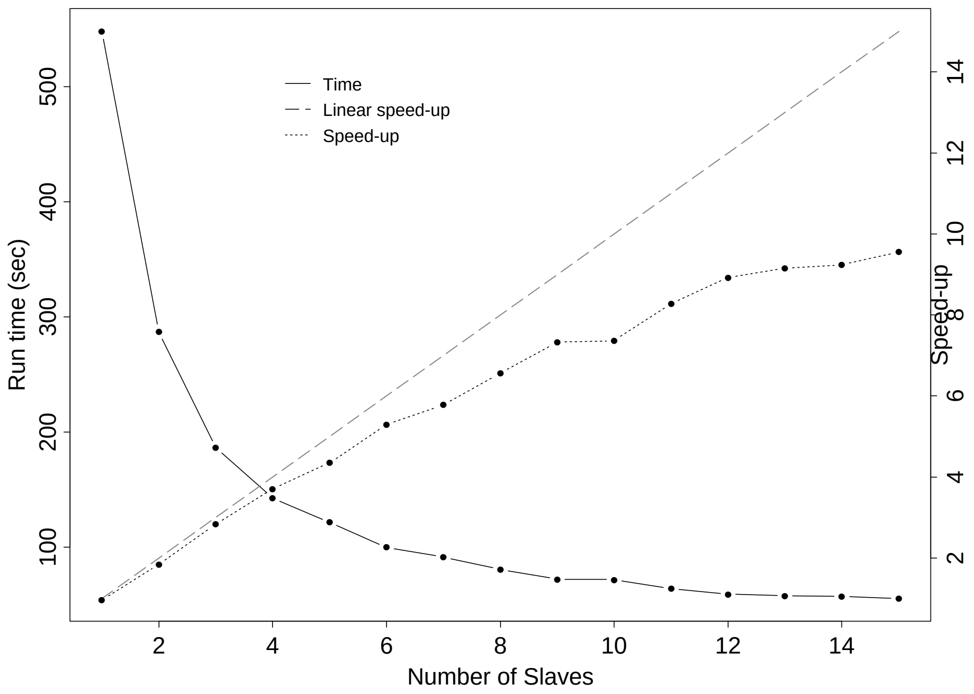


Figure 6.5: Graph of time and speedup against numbers of slaves run on the SP2.

whether to distribute work or not based on knowledge of the entire system.

One issue to take into consideration is that when enumerating large problems, it is best to have the slaves grouped working together on a specific region, as opposed to having the slaves working on areas spread across the search space. This results in the state of the system being smaller, and a smaller size of work packet, which is especially important if there is a limit to the length of time a slave can operate for.

Tests performed in the following sections indicate that the approach of fixing the level of distribution, based on the number of cells fixed, works effectively.

### Instrumentation

The efficiency of the application can be examined by timing the various phases of the application. Firstly, the master process can measure how long it takes to spawn the slave processes and communicate the model matrix and observed table. This delay can be critical since it is a sequential portion of the application that cannot be eliminated.

Once the master has entered into the loop of distributing work and receiving results, it can time how long a work packet takes from the time it is sent to the time the result is received.

From the slave point of view, the initial setup time can be measured as well. This time represents how long, from the time the slave was spawned, it took to get into a state where useful computation can be performed. This includes the time delay in receiving the model matrix and observed table from the master. The processing involved with ordering the matrix and calculating the fitted values is also included in this time, but is generally negligible. The importance of this time delay is that it is unavoidable and no generation of tables can occur until it finishes. At this stage the slave enters into a loop of requesting work, waiting for a response, performing the computation and returning the result. This leads to two timing regions. Firstly, the time between sending a request for work and a work packet being received is measured. This period is the overhead of the distributed system and consists of the communication time and any additional delays incurred, i.e., the delay in waiting for more work to become available. There are two special cases for this time delay, namely the first and last delays. The first delay is the time taken whilst the system maps out the top of the search tree, the first level of which can only performed by a single slave, i.e., effectively sequentially. As the initial search is performed, work packets will be produced and given to the remaining idle slaves. The second special case delay is the last message interaction, when the computation nears the end and there are no remaining work packets. At this point, a number of slaves are idle whilst the remaining slaves complete their work packets. If this delay is large, it indicates that the work packet size is possibly too large, and the delay could be reduced by decreasing the work packet size. The slave can also measure the time it spends operating on a work packet. This quantity observed from the master includes the communication time. Hence, the time delays due to communication can be derived from these two quantities. The work time measured by the slave consists of the overhead of setting up the enumeration from the given work packet, the required enumeration, which is the same as in the sequential version, and the time spent communicating work packets back to the master. These overheads can be examined by varying the depth at which distribution occurs, resulting in more or less work packets. Since the enumeration remains the same, the difference in time measured by the slave for operating on the work packets is just the overheads.

