Generally Applicable Heuristics for Global Optimisation: An Investigation of Algorithm Performance for the Euclidean Traveling Salesman Problem

by

Grant Telfar

A reading paper
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Statistics and Operations Research

October 23, 1994

Institute of Statistics and Operations Research
VICTORIA UNIVERSITY OF WELLINGTON

Abstract

The purpose of this reading paper is to empirically analyse and compare several heuristic algorithms that can be applied to combinatorial optimisation problems. The main focus of the paper is heuristic paradigms that are generally applicable to a wide variety of problem types. The three techniques investigated here are repeated local search, simulated annealing, and genetic algorithms.

Each heuristic is investigated individually, with a small amount of theoretical discussion, to determine how the performance of each method can be changed by altering algorithm parameters. The Traveling Salesman Problem is used as a problem indicative of combinatorial optimisation to test the three heuristics. It is an NP-hard problem that shares many properties common to other combinatorial optimisation problems and it is hoped that the behaviour of the heuristics when applied to the Traveling Salesman Problem will be similar to that experienced for other combinatorial problems. To gain some insight into the relative performance of the three heuristics when applied to combinatorial optimisation problems, the algorithms are run on several Traveling Salesman Problem instances and results from the various algorithms are compared to determine which one is 'best'.

Acknowledgments

I would like to thank my thesis supervisor, Professor G A Vignaux, and all the staff of the Institute of Statistics and Operations Research for their invaluable assistance. All help, small and large, was greatly appreciated but I would especially like to thank Geoff Grimwood whose help in every area was instrumental in the completion of this reading paper. Thanks to all for keeping me sane.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This paper investigates the use of heuristic algorithms for *global optimisation*, especially for *combinatorial optimisation* problems. The prototypical example of combinatorial optimisation is the *Traveling Salesman Problem*, in which the best (i.e. the shortest) tour that visits each of N cities exactly once is to be found, starting and finishing the tour at the same city. Such problems are deceptively difficult and algorithms to solve combinatorial problems exactly, i.e. that find *the* best solution, often take an unacceptably long time. Thus non-exact or *heuristic* algorithms are used to produce 'good' quality solutions to a problem in a 'reasonable' amount of time.

When selecting an heuristic algorithm for a specific problem there is a general decision to be made, the algorithm can be either *problem specific* or *generally applicable*. A problem specific heuristic is very narrow in its application since it exploits information about the problem at hand to ease its search. A robust, flexible algorithm is much more general in its application and can be applied, with only very minor changes, to a wide spectrum of problem types. Since robust and flexible algorithms necessarily use very little information about the structure of a problem, they tend to be slower than problem specific algorithms (potentially orders of magnitude slower), however the quality of solution is often comparable.

Why then would a generally applicable heuristic be used when they often produce solutions of inferior quality in a much longer time than an algorithm tailored to the problem? It ultimately depends on the reason for which the algorithm is being used and is also a matter of personal philosophy. If you are happy writing an algorithm from scratch every time a different problem presents itself, then tailored algorithms are the path to follow. However if you prefer to use the same technique time and again for a huge variety of problems, involving only minor changes in your algorithm, then robust, flexible algorithms are the solution. Also, if the problem under consideration has no known solution method, then designing a tailored algorithm will involve constructing, from scratch, a method to solve (i.e. find 'good' solutions to) the problem. Most robust, flexible algorithms, on the other hand, require only a method of determining the quality of a solution and a method to move between solutions, both of which are relatively simple tasks for the majority of problem types.

Specific heuristic paradigms that are generally applicable include:

- Simulated Annealing
- Genetic Algorithms
- Threshold Accepting Methods
    1. The Great Deluge Algorithm
    2. The Record-to-Record-Travel Algorithm
- Tabu Search
- Artificial Neural Nets
- Repeated local search

In chapter 2 an introduction to the Traveling Salesman Problem, a discussion of its inherent difficulty, and a brief look at some tailored and generally applicable heuristics is given. Chapter 3 contains a rigourous discussion of the Genetic Algorithm paradigm, and an empirical study of its performance when applied to a variety of Traveling Salesman Problems. Threshold Accepting methods, specifically the Great Deluge Algorithm, are discussed in chapter 4 and the performance of the Great Deluge Algorithm when applied to the Traveling Salesman Problems is investigated. A comparison of Genetic Algorithms, the Great Deluge Algorithm, and repeated local search, as well as a tailored Traveling Salesman Problem heuristic, the k-opt algorithm, is made in chapter 5 using a variety of Traveling Salesman Problem instances as a test bed for performance. Appendix A contains details of a common operator used in many generally applicable heuristics when applied to the Traveling Salesman Problem, the 2-change (or 2-opt) neighbourhood generation operator. In appendix B the Traveling Salesman Problem instances used in the empirical study of the various heuristics are given. Finally, appendix C contains full program listings of the code used to study the heuristics.

Chapter 2

The Traveling Salesman Problem and Heuristics

2.1 Introduction

A salesman wishes to visit a number of cities in a plane in a particular order, starting and finishing at the same city and visiting every other city exactly once. It seems sensible to select this *tour* in a manner that minimises the total distance traveled, a quantity known as the *tour length*. Informally this defines an intriguing problem that has captivated researchers for many years, the *Traveling Salesman Problem* (TSP). In the *symmetric* TSP, which is the focus of this paper, the distance from city-i to city-j is the same as the distance from city-j to city-i. Furthermore, an *euclidean* TSP measures distance between cities in using a euclidean metric[1]. Figure 2.1 shows an optimal tour for a 20-city symmetric, euclidean TSP.

The TSP is not often widely applicable in its purest form, but can be successfully applied in a generalised manner to a number of fields, from VLSI chip fabrication to X-ray crystallography. It is of immense theoretical interest for a number of scientific fields, largely because it shares many characteristics typifying other problems in its genre, that of *combinatorial optimisation*. Common properties of combinatorial optimisation problems that make them intriguing are their simplicity of expression, practical application, and implications for other scientific fields.

Combinatorial optimisation (see section 2.2) deals with optimising a function over a discrete set of possible solutions. Normal calculus methods of optimising do not apply since they require a continuous environment. Instead other methods have been developed to deal with combinatorial optimisation problems, methods such as *linear programming* [2] (LP), with the TSP often acting as a test arena. In principle LP and other solution methods can solve combinatorial optimisation problems like the TSP. However due

[1] In an euclidean metric the distance between two points (x_1, y_1) and (x_2, y_2) in R^2 is given by $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Thus a Euclidean TSP is by default symmetric.

[2] Linear programming was developed in the late 1940s, by George Dantzig and others. The method involves finding the optimum of a linear function on a polyhedron described by a system of linear equations and inequalities.

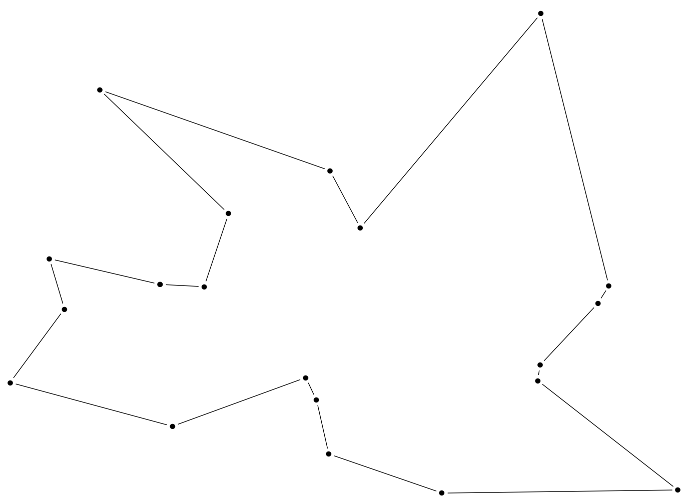

Figure 2.1. The optimal tour for a 20-city TSP instance with a tour length of 4.01616 units.

to the inherent difficulty of the TSP a satisfactorily efficient algorithm has never been found and so the TSP remains 'unsolved' meaning most algorithms used to find solutions for the problem are of an heuristic nature with no guarantees of optimality. The TSP is among the most prominent of the unsolved combinatorial optimisation problems.

For a rigourous description of the TSP and its many variations along with both exact and heuristic solution methods for many of these problems the reader is directed to Lawler *et al.* [27].

2.2 Combinatorial Optimisation

The study of finding the optimal solution to problems which can be be structured as a function of *decision variables* subject to certain *constraints* has attracted much interest in modern times. A general formulation[3]

$$\begin{array}{lll} Minimise & f(x) & \\ such\,that & g_i(x) \geq b_i & i = 1, \cdots, m \\ & h_j(x) = c_j & j = 1, \cdots, n \end{array}$$

where x is a vector of decision variables and f, g_i, and h_j are general functions. By restricting the types of allowable decision variables and functions, many classes of problems fitting the above general scheme are defined. If decision variables are allowed to take continuous non-negative values and

[3]Formulation assumes minimisation of f but the modifications for maximisation are clear.

f, g_i, and h_j are restricted to be linear functions then problems in the class of LP are defined. Another class is defined when decision variables are discrete and all solutions to a problem P (i.e the *solution space* of P) form a set of discrete objects. The problem of finding optimal solutions to such problems is thus termed combinatorial optimisation.

Early attempts at solving combinatorial problems generally involved reformulations in order to apply LP techniques, which at the time were attracting much interest. However transforming many combinatorial problems is often difficult and resulting formulations tend to be large, often requiring integer restrictions, thus defining a class of problems solved through *integer programming* (IP). This was troublesome at the time since the methods used to solve LP and especially IP problems, with the existing technology, were unable to handle large numbers of variables in a satisfactory manner. The IP approach is never-the-less useful for combinatorial optimisation in finding lower bounds to problem instances through the technique of *Lagrangean relaxation*[4]. Due to the practical restrictions encountered in the IP solution method, new optimisation techniques were explored. With modern equipment however the LP and IP techniques are not overly problematic for most real world problems.

In optimising a combinatorial problem the 'best' solution to a problem instance over the solution space for that problem is sought. Solution quality is usually defined in terms of a *cost* or *objective* function. In the quest for optimality it is necessary to distinguish between a *local* and a *global* optimum, but must first define the concept of a *neighbourhood*. A solution x to a problem P has an associated neighbourhood $N(x, \sigma)$, defined as the set of all solutions that can be reached from x by a simple operation σ. If a particular solution y is better than all members of $N(y, \sigma)$ then y is locally optimal. This solution y is globally optimal if there are no other solutions in the entire solution space of P better than y.

Many methods for dealing combinatorial problems have been developed. These methods fall into one of two classes *exact* or *heuristic*. An exact solution method is guaranteed to find *the* optimal solution to any instance of a problem. Heuristic methods seek near-optimal solutions to a problem instance at a reasonable computational cost, with no guarantee of feasibility or optimality.

The question naturally arises of why an heuristic method would be favoured over an exact one. The answer lies in the inherent difficulty of many combinatorial problems.

In any combinatorial problem the brute force approach of *complete enumeration* can be used to list all feasible solutions and their associated costs to a problem instance. Picking the best solution from this list is guaranteed to find the global optimum. However in practice this can require vast amounts of computing time. Consider the TSP. If a problem instance has n cities then there are $\frac{(n-1)!}{2}$ feasible solutions to consider. Putting this in perspective, this means if the solution to a 20-city problem can be found using complete

[4]For an extended discussion on Lagrangean relaxation the reader is directed to Reeves [36].

enumeration in 1 hour, then to find the solution to a 25-city problem would take nearly 6 centuries! This type of growth rate in computing time is termed *exponential*, making complete enumeration a poor choice of solution method.

It is possible to find the exact solution to many problem instances more efficiently than for complete enumeration by taking advantage of the structure of a problem. Such a method is the Simplex method used in LP[5]. With increasing computing power the sizes of the problems tackled in an exact manner have increased and much consideration has been given as to how solution time varies with problem size. The distinction is thus made between techniques which solve a problem in *polynomial-time* and those that solve problems in *exponential-time*. Polynomial-time methods display a computation time that is a polynomial function of the problem size. Exponential techniques have a computation time that is an exponential function of the problem size. All known methods for solving the TSP exactly fall into the latter class.

A question that was first considered in the late 1960s was whether a polynomial optimising *algorithm* existed for problems such as the TSP. This led to the development of the theory of *computational complexity*. It was discovered that if this question was true for the TSP then it was also true for a number of other difficult problems. Despite much research, no such polynomial algorithm has been found and it is widely supposed that the answer to the above question is thus 'no'. However a proof of this hypothesis has never been given. It would seem then that such is the inherent difficulty of problems like the TSP that any reasonable solution method must be an heuristic one.

2.3 Computational Complexity

Computational complexity was developed largely in the 1970s to evaluate whether algorithms should be classified as 'easy' or 'hard'. Combinatorial optimisation has given computational complexity many motivations and insights, especially so the TSP, serving as a test-bed for many new algorithmic ideas. The TSP was one of the first optimisation problems conjectured to be 'hard' in a sense defined below.

2.3.1 Algorithms and Problems

An algorithm is a step-by-step procedure for solving a problem. A problem can be viewed as a domain of problem instances, with a question that can be asked about any of the instances. A problem can be described as a generic instance together with a question that can be asked about this generic formulation. The TSP is thus fully described by[6]:

[5]Although the Simplex method has been shown to have exponential solution time in extreme cases.

[6]This assumes a symmetric TSP and that intercity distances are integers. The second assumption is made without loss of generality since in a computer irrational numbers are

	TSP
INSTANCE	Integer $n \geq 3$ and $n \times n$ matrix $C = (c_{ij})$ where each c_{ij} is a non-negative integer.
QUESTION	Which cyclic permutation π of the integers $1, \cdots, n$ minimises the sum $\sum_{i=1}^{n} c_{i\pi(i)}$?

An algorithm A *solves* a problem P if, given any instance I of P as input data, it will generate the answer of P's question. So for the TSP, given the distance matrix C and the number of cities n the minimum length tour is generated. It is known that for some problems no algorithm exists (i.e. an algorithm that will generate an answer to the problem's question for all instances). In problems like the TSP however, an algorithm does exist, that of complete enumeration.

When estimating the running time of an algorithm A, 'time' is expressed as a function of the problem size (in the case of the TSP this is the number of cities n), and measures 'steps' rather than the actual time it took to run the algorithm. *Order* notation (or O-notation) is used to express this running time. If A has a running time of $O(f(n))$ then there is some constant c such that the algorithm's running time for all instances is bounded by $cf(n)$ (Complete enumeration for the TSP takes time $O((n-1)!)$).

An algorithm may or may not be *well behaved*. It is well behaved if one can accurately predict the number of steps required for a given problem instance, and this prediction does not vary from one instance to another with the same size. Some algorithms however are not well behaved and the number of steps required to solve problem instances of the same size can vary widely. In assigning a *complexity* function to a problem that has a unique value for each problem size, there are two approaches in measuring how long an algorithm takes. We can either count the average or the maximum number of steps over all instances of the same size. The second form of measurement, *worst-case analysis*, has the disadvantage that bad performances might be very rare in practice. *Average-case analysis* involves choosing a probability function for the instances of the same size, a potentially difficult task. Another difficulty of average-case analysis is the dependence between the stages of many algorithms, making analysis complicated.

Generally, when evaluating a problem instance the average running time is not really an issue, but through worst-case analysis there is at least a guarantee of how long it will take. This worst-case approach leads to the theory of $\mathcal{NP}$-completeness.

2.3.2 Polynomial Time

An algorithm is considered 'good' when it is efficient enough to use in practice, here efficiency is defined as having worst-case complexity bounded by

approximated by rationals and with an appropriate scale these rationals can be converted to integers.

a polynomial function of the problem size n. This splits algorithms into two distinct classes, polynomial and exponential. Although exponential algorithms may do better than polynomial algorithms for small n, exponential algorithms will perform much worse than polynomial algorithms as the problem size increases.

The precise definition of problem size can have a large bearing on the function derived for the worst-case complexity. But as long as the size is a measurement in some sense of the problem dimension, a polynomial algorithm will remain a polynomial algorithm and an exponential algorithm will remain an exponential algorithm, regardless of the choice of definition[7].

2.3.3 $\mathcal{P}$ and $\mathcal{NP}$

A problem is 'easy' if it can be solved by a 'good' algorithm, one that has polynomial time complexity. Similarly all known algorithms for a 'hard' problem have exponential-time complexity. This distinction between hard and easy problems leads to the theory of complexity.

Complexity theory restricts itself to *decision problems*, a problem that requires only a yes/no answer. Any optimisation problem can be reformulated in terms of a decision problem with the addition of a bound B to the input data. The decision problem for the TSP then becomes:

	TSP Decision
INSTANCE	Integer $n \geq 3$, $n \times n$ matrix $C = (c_{ij})$ where each c_{ij} is a nonnegative integer, and an integer $B \geq 0$.
QUESTION	Is there a cyclic permutation π of the integers $1, \cdots, n$ such that $\sum_{i=1}^{n} c_{i\pi(i)} \leq B$?

If a polynomial-time algorithm exists for the TSP then one also exists for TSP Decision, a result that is true for all optimisation problems as long as the cost of a solution can be computed in polynomial-time. The converse is also true for the TSP, and many other problems, by the following theorem, a result that can be proved for most optimisation problems[8]:

There is a polynomial-time algorithm for TSP iff there is a polynomial-time algorithm for TSP Decision.

The complexities of different decision problems can be related using the idea of the *polynomial reduction*. Problem A is *polynomial-time reducible* to problem B if there is a polynomial-time algorithm for A which uses an algorithm for B as a subroutine and the time taken for this subroutine is

[7] The theory requires that the measure of input size must reflect (to within a polynomial) the actual length of a concise coding of the problem instance in question as a sequence of symbols (i.e. as it would be stored in a computer).

[8] A discussion of this theorem, but not a formalised proof, can be found in Johnson and Papadimitriou [25].

counted as a single step. This gives the following powerful result[9]:

If there is a polynomial-time reduction from A to B, and there is a polynomial-time algorithm for B then there is a polynomial-time algorithm for A.

Thus three essential types of reductions for a problem A exist:

- Reductions proving A is easy-by reducing A to a problem known to be solvable in polynomial-time.
- Reductions that prove A is hard-by showing some known hard problem reduces to A.
- Reductions that prove nothing -by reducing A to a problem known to be hard.

Polynomial reductions (between decision problems) that call the hypothetical subroutine exactly once, returning the answer to the subroutine as their own answer, are known as *polynomial transforms*. Notationally, if problem A is polynomial transformable to problem B then $A \propto B$.

The classes $\mathcal{P}$ and $\mathcal{NP}$ can now be defined. The class $\mathcal{P}$ consists of all those decision problems for which a polynomial-time algorithm exists, i.e. 'good' problems. The question that must be considered for the TSP is whether or not TSP Decision is in $\mathcal{P}$. Current thought has the answer to this question as 'no', but a proof has never been given. Problems do exist that have been proved to be outside of $\mathcal{P}$, i.e. 'hard', but TSP Decision is qualitatively easier than all of these hard problems.

Characteristics that make TSP Decision easier than known hard problems are shared by many combinatorial problems. Three different but equal formulations for these characteristics exist and define the class of decision problem known as $\mathcal{NP}$.

The succinct certificate property: Any 'yes' instance to a decision problem has a succinct certificate, a mathematical object that establishes that the problem instance warrants a yes answer. If this certificate is succinct, i.e bounded in size by a polynomial of the instance size, and it is possible to check in polynomial-time that a certificate is valid for a given instance, then the decision problem is in $\mathcal{NP}$.

A nondeterministic algorithm: A nondeterministic algorithm can divide the computation into two parallel processes. With repeated divisions the problem branches into a *tree* of parallel computations. If any of these *branches* returns a 'yes' then the answer to the nondeterministic algorithm is 'yes'. The answer is 'no' if none of the branches ever answers 'yes' (See figure 2.2).

A nondeterministic algorithm solves a decision problem in polynomial-time if it gives an answer for each instance and the number of steps used by the first branch to report 'yes' is bounded by a polynomial in

[9] Proved from the fact that the composition of two polynomials is a polynomial.

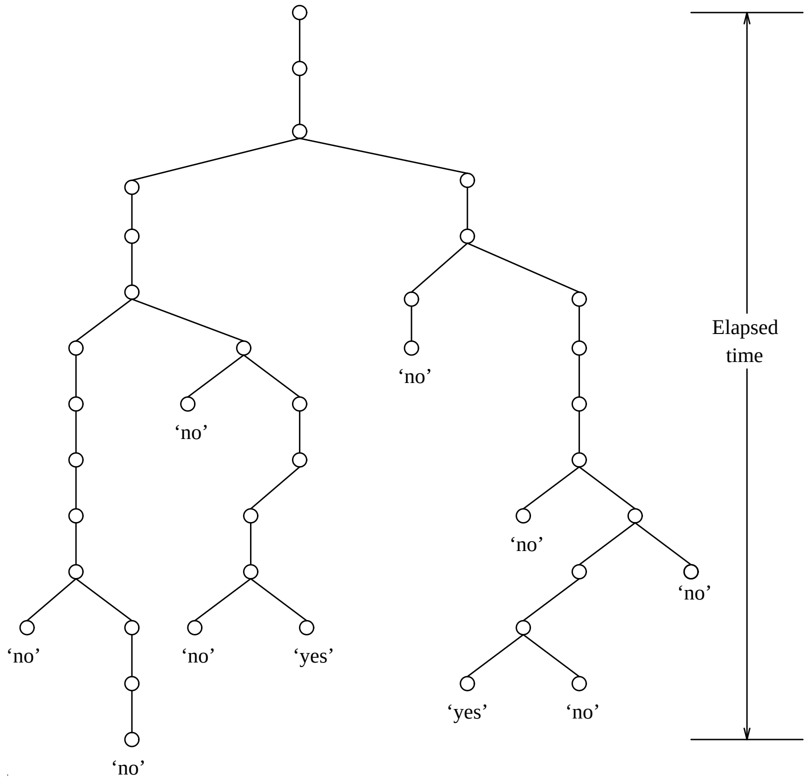

Figure 2.2. A possible execution history of a nondeterministic algorithm. By branching into parallel processes, the nondeterministic algorithm solves a problem if any of the branches returns a 'yes' answer to the question posed by the problem. If all the branches return 'no' then the algorithm does not solve the problem.

the size of the instance. If a problem can be solved in polynomial-time by a nondeterministic algorithm then it belongs in $\mathcal{NP}$.

Reduction to IP: If a problem can be transformed to IP in polynomial-time then the problem belongs in $\mathcal{NP}$.

Since ordinary deterministic algorithms are special cases of nondeterministic algorithm (when no branching occurs) $\mathcal{P}\subseteq\mathcal{NP}$. One of the most important open questions in mathematics is the conjecture that $\mathcal{P}\neq\mathcal{NP}$. Most researchers think it is unlikely, certainly no algorithm solving an $\mathcal{NP}$ problem such as TSP Decision in polynomial-time has been found. Since no proof exists it cannot be shown that a problem like TSP Decision is *intractable* i.e. unable to be solved in polynomial-time, however it is known that if TSP Decision is intractable then $\mathcal{P}\neq\mathcal{NP}$.

2.3.4 $\mathcal{NP}$-completeness

A decision problem A is *$\mathcal{NP}$-complete* if $A \in\mathcal{NP}$ and every problem in $\mathcal{NP}$ is polynomially transformable to A, furthermore if A is $\mathcal{NP}$-complete then it is solvable in polynomial-time if and only if $\mathcal{P}=\mathcal{NP}$, a seemingly unlikely result. An extensive listing, including TSP Decision, of $\mathcal{NP}$-complete problems is given by Garey and Johnson [14].

Since polynomial transformability is a transitive operation, if any known $\mathcal{NP}$-complete problem B transforms to a problem $A \in\mathcal{NP}$ then A is also $\mathcal{NP}$-complete. Thus many variations of TSP Decision are similarly classified $\mathcal{NP}$-complete, and there is strong evidence to support the supposed intractability of TSP Decision (and therefore TSP itself). Note however that for technical reasons many problems that can be solved in polynomial-time if and only if $\mathcal{P}=\mathcal{NP}$ can not be classified as $\mathcal{NP}$-complete, such as the TSP optimisation version (since it is not a decision problem). Similarly a problem in $\mathcal{NP}$ to which a $\mathcal{NP}$-complete problem is reducible (but not transformable) is not $\mathcal{NP}$-complete even though it is just as inherently difficult as any $\mathcal{NP}$-complete problem. Also if a problem A is not known to be in $\mathcal{NP}$ but every problem in $\mathcal{NP}$ is polynomially reducible to A then A is *$\mathcal{NP}$-hard*. $\mathcal{NP}$-hard problems are not restricted to decision problems. The TSP optimisation version is classified $\mathcal{NP}$-hard.

2.3.5 Complexity and Heuristics

Complexity theory offers many more distinctions that can be made between algorithms than have been given in the preceding subsections. Concepts such as *co$\mathcal{NP}$* and *#-complete* are interesting theoretical extensions to the ideas discussed thus far. However there is already sufficient evidence to suggest the TSP is of such an inherently difficult nature that an heuristic approach is warranted. Should a more rigourous description of complexity theory be required the reader is directed to Garey and Johnson [14] or Johnson and Papadimitriou [25].

Aside from the complexity consideration another argument for the use of a heuristic algorithm is of a more philosophical nature. Generally a model of a 'real-world' problem is being optimised. There is no reason to suppose that the optimal answer to the constructed model is in fact the optimal answer to the problem in the real-world. It follows that spending much time finding the exact answer to a problem is often unnecessary when the answer found is not guaranteed to be the exact solution to the real problem under consideration. This idea generally applies to any modeling situation, but in this paper is restricted to the TSP, a problem of a more theoretical nature.

2.4 History of the Traveling Salesman Problem

The Traveling Salesman Problem was first considered in a true sense in connection with Graph Theory, a branch of combinatorial mathematics. A *graph* is defined as a set of *vertices* with some pairs of vertices joined by *edges*. A *cycle* is a set of vertices that can traversed via a set of connecting edges, beginning and finishing at the same vertex, encountering each vertex exactly once. A *Hamiltonian*[10] cycle is a cycle containing all vertices in the graph. Thus the TSP for a graph is the problem of finding the shortest Hamiltonian cycle.