In practise, it is sufficient for the slaves to accumulate the time spent on the different stages and communicate these values to the master prior to leaving the system. These values are the total time spent waiting for work, the total time spent working, the start and end delays, and also the minimum and maximum wait times. The final two values help to give an idea of the range of delays during a run.

### 6.4.5 Linux Clusters vs SP2

This section looks at the comparison of using a cluster of PCs running Linux (a free Unix implementation) and an SP2 running AIX (IBM's Unix implementation).

Applying the timings outlined above to a run on the SP2 nodes reveals some evidence of the inefficiencies that effect the results shown in Figure 6.5. Firstly, when the master process starts up, it takes between 1.5 and 2.0 seconds to enroll in the system. Further, the spawning of the slaves in the system takes 0.8 seconds per slave. The broadcast of the model matrix and observed table takes a total of between 0.7 and 1.0 seconds. These amount to quite significant overheads, especially as the number of slaves is increased. In addition, the first delay that each slave experiences is relatively high. All these timings are significantly higher than those experienced on a cluster of PCs running PVM. This seems to suggest that the PVM implementation on the SP2 has much higher initialisation costs than the standard PVM distribution.

Once the enumeration starts, the time delay between sending a request for work and receiving a reply is on average approximately 0.4ms. This compares favourably with the equivalent time of 8ms on the Linux machines. The other penalty that the distributed version incurs, compared to the sequential version, are the overheads associated with the creation, communication and resumption of a work packet. Timings estimate this delay to be 0.2ms and 4ms for the SP2 and Linux runs respectively. This indicates that the major portion of the delay is due to communication, as expected. For equivalent runs, this translates to 0.5% of the time spent communicating for the SP2 and between

| | Run A | | | Run B | | |
|---|---|---|---|---|---|---|
| Nodes | SP2 | 10x100 | 8x133 | SP2 | 10x100 | 8x133 |
| Sequential | 528.86 | 561.93 | 419.15 | 4371.12 | 2445.85 | 1820.51 |
| 1 | 534.98 | 592.23 | 443.82 | 4475.88 | 2521.10 | 1832.50 |
| 2 | 275.85 | 297.12 | 222.30 | 2222.22 | 1214.94 | 902.48 |
| 3 | 183.51 | 198.04 | 151.65 | 1487.31 | 807.69 | 603.95 |
| 4 | 144.45 | 149.17 | 111.04 | 1116.37 | 611.48 | 459.91 |
| 5 | 113.99 | 120.61 | 90.68 | 897.36 | 492.21 | 369.43 |
| 6 | 97.33 | 100.81 | 75.42 | 761.66 | 410.34 | 305.73 |
| 7 | 86.90 | 86.21 | - | 656.21 | 352.30 | - |
| 8 | 76.92 | - | - | 562.44 | - | - |
| 9 | 69.67 | - | - | 514.21 | - | - |
| 10 | 64.88 | - | - | 475.78 | - | - |
| 11 | 61.47 | - | - | 438.27 | - | - |
| 12 | 58.85 | - | - | 407.10 | - | - |
| 13 | 57.11 | - | - | 385.63 | - | - |
| 14 | 54.26 | - | - | 366.72 | - | - |
| 15 | 52.75 | - | - | 333.50 | - | - |

Table 6.3: Comparisons of timings on an SP2 and two Linux clusters for two runs. Times in seconds

3% and 5% for the Linux cluster. Results for the SP2 are generally more stable, since the system guarantees a quality of serviced, as opposed to using a PC cluster where the application competes for resources with whatever other processes and applications are running.

Two clusters were used for this experiment, one with ten 100MHz Pentium processors and another with eight 133MHz Pentiums. The tests were performed in the evening, when usage was lower, but at any one time, not all machines were available for use. The results are shown in Table 6.3. Two runs were performed, the first “A” consists of the test for independence on Example “D” used in Chapters 2 and 4. The second example was the model of QI+PS on the Great Britain slice of the religion data detailed in Section 4.9.2. The table shows the runtime for each run based on the number of slaves used. The time for the sequential version of the algorithm is also shown. The SP2 results in this table are used in the graph shown in Figure 6.5.

As expected, the integer nature and cache requirements of the algorithm favours faster processors. The SP2 used in this test has 66MHz processors but has better memory bandwidth than PCs. Since the application has relatively small communication bandwidth requirements, the cluster of PCs performs well. This is especially true for Run B, where the SP2 seems to perform comparatively badly. Alternatively, the performance of Run A on Pentium processors could be considered to be worse than expected.

### 6.4.6 Examples

The distributed application significantly reduces the runtime for a wide range of models and tables presented in this thesis. For the examples with a small reference set, and resulting runtime, it is unnecessary to perform these tests on the distributed application.

As an example of a large computation, the model of quasi-symmetry was applied to a $2^6$ table, in which 46 of the 64 cells had a zero count. Complete enumeration yielded 2,346,795,429 tables in the reference set. This took approximately 72 hours to calculate on 5 133MHz Pentium computers.

## 6.5 Further Work

### 6.5.1 Distributed Work Pools

Currently all the work packets are stored centrally in the master processes. This results in all communication of work packets and results going to the master process. Although we are able to adjust the size of work packet such that there is no bottleneck in the master, a more efficient mechanism might be have distributed work pools. Further work could examine how pools of work could be kept locally, whilst still preserving the ability to save the state of the system and be resilient to failures.

### 6.5.2 Memory usage

The model matrix enumeration algorithm used in this chapter has a very low memory requirement. One of the features of using distributed environments and parallel machines is that there is more memory available to the application. Further work could examine how this extra memory can be used to provide more efficient enumeration.

## 6.6 Conclusions

The implementation of the prototype system was useful as a proof of concept. The data structures used to maintain the state of the slaves was reused in the model matrix enumeration implementation. One of the obvious disadvantages of the prototype system was the delay in communications between the slaves and client. This was mainly due to the intermediate server, requiring extra encoding and decoding. This could be eliminated by re-implementing the server process to avoid decoding and encoding of the data communicated between slave and client. Alternatively, higher performance could be achieved by creating a direct communications link between client and slave. This is similar to the direct mode option of communication between processes in a PVM system. PVM has a similar notion of message routing processes which can be a source of inefficient communication, although the message routers do not decode or encode the messages in transit. PVM's direct mode opens a communications channel between two process in the system.

The prototype system's different farm structure is more useful in providing a service on the distributed system. The implementation in PVM of the matrix enumeration application is more suited to a one-time batch-processing situation. PVM has a number of useful *group* operations which could be used to extend the system into a pool of application slaves, as advocated by the prototype. Both systems still require an extra layer of distributed configuration control, to provide a more effective and useful system for the user. The role of this layer would be to start and stop the individual slaves to react to the changing state of the machines in the system. Such facilities could be provided by such systems as Condor (Epema *et al.*, 1996) but these lack knowledge of the application and its current state of computation. A more effective layer would enable the applications programmer to have a closer integration of the management system and the application.