Prior to the consideration given Hamiltonian graphs in the 19th century, similar tours had been considered. In the 17th century Euler and Vandemond discussed the problem of the *knight's tour* a Hamiltonian tour of a chess board using only the moves allowable to a knight as 'edges' between 'vertices' (an individual square).

The Reverand T. P. Kirkman and Sir William Rowan Hamilton independently in the middle of the 19th century where the first to investigate Hamiltonian cycles in a graph, although not much attention was given to the shortest Hamiltonian cycle. Interestingly Hamilton even marketed a game (called the Icosian Game) involving graphs and cycles, a variant of the game even bore the name the 'Traveler's Dodecahedron'.

The term 'Traveling Salesman Problem' was first used in mathematical circles in 1931-1932 but the actual idea of the TSP was first recorded in a German book[11] written by a traveling salesman as a guide to aspiring youngsters. Who actually coined the phrase in connection with a mathematically formalised model is unknown, although Merrill Flood first published the term in mathematical circles as well as in the burgeoning field of Operations Research. He attributes the phrase to A. W. Tucker in 1937 who remembered it from Hassler Whitney in 1931-1932.

Flood in 1948 popularised the TSP at the RAND corporation, formalising the problem as follows:

[10]From the 19th century mathematician Sir William Rowan Hamilton.

[11]"The Traveling Salesman, how he should be and what he should do to get Commissions and to be Successful in his Business. By a veteran Traveling Salesman."

$$\begin{array}{ll} Minimise & \sum_{i=1}^{n} c_{i\pi(i)} \\ where & There\ are\ n\ cities\ and \\ & a\ cyclic\ permutation\ \pi\ of\ the\ n\ cities \\ & and\ a\ n \times n\ distance\ matrix\ C = (c_{ij}) \end{array}$$

Thereafter the TSP became popular largely due to connections with combinatorial problems such as the *assignment problem* (AP), the *transportation problem* (TP) and other LP problems, seemingly similar to these problems but harder to solve, presenting researchers with a challenge.

The seminal paper on the TSP by Dantzig *et al.* [6] in 1954 was of great importance in the history of combinatorial optimisation. Occurring at the same time as the explosion of interest in LP, TSP resisted being solved by these new methods, even with the development of some of the first practical computers[12]. To understand their paper some knowledge of the AP is required.

The assignment problem consists of choosing n elements from an $n \times n$ matrix $C = (c_{ij})$, one from each row and column, in order to minimise the sum of the elements selected. An LP approach to this problem is to consider the polyhedron P in n^2-space given by all matrices $X = (x_{ij})$ subject to certain conditions. The AP can be formalised as follows:

$$\begin{array}{ll} minimise & \sum_i \sum_j c_{ij} x_{ij} \\ such\,that & x_{ij} \geq 0 \\ & \sum_j x_{ij} = 1\ \forall i \\ & \sum_i x_{ij} = 1\ \forall j \end{array}$$

Each matrix X is a vertex of the polyhedron,each row and column containing a single 1, with all other entries 0.

By defining c_{ij} to be the distance between city i and j then the TSP is similar to the AP, since if $x_{ij} = 1$ then the salesman moves from i to j. However solving this problem may generate disjoint sub-tours. To prevent this situation, 2^{n-1} additional constraints are required. With these extra constraints vertices of the new polyhedron R are no longer guaranteed to give integer results, so we need to ensure variables only take the values 0 or 1. Thus the TSP becomes an IP problem, a class that had no solution method in the early 1950s.

Dantzig *et al.* [6] suggested a new method to overcome these difficulties, essentially a branch and bound algorithm. They proposed that starting from an optimal or near optimal tour it might be possible to prove optimality by invoking only a few extra inequalities (*cuts*). Using this technique to find the optimal solution to a 49-city TSP, they showed also that *cutting plane techniques* where relevant to IP generally as well as for the TSP[13]. From these beginnings a theoretically more sensible solution method for the TSP than

[12]LP was one of the influential forces in the sponsoring of the early computers UNIVAC and SEAC at the National Bureau of Standards.

[13]Branch and bound algorithms work as follows: A variable x that must be an integer

that of complete enumeration could now be employed. However for any large problem the time required to produce a solution was still prohibitive.

Soon after these developments, at the end of the 1960s, distinctions between hard problems such as the TSP and easy problems such as the AP were formalised and have been discussed in more detail in section 2.3. Computational complexity theory confirms the inherent difficulty of the TSP, but despite this a solution to the problem is still required. Three immediate options become clear:

1. Reject the hypothesis that the TSP is a hard problem and search for a structure to the problem that makes it easy. Ideas such as cutting planes and branch and bound methods fall into this category.

2. Insist on an exact solution at the expense of time considerations.

3. Find heuristic solutions in reasonable time at the expense of global optimality.

In practice, for any large problem[14], heuristic methods are generally employed. However the border between exact and heuristic algorithms is not all that strict since it is always possible to use an heuristic method in a branch and bound algorithm to find the exact solution of any problem.

2.5 Heuristics

For both exact and heuristic algorithms the distinction can be made between *general* and *tailored* algorithms. A tailored algorithm uses problem-specific information and can only be applied to a small set of problems, requiring the development of a new algorithm for each type of combinatorial problem. Conversely general algorithms are problem independent and will work for any number of different problem types. Again a trade off is generally encountered. Either a tailored algorithm can be used, which makes use of problem structure and is comparatively quick but has limited applicability, or a general algorithm can be used, which is usually slower but has wide potential for application.

2.5.1 Tailored Heuristics

There are a number of successful tailored heuristic algorithms for the TSP. Some are described briefly below:

but is currently not (after some LP solution) is *branched* on nearby integer values to give two more LP problems (with restrictions) which must be evaluated. This process is repeated as often as necessary giving a tree, with LPs as nodes, that doesn't grow beyond some nodes due to bounds on the solution.

[14]With the current level of computing power it is widely accepted that problem instances of up to 1000 cities are solvable exactly in reasonable amounts of time.

The Nearest Neighbour Algorithm: A 'greedy' algorithm that constructs a tour by considering each city in turn and adding it to the existing tour.

1. Begin with a partial tour consisting of a single city a_1.
2. With the current partial tour $a_1 \cdots a_k,\ k < n$ choose a_{k+1}, from the cities not already on the tour, where a_{k+1} is the closest city to a_k.
3. Add a_{k+1} to the tour and return to step 2, halting the procedure when all cities have been accounted for.

The Minimum Spanning Tree Algorithm: An algorithm that begins with a 'good' tour, where some cities are visited more than once, then eliminates these repetitions to give a TSP tour.

A *minimum spanning tree* (MST) for n cities is a set of $n-1$ edges that minimally connect the cities into a single connected component (a minimally *connected graph*).

Depth first traversal of a MST is a tour beginning at a *leaf* of the tree, where a leaf is vertex of *degree* 1 (connected by a single edge). The tour is constructed by following any untraversed edge to a new city. If all edges have been traversed, return along the edge by which the city was first reached until finally returning to the leaf that was started with.

Shortcuts can be introduced into a depth first traversal by skipping over cities already visited in the traversal and proceeding directly to the next unvisited city.

The MST algorithm is then as follows:

1. Find a minimum spanning tree for all cities.
2. Construct a depth first traversal of the MST.
3. Introduce shortcuts into the depth first traversal to eliminate repetition of cities to obtain a TSP tour.

Christofides' Algorithm: Similar to the MST algorithm in that it forces a 'good' tour to be a TSP tour by eliminating repetitions. The difference is in the way the initial tour is selected.

The initial tour is selected using the concept of a *Eulerian graph*, a connected graph in which every city has an even degree. Beginning with the MST it is possible to construct a Eulerian graph by connecting pairs of odd degree cities[15]. Matching the pairs in an optimal manner gives a good (but not the best) Eulerian graph.

A *minimum weight* matching for a set of nodes from a graph is a collection of edges T such that each node in the set is the endpoint of exactly one edge in T, and the total length of the edges in T is minimal.

Thus the algorithm is loosely as follows:

[15] Graph theory says the number of odd degree nodes in any graph is guaranteed to be even. So in matching pairs of cities there will be no cities left out.

1. Find a minimum spanning tree for all cities.
2. Construct a minimum weight matching for the set of all odd-degree nodes in the MST.
3. Calculate an Eulerian tour for the MST with the additional edges and convert to a TSP tour using short cuts.

The Variable k-opt Algorithm: An iterative improvement algorithm that repeatedly improves on the current solution until near optimality is found. Lin and Kernighan's variable k-opt is a generalisation of the k-opt algorithm. This simple algorithm searches the neighbourhood of the current solution as defined by the k-opt (fixed k) neighbourhood move, described in subsection 2.5.2, and accepts any tour that is an improvement over the present configuration. When no better solution in the current neighbourhood can be found the algorithm terminates. The larger the value of k the more likely the final solution is to be optimal. However as k increases the neighbourhood size grows quickly, making values of k beyound 3, time consuming.

Instead of keeping the value of k fixed, the variable k-opt algorithm decides dynamically which value of k to use at each iteration. To determine the value of k for each iteration of the algorithm, if the value $k = l$ has been decided upon, a series of stopping conditions must be met to ensure that $k = l + 1$ is not a better choice. Once a value for k has been selected the neighbourhood is searched for an improved tour. A good description of the algorithm can be found in Lawler *et al.* [27].

2.5.2 General Heuristics

Generally applicable heuristics appropriate for a wide range of combinatorial problems have gained much attention in recent years. The TSP is used as a test arena for comparing some different general heuristics. All these algorithms have the potential to solve other problems with only a small number of alterations given by the problem *structure*. For many heuristics the only structure required is the notion of a cost function while others necessitate a neighbourhood definition. Both of these concepts are discussed in section 2.2.

The TSP has a clearly defined cost function, the sum of the distances between cities in the tour. The definition of a neighbourhood depends on the operation σ. A widespread neighbourhood *move* first used in a heuristic by Lin and Kernighan [28] is the k*-change*. The k-change operation for a solution i to the TSP defines a neighbourhood for i consisting of the set of all solutions generated by removing k edges from the tour associated with i and replacing them with k new edges in a manner that will give a tour.

The 2-change operation is then just the reversal of the order in which the cities are traversed inbetween two arbitrary cities in a tour; see figure 2.3. Appendix A has details of this 2-change operation.

With a suitable structure defined, a generally applicable heuristic method can be applied. Most of these algorithms revolve around the common theme

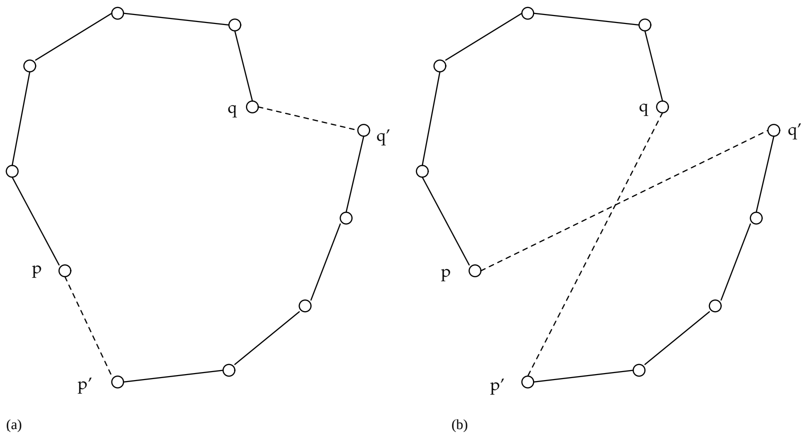

Figure 2.3. Example of a interchange operation on a instance of the TSP between the initial tour (a) and the tour resulting from a 2-change (b). If p and q are the cities selected by the operation then the links pp' and qq' are broken and replaced by pq' and qp'.

of iteratively improving the value of the cost function. Beginning with an initial tour, repeatedly make changes using some operator to ultimately arrive at a solution that is a 'good' (but not usually optimal) tour.

The most primitive of these heuristics is *local search* discussed in section 2.6. Local search has two major drawbacks, firstly the algorithm terminates in a local optimum with no guarantee of even being close to the optimal solution, and secondly, quality of solution is heavily dependent on the choice of initial solution. Thus generally applicable heuristic algorithms are sought that have the potential to avoid being trapped in a local minimum and give solutions close to the true optimum. In the remainder of this paper I will be investigating several such heuristics, analysing performance on the basis of results gained from the TSP. In the following sections of this chapter I will give brief descriptions of some generally applicable heuristic algorithms that are currently in wide use.

2.6 Local Search

A local search algorithm is an iterative improvement technique that can be applied generally to any problem that has a well defined solution space S, an evaluation function $f(x)$ for a solution $x \in S$, and a neighbourhood generation mechanism $N(x, \sigma)$. Given an initial configuration, which is often randomly generated, the algorithm generates a sequence of solutions by repeatedly searching the neighbourhoods of the current solution for a configuration to become the new current solution. A solution is accepted if it is of a better quality[16] than that of the current solution. The algorithm terminates when no solution in the neighbourhood of the current solution

[16]The algorithm can either accept the first solution of better quality found or the solution in the neighbourhood that gives the greatest improvement in quality, termed the *steepest descent* (or *steepest ascent*) approach.

is of a better quality. A generalised algorithm for local search, implemented for minimisation, is given in figure 2.4.

begin
  choose an initial configuration $x \in S$
  repeat
    choose a neighbour $y \in N(x, \sigma)$
    if $f(y) < f(x)$ **then**
      $x := y$
    end if
  until $f(y) \geq f(x) \; \forall \; y \in N(x, \sigma)$
end

Figure 2.4. Pseudo-algorithm for the Local Search algorithm. The algorithm continuously searches the neighbourhood of the current solution for a solution of better quality to replace it. When no such solution can be found the algorithm terminates.

Since the quality of the final solution reached by the local search algorithm strongly depends on the initial solution (and the neighbourhood structure) the algorithm has the potential to perform poorly for some starting positions and well for others. One method of partially solving the problem of local optimality is to restart the algorithm many times with different initial solution, a method which will asymptotically result in global optimisation if all solutions are used as an initial solution. Practical time considerations do not allow for all solutions to be used as starting points, instead the algorithm should terminate after some 'reasonable' number of restarts. Unfortunately the number of restarts required cannot be determined with any guarantee of final solution quality. Nevertheless the algorithm is usefully in that it is extremely simple to use and generally applicable. It will be informative to compare results gained from other heuristic methods with results achieved by repeated local search in the same amount of computation time.

2.7 Tabu Search

Tabu Search is an heuristic algorithm that exploits 'memory' or knowledge of the system under investigation to find a 'good' solution to a problem in a combinatorial situation. The modern Tabu Search paradigm derives from work by Glover [16] with seminal ideas and contributions from various sources. The technique is still comparatively new and alternative methodologies and applications are being investigated, making it somewhat unclear how exactly to implement a Tabu Search algorithm. According to Glover [17] Tabu Search is defined as

> "...a higher level heuristic procedure for solving optimisation problems, designed to guide other methods (or their component

processes) to escape the trap of local optimality."

Tabu Search is also an heuristic search procedure in its own right, with a growing number of theoretical and practical articles investigating the method. A good introduction to the topic is presented in Glover and Laguna [15], covering both the basic algorithm and implementation considerations about tabu memory.

The Tabu Search method seeks to optimise a function under certain constraints. In the case of minimisation this amounts to

$$\min_{s.t.\, x \in X} \; c(x)$$

where $c(x)$ is the objective or cost function and the family of solutions X may include restraints on x.

In the quest for optimality the neighbourhood $N(x)$ of each solution $x \in X$ is searched. Another solution x′ in this neighbourhood can be reached by a *move*[17] (this relationship is usually symmetric).

Tabu Search uses randomisation selectively (or not at all), instead exploiting flexible memory to control the search process. Every solution has a modified neighbourhood so that history determines which solutions may be reached by a move from the current solution. For example x′ $\in N(x)$ may be classified *tabu* (disallowed), when selecting a potential neighbour of x, due to memory considerations. Instead choose $x'' \in N(H,x)$ as an acceptable candidate to move to from x. $N(H,x)$ contains all neighbourhood candidates that the memory (H) will allow the algorithm to consider. Generally, in the short-term, $N(H,x) \subseteq N(x)$, but in the mid-to-long-term $N(H,x) \nsubseteq N(x)$ necessarily. Instead the neighbourhood may contain a set of elite solutions previously encountered, either in a regional cluster or in different clusters, depending on what the mid-to-long-term considerations are. The type of solutions in the modified neighbourhood change over time as the search progresses from a diverse search, looking at many areas of the search space, to a more intense search, concentrating on particular 'good' regions.

Tabu Search also uses memory to modify the cost function, $c(x)$, in a similar manner to the modification of the neighbourhood structure. So use $c(H,x)$ instead of $c(x)$ to measure the goodness of a solution, where history, H, may encourage or penalise solutions in the determination of the 'best' x′ $\in N(H,x)$.

When $N(H,x)$ is large (or costly to examine) create a subset of candidates representative of $N(H,x)$, called $NC(H,x)$, which is examined for a new solution. The basic pseudo-algorithm for Tabu Search is now described, based on an algorithm presented in Glover and Laguna [15], shown in figure 2.5.

The method depends largely on the definition of memory (or history),H, the candidate subset for neighbourhood selection, $NC(H, \mathrm{x}) \subseteq N(H,x)$, and the (modified) cost function $c(H,x)$.

[17] The concept of a move is essentially analogous to selecting an element from $N(x,\sigma)$ discussed in section 2.2.

```
begin
    select x ∈ X
    x* = x
    c* = c(x*)
    H = ∅
    calculate NC(H, x)
    choose x' such that c(H, x') = min{c(H, x) | x ∈ NC(H, x)}
    while cutoff rule is not satisfied
        x := x'
        if c(x) < c* then
            x* := x
            c* := c(x*)
        end if
        update H
        update NC(H, x)
        choose x' such that c(H, x') = min{c(H, x) | x ∈ NC(H, x)}
    end while
end
```

Figure 2.5. Pseudo-algorithm for the Tabu Search method. Tabu Search performs a repeated neighbourhood search, selecting the 'best' solution in each neighbourhood to be the new solution. The neighbourhood solution and determination of solution quality are influenced by the algorithm's memory which either encourages or discourages the algorithm from taking certain moves.

2.7.1 Tabu Search Memory

The way in which memory is implemented and the influence this has on neighbourhood definition and solution evaluation has a great effect on algorithm performance. How a move is identified and whether we wish to encourage or discourage particular moves are all of paramount importance in the search. Generally speaking, the role of memory changes as the algorithm proceeds, with both short-term and mid-to-long term memory considerations. At the beginning of a run, as in most general heuristic algorithms, a 'broad look' at the search space is desired to identify good and bad regions, a concept in Tabu Search known as *diversification*. As the algorithm progresses and a general feeling exists for where 'good' solutions are to be found, the search is refined to produce a 'near-optimal' solution in a 'good' area, an idea in Tabu Search termed *intensification*[18]. It is this byplay between the contradictory ideas of intensification and diversification that is controlled by the algorithm memory, which changes the emphasis on the two concepts as the algorithm proceeds. A widely used and quite natural Tabu Search memory concept is *attribute* based memory, which identifies an *aspect* of a solution that changes as the result of a neighbourhood move.

Attribute Based Memory

An attribute of a move from x to x' is any aspect that changes as a result of this move. For example the cost of a solution, $c(x)$, may alter or if the solution is represented as a sequence of components, then x_i and x_j may swap values. In a particular problem more than one type of move attribute can operate. Some attributes will be quite general (such as an increasing/decreasing cost function) while others will be more specific (swapping two components as mentioned above). In fact there can be many attributes for a single move and can be naturally categorised as *from* or *to* attributes so that a move attribute is an ordered pair (from-attribute,to-attribute). From-attributes are associated with x whereas to-attributes are associated with x'.

It is often via move attributes that *tabu* restrictions are imposed in Tabu Search, preventing the search from reversing or repeating itself. Since a move is effectively represented by its attributes we record the *opposite* attributes to a move just completed and classify them tabu. Any move containing the tabu attributes is not available for selection as a neighbourhood candidate (unless certain *aspiration* criteria are first satisfied).

A tabu restriction is activated if attributes have occurred either *recently* for short-term considerations or *frequently* for mid to long-term considerations thus classifying these attributes as *tabu-active*. The most common tabu restrictions are designed to prevent cycling and to introduce vigour into the search (however avoidance of cycles is not in itself a goal but is often useful in search techniques).

Consider a hypothetical example below in which a solution to a problem

[18] In Genetic Algorithms the ideas of intensification and diversification are mirrored by the concepts exploitation and exploration.

consists of random permutations of 10 numbers such as a solution to a 10 city TSP. If the previous move consisted of swapping the cities 9 and 5 then the values involved in this swapping procedure could be used as a potential move attribute (other attributes could involve the costs of the respective solutions or the values of the city indices swapped). The move attribute of swapping 5 and 9 is now classified tabu-active (in the short-term).

3 6 1 9 4 8 7 10 2 5 6

This means that in the near future these two cities are not allowed to be swapped since the associated move attribute is on the tabu list[19]. However, the tabu status of a move attribute can be overridden in certain circumstances. If the aspiration criteria are met then the move can proceed as if it were not tabu.

On certain occasions in Tabu Search, active tabu restrictions can be overridden. Conditions that must be met before this can occur are called the aspiration criteria. For example a standard implementation is to override a tabu restriction if the potential move yields a solution better than the best solution found so far. Other aspiration criteria can also be considered. Aspiration by default is needed when all neighbourhood candidates are classified tabu, then the algorithm is forced to select the move that violates the tabu restrictions the least. Aspiration by region overrides the tabu restrictions when the potential move yields a better solution than previously found in a given region.

Two complementary ideas in Tabu Search memory concerned with short-term and mid-to-long-term memory are recency based memory and frequency based memory respectively.

Recency: When a tabu restriction is activated, in the short-term it is necessary to ensure that the algorithm does not return to the a solution recently visited. However at a later stage in the search it may be appropriate to return to such a solution so the appropriate move attributes are classified tabu for a short time period t. The time period can be determined statically or dynamically. Statically we could choose $t = 7$ or $t = \sqrt{n}$ where n is a measure of population dimension. Dynamically, t could vary between two bounds, either randomly or systematically. Alternatively t could be made dependent on the quality of the attribute. Often different types of move attributes have different values of t to avoid making a tabu restriction too severe (especially when applied to a very general attribute).

Frequency: To complement recency based memory, it is not desirable to continuously select the same moves too often in the long run. Frequency based memory enables the algorithm to broaden the selection

[19] The algorithm memory can be implemented in a number of ways, the simplest in a problem type such as this is as an $n \times n$ matrix, where n is the number of cities, and each matrix element indicates how long (numbers of iterations) a particular swap is classified tabu.

process to consider regions the search has yet to investigate. In the later stages of the search it can also give an indication of which attributes lead to good quality solutions, allowing the algorithm to 'fine tune' the search process. Generally, for a move attribute, record as a measure of frequency a ratio of the number of occurrences of the move to some measure of the total number of moves made to date. If S is defined to be a subset of solutions produced so far for which frequency statistics are recorded, then depending on what type of moves are included in S, various conclusions can be reached. For example if S is a subset containing high quality solutions then a high frequency count may imply that an attribute is attractive to producing solutions of good quality. Conversely if S contains solutions of both high and low quality then a high frequency count may indicate an entrenched attribute, restricting the search space.

Whereas recency based memory is invoked by means of a tabu list, frequency based memory is typically implemented by either penalising or giving incentives to certain move attributes. Returning to the previous example, look at what frequency based memory can mean. Consider a situation in which the move to swap 4 and 10 has been identified a potential candidate for selection.

$$3 \quad 6 \quad 1 \quad 9 \qquad 4 \qquad 8 \quad 7 \qquad 10 \qquad 2 \quad 5 \quad 6$$

The algorithm frequency memory may indicate that 4 and 10 have often been swapped (in some time interval) and if this move has been made more often than other potential moves then it is penalised to make it less appealing in the selection process. Conversely, if the move has been made relatively infrequently, then it becomes more attractive for selection. This holds true as long as diversification is most important, trying to push the search into unexplored areas, if intensification is being emphasised, then frequency based memory may indicate that swapping 4 and 10 is a common property leading to other good solutions.

It becomes apparent the the exact role of memory in Tabu Search depends on the stage the algorithm is at and whether diversification or intensification is more important. Intensification is designed to create better solutions by encouraging the inclusion of 'good' attributes. Short-term this means incorporating attributes with the highest evaluations. Long-term this means incorporating attributes of solutions from elite subsets forcing the search into 'good' subregions. To achieve this via frequency based memory, S can be considered to be a small subset of elite solutions (high quality local optima). These solutions share common attributes and are able to reach each other in a small number of moves (not to be confused with a concept of distance in the search space).

Opposed to the goal of intensification, diversification is needed to generate solutions that contain attributes different from those already encountered in the search so far. Using frequency based memory to achieve this goal, S is defined to be some significant subset of all solutions thus-far encountered.

For example S can be selected to be a subsequence of all local optima that do not necessarily share common properties guiding the algorithm to different 'good' areas in the search space. A penalty/incentive function can then be created which, when used in combination with $c(H, x)$, can penalise or encourage a particular search strategy depending on whether at this stage of the search intensification or diversification is more desirable.

2.8 Simulated Annealing

Simulated Annealing is an heuristic method for finding 'good' solutions to difficult optimisation problems. The technique was first applied to combinatorial optimisation problems by Kirkpatrick *et al.* [26] and independently by Černy [5]. They showed that ideas used in simulating the physical process of *annealing* as applied to solids, first proposed by Metropolis *et al.* [30], could be extended to solve optimisation problems in general, especially combinatorial optimisation problems.