The increased communication patterns of the matrix enumeration application was better suited to the more closely coupled implementation using PVM. The modular implementation of this system meant that the application specific code is easy to replace. This extends from the application, right down to the encoding of work packets and the slave state.

The approach of having application-generated state capture provides a much more portable method than relying on the operating system runtime library to provide the required support, which is currently not widely available. This requires that the application programmer to do more complex coding. This can lead to transformations of the implementation of the algorithm, as seen with the model matrix enumeration. Not all applications may be amenable to this approach.

Both the applications demonstrate that major benefits occur when using a distributed system. The nature of the exact tests by both simulation and complete enumeration are particularly suited to running on a parallel or distributed system. The speed-up benefits the technique in two areas. Firstly, the reduced run-time brings a wider range of models that can be tested, enabled by the powerful and generic nature of enumerating from a model matrix. Secondly, the size of models that can be tested has been significantly increased. The low memory and communications requirements result in this application running efficiently on a cluster of workstations.

The distributed approach enables these applications to run on a wide variety of machines, operating systems and networks. The implementation of state capture enables long computations to be stopped and restarted, as well as enabling machines in the distributed system to come and go, as their usage patterns vary.

Both the prototype system and communication code in the distributed model matrix enumeration system could be used in a wide variety of statistical algorithms, providing there is suitable decomposition of the problem.

# Chapter 7

# Conclusions

This thesis has presented a number of new algorithms for the analysis of contingency tables via exact tests and complete enumeration.

Firstly, an algorithm for enumerating tables under the model of independence was developed. This algorithm uses a recursive technique, constructing the table a row at a time, and finding all subtables. Investigation into the inefficiencies of this algorithm, lead to development of a different method of constructing a table. This algorithm generates tables by splitting tables into two and generating lists of the top and bottom subtables. The two lists can be combined to provide a list of possible tables. This provides a more efficient enumeration method than the first algorithm since less work is repeated. This algorithm is less efficient than the network algorithm (Mehta & Patel, 1983), but due to smaller memory requirements, it can calculate exact p-values for examples that require too much memory when run with the network algorithm.

Following the development of algorithms for models of independence, algorithms for more complex models such as quasi-independence and quasi-symmetry are developed. For quasi-independence, where certain cells are fixed, a small change can be made to the basic algorithm for enumerating under independence, to enable enumeration of tables under quasi-independence. For more complex models, such as quasi-symmetry and uniform association, a process called enumerate-and-reject was developed. In these models, tables are enumerated from a model whose reference set encompasses the reference set for the model of interest, and tables which do not meet the extra constraints of this model are rejected. For example, for quasi-symmetry, tables are enumerated under quasi-independence with cells on the diagonal fixed and tables are rejected if the pair-wise sums for symmetric opposite cells are not the same as the observed table. This simple extension to the enumeration process further extends the range of models that can be performed using complete enumeration.

The enumerate-and-reject algorithm enumerates tables inefficiently, since it has to enumerate all tables in the encompassing set. Chapter 4 began by establishing how the constraints on the cells in a table can be specified as a model matrix. An algorithm to enumerate all possible tables that satisfy a model matrix was developed. This enables an arbitrary model to be tested, providing it can be specified using a model matrix. The example section demonstrates that a wide range of models can be tested, including the models from previous chapters. The model matrix algorithm performs much better than the enumerate-and-reject algorithm. Inefficiencies of the algorithm were identified and optimisations to the model matrix algorithm were established to minimise the impact of the inefficiencies.

The use of complete enumeration can be computationally intensive. This motivates a distributed approach to enable calculations to be performed on a number of computers, and hence return the answer in a shorter time. A prototype application using a process farming technique was developed to perform a Monte Carlo simulation to estimate exact p-values across a number of machines. The main distributed application was a distributed version of the model matrix algorithm. The tree structure of the algorithm mapped well on to a distributed environment and good performance was obtained. The algorithm was developed such that it is possible to capture the state of the system. This powerful feature lets the computation be frozen and resumed at a different location or at a later time. This enables the application to react to the dynamic nature of the distributed system.