2.8.1 Physical Analogy and the Metropolis Algorithm

Annealing is a thermal process used in *condensed matter physics* to obtain solids with low energy states. The process has two main steps, melting the solid and then **carefully** reducing the temperature so that the particles of the solid arrange themselves into the *ground state*. When the solid is melted to a liquid, the particles are arranged randomly, whereas in the ground state of the solid particles are arranged in an organised lattice that achieves a minimum energy configuration. If cooling is not done carefully, the ground state will not be reached, instead *quenching* occurs, whereby the solid takes on a *meta-stable state* with less than minimum energy. In either case we have reached a steady *frozen* state.

Metropolis *et al.* [30] modeled this physical process with the introduction of an algorithm based on *Monte Carlo*[20] techniques. The algorithm generates a sequence of simulated states of the solid by successively perturbing states in the following manner:

- A particle of the present state is given a small displacement to produce a new state and the resulting change in the energy of the system, δ, is calculated.

- If $\delta < 0$ then the new state is accepted as the current state, otherwise the state is accepted according to the *Metropolis criterion (test)*. The change is accepted with a certain probability given by

  $$\exp\left(\frac{-\delta}{\kappa_B T}\right),$$

  where T is the *temperature* and κ_B a physical constant known as the *Boltzmann constant*.

[20]Monte Carlo techniques involve the selective application of randomisation.

- If the temperature is slowly and carefully lowered (achieved in the algorithm by carrying out many iterations at each stage) then the solid will reach thermal equilibrium at each temperature. Thermal equilibrium is categorised by a Boltzmann probability distribution where the probability of being in state i with E_i, the energy of state i, at temperature T is
$$\frac{1}{Z(T)} \exp\left(\frac{-E_i}{\kappa_B T}\right),$$
and $Z(T)$ is a normalising partition function defined as,
$$Z(T) = \sum_i \exp\left(\frac{-E_i}{\kappa_B T}\right).$$

2.8.2 The Simulated Annealing Algorithm

The application of the Metropolis algorithm to optimisation problems, originally suggested by Kirkpatrick *et al.* [26] and Černy [5], is possible after making an analogy between the problem and the Metropolis algorithm determined by the following equivalences.

- Solutions in the optimisation problem are equivalent to states of the physical system.
- The cost of a solution is equivalent to the energy of a state.
- A neighbouring solution is equivalent to a change of state.
- The *control parameter* is equivalent to the temperature.
- An heuristic solution to the problem is equivalent to a frozen state.

An optimisation problem can be solved using the Metropolis algorithm by repeatedly sampling neighbourhoods randomly, starting from an initial solution, and accepting as a new solution any configuration of superior quality and any inferior solution that passes the Metropolis test. The Metropolis test is dependent on the control parameter and the magnitude of the increase in such a way that small increases (assuming minimisation is the goal) are more likely to be acceptable than large increases. Also, when the temperature is high, the algorithm is likely to accept almost all configurations regardless of quality, but when the temperature is low, only solutions of a better quality are likely to pass the test. By accepting some inferior solutions the annealing algorithm seeks to avoid becoming trapped in a local optimum that is far from globally optimal (i.e. quenched). For a combinatorial optimisation problem, assuming minimisation of the objective function is the goal, a formulation is:

$$\min_{s.t.\, x \in S} f(x),$$

where S is the solution space, x a solution, $f(x)$ the objective function, and $N(x)$ the neighbourhood structure. If the number of iterations spent at each

```
begin
    select an initial solution i ∈ S
    select an initial temperature T
    while stopping criterion not true
        count:=0
        while count< L
            choose a neighbour j ∈ N(i)
            δ = f(j) − f(i)
            if δ < 0 then
              i := j
            elsif random[0,1) < exp(−δ/T) then
              i := j
            end if
            count:=count+1
        end while
        reduce temperature T
    end while
end
```

Figure 2.6. Pseudo-algorithm for the Simulated Annealing method. The algorithm repeatedly searches the neighbourhood of the current solution for a configuration to become the new current solution. Better quality configurations are accepted automatically, solutions of a lower quality are accepted probabilistically, dependent on T and the magnitude of the difference in solution quality.

temperature is identified as L and the temperature as T then a generalised annealing algorithm for such a problem is shown figure 2.6.

Since T is now a control parameter, Boltzmann's constant has no analogy in the optimisation problem and κ_B has been dropped from the Metropolis test. However it is still usual to refer to T as the temperature and the rate and manner by which T is reduced as the *cooling (or annealing) schedule*. The speed of convergence of the algorithm will depend largely on the choice of L and T for each iteration. The Simulated Annealing algorithm in figure 2.6 is both simple and generally applicable since it is essentially a *local search* algorithm with the additional feature of the control parameter regulating some amount of uphill moves. The success of the algorithm has been accounted for by applying a Markov Chain[21] model to the technique and demonstrating the asymptotic convergence properties of Simulated Annealing. In purely theoretical terms it has been shown that the algorithm converges to a set of globally optimal solutions as time tends to infinity. However when implementing an annealing algorithm practical considerations must be given to the annealing schedule to ensure convergence to a 'good' quality solution occurs in finite time.

2.8.3 Implementing the Algorithm

To implement the algorithm for a particular problem it is necessary to first address two issues, *generic* and *problem specific* considerations. Problem specific considerations include the choice of a solution space, the form of the cost function, and an appropriate neighbourhood scheme. Generic considerations involve the choice of the initial temperature, the manner in which temperature is reduced, and the stopping condition. Both factors can greatly affect the speed of convergence.

Problem specific choices: The solution space for any given problem must be clearly defined. In the case of the TSP the solution space is all solutions resulting from different orderings of a sequence of cities in a tour. Once the solution space is determined unambiguously, a neighbourhood structure and a method of evaluating solution quality must be imposed. The correct choice of neighbourhood structure is important if the annealing process is to work effectively. The neighbourhood should not be too large or too small. A neighbourhood that is too small will potentially give rise to slow or no convergence and a neighbourhood that is too large could lead to premature convergence. Ideally the neighbourhood should also be smooth in the sense that the quality of configurations in the neighbourhood of a solution should not be vastly different from either the quality of the solution itself or from the qualities of other solutions in the neighbourhood.

[21] Any system that consists of random transitions from one state to another over an index (usually time) together with the Markov property is a *Markov Chain*. The Markov property essentially says that the choice of states available at the next stage of the system is unaffected by past history. Only the current state affects the selection of the next state.

The form of the evaluation function is quite clear for many problems, taking the value of the function that is being optimised. In the TSP case this is simply the sum of the intercity distances in a (cyclic) tour. With many combinatorial optimisation problems however, the evaluation function is not as straightforward, especially when searching for *any* feasible solution to some problem. In these cases, a penalty function is generally applied, which can distinguish between two infeasible solutions to determine which one is closer to feasibility. Thus the solution space for the annealing algorithm does not necessarily consist entirely of feasible solutions, although the algorithm will tend to discourage the selection of infeasible solutions as the temperature decreases.

With a solution space , a neighbourhood scheme, and an evaluation function defined, the algorithm can now determine which other solutions can be reached in a single move, from a particular solution, and whether they are better or worse than the current position. Also, to begin the algorithm, a method of constructing an initial solution is required. This is usually achieved in either a randomised or a systematic manner.

Generic choices: The set of implementation decisions that remain unchanged for all problem types is collectively known as the cooling or annealing schedule. The annealing schedule consists of:

- An initial temperature.
- The number of iterations spent at each temperature, L.
- A function to reduce the temperature.
- A decision rule to halt the search.

Finding an initial temperature is analogous to heating a solid in the true annealing process to a temperature when all particles are free to move about in the solid, meaning the system is equally likely to be in any state. This can be achieved in the algorithm by selecting some arbitrary initial temperature and performing a preliminary 'heatup' process. This typically involves repeatedly increasing the temperature until a certain high percentage of moves are being accepted.

Once the initial temperature has been determined and the algorithm proper has begun, the time spent at each value of the temperature needs to be decided upon. Returning to the true annealing process, notice that temperature reduction must be performed *carefully* in order to reestablish the thermal equilibrium of the system, categorised by the Boltzmann distribution. Every time the temperature is reduced in the algorithm, L should be the number of iterations needed before the distribution of states returns to equilibrium. This is quite a theoretical consideration and L is often determined arbitrarily, e.g. fixed at some constant value. Alternatively some degree of feedback can be used in the algorithm to determine when equilibrium has (approximately) been reached.

As noted above, temperature reduction needs to be performed carefully. If the temperature is reduced too quickly then the system can quickly become trapped in a local optimum of poor quality, analogous to quenching in the physical system. If the temperature is reduced too slowly then time until the algorithm converges to a local optimum can become prohibitively long. There is a trade off between the size of the incremental changes in the temperature and the number of iterations required to reestablish thermal equilibrium. The larger the change, the longer it will take to reestablish equilibrium at each temperature. If small changes are used then equilibrium is found relatively quickly, larger changes require more time to regain the Boltzmann distribution. Again the exact manner in which temperature is reduced can be determined arbitrarily, such as a geometric decrease $T = \alpha T$ where $\alpha < 1$ (although usually quite close to 1), or through a feedback mechanism. In general it is advisable not to spend too long at high temperatures, since then the algorithm is effectively performing a random search. Too much time should also not be spent at low temperatures, since the algorithm is then simply performing a local search.

The algorithm should terminate when there is little chance of escaping from the current solution to a better quality solution, and if temperature reduction has been done carefully, the quality of this local optimum will be close to that of the global optimum. Many methods of determining when to stop can be used. For example, if the current solution has not changed for a certain number of temperature reductions then the algorithm terminates, an arbitrary method. Alternatively, information from the algorithm can be used to measure the rate of change in solution quality (as a function of temperature) at successive temperatures, and when this rate has fallen below a certain threshold then the algorithm terminates.

2.9 Threshold Accepting

Threshold accepting (TA) is a generally applicable heuristic algorithm developed by Dueck and Scheuer [10], that can be used to solve combinatorial optimisation problems. An iterative improvement method, TA is formally very similar to simulated annealing (SA), discussed at length in section 2.8. Given an arbitrary solution to the problem both methods then involve the selection of a neighbouring solution. This neighbourhood configuration replaces the existing configuration if a certain criterion is achieved and it is at this level that the essential difference between the two algorithms becomes apparent.

In SA a new solution is accepted automatically if it is better (according to the objective function) than the current configuration, if it is worse then acceptance is governed by a probability. The probability is dependent on a control parameter called *temperature* that is slowly decreased (cooled) as the algorithm proceeds. Cooling has the effect that the probability of a worse solution being accepted is reduced as time increases.

TA accepts a new solution if it is better than or 'not much worse' than the the existing configuration, formalised in the algorithm by a *threshold* parameter. In the case of minimisation, any new solution less than the current configuration quality plus the threshold is accepted as the new current solution. No stochastic test is required in the neighbourhood selection, acceptance is deterministic. The threshold is decreased with time so that fewer worse solutions are accepted as the algorithm proceeds. The TA algorithms are discussed extensively in chapter 4.

2.9.1 The Great Deluge Algorithm and the Record-to-Record Travel Algorithm

The TA principle can be simplified further resulting in two new generally applicable heuristics introduced by Dueck [11], the Great Deluge algorithm (GDA) and the Record-to-Record Travel algorithm (RTRT). These two algorithms differ from TA in the decision rule by which a neighbourhood candidate is accepted to replace the current solution.

It is easy to visualise the GDA, especially for maximisation. The quality of all solutions in the sample space together define a surface or a *landscape* of which the maximum or highest point is sought. By letting it 'rain' continuously on this landscape. The algorithm 'walks' around the fictitious landscape ensuring the solution never falls below this rising water level. At some point the algorithm will become trapped on a 'peak' surrounded by water and the hope is that the algorithm has then found a peak that is nearly optimal. This analogy is formalised by the pseudo-algorithm given in chapter 4.

Record-to-Record Travel is very similar to the GDA, the difference again lies in the acceptance rule. RTRT accepts any new solution that has a quality 'not much worse' than the best solution (or *record*) found so far. The pseudo-algorithm given in chapter 4 demonstrates this method as implemented for minimisation.

The problem specific considerations of defining a solution space, a neighbourhood structure, an evaluation function, and an initial solution for the Threshold algorithms are the same as for SA, discussed in section 2.8.3. Generic concerns of the algorithm involve selection of an initial threshold, a method to reduce the threshold parameter, and a termination criterion, although Dueck and Scheuer [10] make the claim that the vital element here is the threshold lowering scheme. Further discussion of parameter selection for both algorithms is given in section 4.3.

2.10 Genetic Algorithms

Genetic Algorithms (GAs) are adaptive heuristic search algorithms based on the evolutionary ideas of natural selection and genetics. They represent an intelligent exploitation of a random search used to solve optimisation problems. GAs exploit historical information to direct the search into regions of better performance within the search space. The basic techniques of GAs are designed to simulate processes in natural systems necessary for evolution, specifically those that follow the principles of "survival of the fittest". The GA paradigm is investigated empirically, together with a discussion of theoretical and practical issues, in chapter 3.

2.10.1 Genetic Algorithm Overview

GAs are based on an analogy with the genetic structure and behaviour of chromosomes within a population of individuals using the following foundations:

- Individuals in a population compete for resources and mates.
- Those individuals most successful in each 'competition' will produce more offspring than those individuals that perform poorly.
- Genes from 'good' individuals propagate throughout the population so that two good parents will sometimes produce offspring that are better than either parent.
- Thus each successive generation will become more suited to their environment.

In a GA a population of *individuals* is maintained, each representing a possible solution to a given problem. A *fitness* score is assigned to each solution representing the abilities of an individual to 'compete'. The individual with the optimal (or generally near optimal) fitness score is sought. The GA aims to use selective 'breeding' of the solutions to produce 'offspring' better than the parents by combining information from the chromosomes.

Parents are selected to mate, on the basis of their fitness, producing offspring via a *reproductive plan*. Consequently highly fit solutions are given more opportunities to reproduce, so that their offspring inherit characteristics from each parent. As parents mate and produce offspring, room must be made for the new arrivals since the population is kept at a static size. Individuals in the population die and are replaced by the new solutions, eventually creating a new *generation* once all mating opportunities in the old population have been exhausted. In this way it is hoped that over successive generations better solutions will thrive while the least fit solutions die out.

New generations are produced of solutions containing, on average, more good characteristics than a typical solution in a previous generation. Eventually, once the population has *converged* and is not producing offspring noticeably different from those in previous generations, the algorithm itself is said to have converged to a near optimal solution to the problem at hand.

Chapter 3

Genetic Algorithms

3.1 Introduction

Genetic Algorithms (GAs) are adaptive heuristic search algorithms based on the evolutionary ideas of natural selection and genetics. As such they represent an intelligent exploitation of a random search used to solve optimisation problems. Although randomised, GAs are by no means random, instead they exploit historical information to direct the search into regions of better performance within the search space. The basic techniques of GAs are designed to simulate processes in natural systems necessary for evolution, specifically those that follow the principles first laid down by Charles Darwin of "survival of the fittest".

The initial developments in the GA field occurred in the 1960s and 1970s with the first rigourous description of the Genetic Algorithm process described by Holland [24]. In the early years of their existence GAs where primarily used in the Artificial Intelligence arena, more specifically for game playing and pattern recognition. Only recently has the GA approach been applied to problems in combinatorial optimisation.

What follows is a summary (by no means complete) of issues involved in the GA field. The amount of research into the GA paradigm, part of a larger field of study known as *evolutionary computing*, is extensive and brings together ideas from many varied academic disciplines, biology and genetics, computer science, mathematics, statistics and operations research to name a few. Unfortunately much of the work done on GAs is empirical in nature and many of the latest developments lack theoretical grounding. Good introductions to the topic, of various sizes, include books such as Goldberg [19], chapters in books such as Reeves [35], and introductory papers such as Beasley *et al.* [3].

3.2 Genetic Algorithm Overview

GAs are based on an analogy with the genetic structure and behaviour of chromosomes within a population of individuals using the following foundations:

- Individuals in a population compete for resources and mates.
- Those individuals most successful in each 'competition' will produce more offspring than those individuals that perform poorly.
- Genes from 'good' individuals propagate throughout the population so that two good parents will sometimes produce offspring that are better than either parent.
- Thus each successive generation will become more suited to their environment.

In a GA a population of *individuals* are maintained, each representing a possible solution to a given problem. Each individual is coded as a finite length vector of components, or variables, in terms of some alphabet, usually the binary alphabet $\{0, 1\}$. To continue the genetic analogy these individuals are likened to *chromosomes* and the variables are analogous to *genes*. Thus a chromosome (solution) is composed of several genes (variables). When a gene takes on a specific value it is known as an *allele*, and the physical position of a gene in the chromosome is known as its *locus*. The *genotype* is the coded string that is processed by the algorithm whereas the *phenotype* is the actual interpretation of this representation in terms of the particular problem being dealt with. A *fitness* score is assigned to each solution representing the abilities of an individual to 'compete'. The individual with the optimal (or generally near optimal) fitness score is sought. The GA aims to use selective 'breeding' of the solutions to produce 'offspring' better than the parents by combining information from the chromosomes.

The GA maintains a population of n chromosomes (solutions) with associated fitness values. Parents are selected to mate, on the basis of their fitness, producing offspring via a *reproductive plan*. Consequently highly fit solutions are given more opportunities to reproduce, so that offspring inherit characteristics from each parent. As parents mate and produce offspring, room must be made for the new arrivals since the population is kept at a static size. Individuals in the population die and are replaced by the new solutions, eventually creating a new *generation* once all mating opportunities in the old population have been exhausted. In this way it is hoped that over successive generations better solutions will thrive while the least fit solutions die out.

New generations of solutions are produced containing, on average, more good genes than a typical solution in a previous generation. Each successive generation will contain more good 'partial solutions' (*schemata*) than previous generations. Eventually, once the population has *converged* and is not producing offspring noticeably different from those in previous generations, the algorithm itself is said to have converged to a set of solutions to the problem at hand.

From this brief description of a Genetic Algorithm, Goldberg [19] recognises that GAs differ from more traditional search/optimisation methods in four main ways:

- GAs work with a problem coding, not the actual variables, making them largely problem independent. A solution is represented as a finite length string over some alphabet, usually of cardinality 2, the binary alphabet.
- GAs search from a population of solutions, not a single point. This 'safety in numbers' approach means the algorithm is less likely to become trapped in a local optimum that is far from optimal.
- GAs are guided in their search only by fitness scores, not derivatives or other information. Many methods make assumptions about the existence of such auxiliary information which can often be misleading. GAs make no such assumptions, so are applicable to a wide variety of problems, unfortunately at the expense of algorithm run time.
- GAs use stochastic rules rather than deterministic choices when searching for the optimum. Random choice is used as a tool to guide the algorithm towards areas in the search space of likely improvement.

These properties make GAs a robust search technique that will produce good quality solutions to a problem with reasonable speed for many different problem classes, from combinatorial optimisation to Artificial Intelligence. A pseudo Genetic Algorithm algorithm due largely to Beasley *et al.* [3] is shown in figure 3.1.

```
begin
    generate the initial population (of n individuals)
    compute the fitness of each individual
    current generation:=initial population
    while generation not converged
        repeat
            begin new generation
            select two parents from the current generation to mate,
               biased towards the fitter individuals
            apply crossover to parents to produce offspring
            apply mutation to the offspring
            compute the fitness of the two offspring
            place offspring in the new generation
        until new generation is full
        current generation:=new generation
    end while
end
```

Figure 3.1. Pseudo-Algorithm for the Genetic Algorithm Method. Successive generations of solutions to the problem are produced by mating individuals selectively to give offspring with characteristics from both parents. Over time the generations evolve towards optimality.

3.3 Implementation Details

Beasley *et al.* [3] identifies several considerations important to the Genetic Algorithm technique

- A suitable *coding* (representation) for the problem.
- A *fitness function* to compare the solutions which are being optimised (i.e. the objective function).
- A *reproductive plan* for redistributing genetic material contained in the population. This is composed of three parts:
  - *Reproduction*: a selection process for choosing parents for mating purposes.
  - *Crossover*: recombines genetic information from two parents to produce offspring with characteristics from both.
  - *Mutation*: makes random changes to the offspring to ensure variety is maintained in the population.
- A method to determine population *convergence*.

There are additional details to consider when implementing a GA, however the driving force of the GA is largely due to those mentioned above. Depending on the particular problem under consideration, specifics of the GA process will vary. However generally the coding and the fitness function relate to the problem under consideration while the reproductive plan and convergence processes are implemented generically.

3.3.1 Coding

Any solution to a problem is fully represented as a finite collection of variables, where the variables together make a vector of values. The solution is coded in an alphabet (usually binary) generally making the reproductive plan, which operates on this coding, problem independent. For example a solution to a problem may be coded in the binary alphabet $\{0, 1\}$ by the following vector of length 10.

$$0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1$$

The precise coding used to describe a problem is not considered vital since the robust nature of the algorithm means GAs are forgiving to the vagaries of coding alternatives.

Two basic principles, due to Goldberg [19], guide the choice of a suitable problem coding, the *principle of meaningful building blocks* and the *principle of minimal alphabets*. The principle of meaningful building blocks states that the programmer should select a coding so that short, low-order *shemata* are relevant to the underlying problem and relatively unrelated to schemata over fixed positions. Section 3.4 contains details of schemata and how they pertain to the GA process. However this guiding rule is seldom practical making good coding choices hard to construct. The principle of minimal alphabets states that the programmer should choose the smallest alphabet that permits a natural expression of the problem. As shall be seen in section 3.4 GAs operate by implicitly recognising similarities (schemata) in the best solutions and giving these similarities a greater chance to survive to the next generation by means of a reproductive process biased towards fitter individuals. By using as low a cardinality alphabet as possible, the number of schemata per bit of information in any problem coding[1] is maximised. Thus the implicit information analysis that drives the GA process is maximised, leading to a faster, more efficient search.

3.3.2 Fitness Function

Given a particular chromosome x, a single fitness value $f(x)$ can be calculated to indicate the 'utility' of the solution or how well that individual solves the particular problem at hand. Generally, the fitness function is the value of the objective function of the problem that needs optimising, but this is not always the case. In some problems, especially combinatorial optimisation problems, the choice of fitness function is not as clear, a number of different objectives may need to be met simultaneously (usually resulting in some form of multi-objective utility function), or a feasible solution among many infeasible ones may simply be required (in which case a penalty function is often employed).

3.3.3 Reproduction

During reproduction individuals are chosen to be *parents* and are subsequently recombined to produce offspring for the next generation. Parents are selected using a scheme that favours the fitter individuals, giving good solutions a higher probability of selection than those solutions that do not perform as well. Better solutions are more likely to mate several times and poor solutions are more likely to mate seldom if at all. Thus a *mating pool* is formed. Only those solutions that are in the mating pool will have their genetic information recombined via the techniques of crossover and mutation to produce offspring for the next generation.

The manner in which individuals are selected for the mating pool is determined by the *reproductive operator*. There are many ways this operator may be implemented, but it must be done carefully if the the algorithm is to converge correctly. Theoretically the actual number of times an individual x, in a population G, appears in the mating pool should be equal to the expected number as determined by the reproductive operator. However as the expected value is seldom an integer amount, bias is often introduced by the way in which an integer number of berths in the mating pool is allocated

[1] Goldberg [19] provides an explanation of this claim

to each solution. Also, selection on the basis of fitness alone, where the expected frequency of a solution x in the mating pool is

$$n\frac{f(x)}{\sum_{x \in G} f(x)}$$

with a population size n and a fitness function $f(x)$ applied to a solution x, generates problems, that of *premature converge* and its counterpart *slow convergence*.

At the start of the process, when one or two solutions are very good in comparison to others in the population, these 'super-fit' individuals will take over the population, leading to premature convergence to a poor local optimum. At the end of a GA run, when the population has converged to a set of similar solutions, selection on the basis of fitness alone cannot distinguish sufficiently between the solutions, resulting in slow convergence. A good summary of various reproductive techniques used to minimise the dual problem of premature and/or slow convergence and unbiased integer allocation are given in Beasley *et al.* [3]. Three main methods used to solve the need to overcome slow and/or premature convergence are *fitness scaling*, *fitness ranking*, and *tournament selection*.

Fitness scaling: The existing fitness function f is transformed into a more robust function g, such as the linear transform shown below:

$$g(x) = af(x) + b$$

with the constants a and b determined by

$$\overline{g} = \overline{f}$$

$$f_{max} = \mu\overline{f}$$

An individual is then chosen for inclusion in the mating pool with a probability $\frac{g(x)}{\sum_{x \in G} g(x)}$. This particular transformation ensures that the fittest individual in a population will be selected on average μ times[2].

Fitness ranking: Individuals are sorted in order of fitness, and a probability distribution is associated with this sorted list. Individuals are then selected for mating in accordance with this distribution. Several papers such as Whitley [41] have claimed ranking to be superior to scaling since adequate selective pressure is applied in proportion to relative solution fitness regardless of the actual fitness score, which may be a somewhat arbitrary measure in the first place.

Tournament selection: Groups of solutions of size T are selected randomly from the population and the fittest individual in this tournament is placed into the mating pool. All individuals are returned to the population and another tournament is performed. This process continues until the mating pool is full. A binary tournament, with $T = 2$ is similar to the ranking procedure without the same overheads.

[2]Certain transforms can result in negative fitness functions, a problem easily overcome. See Goldberg [19].

Fitness ranking and fitness scaling can both produce non-integer numbers of mating opportunities for a solution, so a technique is required to convert these real numbers into integers without introducing bias. Two widely used methods[3] are *stochastic remainder sampling without replacement* and *stochastic universal sampling*. Tournament selection produces integer numbers of mating opportunities by construction so no bias is introduced and the need for such methods is negated.

All three methods have advantages and disadvantages and any individual method can be made to produce similar results to another method. As long as the reproductive operator is implemented carefully any of the above three will give good results.

So far it has been assumed that any generation will be replaced en masse by the successive population created through reproduction, crossover, and mutation. In this manner parents from one generation are given no opportunity to compete with their offspring in the next generation. This evolutionary model is sound for some short-lived species, such as many insects, but in longer-lived species, such as mammals, parents and offspring are often alive simultaneously and can compete for resources. To allow for this possibility the notion of the *generation gap* was introduced to define the proportion of a population replaced every generation[4]. If taken to extremes only a very few (i.e. two) individuals are replaced in each 'generation' producing *steady-state* replacement. Beasley *et al.* [3] gives several references for papers investigating the generation gap and steady-state replacement. However the claim made in one paper, that performance achieved by a steady-state GA can be mirrored in a generational replacement GA by the selection of an appropriate reproductive scheme, seems to indicate that the generation gap is perhaps not a major consideration when designing a GA.

3.3.4 Crossover

Two individuals, chosen at random from the mating pool, reproduce, by means of a crossover operator, to generate a number of offspring. The standard example of a crossover technique is *single-point* or *simple* crossover. In this crossover scheme each parent has their chromosome cut randomly at a fixed position and the tail chromosome segments are swapped to produce two new offspring, see figure 3.2. Simple crossover is usually quite effective but in some problems more complicated operators have proved useful (see below). Crossover ensures that individuals from the mating pool recombine genetic information to produce new solutions to the problem. Some genes or parts of genes from each parent are inserted into each of the offspring. To remain true to the evolutionary analogy, an individual in any given generation should be able to compete with individuals from future generations. For this reason the crossover operator is only applied to a mating couple with a probability p_c, usually a fixed number between 0.6 and 1.0. If the operator is not applied the offspring are simply replications of the parents.

[3]See Beasley *et al.* [3] for references to theses techniques.

[4]By this definition, the simple generational replacement processes a generation gap of 1.

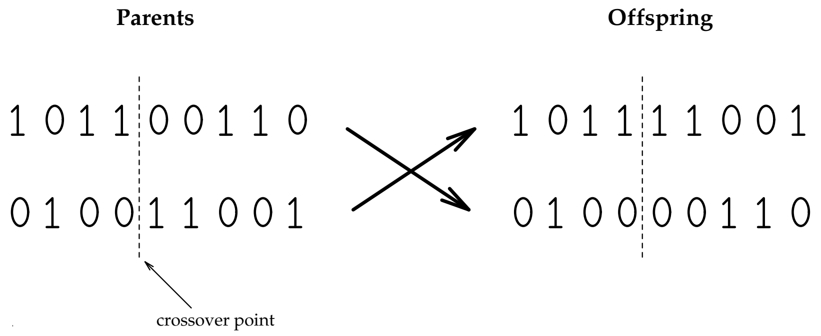

Figure 3.2. Single-point crossover. Simple crossover involves the exchange of information between two parent chromosomes to produce two offspring, each with characteristics from both parents. The parents are split randomly into two parts, a front section and a tail section. One offspring inherits the front section from the first parent and the tail section from the second parent, while the other inherits the front section from the second parent and the tail section from the first parent.

Other variations of the crossover operator exist. A discussion of alternative crossover techniques, with references, is given by Reeves [35]. A brief description of several of these is given below . Some are inappropriate for certain problem types while others are generally applicable.

- A direct extension of the single-point crossover is the *multi-point* crossover, a similar technique to simple crossover with the exception that a chromosome is partitioned into k segments rather than 2. Eshelman *et al.* [12] investigated the multi-point crossover and reached the conclusion that simple crossover, by choosing the crossover point in a random uniform manner, can limit the exchange of information between parents. Also if a basic tenant of GA theory, the building block hypothesis (see section 3.4), is not true, simple crossover may not result in the construction of better solutions. Multi-point crossover has the potential to overcome both these problems.

- *Generalised* crossover, in essence, is an extension of the multipoint crossover operator. If a bit string is generated stochastically using the Bernoulli distribution, then wherever a “0” occurs in the string copy the corresponding information from the first parent into one of the offspring and copy the corresponding information from the second parent into the other offspring. Where a “1” occurs in the bitstring this procedure is reversed. Also called *uniform* crossover by Syswedra [40], this operator has the property that schemata (see section 3.4) do not have to be short to be able to survive into the next generation. Any schema can be disrupted or copied into the offspring, a property which may or may not be an advantage.

- The *‘String-of-change’* crossover operator attempts to deal with a problem occurring late in the GA process, when many solutions are similar. At this stage, simple crossover may not produce offspring different from the parents. The 'string-of-change' operator effectively limits the crossover point to positions that will generate solutions not identical to the parents and was proposed independently by Booker [4] and Fairley [13].

- A variation on the uniform crossover theme is the 'threshold' crossover operator proposed by Sirag and Weisser [39]. When constructing an offspring it is necessary to decide, at each string position, which parent to copy the information from. If the i^{th} bit is selected from the first parent, then select the $(i+1)^{th}$ bit from the other parent with probability

$$\exp \frac{-\theta_c}{T}$$

where θ_c is the *threshold energy* and T is the *temperature*, similar in nature to the spirit of simulated annealing (see section 2.8). At high temperatures, the operator behaves like the generalised crossover, while at low temperatures it simply copies a single parent.

3.3.5 Mutation

Mutation is a reproductive operator, applied after crossover to each offspring, designed to maintain diversification in the search. It provides a small amount of random search in the GA process, ensuring that all points in the search space can be reached. *Simple mutation* is the usual operator applied for binary codings, whereby each child has a gene altered with some small probability p_m, generally of the order 10^{-3}. The alteration of a gene amounts to "flipping" the value at a given position in the gene when represented as a binary number (either a "0" flips to a "1" or a "1" flips to a "0"). Figure 3.3 shows a possible outcome of applying simple mutation to a binary string.

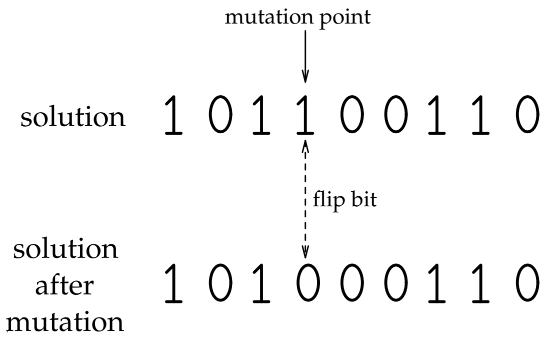

Figure 3.3. Simple mutation. For binary codings, single mutation randomly alters each bit by flipping it with a small probability p_m.

Other mutation operators exist. Many are used only for a particular problem type which defeats the generic aim of the reproductive plan, but is nonetheless necessary in some situations. Two operators that have been used in Traveling Salesman Problem applications (see section 3.5) are *scramble* mutation and *swap* mutation. Scramble mutation randomly rearranges the genes between two arbitrary loci on the chromosome, while swap mutation swaps the loci of two randomly selected genes.

It is generally assumed that the mutation operator plays a secondary role to crossover in the construction of new solutions for the next generation, although the precise role of mutation in the GA framework is not clearly understood other than that of its broad diversification role. It is a open question as to whether an optimal mutation rate exists. Suggestions have been made to use $p_m = \frac{1}{c}$ where c is the length of the chromosome or to simply use some small, length independent, probability such as $p_m = 0.001$.

3.3.6 Convergence

The GA process terminates when the population has converged. As noted by Beasley *et al.* [3], the population will evolve over each generation so that the fitness levels of the best and the average individual increase towards the global optimum, and hence the distance between the two measures decreases.

Convergence in a GA is the progression toward uniformity as individuals in the population become more and more similar. Few definitions of convergence exist in the literature, reflecting the fact that researchers are investigating how altering sections of the algorithm, such as the reproductive plan, affect the quality of solution. When investigating such changes, the algorithm is simply run for a fixed number of generations and the final qualities are compared, consideration is not given to whether the algorithm has stopped too soon or has continued processing for an unnecessary number of additional generations.

One definition of convergence, due to Beasley *et al.* [3], is given as follows: a gene is said to have converged when 95% of the population in a given generation share the same value. The population has converged when all genes have converged. In this way the average individual "converges" towards the best individual in successive populations.

Other (less computationally intensive) convergence measures can be used. Suggestions by Hobbs [23] below involve monitoring information that is already at hand in the algorithm (or can be easily computed), such as the best and average fitness values in a generation. Terminating the GA when the distance between the average fitness and best fitness falls below a certain threshold is a possibility. Similarly monitoring the variation of the solutions could be used as a convergence measure, again stopping once a threshold has been reached. Natural variation in the algorithm due to the reproductive plan (especially mutation) means that these stopping conditions have the potential to never be satisfied if the thresholds are too strict. So an alternative is to record the means and variations of the last $2N$ population averages, stopping when there is no *statistical* difference between the mean of the last N values and the mean of the N values before that.

While GA convergence necessarily means that individuals in the population become more similar as time progresses, it is important in the GA method to maintain a sufficient level of population *diversity* if the algorithm is to be adequately guided towards optimality. This is especially important in the early stages of the algorithm to avoid premature convergence. So although the population does progress towards uniformity, this is achieved implicitly in a carefully controlled manner.

3.3.7 Additional Considerations

An important factor in a practical implementation of a GA, that is often overlooked, is how to select the initial population and more importantly what size population should be used.

Initial populations are usually randomly generated. In a bit string representation this means generating random strings of "0"s and "1"s but can sometimes be more complicated in combinatorial optimisation if feasibility is a concern. Seeding the initial population with good quality solutions will almost certainly produce faster convergence, however this convergence may be premature. Any initial population seeding must be done carefully if reasonable population diversity is to be maintained throughout the GA run and good quality solutions produced.

Another concern when implementing a GA is population size. Theoretical guidelines for selecting this parameter are at present rather impractical in terms of computation time and in many applications the choice of population size is rather arbitrary. Reeves [35] mentions several studies looking at population size, among them empirical studies suggesting that for many problem types populations as small as 30 produce adequate results, while other studies maintain that a population size between l and $2l$, where l is the chromosome length, is best.

Apart from the standard reproductive operators reproduction, crossover, and mutation other operators and techniques can be used in the GA process. Goldberg [19] devotes a chapter to such considerations. Topics discussed are dominance, inversion, segregation, duplication and deletion, niche and speciation, multiobjective optimisation, hybridisation and knowledge-based techniques, and parallel implementations.

3.4 Theory

In previous sections it has been claimed that via the operations of reproduction, crossover, and mutation the GA will converge over successive generations towards the global (or near global) optimum. Why these simple operators should produce a fast, useful, and robust technique is largely due to the fact that GAs combine direction and chance in the search in an effective and efficient manner. This is achieved by building new solutions from the best partial solutions of previous generations in a randomised manner. Since populations implicitly contain much more information than simply the individual fitness scores, GAs combine the good information hidden in a solution with good information from another solution to produce new solutions with good information inherited from both parents, inevitably (hopefully) leading towards optimality.

The theory of GAs is still largely based on the original work by Holland [24] and deals with the basic GA paradigm of reproduction, crossover, and mutation in a binary coding. What follows is essentially a précis of Holland's original work as reported by Goldberg [19] and Reeves [35], emphasising the fact the following arguments involve uncertainty, and as such are stated in terms of expectation.

3.4.1 Schemata

When looking at a population of solutions to a problem it would be useful to be able to recognise similarities in the chromosomes representing good solutions. These similarities could then be exploited to construct new solutions that will hopefully also share these good characteristics. It is claimed that this is precisely what a GA does by selecting individuals to reproduce on the basis of their fitness.

A *schema* (plural *schemata*) describes a subset of strings with similarities at certain positions. Schemata can be created by adding a wild card symbol, "$*$", to the alphabet used in the coding of the problem and create strings from this new extended alphabet. A schema is essentially a pattern matching device, matching a particular string from a binary alphabet if in every position a "1" matches a "1", a "0" matches a "0", or a "$*$" matches either. Hence the schema

$$* \quad 0 \quad 1 \quad 1 \quad 0 \quad * \quad 1 \quad 0 \quad 1$$

matches the 4 strings below , since each of the two wild card symbols can take on either of the two values in the binary alphabet.

$$\begin{array}{ccccccccc} 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \end{array}$$

In order to distinguish between different schema two properties are used, schema *order* and *defining length*. The order of a schema S, $o(S)$ is the number of fixed (non wild card) positions in its length. So the order of the above schema is 7. The defining length of a schema S, $\delta(S)$, is the distance between the first and last fixed string position. In the above schema, the defining length is $9 - 2 = 7$.

For any alphabet of cardinality k there are k^l possible chromosomes of length l and $(k+1)^l$ schemata. Instead of making the search space larger by considering schemata, a wealth of new information contained in a population has been discovered and can be exploited by the GA process. How much of this hidden information is processed by a GA is related to the number of schemata contained in a population. An individual string of length l is an instance of 2^l schemata, since at each position along its length a potential schema can take either the actual value of the string at that location or the wild card symbol. Hence in a population of size n there are between 2^l and $n2^l$ schemata present depending on how many of the schemata overlap due to population diversity (more than one string can be an instance of the same schema). So it seems a great number of possible search directions are tested using only a few trials. This property is termed *implicit* or *intrinsic* parallelism[5], and is the key ingredient in the strength of the GA technique.

However, it must still be determined how many schemata present in a population are processed usefully by the algorithm. This is answered to a certain extent by the Fundamental Theorem of Genetic Algorithms, or as it is more commonly known, the *Schema Theorem*.

3.4.2 The Schema Theorem

To first determine the effect of reproduction on the expected number of schemata in a population of size n, let $m(S,t)$ be the number of instances of a particular schema S in the population at time t, and note that if a parent is selected for mating in proportion to its fitness then a particular string x will get chosen with probability $p(x) = \frac{f(x)}{\sum f(x)}$ where f is the fitness function. Then the expected number of instances of the schema S at time $t+1$ is given by $E(S,t+1) = m(S,t)n\frac{f(S)}{\sum f(x)}$ where $f(S)$ is the average fitness of all strings that are instances of the schema. As the average population fitness may be written $\overline{f} = \frac{\sum f(x)}{n}$ then:

$$E(S,t+1) = m(S,t)\frac{f(S)}{\overline{f}}$$

This shows that (in the absence of crossover and mutation) schemata with above average fitness values will, on average, receive an increasing number of representations in the next generation, while schemata with below average fitness values will, on average, receive a decreasing number of representations in the next generation. This expected behaviour is carried out in parallel for all schemata present in the population. The growth (decay) in schemata representation propagates over time in a geometric manner, and hence reproduction is said to allocate exponentially increasing (decreasing) numbers of mating opportunities to above-average (below-average) schemata.

Once reproduction has allocated good schemata a greater chance to reproduce and poor schemata a smaller chance to mate, the operators crossover and mutation are applied (with probabilities p_c and p_m respectively), thus affecting the representation of schemata in the next generation.

Restricting ourselves to simple crossover, it is not hard to see that whether a schema is disrupted is largely dependent on its defining length. Consider

[5] Holland [24] originally used the term "intrinsic parallelism" but later changed his terminology to "implicit parallelism" to avoid confusion with notions in the parallel computing field.

the following chromosome C, and two representative schemata S_1 and S_2:

$$\begin{array}{cccccccc} C & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ S_1 & * & 1 & * & * & * & * & 0 \\ S_2 & * & * & * & 1 & 0 & * & * \end{array}$$

The location of the crossover point affects whether the two schemata will survive into the next generation. S_1 is destroyed if crossover occurs in any of the 5 locations between 1 and 0 (unless of course the other parent has a 0 and a 1 in the same positions). On the other hand S_2 is destroyed only if crossover occurs in a single position, between the 1 and the 0. Assuming simple crossover occurs uniformly over the schema S of length l, destruction occurs with a probability not greater than $\frac{\delta(S)}{l-1}$. Since crossover is applied with probability p_c then a lower bound on the probability of S surviving the crossover operator is

$$p(S\ survives) \geq 1 - p_c \frac{\delta(S)}{l-1}$$

Thus the combined effect of crossover and reproduction is that a lower bound on the expected number of schemata surviving into the next generation is given by:

$$E(S, t+1) \geq m(S,t) \frac{f(S)}{\overline{f}} \left[1 - p_c \frac{\delta(S)}{l-1} \right]$$

This relationship implies that those schemata with above-average fitness *and* short defining lengths are likely to survive into the next generation in an exponentially increasing manner.

Mutation is the final operator in a standard GA that will affect the analysis of schema representation. Consider the effects of simple mutation, applied to a gene with probability p_m. The gene remains unchanged by mutation with probability $1 - p_m$. As all mutations in a chromosome are statistically independent, and there are $o(S)$ fixed positions in a schema that must survive if the schema is to remain intact, then the probability of a schema S surviving mutation is $(1 - p_m)^{o(S)}$. For $p_m << 1$ this probability has a lower bound of $1 - p_m o(S)$.

Thus the combined effect of reproduction, crossover, and mutation is, for any schema S in the current population, the expected number of representations of S in the next generation is given by the *Schema Theorem*:

$$E(S, t+1) \geq m(S,t) \frac{f(S)}{\overline{f}} \left[1 - p_c \frac{\delta(S)}{l-1} - p_m o(S) \right]$$

The conclusions of this fundamental theorem are that short defining length, low order schemata of above average fitness will receive (in expectation) an exponentially increasing representation in subsequent generations.

It is proposed that short, low order shemata combine together to form better solutions as the number of generations increases. This conjecture, discussed in length below, has its own name, the *building block hypothesis*.

3.4.3 The $O(n^3)$ Estimate

From the Schema Theorem it can be seen that not all schemata have the same probability of surviving under the standard reproductive plan. Also, in a population of size n with a chromosome length l, there are between 2^l and $n2^l$ schemata present depending on population diversity. Schemata that are short, low order, and have above average fitness are more likely to survive into the next generation. Furthermore the representation in successive generations of such schemata increases in an exponential manner. A lower bound on the number of schemata that are usefully processed by the GA is this manner is now derived.

To simplify the procedure count only those schemata that have probability $1-\epsilon$ of surviving reproduction (where ϵ is a transcription error, a function of schema fitness, length, and order), meaning the counting process is restricted to schemata with length $\delta(S) < 2L$ where L is a function of ϵ. Also count only those schemata which are of order L.

Consider a chromosome of length l and a 'window' of length $2L$ placed at the left hand side of the chromosome, i.e. encompassing $2L$ positions of the chromosome counting from the leftmost end of the window. There are ${}^{2L}C_L$ possible ways to choose positions for L fixed values within the window giving rise to a number of order L schemata, since any combination of L fixed values and L wildcard symbols, "$*$", within the window and $n-2L$ wild card symbols outside the window defines an order L schema over the chromosome. All of these fixed values take on a value in the alphabet. In the binary alphabet for a given combination of fixed values inside the window there are an associated 2^L order L schemata since any fixed position can take on two values. Now add a new character (as well as the wild card symbol) to the alphabet used for coding. A "#" is used inside the window to represent a position that can either take on a fixed value in the alphabet or the wildcard symbol. The window is then 'slid along' the chromosome until the far end is reached while counting numbers of schemata. Figure 3.4 shows an instance of this procedure for a chromosome of length 10, and a window of length 6.

In the first window shown in figure 3.4 each "#" symbol can take on the actual value of the associated gene or the wildcard symbol. Since counting is restricted to schemata of order 3, inside the window 3 fixed values must be selected giving a possible $2^3 = 8$ schemata for each set of fixed values. There are a possible ${}^6C_3 = 20$ ways of positioning the fixed values in the window, giving a total of $8 \times 20 = 160$ order 3 schemata. A similar process, achieved by sliding the window along the length of the chromosome leads to the total number of order 3 schemata of length less than 6.

Generally speaking, there are $l - 2L + 1$ positions that the window can take. However in any other position but the first there are only ${}^{2L-1}C_{L-1}$ choices in selecting positions for the fixed values. At the right hand side of the window the last "#" symbol cannot take on the wildcard value since the previous window has already counted these schemata when its left most "#" took on the "$*$" symbol. Allowing the farthest right "#" symbol to take the "$*$" value would result in double counting of schemata. Hence a fixed

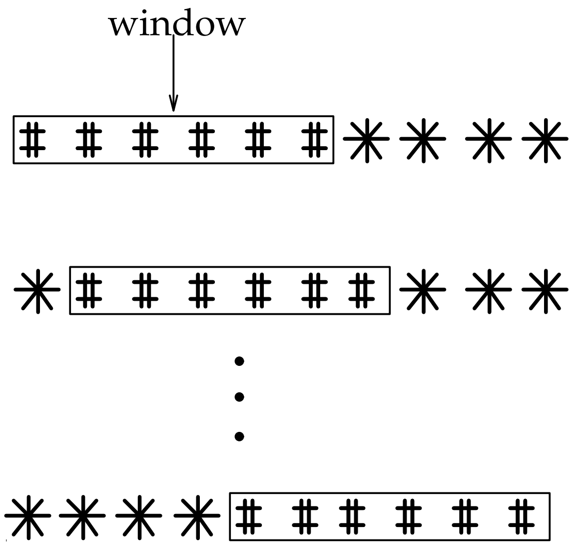

Figure 3.4. Schema counting. For a chromosome of length l, restricting the process to counting schemata of defining length $\delta(S) \leq 2 \times 3$ and order $o(S) = 3$, consider a 'window' of length 6 and count the schema contained inside. This process is repeated by sliding the window along the chromosome until the far end is reached. A "#" symbol can stand for a fixed alphabet value or the wildcard symbol.

value must be assigned to this position, leaving $L - 1$ fixed values to place in a possible $2L - 1$ positions.

After sliding the window to the right $l - 2L$ times the count is:

$$\left[\binom{2L}{L} + (l - 2L)\binom{2L-1}{L-1}\right] 2^L$$

schemata of order L with defining length $\leq 2L$. Using Stirling's approximation[6] the total number of schemata S with $o(S) = L$ and $\delta(S) \leq 2L$ for a string of length l is approximately:

$$\frac{2^{3L-1}}{\sqrt{\pi L}}$$

Finally note that for a particular chromosome all 2^L schemata are not represented for a particular choice of fixed values, hence the need to estimate the number of relevant schemata present in a population of size n. Consider a particular chromosome from a uniformly random population and a given schema of order L. Each locus of the chromosome has a probability of $\frac{1}{2}$ of matching the relevant schema value, for each of the L fixed values along the schema string (the other $l - L$ positions are "$*$"s and will match anything). So there is a $\left(\frac{1}{2}\right)^L$ probability of the chromosome matching the given schema. Thus by choosing $n = v2^L$ (for v a small integer) then in a random population it is expected there will be v instances of the order L

[6]James Stirling in *Methodus Differentialis* (1730) showed $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

schemata[7]. Using this definition of population size it is seen that an estimate of the number of schemata of length $2L$ or less, and order L is

$$\frac{2^{3L-1}}{\sqrt{\pi L}} = \frac{2^{3L}}{2\sqrt{\pi L}} = \frac{n^3}{2v^3\sqrt{\pi L}} = O(n^3)$$

This (probable) lower bound on the number of usefully processed schemata in a population is known as the $O(n^3)$ estimate, demonstrating the intrinsic parallelism of a GA, the ability to process a large number of schemata while considering a relatively small number of chromosomes.

3.4.4 The Building Block Hypothesis

With perspective granted by schema analysis, the Schema Theorem and the $O(n^3)$ estimate have made two points clear. Firstly, that short defining length, low-order schemata of above average fitness are selected for reproduction, crossover, and mutation in successive generations in an exponentially increasing manner, and secondly that these schemata are effectively processed by the algorithm in an implicitly parallel way.

The above mentioned schemata are termed *building blocks*, and it is claimed that a GA combines these building blocks together to produce solutions of fitness potentially higher than that of solutions in previous generations. The notion that better and better solutions are constructed from the best partial solutions (building blocks) of past solutions is known as the *building block hypothesis*.

Termed a hypothesis, since no proof has ever been given, it is nevertheless widely believed that the building block hypothesis is responsible for the success of the GA technique. The fact that the algorithm has been shown empirically to work well for many different problem types gives strong evidence in support of the hypothesis in practical situations. Two recent theoretical advances that tend to give credence to this belief and greater theoretical background to the entire GA method are *Walsh function analysis* and the *deceptive problem*.

Walsh functions are essentially discrete analogues of trigonometrical functions originally developed by mathematicians and engineers for the field of signal processing. They are applicable to schemata analysis since one way to envisage schemata is as 'periodic' functions or as different 'frequencies'. Consider figure 3.5. In a situation where binary coding represents integers, the schema $**\cdots**1$ represents all the odd integers.

Since the Schema Theorem is an expression about expected values of numbers of schemata, it is possible that even the fittest schemata have a reasonable chance of failing to survive due to natural variation. Walsh function analysis can be used to determine the variation in the number of schemata surviving through to the next generation, and these theoretical considerations can lead to a greater understanding of effective parameter choices and the mechanics of the algorithm.

[7] Since $\left(\frac{1}{2}\right)^L v2^L = v$

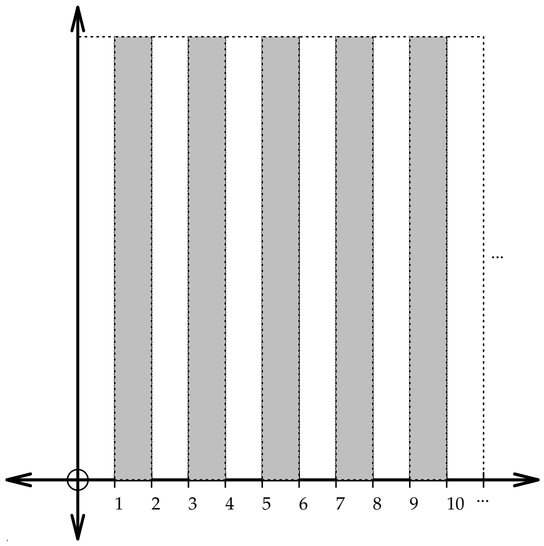

Figure 3.5. A periodic schema. When the binary coding represents integer values, the schema $* * \cdots * * 1$ represents all odd integers, shown here shaded in gray strips. The white strips showing the even integers, are given by the schema $* * \cdots * * 0$. Other schemata can be similarly represented.