All calculations of exact p-values were verified, where possible, with results in the literature. This demonstrates that the algorithms are enumerating the correct reference sets.

## 7.1 Discussion

Exact tests are an important and active field of research. As computing power increases, the range of tables and models for which exact p-values can be calculated with existing programs and algorithms is extended, increasing the usefulness of the technique to wider and wider research communities. In addition greater processing power encourages the development of more complex techniques, again, benefiting the research community.

The algorithms in this thesis have significantly extended the range of models for which the reference set can be enumerated. One of the aspects that makes complete enumeration attractive with complex models is that these models impose additional constraints over simpler models which often significantly reduces the size of the reference set so that the enumeration becomes feasible. For example, with the model of quasi-symmetry tests on much larger tables could be performed.

Complete enumeration can be very computationally intensive. This leads to the problem of whether it is feasible to perform an exact test or not. In the absence of a means to calculate or approximate the number of tables in a reference set, it is not an easy decision to make. Factors such as the number of cells, degrees of freedom and nature of the model matrix have to be taken into consideration. One practical approach is to run the enumeration for a period of time, and inspect how far into the reference set the enumeration has progressed.

With the constant increases in computing power, larger problems becomes feasible. The distributed version of the model matrix enumeration algorithm allows the enumeration to take place across a number of machines. The algorithm has been shown to run efficiently on networks of desktop PCs. This enables researchers to use the resources that are close to hand. Desktop PCs can be found in abundance compared with super-computers. The flexibility of the algorithm means that a researcher can potentially start an analysis on a personal machine, and the computation spreads to local machines that are idle, at night, for example. Large enumerations could occur over a period of days, weeks and months. The use of distributed computing and the farm structure used in this thesis can be applied to other statistical processes such as Monte Carlo simulation.

Although these advances in processing power and the use of distributed computing may give one or two orders of magnitude increase in performance, this is not sufficient for a large number of models and tables. For these problems, the use of simulation to estimate the exact p-value is vital. However, complete enumeration has an important role to play in this area. Being able to verify that the result of a simulation contains the true exact p-value within its confidence interval, is vital to establish the validity of the simulation technique. Hence, complete enumeration can be used intensively on larger problems by developers of less computationally intensive (and hence more widely usable) estimation routines.

## 7.2 Further Work

There remains much work on the development of new algorithms for the complete enumeration of complex models. This may take the form of developing specific algorithms for a specific model, or may adopt the generic approach used by the model matrix algorithm.

A number of areas of further work on the model matrix algorithm were proposed at the end of Chapter 4. The development of more efficient algorithms is vital, but there are other areas that need to be addressed, such as the ability to test for a number of related models simultaneously or being able to calculate or estimate the number of tables in the reference set.

An important avenue could be the investigation of combining simulation techniques and complete enumeration to develop a hybrid algorithm. Such an algorithm would simulate cells until the problem was within the reach of complete enumeration.

One of the significant problems with complete enumeration is that it is hard for the user to estimate how far into the reference set a computation is, and how long it will take to complete. Ideally, some form of visual indication would allow the user to monitor the process. This monitoring could be made more interactive by allowing the user to halt enumeration, or perhaps switch to simulation or another technique if available. In a distributed environment this can be extended to the management of processors and machines, allowing the user to allocate processing resources to techniques and models. Many of these decisions could be automated or take the form of advice to the user. The state capture facilities and dynamic nature of the algorithm can enable the computation to spread across a number of machines, allowing the user to initiate a long computation which expands to utilise idle resources on the network. Further work could examine tools to support and manage this migration process more effectively.

Chapter 5 identified that there are a number of features of statistical operations and applications that are well suited to distributed computation. This suggests that there is a need for a general purpose system to develop and run statistical applications in a distributed environment.

# Bibliography


Abelson, H., Sussman, G. J., & Sussman, J. 1985. *Structure and Interpretation of Computer Programs*. Cambridge, MA: The MIT Press.

Adams, N. M., Kirby, S. P. J., Harris, P., & Clegg, D. B. 1996. A review of parallel processing for statistical computation. *Statistics and Computing*, **6**, 37–49.

Agresti, A. 1990. *Categorical Data Analysis*. Probability and Mathematical Statistics. New York: Wiley.

Agresti, A. 1992. A survey of exact inference for contingency tables. *Statistical Science*, **7**(1), 131–177.

Agresti, A. 1996. *An Introduction to Categorical Data Analysis*. Probability and Mathematical Statistics. New York: Wiley.

Agresti, A., Wackerly, D., & Boyett, J. M. 1979. Exact conditional tests for cross-classifications: Approximation of attained significance levels. *Psychometrika*, **44**(1), 75–83.

Agresti, A., Mehta, C. R., & Patel, N. R. 1990. Exact inference for contingency tables with ordered categories. *Journal of the American Statistical Association*, **83**, 453–458.

Archer, P. G., Koprowska, I., McDonald, J. R., Naylor, B., Papanicolaou, G. N., & Umiker, W. O. 1966. A study of variablility in the interpretation of sputum cytology slides. *Cancer Research*, **26**, 2122–2144.

Bal, H. 1990. *Programming distributed systems*. New York: Prentice Hall.

Balmer, D. W. 1988. Algorithm AS 236 – Recursive enumeration of $r \times c$ tables for exact likelihood evaluation. *Applied Statistics*, **37**(2), 290–301.

Becker, M. P. 1990. Quasisymmetric models for the analysis of square contingency tables. *Journal of the Royal Statistical Society Series B*, **52**, 369–378.

Bedrick, E. J., & Hill, J. R. 1990. Outlier tests for logistic regression: A conditional approach. *Biometrika*, **77**(4), 815–827.

Birell, A.D., & Nelson, B.J. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, **2**(1), 39–59.

Bishop, Y. M. M., & Fienberg, S. E. 1969. Incomplete two-dimensional contingency tables. *Biometrics*, **25**, 119–128.

Boulton, D. M. 1974. Remark on Algorithm 434 [G2]– Exact probabilities for $R \times C$ contingency tables. *Communications of the ACM*, **17**(6).

Boulton, D. M., & Wallace, C. S. 1973. Occupancy of a rectangular array. *The Computer Journal*, **16**(1), 57–63.

Boyett, J. M. 1979. Random $R \times C$ tables with given row and column totals. *Applied Statistics*, 329–332.

Breen, R., & Hayes, B. C. 1996. Religious mobility in the UK. *Journal of the Royal Statistical Society Series A*, **159**(3), 493–504.

Contegean, E., & Devie, H. 1994. An efficient incremental algorithm for solving systems of linear diophantine equations. *Information and computation*, **113**(1), 143–172.

Davis, L. J. 1986. Exact tests for $2 \times 2$ contingency tables. *The American Statistician*, **40**(2), 139–141.

De Roure, D. C., & Michaelides, D. T. 1994. A distributed LISP-STAT environment. *Pages 371–382 of:* Dutter, R., & Grossman, W. (eds), *Proceedings of COMPSTAT 1994*. Heidelberg: Physica-Verlag.

Dybvig, R. K. 1987. *The Scheme programming language*. Englewood Cliffs, New Jersey: Prentice-Hall.

Epema, D. H. J., Livny, M., van Dantzig, R., Evans, X., & Pruyne, J. 1996. A worldwide flock of condors : Load sharing among workstation clusters. *Journal on Future Generations of Computer Systems*, **12**.

Flynn, M. J. 1972. Some computer organisations and their effectiveness. *IEEE Transactions on Computers*, **21**(9), 948–960.

Forster, J. J., McDonald, J. W., & Smith, P. W. F. 1996. Monte Carlo exact conditional tests for log-linear and Logistic Models. *Journal of the Royal Statistical Society Series B*, **58**(No. 2), 445–453.