Walsh function analysis also leads to another tool used in GA theory, the deceptive problem. It is possible to construct (with the help of Walsh function analysis) problems for which the building block hypothesis appears to fail, problems with isolated optima surrounded by the worst solutions. Partial solutions will lead the algorithm away from the optimal value rather than towards it. Although this sort of problem will be difficult for any search algorithm to solve, and practical problems are rarely likely to have this peculiar type of structure, deceptive problems give insight into the dependence of GAs on the building block hypothesis and what is required for the algorithm to succeed.

Another perspective on schemata may be reached by taking a geometrical look underlying space (a bit space if considering the binary alphabet). Consider schemata constructed from the binary alphabet of string length 3. Geometrically order 3 schemata are points in the space, order 2 schemata are lines, order 1 schemata are planes, and the order 0 schema is the whole space. As shown in figure 3.6 schemata *hyperplanes* are formed in three dimensions. In general schemata from the binary alphabet $\{0,1\}$ of string length l define hyperplanes in a space of dimension l with the order of the schema defining the dimension of the hyperplane. By considering and recombining schemata the GA technique can be seen to cut across various hyperplanes to search for improved performance.

A more recent development in the understanding of why crossover and the building block hypothesis works is the concept of a *forma* (plural *formae*), given by Radcliffe [34]. Schemata represent similarities in gene value (allele) at a specified position on the chromosome (locus). Formae represent similarities in a more generalised manner, such as the fact that a chromo-

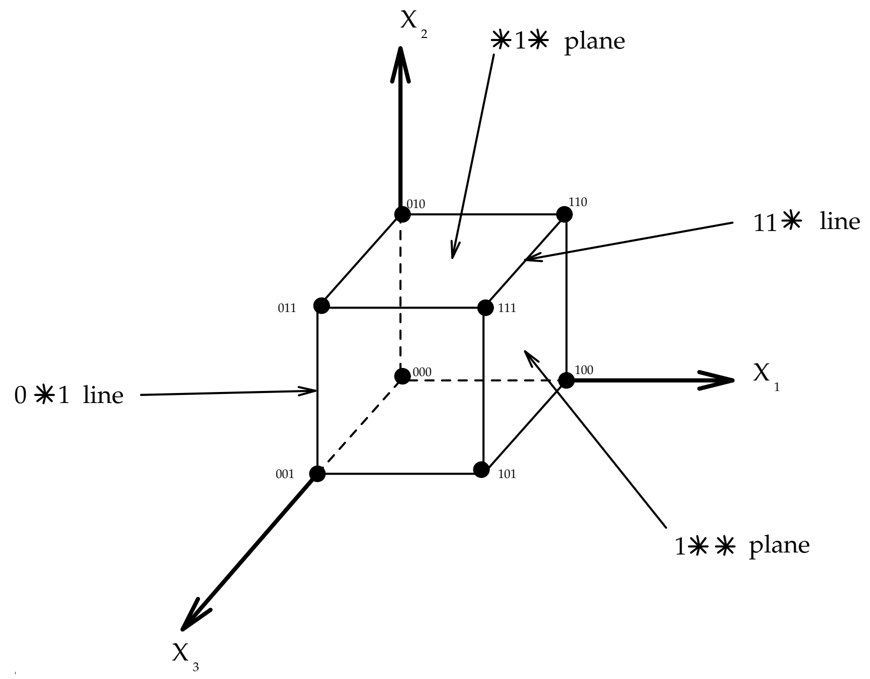

Figure 3.6. Schemata hyperplanes. Schemata hyperplanes in three dimensions representing schemata of string length 3 from the binary alphabet $\{0,1\}$. The schema order defines the dimension of the hyperplane, producing (in three dimensions) points, lines, planes, and the entire space.

some contains a specific gene regardless of its locus. Through the formae approach it is possible to design operators with desirable properties, properties such as *respect* and *proper assortment*. In general terms respect is the property that whenever both parents have a certain characteristic the offspring should always inherit this characteristic. Sometimes in conflict with respect is proper assortment, which says that whenever parents have different characteristics it should be possible for the offspring to inherit both characteristics (essentially a less general interpretation of the building block hypothesis).

3.4.5 The Genetic Algorithm–A Robust Technique

After consideration of the GA approach using schemata analysis, the main result given by the Schema Theorem is that high-performance, short-defining-length, low-order schemata receive exponentially increasing numbers of opportunities to reproduce in successive generations. Looking at the number of such schemata effectively processed by the algorithm, it is estimated that there are at least $O(n^3)$ schemata processed in what is described as an implicitly parallel manner. Processing schemata in this manner means that problem complexity is reduced by building better solutions from the best partial solutions of previous generations. The assumption that these building blocks do in fact lead to increasingly better solutions is known as the building block hypothesis. The hypothesis has not been proved, but after successful GA application to many different problem types and consideration of the GA process in different theoretical lights, such as Walsh function analysis and the deceptive problem, the general conclusion is that the hypothesis is reasonable in most situations.

As iterative algorithm attempting to find the global optimum needs to *explore* the unknown areas in the search space while *exploiting* knowledge of the space searched so far in order to find better solutions. These two contradictory requirements have been shown by Holland [24] to be optimally combined by a GA so that both aspects are simultaneously met. This is true in theory but in practical situations there are the limitations of the need for a finite population, no interaction between genes in a chromosome, and the ability of the fitness function to reflect the true utility of a solution. The ability of the algorithm to explore and exploit simultaneously, a growing amount of theoretical justification, and successful application to real-world problems strengthens the conclusion that GAs are a powerful, robust optimisation technique.

3.5 Genetic Algorithms and the TSP

A *weak* method, borrowing from the terminology of artificial intelligence, is a problem solving technique that makes few assumptions about the actual problem structure so is generally applicable to a wide variety of problems. Most weak methods are also *low power* since they are not very efficient. GAs are an anomaly; a powerful, weak method, efficiently sampling the search space, even with knowledge of the problem restricted to the coding and fitness function. However with many problem types it is not possible to restrict GAs to such minimalism, additional problem knowledge may need to be incorporated into some of the operators, especially crossover and mutation. It may also be necessary to abandon the binary coding that is part of the standard GA paradigm.

Binary representation has generally been used for four reasons, simplicity of schema analysis, elegance and simplicity of the genetic operators, computational speed, and the fact that binary encoding maximises the number of schemata sampled by a set of individuals in a population. Antonisse [1] questions the validity of this last fact, which in turn calls into question the basis of implicit parallelism. By redefining the wildcard symbol to represent an element from *any* subset of the coding alphabet (rather than simply representing any element of the *entire* alphabet), the claim is made that alphabets of higher cardinality are needed for the algorithm to perform optimally.

The prevalence of bit string representation in current GA theory and in many GA applications has been largely responsible for the technique being ignored by potential users (in artificial intelligence for example) who would like a more meaningful coding. Indeed binary coding is not a natural way to express many problems, and consideration often needs to be given to alternative coding methods when implementing a GA. This can occur in many combinatorial optimisation problems, especially when a sequence representation is a natural solution to the problem, as in the Traveling Salesman Problem (TSP).

3.5.1 Implementing a GA for the TSP

The TSP, described in chapter 2, involves choosing the optimal order to tour n cities, visiting each city exactly once. By labeling the cities $1, 2, \ldots, n$ then a natural way to express a solution to the problem is as a sequence of cities. Thus the sequence

$$2 \quad 3 \quad 7 \quad 4 \quad 1 \quad 6 \quad 9 \quad 8 \quad 5 \quad 10$$

represents the (cyclic) tour

$$2 \mapsto 3 \mapsto 7 \mapsto 4 \mapsto 1 \mapsto 6 \mapsto 9 \mapsto 8 \mapsto 5 \mapsto 10 \mapsto 2$$

The sum of the intercity distances for the sequence, the tour length, represents the quality (or fitness) of the solution. However applying the standard genetic operators, crossover and mutation, to this particular coding does not (usually) yield feasible solutions. Consider the two solutions P_1 and P_2.

P_1	2	3	7	4	1	6	9	8	5	10
P_2	5	9	1	6	7	3	2	10	4	8

Applying single point crossover to these two solutions between the 4^{th} and 5^{th} string positions gives O_1 and O_2 as offspring.

O_1	2	3	7	4	7	3	2	10	4	8
O_2	5	9	1	6	1	6	9	8	5	10

Neither of these are valid TSP solutions. Similarly by applying simple mutation to the 6^{th} position of P_1 an infeasible solution will usually result, such as the following possible outcome:

$$2 \quad 3 \quad 7 \quad 4 \quad 1 \quad 1 \quad 9 \quad 8 \quad 5 \quad 10$$

For sequence solutions to the TSP it is necessary to redefine these two operators to produce feasible solutions. It is desirable for crossover to combine genetic information in a purposeful manner, so that the building block hypothesis is at least possible. Various attempts have been made to produce a 'good' crossover operator for the TSP, four of these are given below:

The Order Crossover: This operator produces two offspring, O_1 and O_2, from two parents, P_1 and P_2, proceeding as follows. Both parents are cut into three sections each, determined by the selection of two randomly selected cut points. The middle section of P_1 is copied intact into the corresponding position of O_1. The rest of O_1 is filled in with values taken in order from the third, first, and second sections of P_2, skipping values that have already been copied from the first parent. O_2

is produced in the same manner by reversing the roles of the parents. With specific instances of P_1 and P_2 below, randomly choosing the cut points between the 3^{rd} and 4^{th} and 7^{th} and 8^{th} elements, the following offspring are produced.

P_1	2	3	7	4	1	6	9	8	5	10
P_2	5	9	1	6	7	3	2	10	4	8
O_1	7	3	2	4	1	6	9	10	8	5
O_2	4	1	9	6	7	3	2	8	5	10

Consider O_1. The middle section of P_1 is copied intact into the middle section of O_1. The rest of O_1 is now created from looking at P_2 ignoring values that have already been used. The third, first, and second sections of P_2 define the sequence

$$10 \quad 4 \quad 8 \quad 5 \quad 9 \quad 1 \quad 6 \quad 7 \quad 3 \quad 2$$

Removing values already present in O_1 leaves

$$10 \quad 8 \quad 5 \quad 7 \quad 3 \quad 2$$

So the third and first sections of O_1 are filled by taking elements in order from this sequence. O_2 is created in a similar manner.

The order crossover is a generalisation by Oliver *et al.* [33] of the "modified" crossover first presented by Davis [8]. However the operator has been independently discovered by a number of researchers, refer to Reeves [35] for a full description.

The rationale behind the crossover is it preserves the absolute positions of jobs taken from one parent while maintaining some of the relative positions of jobs taken from the other parent. It is hoped that this will provide enough scope to modify schemata without too much disruption to the building blocks. The operator seems to preserve compactly represented schemata well, which is reflected in Oliver *et al.* [33] by its reasonable performance.

The PMX Crossover: By choosing two random cut points in the same manner as for the order crossover, each parent is split into three sections. Offspring are produced by considering the two middle sections as an *interchange mapping*. In those positions in each parent where an element from either one of the two middle sections appears, this value is swapped with the corresponding value from the other middle section. With the same two parents as for the order crossover, and the same two cut points, the PMX crossover produces O_1 and O_2 as offspring.

P_1	2	3	7	4	1	6	9	8	5	10
P_2	5	9	1	6	7	3	2	10	4	8
O_1	9	4	1	6	7	3	2	8	5	10
O_2	5	2	7	4	1	6	9	10	3	8

In the above instance, the middle sections define the following interchange map:

$$\begin{array}{c} 4 \leftrightarrow 6 \\ 1 \leftrightarrow 7 \\ 6 \leftrightarrow 3 \\ 9 \leftrightarrow 2 \end{array}$$

Applying this swapping action to P_2, O_2 is produced as follows: 5 is unaffected, $9 \mapsto 2$, $1 \mapsto 7$, $6 \mapsto 4$, $7 \mapsto 1$, $3 \mapsto 6$, $2 \mapsto 9$, 10 is unaffected, $4 \mapsto 6 \mapsto 3$, and finally 8 is left unchanged. O_1 is produced in a similar way.

Generally recognised as the first crossover attempt in the application of GAs to the TSP, the *partially-mapped crossover* (PMX) of Goldberg and Lingle [18] attempts to recognise the fact that for TSP codings, unlike standard GA problem codings, gene order and allele information can affect the fitness evaluation. The claim is made in Goldberg and Lingle [18] that the operator preserves ordered building blocks in a manner that enables different orders and allele combinations to be explored with implicit parallelism. The PMX crossover preserves the absolute positions of some elements from both parents, so is able to preserve compact schemata with reasonable effectiveness. Using only reproduction and the PMX crossover Goldberg and Lingle solved, to near optimality, a well known 10 city TSP.

The Cycle Crossover: With two parents selected for reproduction it is possible to create offspring such that each element in an offspring takes the value of the corresponding position of one of the parents. Each mating couple defines a number of *cycles* and by randomly choosing a parent for each cycle, offspring are constructed. Consider a particular instance of this operator with P_1 and P_2 below.

$$\begin{array}{ccccccccccc} P_1 & 2 & 3 & 7 & 4 & 1 & 6 & 9 & 8 & 5 & 10 \\ P_2 & 5 & 9 & 1 & 6 & 7 & 3 & 2 & 10 & 4 & 8 \end{array}$$

This couple defines the following cycles:

$$\begin{array}{l} 2 \mapsto 5 \mapsto 4 \mapsto 6 \mapsto 3 \mapsto 9 \mapsto 2 \\ 7 \mapsto 1 \mapsto 7 \\ 8 \mapsto 10 \mapsto 8 \end{array}$$

Thus each parent is partitioned into three sets A, B, and C represented by the following cycle label vector.

$$A \quad A \quad B \quad A \quad B \quad A \quad A \quad C \quad A \quad C$$

By randomly selecting a parent for each partition, two offspring are created. For example one O_1 below is created by choosing A from P_1 and B and C from P_2 whereas O_2 is created by choosing A from P_2 and B and C from P_1.

$$\begin{array}{ccccccccccc} O_1 & 2 & 3 & 1 & 4 & 7 & 6 & 9 & 10 & 5 & 8 \\ O_2 & 5 & 9 & 7 & 6 & 1 & 3 & 2 & 8 & 4 & 10 \end{array}$$

Oliver *et al.* [33] propose this novel approach to a crossover for the TSP by attempting to produce offspring that are different from the parents but where every position in an offspring is occupied by a corresponding element from one of the two parents. On average the absolute positions of half the elements from each parent are preserved. Cycle crossover can preserve schemata independent of length, which is ideal when schemata compactness is not important, seemingly responsible for the ordinary performance of the operator in the series of tests carried out by Oliver *et al*.

The Edge Recombination Crossover: A single offspring is generated from the mating of two parents by following the algorithm given below:

1. Construct an *edge map* from the parent tours. For each city the edge map stores an *edgelist* of all the connections from the two parents that lead into and out of the city.
2. Choose the initial city from one of the two parent tours. (This can be done randomly or according to criteria outlined in step 5.) Call this the "current city".
3. Remove all instances of the current city from the right-hand side of the edge map. (These can be found by referring to the edgelist of the current city.)
4. If the current city has entries in its edgelist go to step 5, otherwise go to step 6.
5. Determine which of the cities in the edgelist of the current city has the fewest entries in its own edgelist. The city with the least number becomes the new current city (ties are broken randomly). Go to step 3.
6. If the tour has visited every city then stop; otherwise randomly select a city not yet visited and go to step 3.

Consider P_1 and P_2 as given below.

P_1	2	3	7	4	1	6	9	8	5	10
P_2	5	9	1	6	7	3	2	10	4	8

The edge map for these tours is:

1	$\mapsto$	4	6	9	
2	$\mapsto$	3	10		
3	$\mapsto$	2	7		
4	$\mapsto$	1	7	8	10
5	$\mapsto$	8	9	10	
6	$\mapsto$	1	7	9	
7	$\mapsto$	3	4	6	
8	$\mapsto$	4	5	9	
9	$\mapsto$	1	5	6	8
10	$\mapsto$	2	4	5	

Select an initial city from the parent tours to be the current city. Since 2 has 2 cities on its edgelist and 5 has 3 cities, select 2 as the current city. Delete all instances of 2 from the edgemap. Looking at the edgelist of 2 gives the cities 3 and 10. 3 now has 1 city in its edgelist while 10 has 2, so choose 3 to be the new current city and delete all instances of 3 from the edge map. 3 only has an edge to 7, so choose 7 as the new current city and delete all instances of 7 from the map. Then in the same manner choose 6, then 1, then break the tie between 4 and 9 arbitrarily by selecting 9, similarly break the tie between 5 and 8 by choosing 5, then break the tie between 8 and 10 by choosing 10, then choose 4, and finally 8. Thus the edge recombination crossover produces O_1 as offspring.

$$O_1 \quad 2 \quad 3 \quad 7 \quad 6 \quad 1 \quad 9 \quad 5 \quad 10 \quad 4 \quad 8$$

Whitley *et al.* [42] attack the sequence crossover problem in a altogether new way by producing the edge recombination operator described above. Instead of emphasising the relative or absolute positions of an element in a string, the authors attempt to improve on the performance of a previous "edge list" approach by Grefenstette [20] by attempting to preserve the *adjacency* of string elements. They hypothesize that since it is the *edges* that reflect the goodness of a solution, offspring should be created by recombining edges. Using an edgemap to choose cities with the fewest remaining connections, the procedure ensures that most cities are connected by edges from the parent tours and that the offspring are composed almost entirely of relevant information from the parent tours.

Whitley *et al.* [42] also show that the search space can be modeled as a binary hyperspace meaning the success of their operator can be explained using a variation on the traditional schemata theory. Whitley shows that the edge recombination operator seems to work very well, solving moderate to large TSP instances to optimality . In a later paper Whitley *et al.* [43] note that even better performance is achieved by increasing the size of the population.

The first question that naturally springs to mind after considering the various sequence crossovers is which one produces the best quality solutions. The order crossover and the PMX crossover operators emphasise preservation of relative or absolute string positions. They both ignore the difference between the TSP and the kind of ordering independent problems to which GAs are more naturally suited. Tours are simply 'crossed' to give offspring which are generally illegal (some cities are visited either twice or not at all) and then corrected to remove city duplicates or insert omissions.

The cycle crossover is quite different from the first two operators. Cycles can be exchanged between parents, producing offspring, without introducing duplicates or omissions. However this preserves absolute city positions while unfortunately destroying many links between cities that are present in the parents. Information is extracted from the parents but seemingly the wrong information, leading to poor performance.

Edges are critical in evaluating the fitness of a solution not city position, an idea reinforced empirically by Whitley *et al.* [42], who go on to point out that the cycle operator tends to destroy edges from the parents in greater quantities than the PMX operator, which in turn breaks more edges than the order crossover. This edge preserving capability is directly reflected in Oliver *et al.* [33] by the relative performance of the three operators (the order crossover worked best, followed by the PMX crossover, and finally the cycle crossover).

The edge recombination operator attempts to directly use this idea of edge preservation and constructs offspring by directly transferring edges from the parents as effectively as possible. Interestingly, the edge recombination operator follows the idea of *respect* given by Radcliffe [34] since whenever both parents have a common edge the offspring will always inherit that edge.

Since the edge recombination operator preserves edges better than the other crossover techniques discussed it is expected to produce the best results. However as edge recombination seems more computationally intensive than some of the other approaches a balance must be met between algorithm speed and quality of solution. This balance will depend on the exact purpose for which the GA is implemented.

Because ordinary mutation is applied to each string position independently, the mutation operator also needs to be re-defined for sequence representations. Reeves [35] mentions three potential operators with references, *shift* mutation, *exchange* mutation, and *a scramble sublist* mutation. Shift mutation moves a randomly selected element and inserts it elsewhere in the sequence. In the example below the shift operator moves the fourth city of the tour O_1, city 6, 3 places to the right to produce O_2.

O_1	2	3	7		6		1	9	5	10	4	8
O_2	2	3	7	1	9	5		6		10	4	8

The exchange mutation operator simply swaps two randomly selected cities in a tour. Applying this to O_1 on the second and seventh cities produces O_3 below.

O_1	2	3	7	6	1	9	5	10	4	8
O_3	2	5	7	6	1	9	3	10	4	8

A scramble sublist mutation operator scrambles all the elements in the sequence between two randomly selected cities. Choosing the third and the seventh cities from O_1 could produce O_4 as a result of mutation.

O_1	2	3	7	6	1	9	5	10	4	8
O_4	2	3	6	5	1	7	9	10	4	8

For flowshop scheduling[8] the exchange operator was found to be inferior to a shift mutation. The scramble sublist operator is more disruptive than

[8]Flowshop scheduling is a problem in combinatorial optimisation which attempts to find the best order a sequence of jobs should be processed on a number of machines.

the other two operators so in some cases the length of the sublist may need to be limited.

The final reproductive operator often used in GA applications, dependent on the problem coding, is the inversion operator, which simplistically is the reversal of a randomly chosen subsequence of the chromosome[9]. In terms of a GA coding of the TSP, this is identical to the 2-opt neighbourhood operator discussed in length in appendix A. Applying this operator to the sequence defined between the second and seventh cities on the tour represented by O_1 results in O_5.

O_1	2	3	7	6	1	9	5	10	4	8
O_5	2	3	9	1	6	7	5	10	4	8

3.6 Empirical Results

In order to investigate how altering the parameter values affects the performance of a GA, both in quality of solution and run time, a comparatively simple GA was implemented for the TSP. Details are as follows:

- Since sequence representation is the most natural expression for the TSP, bit string encoding was not used, instead sequence solutions as described in section 3.5 were used directly in the GA.

- The fitness function used is the value of the objective function for the TSP, the sum of the intercity distances in a (cyclic) tour of all cities.

- Using the generational replacement approach, binary tournament selection was implemented for constructing the mating pool.

- Reflecting the sequence coding needed for the TSP, the edge recombination operator has been implemented as a crossover method.

- Simple bit flipping mutation is not relevant for this implementation, so the mutation operator used is the 'exchange' mutation.

- A simplification of a technique described above in section 3.3 was used to determine when the GA had converged. If $X_1, X_2, \cdots, X_N$ are the average population solutions from the last N generations and $Y_1, Y_2, \cdots, Y_N$ are the average population solutions from the N generations before that, then the algorithm is deemed to have converged when $\frac{|\overline{X}-\overline{Y}|}{\overline{X}} < \epsilon$, where ϵ is some small amount.

The particular parameters investigated empirically are as follows:

Population size: An integer value representing n, the number of individuals in each generation.

[9]See Goldberg [19] for a formal definition of the inversion operator.

Crossover probability: A real number representing p_c, the probability that crossover is applied to a mating pair. If crossover is not applied then the offspring is identical to one of the parents (selected randomly).

Mutation probability: A real number representing p_m, the probability that mutation is applied to an offspring.

Convergence value: A real number representing ϵ, the critical relative difference between the means $\overline{X}$ and $\overline{Y}$ (see above) used in the determination of algorithm convergence.

Convergence set size: An integer amount representing N, the size of the sets used when determining convergence.

The empirical performance of the GA was investigated for three instances of the TSP to determine if altering the size of the problem instance has any bearing on the appropriate selection of parameters used for a GA run. Three Euclidean TSP instances of sizes 29, 51, and 100 are used in this investigation and will be henceforth referred to as bays29, eil51, and kroC100 respectively. Details of all three TSP instances can be found in appendix B.

For a complete investigation into the effects of the GA parameters, a linear style search is perhaps unsatisfactory. All possible parameter settings should be investigated to see if altering a particular parameter affects the relative performance of another parameter. Due to time constraints however, such a thorough study is infeasible and it is hoped that a linear search through the parameter space will produce sufficient information in the quest for 'optimal' parameter settings.

3.6.1 The bays29 TSP

Each of the GA parameters given above was varied, while keeping the other parameters fixed, for ten independent runs using the same ten sets of random seeds for each parameter setting. The default[10] parameter settings used in all GA runs of the bays29 problem were[11] $n = 300$, $p_c = 0.6$, $p_m = 0.05$, $\epsilon = 0.0005$, and $N = 50$.

The effects on algorithm solution quality of varying the population size are shown in figure 3.7. As the population size increases, the quality of both the best and the average solutions improves. However beyound a population size of about 250 there is little noticeable difference in either measure of algorithm quality. Notice also that the variation in the algorithm solution quality decreases as the population size increases. Both these observations are to be expected. A larger population size means that there is more opportunity to escape the trap of local optimality leading to improved solution quality. Since the population is a sample of the solution space, a larger population is more likely to reflect the area about the true minimum in an accurate manner, reducing solution variation and improving quality.

[10]Default here means the value a parameter held when not under investigation.

[11]The one exception to the convergence set size of $N = 50$ was when investigating the effect of ϵ, where $N = 25$ was used for reasons which will be explained later.