Foster, I., Kesselman, C., & Tuecke, S. 1994. The Nexus task-parallel runtime system. *In: Proceedings of the 1st International Workshop on Parallel Processing.*

Foster, I., Geisler, J., Kesselman, C., & Tuecke, S. to appear. Multimethod communication for high-performance networked computer systems. *Journal of Parallel and Distributed Computing.*

Freeman, G. H., & Halton, J. H. 1951. Note on an exact treatment of contingency, goodness of fit and other problems of significance. *Biometrika*, **38**, 141–149.

Gail, M., & Mantel, N. 1977. Counting the number of $r \times c$ contingency tables with fixed margins. *Journal of the American Statistical Association*, **72**, 859–862.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., & Sunderam, V. 1994. *PVM: Parallel virtual machine.* MIT Press.

Gelernter, D. 1985. Generative communication in Linda. *ACM Transactions on Programming Language and Systems*, **7**(1), 80–112.

Gentleman, J. F. 1975. Algorithm AS 88. Generation of all ${}_nC_r$ combinations by simulated nested FORTRAN DO loops. *Applied Statistics*, **24**, 374–376.

Good, I. J. 1976. On the applications of symmetric Dirichlet distributions and their mixtures to contingency tables. *Annals of Statistics*, **4**, 1159–1189.

Goodman, L. A. 1968. The analysis of cross-classified data: independence, quasi-independence, and interactions in contingency tables with or without missing entries. *Journal of the American Statistical Association*, **63**, 1091–1131.

Goodman, L. A. 1979. Simple models for the analysis of association in cross-classifications having ordered categories. *Journal of the American Statistical Association*, **74**, 537–552.

Grimshaw, A. S. 1993. Easy to use object-oriented parallel programming with Mentat. *Computer*, **26**(5), 39–51.

Halstead, Jr, R. H. 1984. Implementation of multilisp: Lisp on a multiprocessor. *Pages 9–17 of: LFP '84 – ACM Symposium on Lisp and Functional Programming.*

Hout, M., Duncan, O. D., & Sobel, M. E. 1987. Association and heterogeneity: structural models of similarities and differences. *Pages 145–185 of:* Clogg, C. C. (ed), *Sociological Methodology.* Washington DC: American Sociological Association.

Joe, H. 1985. An ordering of dependence for contingency tables. *Linear Algebra and its Applications*, **70**, 89–103.

Joe, H. 1988. Extreme probabilities for contingency tables under row and column independence with application to Fisher's exact test. *Communications in Statistics-Theory and Methods*, **17**(11), 3677–3685.

Klotz, J., & Teng, J. 1977. One-way layout for counts and the exact enumeration of the Kruskal-Wallis $H$ distribution with ties. *Journal of the American Statistical Association*, **72**(357), 165–169.

Mantel, N. 1970. Incomplete contingency tables. *Biometrics*, **26**, 291–304.

March, D. L. 1972. Algorithm 434 [G2] – Exact probabilities for $R \times C$ contingency tables. *Communications of the ACM*, **15**(11).

McDonald, J. W., & Smith, P. W. F. 1995. Exact conditional tests of quasi-independence for triangular contingency tables: Estimating attained significance levels. *Applied Statistics*, **44**, 143–151.

McDonald, J. W., Smith, P. W. F., & Forster, J. J. 1996 (August). *Markov Chain Monte Carlo exact tests for logistic regression models.* Tech. rept. No. 96-3. Department of Social Statistics, University of Southampton.

Mehta, C. R., & Patel, N. R. 1980. A network algorithm for the exact treatment of the $2 \times k$ contingency table. *Communications in Statistics, Series B*, **9**(6), 649–664.

Mehta, C. R., & Patel, N. R. 1983. A network algorithm for performing Fisher's exact test in $r \times c$ contingency tables. *Journal of the American Statistical Association*, **78**(382), 427–434.

Mehta, C. R., Patel, N. R., & Senchaudhuri, P. 1988. Importance sampling for estimating exact probabilities in permutational inference. *Journal of the American Statistical Association*, **83**, 999–1005.