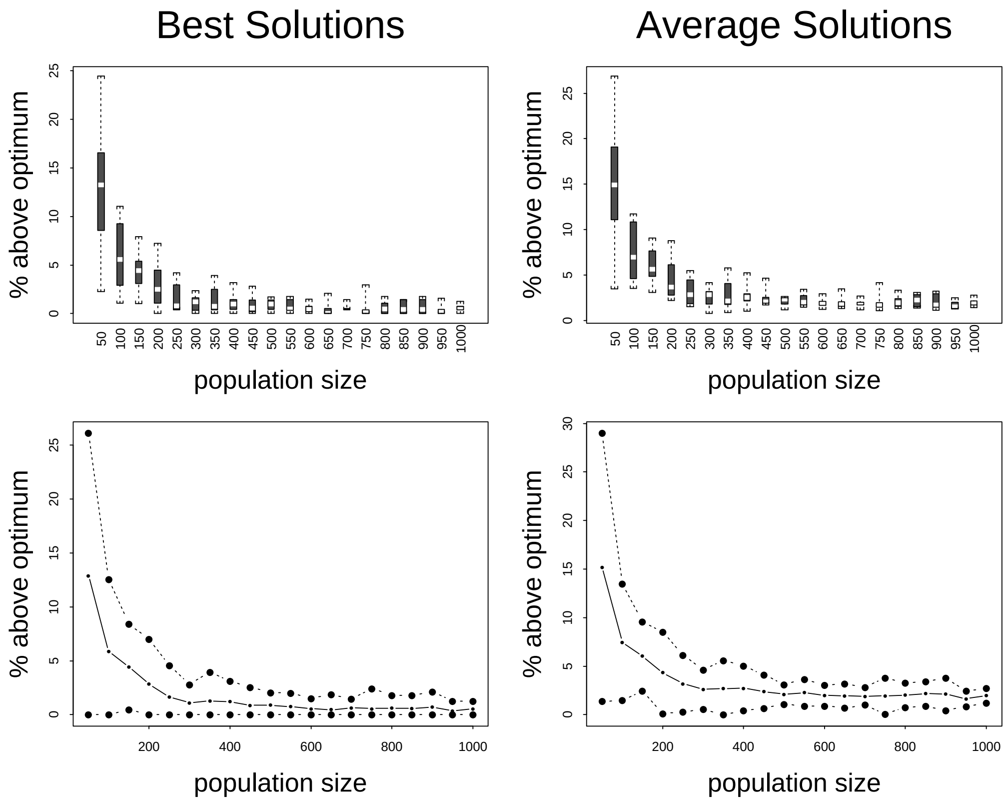

Figure 3.7. Effects of population size on solution quality for the bays29 TSP. The diagrams show the variation in GA performance as the population size changes. The top graphs show boxplots, for varying population sizes, of the final best solutions and final average solutions recorded from ten runs of the GA. The bottom graphs show means (solid lines) and 95% confidence bounds (dashed lines), for varying population sizes, of the final best solutions and final average solutions recorded from ten runs of the GA. Parameter settings: $p_c = 0.6$, $p_m = 0.05$, $\epsilon = 0.0005$, and $N = 50$.

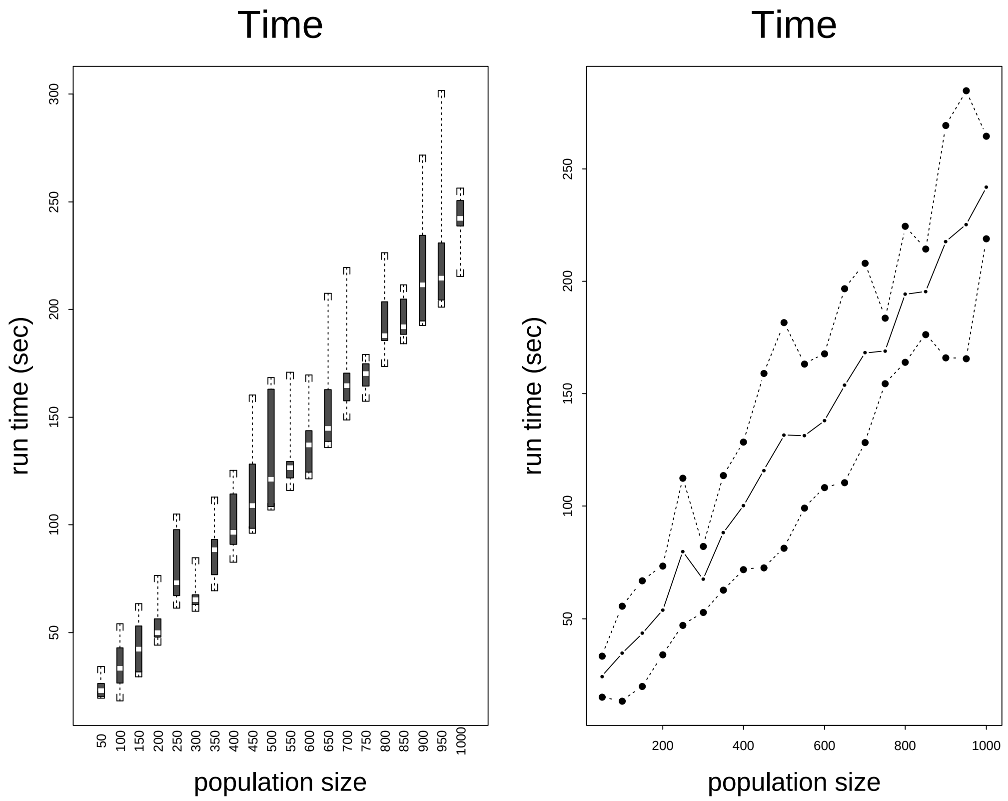

Figure 3.8. Effects of population size on run time for the bays29 TSP. The diagrams show the variation in GA run time as the population size changes. The left graph shows boxplots, for varying population sizes, of the GA run times recorded from ten runs of the GA. The right graph shows means (solid lines) and 95% confidence bounds (dashed lines), for varying population sizes, of the GA run times recorded from ten runs of the GA. Parameter settings: $p_c = 0.6$, $p_m = 0.05$, $\epsilon = 0.0005$, and $N = 50$.

The effects on algorithm run time of varying the population size are shown in figure 3.8. A clear increasing linear relationship exists between the population size and the run time. With a larger population there are more individuals to grind through the genetic mill, directly reflected in longer run times.

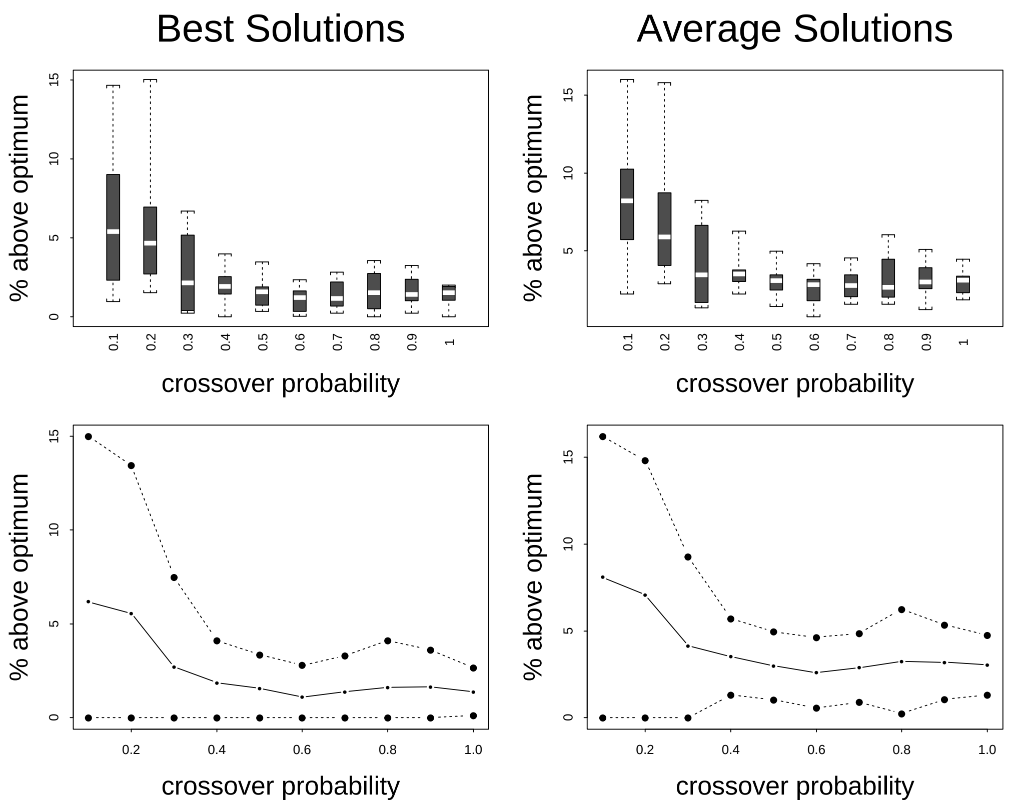

Figure 3.9. Effects of crossover probability on solution quality for the bays29 TSP. The diagrams show the variation in GA performance as the crossover probability changes. The top graphs show boxplots, for varying crossover probabilities, of the final best solutions and final average solutions recorded from ten runs of the GA. The bottom graphs show means (solid lines) and 95% confidence bounds (dashed lines), for varying crossover probabilities, of the final best solutions and final average solutions recorded from ten runs of the GA. Parameter settings: n = 300, $p_m = 0.05$, $\epsilon = 0.0005$, and N = 50.

The effects on algorithm solution quality of varying the crossover probability are shown in figure 3.9. The 'best' crossover probability appears to be $p_c = 0.6$, agreeing with the 'optimal' probability level often reported. At low levels of p_c exploitation dominates the algorithm. Rather than searching for unexplored good areas of the search space, the algorithm exploits the information it has to produce solutions which are locally good but probably not globally optimal. At high levels of p_c exploration is the dominant factor guiding the search. Now the algorithm continuously seeks new good areas without fully examining any areas found, meaning the best solutions are not usually encountered. An intermediate value of $p_m = 0.6$ strikes a compromise between exploration and exploitation, optimally searching the solution space for good areas which are thoroughly examined for the best local solution, leading to solutions which are likely to be close to global optimality.

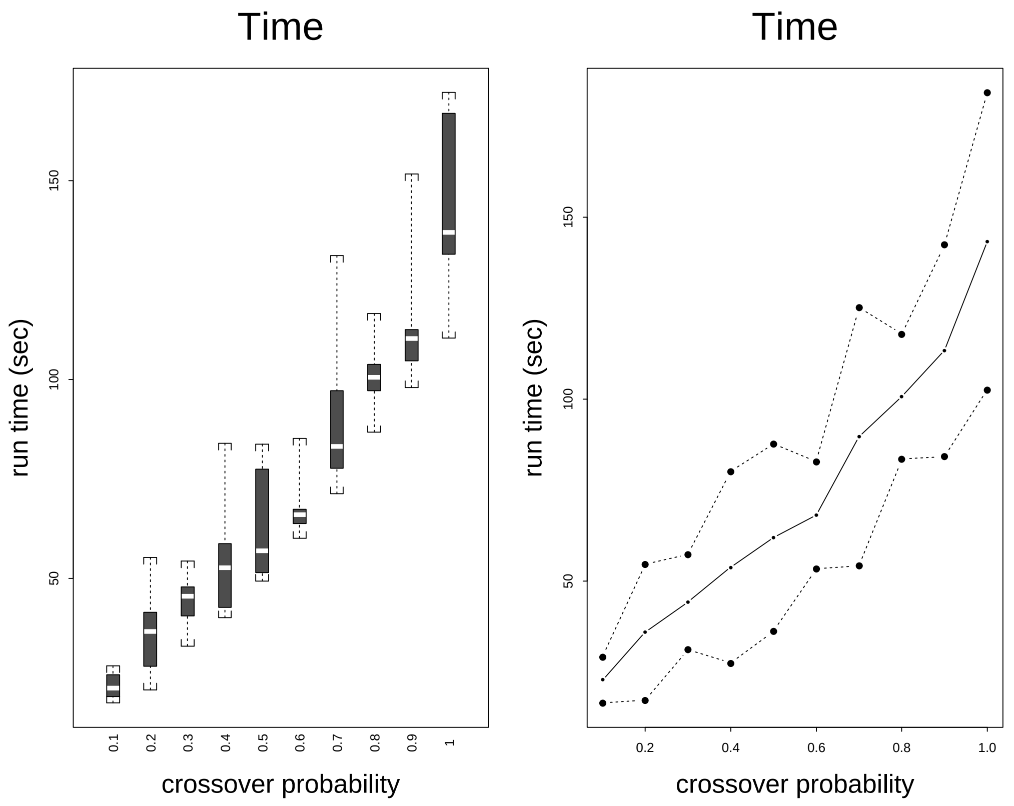

Figure 3.10. Effects of crossover probability on run time for the bays29 TSP. The diagrams show the variation in GA run time as the crossover probability changes. The left graph shows boxplots, for varying crossover probabilities, of the GA run times recorded from ten runs of the GA. The right graph shows means (solid lines) and 95% confidence bounds (dashed lines), for varying crossover probabilities, of the GA run times recorded from ten runs of the GA. Parameter settings: $n = 300$, $p_m = 0.05$, $\epsilon = 0.0005$, and $N = 50$.

The effects on algorithm run time of varying the crossover probability are shown in figure 3.10. An increasing linear relationship seems to exist between the crossover probability and algorithm run time. As the crossover probability increases, more work is done by the algorithm, since the crossover operation is invoked more often, leading to increased run time.

The effects on algorithm solution quality of varying the mutation probability are shown in figure 3.11. The quality of the best solutions produced by the GA is greatest when $p_m = 0.05$, whereas the quality of the average solutions produced by the GA is greatest when $p_m = 0$. As the mutation probability increases a 'phase-shift' seems to occur somewhere between $p_m = 0.3$ and $p_m = 0.35$, causing the algorithm to suddenly produce solutions of much poorer quality. Beyond this point the variation in the solution

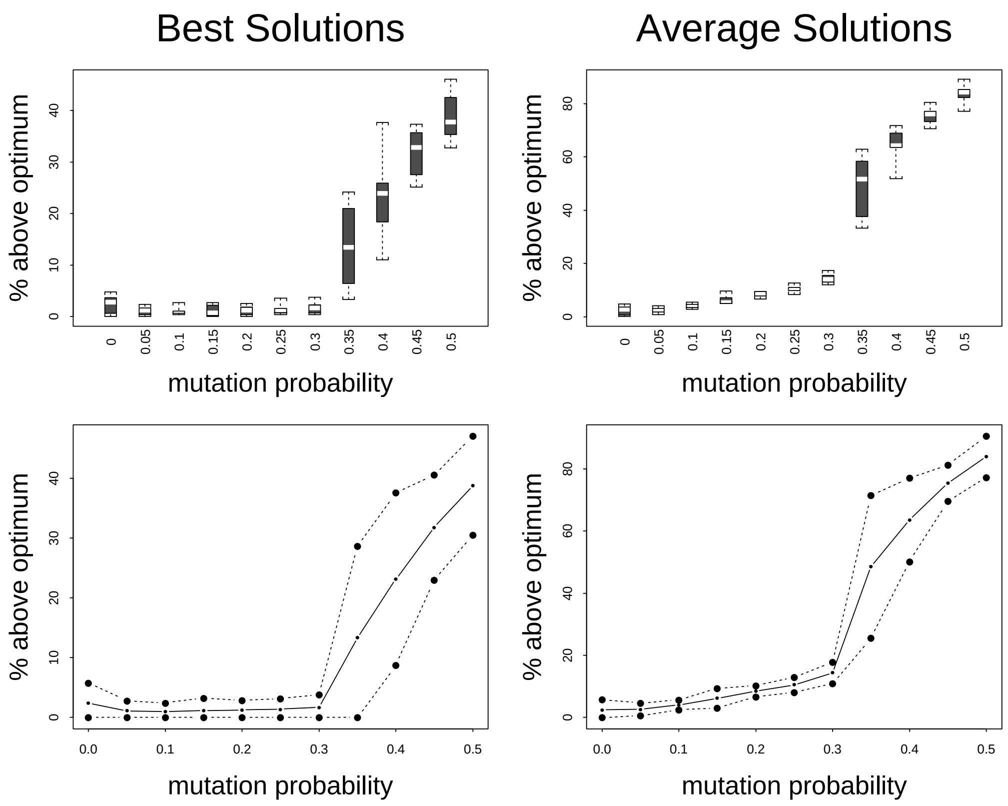

Figure 3.11. Effects of mutation probability on solution quality for the bays29 TSP. The diagrams show the variation in GA performance as the mutation probability changes. The top graphs show boxplots, for varying mutation probabilities, of the final best solutions and final average solutions recorded from ten runs of the GA. The bottom graphs show means (solid lines) and 95% confidence bounds (dashed lines), for varying mutation probabilities, of the final best solutions and final average solutions recorded from ten runs of the GA. Parameter settings: $n = 300$, $p_c = 0.6$, $\epsilon = 0.0005$, and $N = 50$.

quality also suddenly increases. An increase in variation and solution quality is expected as the mutation operator is introducing randomness into the search, destroying some of the good work done by reproduction and crossover. Why a sudden phase-shift exists is not clear. Mutation aids the algorithm in exploring the search space for new regions. For comparatively low levels of p_m the mutation can prevent the algorithm from becoming prematurely trapped in a local optimum. High levels of p_m force the algorithm to become little more than a random search. Debate is on-going about the precise role and importance of mutation in the GA paradigm, although many researchers seem to feel the importance of mutation is secondary to that of crossover. Some degree of mutation is generally necessary if the algorithm is to have a non-zero probability of reaching any solution in the search space, a condition often necessary for the theoretical asymptotic convergence of any search heuristic to a set of globally optimal solutions.

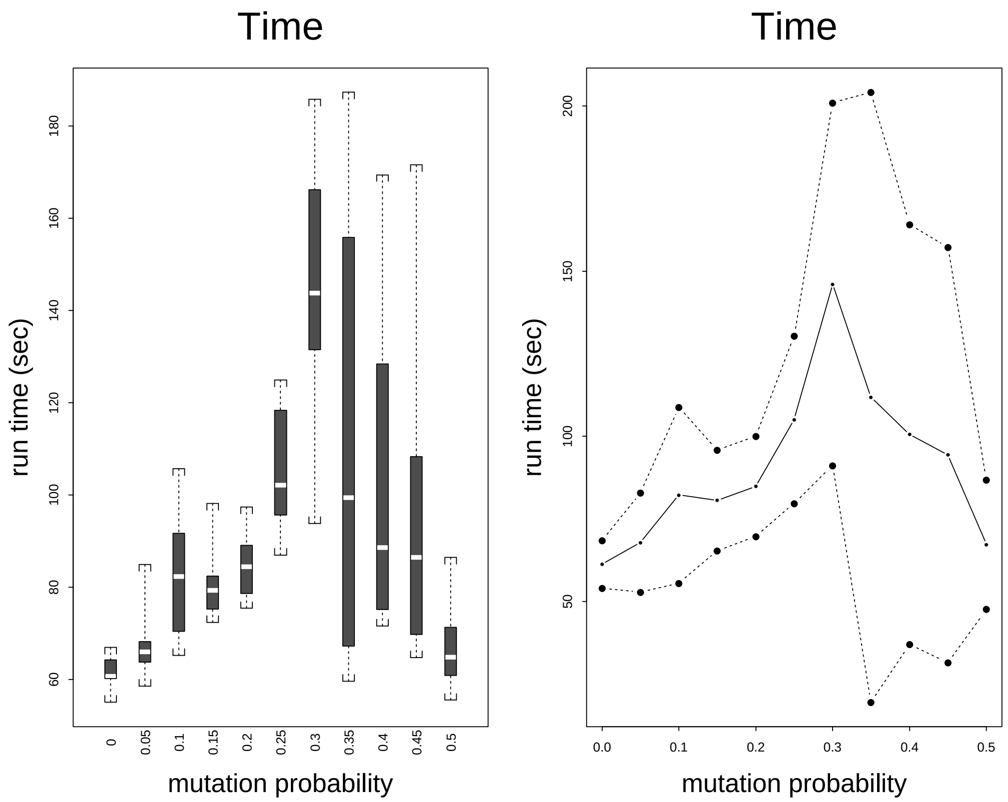

Figure 3.12. Effects of mutation probability on run time for the bays29 TSP. The diagrams show the variation in GA run time as the mutation probability changes. The left graph shows boxplots, for varying mutation probabilities, of the GA run times recorded from ten runs of the GA. The right graph shows means (solid lines) and 95% confidence bounds (dashed lines), for varying mutation probabilities, of the GA run times recorded from ten runs of the GA. Parameter settings: n = 300, p_c = 0.6, ϵ = 0.0005, and N = 50.

The effects on algorithm run time of varying the mutation probability are shown in figure 3.11. As p_m increases it seems reasonable to expect the algorithm to take longer to reach a convergent state, and to a certain extent this is reflected in the results, with a steady increase in run time up to the point when the phase shift occurs at about $p_m = 0.3$. After this point the run time fluctuates widely, which is perhaps to be expected of a process that is becoming more and more random.

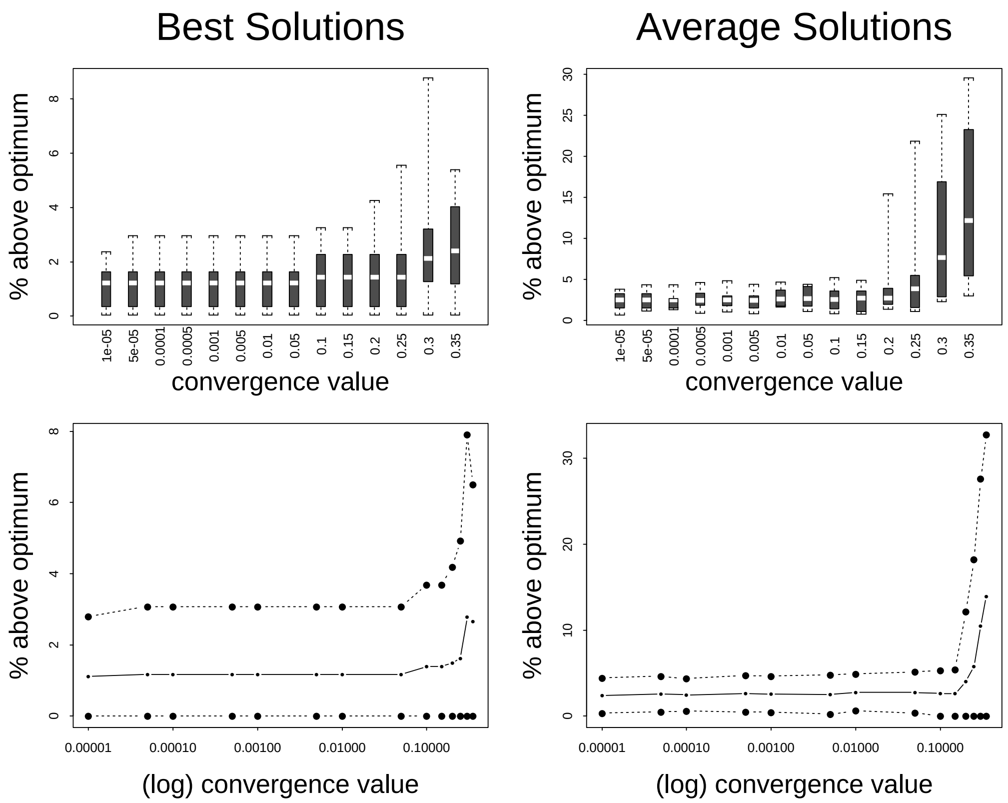

Figure 3.13. Effects of convergence level on solution quality for the bays29 TSP. The diagrams show the variation in GA performance as the convergence level changes. The top graphs show boxplots, for varying convergence levels, of the final best solutions and final average solutions recorded from ten runs of the GA. The bottom graphs show means (solid lines) and 95% confidence bounds (dashed lines), for varying convergence levels, of the final best solutions and final average solutions recorded from ten runs of the GA. Parameter settings: $n = 300$, $p_c = 0.6$, $p_m = 0.05$, and $N = 25$.

The effects on algorithm solution quality of varying the convergence level are shown in figure 3.13. There is little noticeable difference between results produced for $\epsilon < 0.15$. This has the implication that the level of change between successive sets of average population solutions can be quite large and good quality solutions are still produced. Forcing the algorithm to continue for longer until the sets are more similar does not have any great effect on solution quality. This indicates that the GA quickly finds good regions of the search space, and once a good region has been discovered little improvement is gained through searching the region more carefully, i.e. the algorithm is not good at 'fine-tuning' the search. A convergence set size of

$N = 25$ is used when investigating ϵ since for higher ϵ values the algorithm terminates quite quickly, and by construction the GA cannot terminate until $2N$ generations have expired. A lower value of N was required to ensure that the algorithm terminates due to the convergence criterion and is not artificially forced to continue until $2N$ generations have been evaluated.

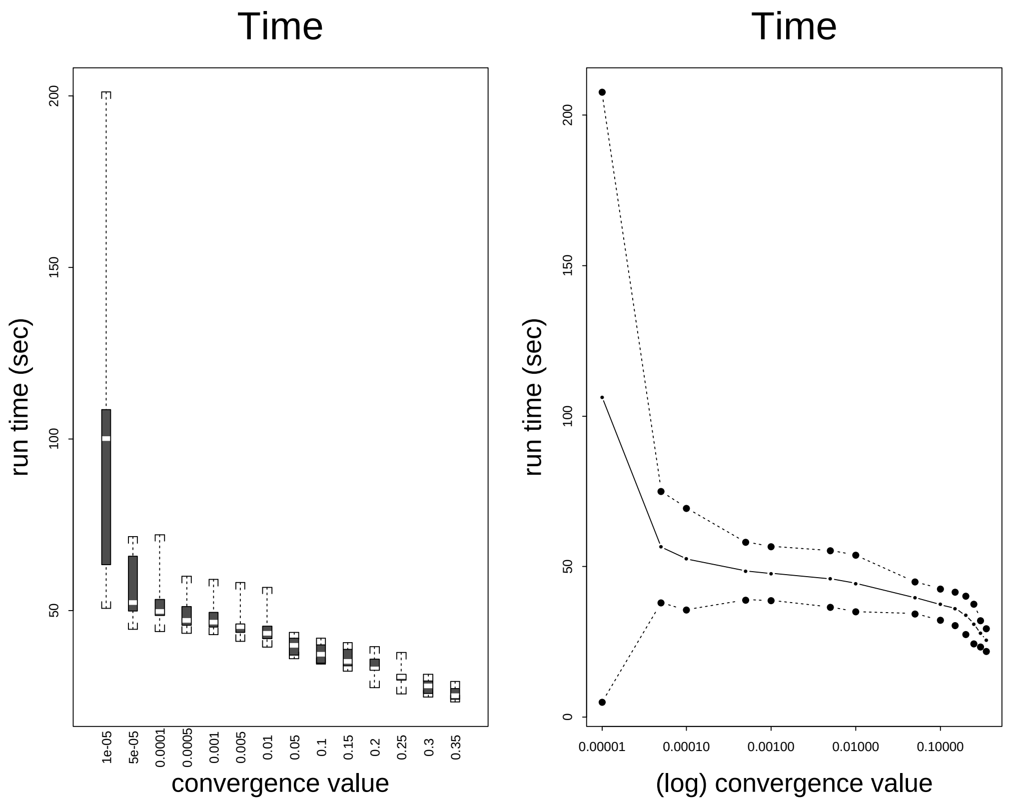

Figure 3.14. Effects of convergence level on run time for the bays29 TSP. The diagrams show the variation in GA run time as the convergence level changes. The left graph shows boxplots, for varying convergence levels, of the GA run times recorded from ten runs of the GA. The right graph shows means (solid lines) and 95% confidence bounds (dashed lines), for varying convergence levels, of the GA run times recorded from ten runs of the GA. Parameter settings: $n = 300$, $p_c = 0.6$, $p_m = 0.05$, and $N = 25$.