Mehta, C. R., Patel, N. R., & Senchaudhuri, P. 1991. Exact stratified linear rank tests for binary data. *Pages 200–207 of:* Keramidas, E. M. (ed), *Computing Science and Statistics: Proceedings of the 23rd Symposium on the Interface*. Interface Foundation.

Morgan, W. M., & Blumenstein, B. A. 1991. Exact conditional tests for hierarchical models in multidimensional contingency tables. *Applied Statistics*, **40**(3), 435–442.

Mount, J. 1995. *Application of convex sampling to optimisation and contingency table generation/counting*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

MPI Forum. 1996. MPI: A message passing interface standard. *International Journal of Supercomputer Applications*, **8**(3/4), 164–416.

Paganini-Hill, A., Krailo, M. D., & Pike, M. C. 1984. Age at natural menopause and breast cancer risk: The effect of errors in recall. *American Journal of Epidemiology*, **119**(1), 81–85.

Pagano, M., & Taylor-Halvorsen, K. 1981. An algorithm for finding the exact significance levels of $r \times c$ contingency tables. *Journal of the American Statistical Association*, **76**(376), 931–934.

Patefield, W. M. 1981. Algorithm AS 159. An efficient method of generating $R \times C$ tables with given row and column totals. *Applied Statistics*, **30**, 91–97.

Queinnec, C., & De Roure, D. C. 1992. Design of a concurrent and distributed language. *Pages 234–259 of:* Halstead, Jr, R. H., & Ito, T. (eds), *Parallel symbolic computing: Languages, systems, and applications.* Lecture Notes in Computer Science. Springer-Verlag. LNCS 748.

Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P., & Neuhause, W. 1991. *Overview of the CHORUS distributed operating systems.* Tech. rept. CS/TR-90-25. Chorus systemes.

Saunders, I. W. 1984. Algorithm AS 205 – Enumeration of $R \times C$ with repeated row totals. *Applied Statistics*, **33**, 340–352.

Senchaudhuri, P., Mehta, C. R., & Patel, N. R. 1995. Estimating exact p values by the method of control variates or Monte Carlo rescue. *Journal of the American Statistical Association*, **90**(430), 640–648.

Smith, P. W. F., & McDonald, J. W. 1994. Simulate and reject Monte Carlo exact conditional test for quasi-independence. *Pages 509–514 of:* Dutter, R., & Grossman, W. (eds), *Proceedings of COMPSTAT 1994.* Heidelberg: Physica-Verlag.

Smith, P. W. F., & McDonald, J. W. 1995. Exact conditional tests for incomplete contingency tables: estimating attained significance levels. *Statistics and Computing*, **5**, 253–256.

Smith, P. W. F., McDonald, J. W., Forster, J. J., & Berrington, A. M. 1996a. Monte Carlo exact methods used for analysing interethnic unions in Great Britain. *Applied Statistics*, **45**(No. 2), 191–202.

Smith, P. W. F., Forster, J. J., & McDonald, J. W. 1996b. Monte Carlo exact tests for square contingency tables. *Journal of the Royal Statistical Society Series A*, **159**(Part 2), 309–321.

Sunderam, V. S. 1990. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, **2**(4).

Tanenbaum, A. S., & Mullender, S. J. 1981. An overview of the Amoeba distributed operating system. *Operating systems review*, **15**(3), 51–64.

Tierney, L. 1990. *LISP-STAT: An object-oriented environment for statistical computing and dynamic graphics.* New York: J. Wiley and Sons.

Verbeek, A., & Kroonenberg, P. M. 1985. A survey of algorithms for exact distributions of test statistics in $r \times c$ contingency tables with fixed margins. *Computational Statistics and Data Analysis*, **3**, 159–185.

von Eye, A., & Spiel, C. 1996. Standard and nonstandard log-linear symmetry models for measuring change in categorical variables. *The American Statistician*, **50**(4), 300–305.

Whittaker, J. 1990. *Graphical models in applied multivariate statistics.* Chichester: Wiley.