The effects on algorithm run time of varying the convergence level are shown in figure 3.14. The lower ϵ is, the longer the algorithm will take to reach the critical level of convergence since the GA is being forced to reach a more homogeneous state. From the figure it can be seen that as ϵ decreases the increase in run time does not occur rapidly which tends to back up a past comment that the algorithm, once in a good region of the search space, is not particularly good at fine-tuning the search. It does not take much time to reach a stricter level of convergence because better solutions are not being discovered.

The effects on algorithm solution quality of varying the convergence set size are shown in figure 3.15. The fact that increasing the set size has very

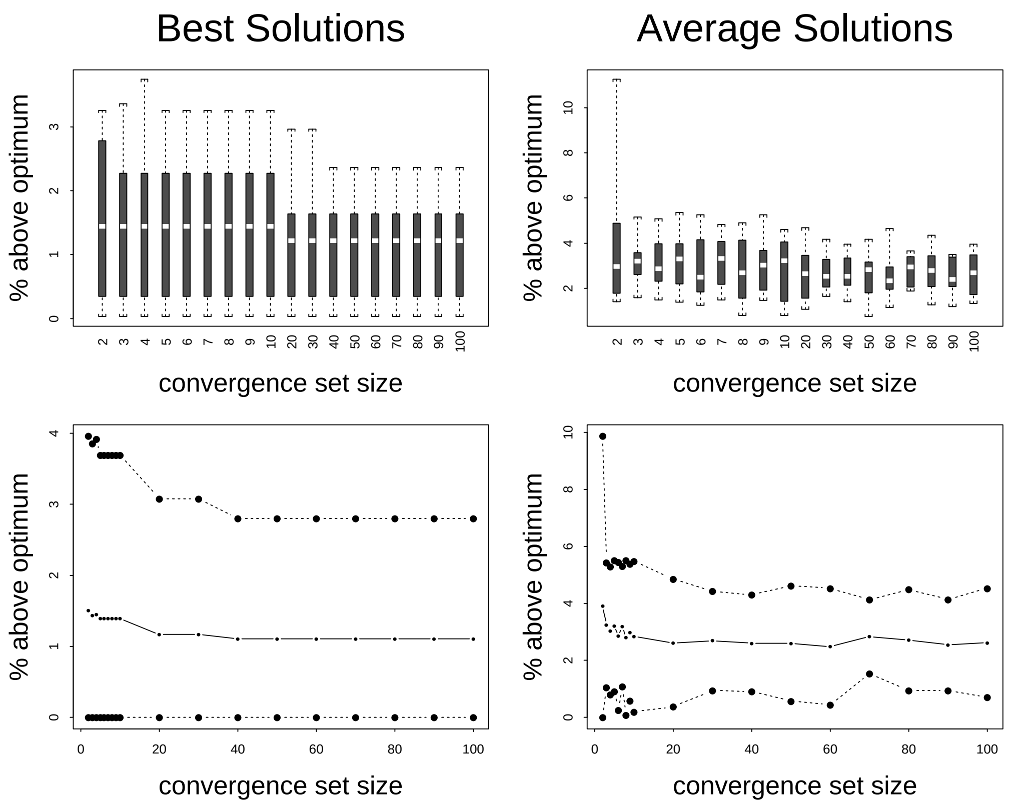

Figure 3.15. Effects of convergence set size on solution quality for the bays29 TSP. The diagrams show the variation in GA performance as the convergence set size changes. The top graphs show boxplots, for varying convergence set sizes, of the final best solutions and final average solutions recorded from ten runs of the GA. The bottom graphs show means (solid lines) and 95% confidence bounds (dashed lines), for varying convergence set sizes, of the final best solutions and final average solutions recorded from ten runs of the GA. Parameter settings: $n = 300$, $p_c = 0.6$, $p_m = 0.05$, and $\epsilon = 0.0005$.

little effect on solution quality would imply once more that the algorithm can reach a good region easily but is poor at fine-tuning the search. Since a large set size does not perform much better than a small set size the quality of solution found by the algorithm does not improve much beyound the stage where a 'good' solution has been found.

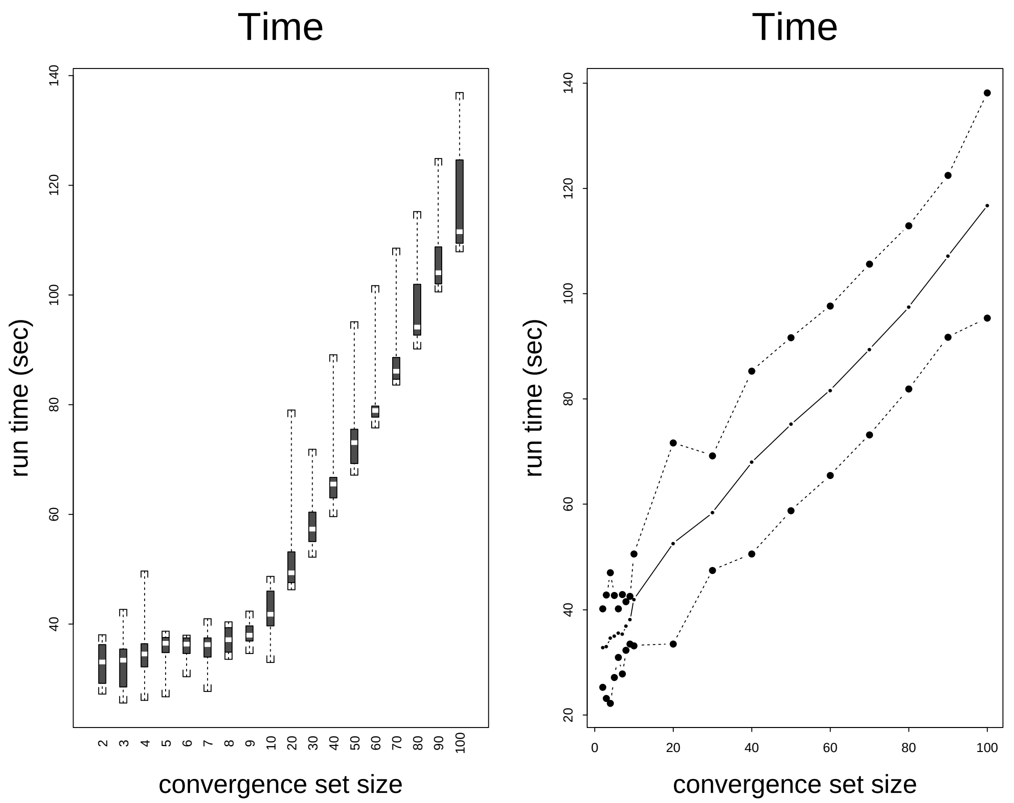

Figure 3.16. Effects of convergence set size on run time for the bays29 TSP. The diagrams show the variation in GA run time as the convergence set size changes. The left graph shows boxplots, for varying convergence set sizes, of the GA run times recorded from ten runs of the GA. The right graph shows means (solid lines) and 95% confidence bounds (dashed lines), for varying convergence set sizes, of the GA run times recorded from ten runs of the GA. Parameter settings: $n = 300$, $p_c = 0.6$, $p_m = 0.05$, $\epsilon = 0.0005$.

The effects on algorithm run time of varying the convergence set size are shown in figure 3.16. Increasing the set size forces the algorithm to terminate when a stricter level of convergence has been reached, leading to longer run times.

3.6.2 The eil51 TSP

Each of the GA parameters given above was varied, while keeping the other parameters fixed, for ten independent runs using the same ten sets of random seeds for each parameter setting. The default parameter settings used in all

GA runs of the eil51 problem were[12] $n = 500$, $p_c = 0.6$, $p_m = 0.05$, $\epsilon = 0.0005$, and $N = 10$.

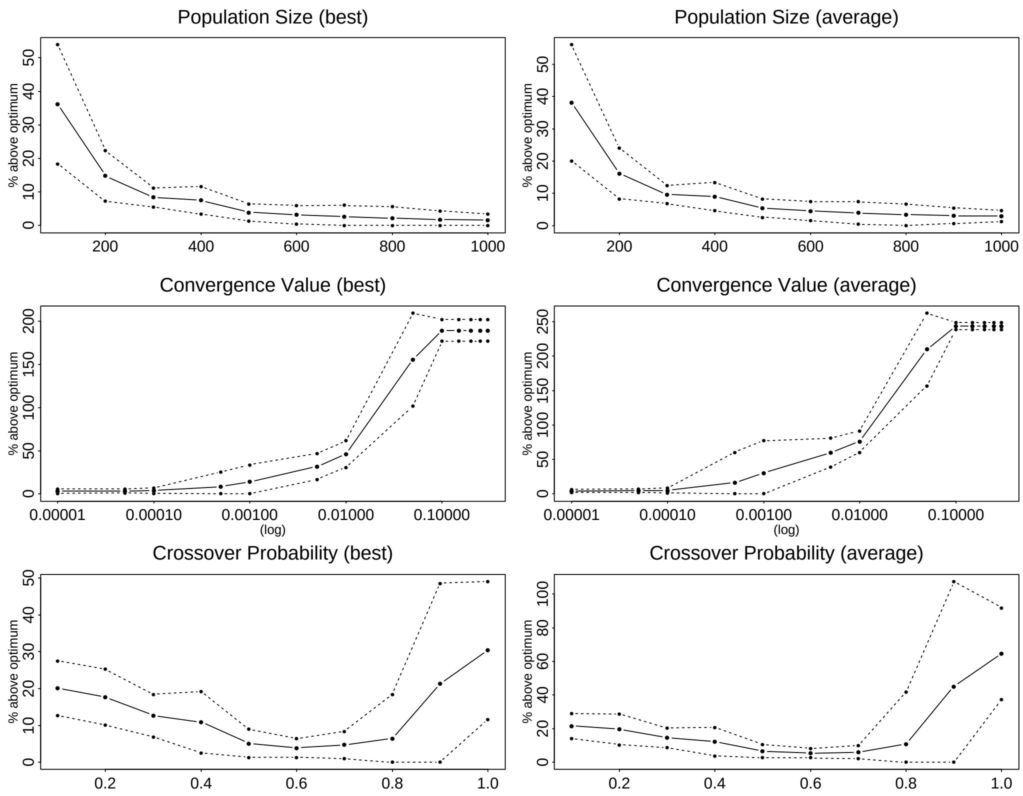

Figure 3.17. Parameter settings for the eil51 TSP. The graphs show means (solid lines) and 95% confidence bounds (dashed lines), for the parameters population size, convergence level, and crossover probability, of the best (left graphs) and average solutions (right graphs) recorded from ten runs of the GA. Default parameter settings: $n = 500$, $p_c = 0.6$, $p_m = 0.05$, $\epsilon = 0.0005$, $N = 10$.

The effects on solution quality of varying the GA parameters population size, convergence level, and crossover probability for the eil51 TSP are shown in figure 3.17. Similar behaviour to that observed for the bays29 problem is observed. Solution quality improves as the population size increases, with little difference in algorithm performance given by population sizes above $n = 500$. Below a convergence level of about $\epsilon = 0.0005$ little improvement is found while performance sharply deteriorates for higher convergence levels. Performance produced through varying the crossover probability is the same as for the bays29 problem, poor performance at low levels when exploitation dominates, poor performance at high levels when exploration is emphasised and good performance at the compromise level of $p_c = 0.6$.

The effects on solution quality of varying the GA parameters mutation probability and convergence set size for the eil51 TSP are shown in fig-

[12]The one exception to the convergence set size of $N = 10$ was when investigating the effect of ϵ, where $N = 2$ was used for reasons which will be explained later.

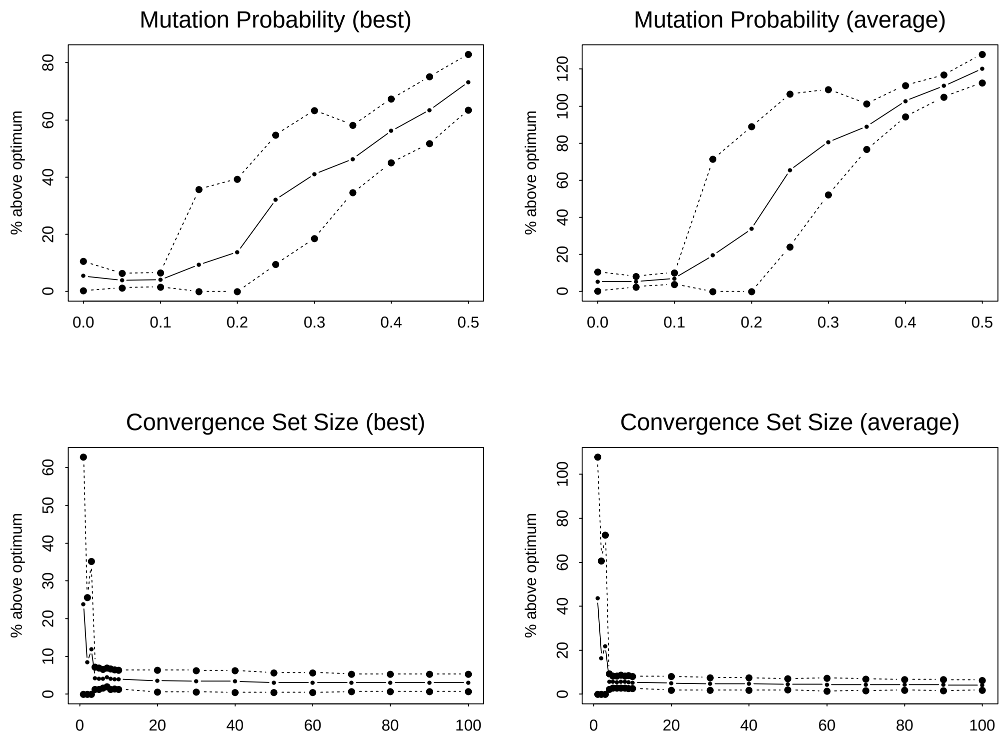

Figure 3.18. More parameter settings for the eil51 TSP. The graphs show means (solid lines) and 95% confidence bounds (dashed lines), for the parameters mutation probability and convergence set size, of the best (left graphs) and average solutions (right graphs) recorded from ten runs of the GA. Default parameter settings: $n = 500$, $p_c = 0.6$, $p_m = 0.05$, $\epsilon = 0.0005$, $N = 10$.

ure 3.18. As previously noted, for high levels of mutation the algorithm degenerates into a random search giving poor quality results. An 'optimal' level of mutation appears to exist at $p_m = 0.05$. Results gained from convergence set sizes over $N = 10$ are very similar. Solution quality is consistently good until N drops below 5 or so when quality very quickly becomes poor.

All these results tend to support the conclusions of the previous problem that GAs are excellent at identifying regions of a general good quality, while poor at fine-tuning the search to identify specific near-optimal solutions.

Results for GA run time for the eil51 TSP are not given here since they display trends similar to those observed for the bays29 TSP.

3.6.3 The kroC100 TSP

Each of the GA parameters given above was varied, while keeping the other parameters fixed, for five independent runs using the same five sets of random seeds for each parameter setting. The default parameter settings used in all GA runs of the kroC100 problem were $n = 1500$, $p_c = 0.6$, $p_m = 0.05$, $\epsilon = 0.000001$, and $N = 10$.

Due to time restrictions it was only feasible to perform a linear search of the parameter space using 5 runs of the GA for each of the parameter settings. While this is somewhat unsatisfactory when attempting to make conclusions about trends in algorithm solution quality and run time it is hoped that 5 repetitions is enough to gain some insight into the general performance of the algorithm.

Figure 3.19 and figure 3.20 show the effects on GA solution quality of varying the parameters. The performance of the algorithm again largely mirrors that of both the bays29 and the eil51 TSPs. A larger population size gives better quality solutions, a crossover probability of $p_c = 0.6$ appears to represent a good compromise between exploration and exploitation, a low mutation probability at about $p_m = 0.05$ gives enough randomisation to provide diversification in the search without making the algorithm into a random search, and a convergence level below a certain limit yields little improvement in algorithm performance.

3.6.4 Parameter Selection

Empirically the 'optimal' levels of mutation probability and crossover probability are largely unaffected by an increase in problem size. A level of $p_c = 0.6$ appears to represent a good compromise between exploration and exploitation, never allowing the algorithm to concentrate too much on one aspect to the detriment of the other. Both aspects are necessary when searching for optimality in a large search space, and different algorithmic techniques have different ways of combining the two. As has been mentioned earlier in this chapter, GAs have been shown to optimally combine these two features simultaneously.

Which of the mutation or the crossover operators is more important to the GA paradigm is currently the subject of much debate. Some level

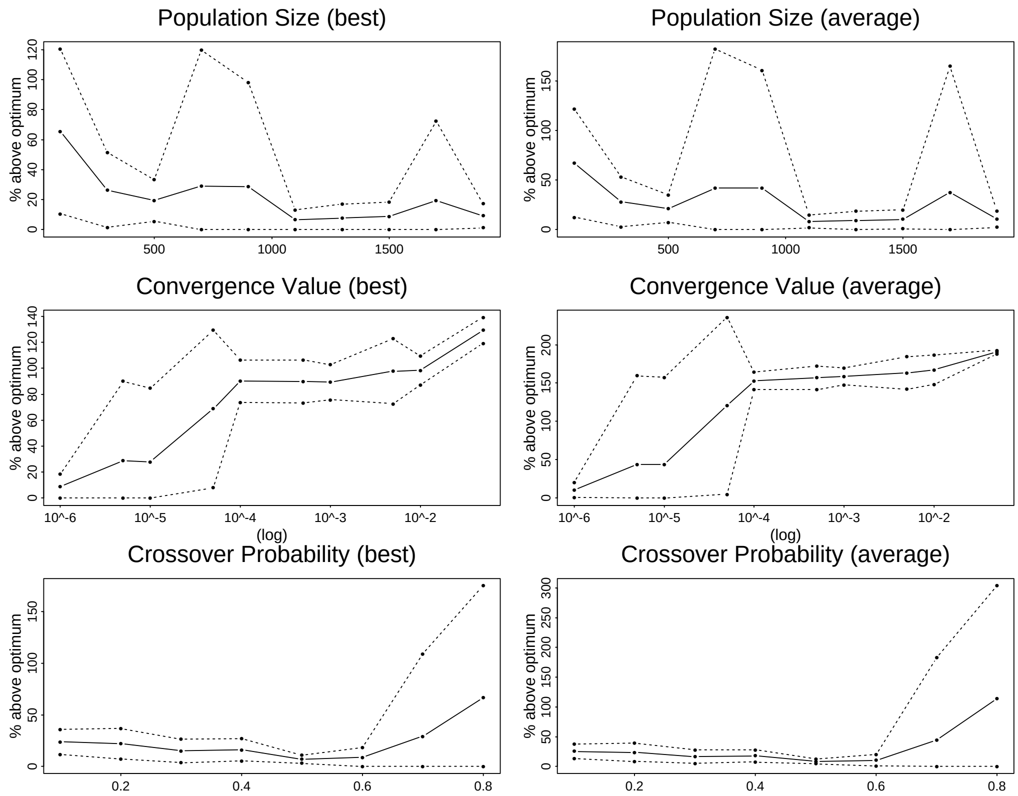

Figure 3.19. Parameter settings for the kroC100 TSP. The graphs show means (solid lines) and 95% confidence bounds (dashed lines), for the parameters population size, convergence level, and crossover probability, of the best (left graphs) and average solutions (right graphs) recorded from ten runs of the GA. Default parameter settings: $n = 1500$, $p_c = 0.6$, $p_m = 0.05$, $\epsilon = 0.000001$, $N = 10$.

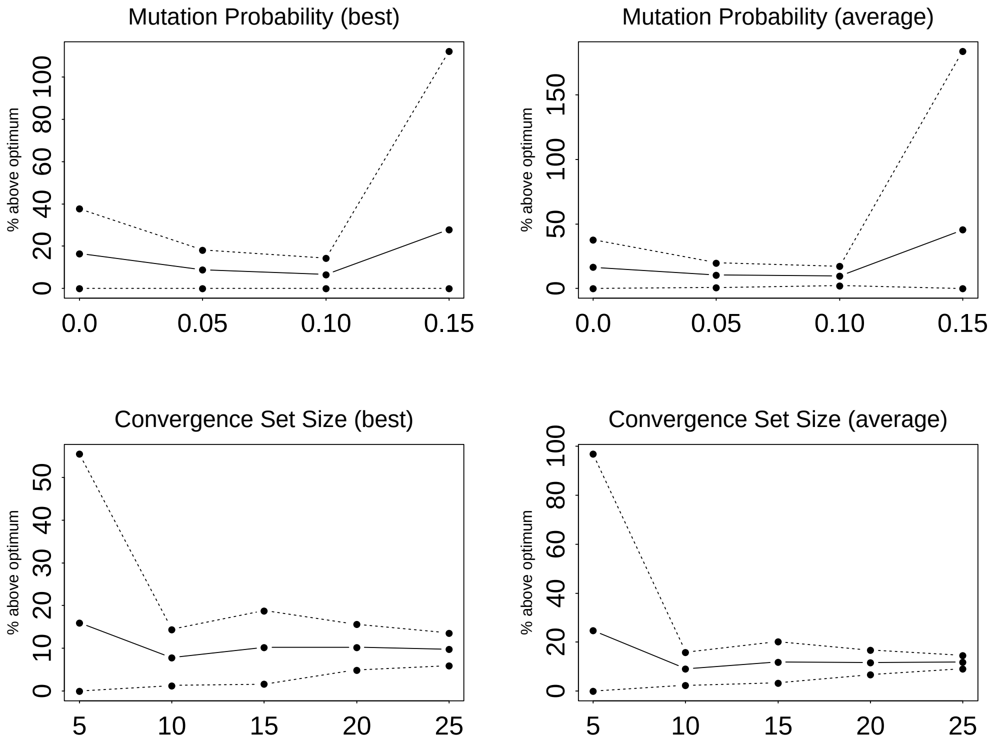

Figure 3.20. More parameter settings for the kroC100 TSP. The graphs show means (solid lines) and 95% confidence bounds (dashed lines), for the parameters mutation probability and convergence set size, of the best (left graphs) and average solutions (right graphs) recorded from ten runs of the GA. Default parameter settings: $n = 1500, p_c = 0.6, p_m = 0.05, \epsilon = 0.000001, N = 10$.

of mutation is theoretically required for convergence to global optimality, and it would appear in practice a small amount of mutation, at a level of $p_m = 0.05$, helps the algorithm to maintain enough diversification in the search to reach solutions that crossover alone would not have found.

The size of the 'optimal' population is affected by the dimension of the problem. This seems reasonable since at each generation the population represents a biased sample of the solution space. The solution space grows rapidly as the problem dimension increases and and maintaining a population that is capable of sustaining a level of diversity sufficiently indicative of the solution space (or regions of it) to guide the search to good quality solutions necessitates a larger population size for a larger problem.

Although the same general behaviour is observed for all three problems when determining an 'optimal' level of convergence, the level at which the 'phase-shift' occurs is not the same in all cases. The required level of ϵ needed to maintain 'good' results progressive becomes smaller as problem dimension increases. For the bays29 TSP $\epsilon = 0.1$ is sufficient, for the eil51 TSP $\epsilon = 0.0001$ is sufficient, and for the kroC100 TSP $\epsilon = 0.000001$ is sufficient. A steady decrease in ϵ is to be expected under the current termination procedure. For the bays29 TSP with a population size of 300 the fraction of the solution space represented by the population is $\frac{2\times300}{28!}$, for the eil51 TSP with a population size of 500 this fraction has reduced to $\frac{2\times500}{50!}$, and the fraction is further reduced for the kroC100 TSP with a population size of 1500 to $\frac{2\times1500}{99!}$. As the fraction of the solution space that is represented by the population reduces it seems sensible that a stricter level of convergence within the population is required if solutions of a comparable quality are to be produced.

The size of the convergence set required to produce good solutions does not seem a particularly important consideration as long as the level is greater than $N = 10$.

From the empirical analysis of GA parameter selection two conclusions are apparent. The first is that GAs are not good at precisely fine-tuning a search to find a particular solution in a good region. This is supported by the lack of improvement in the GA below a certain convergence level and by the indifference of the algorithm to the size of the convergence set. Secondly, the convergence procedure, while sufficient for termination in the problems presented, is fraught with potential difficulty. As mentioned above, larger problems require smaller levels of ϵ, meaning successive convergence sets must be *very* similar at high dimensions for the algorithm to terminate with good quality solutions. Since the GA paradigm requires a certain level of mutation, the random element inherent in the algorithm may prevent convergence completely if ϵ is *too* small and the level of mutation is *too* high. It was hoped that using a convergence set of reasonable size would dampen the natural variation enough for convergence to occur. This does not always work, since for the kroC100 problem no mutation level of above $p_m = 0.15$ was observed to converge. It seems a termination technique such as that discussed in section 3.3, allowing for the GA's natural variation, is required.

Chapter 4

Threshold Accepting Algorithms

4.1 Threshold Accepting

Threshold accepting (TA) is a generally applicable heuristic algorithm developed by Dueck and Scheuer [10], that can be used to solve combinatorial optimisation problems. TA is an iterative improvement method formally very similar to simulated annealing (SA), discussed at length in section 2.8. Given an arbitrary solution to the problem both methods involve the selection of a neighbouring solution. This neighbourhood configuration replaces the existing configuration if a certain criterion is achieved and it is at this level that the essential difference between the two algorithms becomes apparent.

In SA a new solution is accepted automatically if it is better (according to the objective function) than the current configuration, if it is worse then acceptance is governed by a probability. The probability is dependent on a control parameter called *temperature* that is slowly decreased (cooled) as the algorithm proceeds. Cooling has the effect that the probability of a worse solution being accepted is reduced as time increases.

TA accepts a new solution if it is better than or 'not much worse' than the the existing configuration, formalised in the algorithm by a *threshold* parameter. In the case of minimisation, any new solution less than the current configuration quality plus the threshold is accepted as the new current solution. No stochastic test is required in the solution replacement procedure, acceptance is totally deterministic, although selection of a specific neighbourhood solution is stochastic. The greater simplicity of the algorithm leads to a claim of superiority of TA over SA made by Dueck and Scheuer [10]. The threshold is decreased with time so that fewer worse solutions are accepted as the algorithm proceeds. Given a problem structure with cost function $f(x)$ and neighbourhood structure $N(x, \sigma)$ for a given configuration x and a neighbourhood generation operator σ, the pseudo-algorithm for TA in the case of minimisation is given in figure 4.1.

begin
choose an initial configuration i
choose an initial threshold $T > 0$
repeat
choose a neighbour $j \in N(i, \sigma)$
$\delta := f(j) - f(i)$
if $\delta < T$ **then**
$i := j$
end if
if quality has not improved for some time **or**
too many iterations **then**
lower threshold T
end if
until quality has not improved for a long time
end

Figure 4.1. Pseudo-algorithm for the Threshold Accepting method. The algorithm repeatedly searches the neighbourhood of the current solution for a configuration to become the new current solution. All solutions that are of a quality 'not much worse' than that of the current configuration are accepted, regulated by the threshold parameter T.

4.2 The Great Deluge Algorithm and the Record-to-Record Travel Algorithm

The TA principle can be simplified further resulting in two new generally applicable heuristics introduced by Dueck [11], the Great Deluge algorithm (GDA) and the Record-to-Record Travel algorithm (RTRT). These two algorithms differ from TA in the decision rule by which a neighbourhood candidate is accepted to replace the current solution.

It is easy to visualise the GDA, especially for maximisation. The quality of all solutions in the sample space together define a surface or a *landscape* of which the maximum or highest point is sought. By letting it 'rain' continuously, the amount of water covering the landscape, the *water level*, increases. The algorithm 'walks' around the fictitious landscape ensuring the solution never falls below this rising water level. At some point the algorithm will become trapped on a 'peak' surround by water and the hope is that the algorithm has then found a peak that is nearly optimal. This analogy is formalised by the pseudo-algorithm given in figure 4.2, showing the minimisation variant of the GDA, with cost function $f(x)$ and neighbourhood structure $N(x, \sigma)$ for a solution x.

Record-to-Record Travel is very similar to the GDA, the difference again lies in the acceptance rule. The RTRT algorithm accepts any new solution that has a quality not much worse than the best solution (or *record*) found so far. The pseudo-algorithm given in figure 4.3 shows this method implemented for minimisation, with the same problem structure as given for the GDA.

```
begin
    choose an initial configuration i
    choose an initial rain speed UP > 0
    choose an initial water level LEVEL > 0
    repeat
        choose a neighbour j ∈ N(i, σ)
        if f(j) < LEVEL then
            i := j
            LEVEL := LEVEL − UP
        end if
    until quality has not improved for a long time or
        too many iterations
end
```

Figure 4.2. Pseudo-algorithm for the Great Deluge method. The algorithm repeatedly searches the neighbourhood of the current solution for a configuration to become the new current solution. All solutions in the neighbourhood that are 'under water' are accepted, regulated by the water level $LEVEL$.

```
begin
    choose an initial configuration i
    choose an allowed deviation DEV > 0
    set RECORD := f(i)
    repeat
        choose a neighbour j ∈ N(i, σ)
        if f(j) < RECORD − DEV then
            i := j
        end if
        if f(j) < RECORD then
            RECORD := f(j)
        end if
    until quality has not improved for a long time or
        too many iterations
end
```

Figure 4.3. Pseudo-algorithm for the Record-to-Record Travel method. The algorithm repeatedly searches the neighbourhood of the current solution for a configuration to become the new current solution. All solutions in the neighbourhood that are of a quality 'not much worse' than the current best solution are accepted, dependent on the allowed deviation DEV.

4.3 Parameter Selection

The claim is made that the GDA and the RTRT algorithm depend only on a single parameter, for the GDA this is the rain speed and for the RTRT algorithm it is the allowed deviation. Selecting a small deviation and large rain speed, for the two algorithms respectively, poor quality solutions are quickly produced. Alternatively, by choosing a large deviation and a small rain speed good quality solutions are slowly generated. The job then lies in finding appropriate values for these parameters, balancing the two goals of speed and quality of solution. The exact balance will depend on the specific problem under consideration, though generally it is hoped the algorithm will find good solutions in a 'reasonable' amount of time.

Dueck [11] makes several suggestions concerning the selection of appropriate parameter values. For the GDA the rain speed UP is proposed to be on average somewhat smaller than 1% of the average distance between the quality of the current solution and water level. This water level update is achieved by replacing the current $LEVEL$ update procedure given in figure 4.2 with:

if $f(j) < LEVEL$ **then**
  $i = j$
  $LEVEL = LEVEL - UP(LEVEL - f(j))$
end if

where UP now represents a fraction. In the RTRT algorithm the allowed deviation DEV is taken to be a fixed fraction of the quality of the current best solution (i.e. a fraction of $RECORD$), although a further suggestion is made that the size of the fraction should be reduced as the algorithm proceeds. This first proposal is achieved by replacing the $RECORD$ update procedure in figure 4.3 with:

if $f(j) < RECORD$ **then**
  $RECORD = f(j)$
  $DEV = RECORD \times FRACTION$
end if

4.4 TA-An Intuitive Approach?

After some consideration of the three TA algorithms above, it does not seem obvious why they should produce good quality solutions to a problem. Also the claims of the authors need to be addressed. Firstly—are the TA algorithms superior to the SA approach?

Consider the following situation. Assume in minimising a problem P there is a current solution to P, x. Associated with x is a measure of quality $f(x)$ and a neighbourhood structure $N(x, \sigma)$. Randomly generate a neighbouring solution $y \in N(x, \sigma)$ as a candidate to replace x as the present solution. Two cases result, either $f(y) \leq f(x)$ or $f(y) > f(x)$. The SA and TA algorithms differ in the manner in which the second situation is handled, i.e. in the acceptance of a configuration worse than the current one.

- Simulated annealing handles the acceptance of a worse solution with a stochastic test dependent on the temperature parameter.

  $f(y) \leq f(x)$: Accept y as the new solution with probability 1.

  $f(y) > f(x)$: Accept y as the new solution with probability $\exp(\frac{\triangle E}{T})$, where $\triangle E$ is the difference in quality between the two configurations and T is the temperature.

- Threshold accepting methods accept an inferior solution if it is "not much worse" than the current configuration.

  $f(y) \leq f(x)$: Accept y as the new solution with probability 1.

  $f(y) > f(x)$: Accept y as the new solution if $f(y) - f(x) \leq T$ where T is the threshold. Thus a general neighbourhood solution y is accepted with a probability $\frac{|m(x)|}{|M(x)|}$ where
  $M(x) = \{i|f(i) > f(x), i \in N(x,\sigma)\}$ and
  $m(x) = \{i|f(i) - f(x) \leq T, i \in N(x,\sigma)\}$.

In either case a worse neighbouring solution is accepted with a given probability, $\exp(\frac{\triangle E}{T})$ for SA and $\frac{|m(x)|}{|M(x)|}$ for TA. Both algorithms have a probabilistic acceptance of an inferior solution, it is just that in the case of TA the stochastic element of the test to accept a candidate is 'hidden' in the neighbourhood generation process. The claim then that TA methods are deterministic in their acceptance rules is debatable. It seems from this point of view that the TA algorithms are not a new concept in heuristic algorithms but are a variation on the annealing theme, an iterative improvement algorithm that has a probabilistic acceptance of inferior solutions. Nevertheless the TA algorithms have only one stochastic process at work as opposed to the two processes for standard SA, resulting in a simpler algorithm.

The second claim that Dueck [11] makes concerning the GDA and the RTRT algorithm is that the heuristics depend only on a single parameter. Looking at the algorithms presented in figure 4.2 and figure 4.3 it seems conceivable that other parameters might have an effect on both the solution quality and the run time. For the GDA the following should be considered:

- The initial configuration. In itself the starting solution shouldn't affect the algorithm outcome greatly, since the purpose of the heuristic is local optima can be escaped by accepting inferior solutions in a controlled manner. However in combination with some of the other parameters the initial configuration could present a problem.

- The initial water level. Dueck [11] suggests that the initial water level should be the quality of the initial solution. But if the initial solution happens to be a local optimum or near local optimum then the algorithm will never escape to find a better solution. Consider figure 4.4.

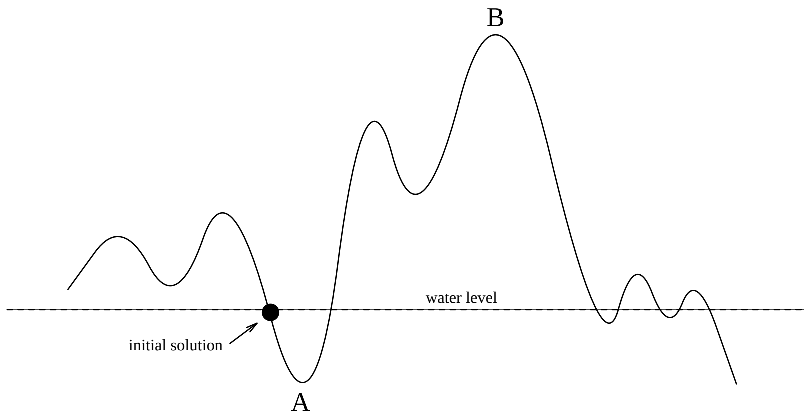

Figure 4.4. A GDA example. Consider finding the maximum of this solution landscape using the GDA. By setting the initial water level to be the cost of the start configuration it is not possible to reach the maximum value at B, since to do so the algorithm must pass through A which is currently 'under water'. As the water level will only ever increase, B can never be reached.

- The rain speed and the way in which the rain speed can change during a run of the algorithm. The rain speed is the obvious parameter that determines both the speed and the quality of the solution. If it is large the water level will increase rapidly trapping the algorithm in a local optimum giving a poor result. A small rain speed allows the algorithm more time to investigate the solution space and gives greater opportunity to escape local optima, producing better quality solutions.

  It does not seem obvious why a slow rise in the water level should prevent the algorithm from becoming trapped in a local minimum. Also the manner in which the water level increases needs consideration. Is a constant rain speed (perhaps linked to the problem size) inherently worse than a rain speed linked to the distance from the water level? If so why?

- The stopping conditions. The GDA has two specified stopping conditions, either no increase in solution quality for some time or too many iterations have occurred. Specifying a maximum number of iterations seems to be a failsafe condition, ensuring that the algorithm will finish in a reasonable amount of time. If no increase in quality has occurred for some time the algorithm is trapped by the rising water level in a local optimum, so no further improvements can be made, it is just a question of how much of the neighbourhood to search before admitting defeat. Both these parameters are dependent on the problem dimension and will affect solution quality if set too low. Whether there is any interaction of the stopping conditions with other parameters (such as the rain speed) remains to be seen.

The RTRT algorithm has similar considerations for its parameters. The concerns for the initial configuration and the stopping conditions are similar to those for the GDA. Other parameters to consider in the RTRT approach are:

- The initial record. The initial record is set to be the quality of the initial solution which is correct since the first solution selected is also the best one found so far. But if the initial solution is already a good solution what problems does this present, especially in combination with the allowed deviation?

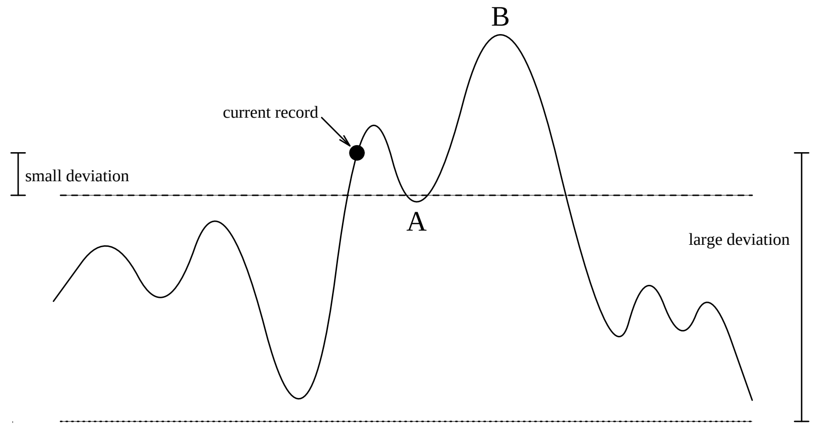

Figure 4.5. A RTRT example. Consider finding the maximum of this solution landscape using the RTRT algorithm. If a small deviation is allowed the algorithm can never reach the maximum at B since it cannot go past the minimum at A. Alternatively, by having a large deviation the algorithm randomly searches the entire solution space.

- The allowed deviation. Dueck [11] makes the claim that a small deviation rapidly gives results of low quality which is reasonable since algorithm will not be able to escape from some local optimum. Large deviations are touted to give better quality solutions at the expense of run time. If the deviation is large enough the algorithm is effectively performing a random search over the entire solution space, the algorithm does not seem to converge in any sense to the global optimum. Figure 4.5 demonstrates these extremes. The suggestion of Dueck [11] to have a deviation that decreases with time makes sense, as this will *diversify* the search at the beginning of the algorithm, broadly searching the solution space, and *intensify* the search at the end, refining the process to produce a locally optimal solution.

4.5 Empirical Analysis

To investigate which parameters have the greatest effect on the solution quality and time taken by the GDA and the RTRT algorithm, a series of empirical experiments have been carried out. These experiments use the Traveling Salesman Problem (TSP) to analyse how the “optimal” parameters behave, especially as the problem size (in this case the number of cities) increases.

A full description of the TSP is given in chapter 2. The performance of these two TA algorithms is considered with respect to three Euclidean TSP problems of sizes 51, 100 and 150, henceforth referred to as eil51, kroC100, and kroA150 respectively. Descriptions of these three TSP problems can be found in appendix B. In each case the best cyclic tour of minimum length joining all cities is sought. To achieve this goal both GDA and the RTRT algorithm begin with a random tour and by use the 2-change operation, described in appendix A, to perturb the current tour to produce a candidate for selection to replace the present solution. The acceptance rules for a particular perturbation for either algorithm are described in section 4.2. Where possible, tour lengths are expressed in terms of a percentage above the optimum tour[1] and times are given in seconds of CPU time[2].

For the GDA the parameter UP represents a fraction as described in section 4.3, and $NSize$ is related to the first stopping condition, if no new solution is found to replace the current configuration in $NSize$ iterations the algorithm terminates. $NSize$ determines how many times the neighbourhood is sampled before the decision is made that no better solution will be found. The maximum number of iterations is set arbitrarily large in all runs so as not to interfere with the process proper, allowing the algorithm to converge to a local optimum rather than terminating prematurely.

Since the RTRT algorithm is a threshold accepting algorithm similar to the GDA algorithm with a "water-level" in the RTRT algorithm given by $RECORD - DEV$, similar results to those found by the GDA are expected. For this reason the following empirical study will look only at the GDA.

4.5.1 GDA Results

eil51

Ten runs of the eil51 problem were performed for fractional UP values of 0.005, 0.01, 0.05, 0.1, 0.5, and 1 and for $NSize$ values of 250, 500, 750, 1000, 1250, and 1500. The maximum number of iterations was set at 1000000, and none of the runs came close to this limit, allowing all runs of the algorithm to terminate because of some form of convergence rather than to this more arbitrary stopping condition. The GDA was run for each set of parameter values on the same ten *seeds*, used in the *random number generator*[3], so any conclusions made about the GDA are more likely due to different parameter settings rather than to different random number streams.

[1]Where the optimum tour is not known the best possible tour found is used in its place.

[2]Details of the machine used for the analysis and the function used to record CPU time are found in appendix C.

[3]A random number generator is a structure provided by most programming languages to produce a sequence of randomly ordered bits, the exact sequence determined by the seed provided to the generator. Random bits can be interpreted as numbers. A random number distribution uses the bits produced by the generator to give numbers from a given distribution, discrete uniform numbers for example.

Results from the runs are shown in table 4.1 and table 4.2, with graphical interpretations of the results given in figure 4.7 and figure 4.6.

UP	NSize	250	500	750	1000	1250	1500
0.005	mean	60.5	113	154	169	200	225
	var	144	316	216	440	827	577
0.01	mean	36.2	64.3	80.2	98.5	104	128
	var	23	111	184	379	532	346
0.05	mean	10.1	16.8	22.3	25.9	30.6	34.8
	var	8.06	20.5	18.9	30.2	43	47.7
0.1	mean	6.08	10.1	13.2	15.7	19.3	19.2
	var	0.82	4.73	9.97	12.9	60.3	12.6
0.5	mean	2.25	3.24	4.73	5.04	5.69	6.29
	var	0.31	0.336	4.72	1.48	2.61	2.74
1	mean	1.34	2.04	2.7	2.86	3.39	3.74
	var	0.0868	0.146	0.11	0.122	0.81	0.841

Table 4.1. Running times for the GDA on the eil51 TSP. Ten runs where made for each parameter setting and the run times, measured in seconds of CPU time were recorded with the means and variances displayed above

UP	NSize	250	500	750	1000	1250	1500
0.005	mean	21.91	9.49	5.74	5.06	3.70	2.72
	var	23.20	4.14	1.72	1.96	0.42	0.90
0.01	mean	18.13	8.51	6.28	4.18	3.90	2.36
	var	17.15	5.65	5.51	6.73	7.60	1.63
0.05	mean	15.32	6.57	4.17	3.37	2.75	2.22
	var	87.83	8.07	3.95	3.31	2.10	1.34
0.1	mean	12.07	6.99	5.09	4.70	4.09	3.72
	var	7.58	3.91	3.17	2.92	2.80	1.83
0.5	mean	11.00	7.53	6.29	5.64	5.47	5.24
	var	19.02	12.81	7.61	8.43	8.65	8.42
1	mean	19.21	12.16	9.18	9.18	8.40	7.89
	var	68.24	36.78	29.84	29.84	24.57	14.86

Table 4.2. GDA solution quality for the eil51 TSP. Ten runs where made for each parameter setting and the best tour lengths found, measured as percentages above the optimal tour, were recorded with means and variances displayed above.

Figure 4.6 displays the run times of the GDA algorithm. For a given UP value, as log($NSize$) increases, there is a linear relationship with log($time$) such that log($time$) increases as log($NSize$) increases. Further more there is a linear relationship for a given $NSize$ value between log($time$) and log(UP), log($time$) decreases in a linear manner as log(UP) increases.

Figure 4.7 shows the trends in tour lengths for varying $NSize$ and UP values using cost measured as a fraction above the optimal tour length. For a given $NSize$ value it is expected that smaller UP parameters will yield

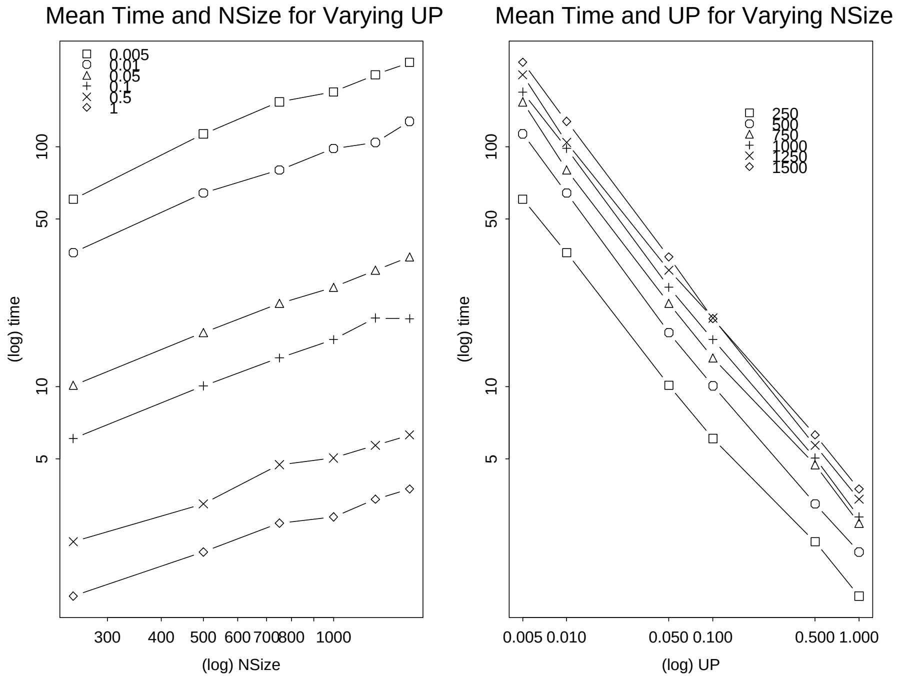

Figure 4.6. Graphical representation of GDA run times on the eil51 problem. See table 4.1 for greater detail.

better results. Contrary to expectations, for all values of $NSize$ shown, smaller parameters do not produce better results even though the running times for such parameters are higher. However relative rankings of which parameters do better appear to change as $NSize$ increases, smaller UP values improving more than larger values. If this trend continues then by increasing the $NSize$ value sufficiently smaller UP values should be observed to give the best results. Figure 4.8 shows mean times and costs for 10 repetitions of the eil51 TSP for $NSize$ taking values up to 10000.

It would seem that better results are produced for smaller UP values only if $NSize$ is sufficiently large. It is also clear from figure 4.8 that increasing $NSize$ beyound a certain size is of little use since this will only produce the same results in more time. By sampling this neighbourhood with replacement the algorithm is more likely to repeat the selection of a given candidate if the sampling continues for too long, thus producing the same result as sampling for a shorter interval. The ideal value of $NSize$ to use intuitively should be linked to the dimension of the problem, for a 51-city TSP the neighbourhood size, using the 2-change operation[4], is $51 \times 48 = 2448$.

The manner in which time increases as $NSize$ increases and UP decreases retains its linear nature, when taking log values, for larger $NSize$ values. However this linear relationship begins to fail for $NSize$ values close to 10000. Tour lengths being produced by the algorithm for $NSize$

[4]See appendix A for description of neighbourhoods using the 2-change operator.

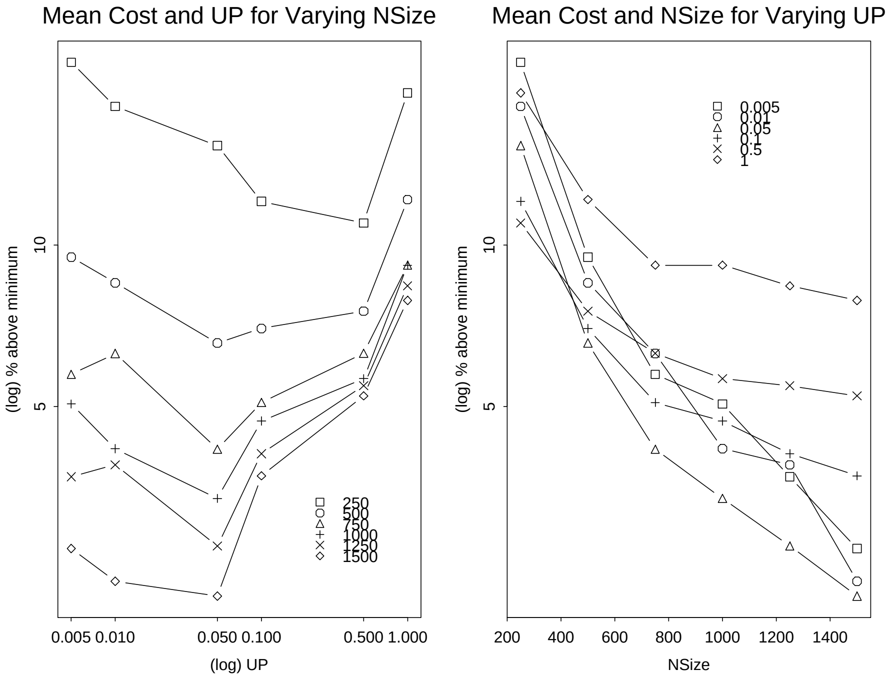

Figure 4.7. Graphical representation of GDA tour lengths on the eil51 problem. See table 4.2 for greater detail.

beyond a certain limit are unchanged and the algorithm is forced to continue unnecessarily searching the neighbourhood for a new solution when it has already sampled a good proportion of the neighbourhood without success. The conclusion can be made that these time values are not likely to increase in a log linear manner since they are not reflective of the algorithm proper but instead an artificial and unnecessary extension of a prolonged neighbourhood search.

Exploiting the linear relationships shown in figure 4.6 an expression to predict run times for the GDA can be formulated. The following expressions are clear from the graphs:

$$\begin{aligned} \log(time) &= A\log(NSize) + B \; where \; B = f(UP) \\ \Rightarrow time &= 10^{f(UP)} NSize^{A} \end{aligned}$$

$$\begin{aligned} \log(time) &= C\log(NSize) + D \; where \; C = g(NSize) \\ \Rightarrow time &= 10^{g(NSize)} UP^{C} \end{aligned}$$

The slopes A and C, which can be measured from the graphs, are constant for varying UP and $NSize$ respectively. The y-intercepts, given by the functions $f(UP)$ and $g(NSize)$, are dependent on parameter values, but by equating the two above equations the functions can be approximated by $f(UP) = UP^{C}$ and $g(NSize) = NSize^{A}$. If true, the y-intercepts should

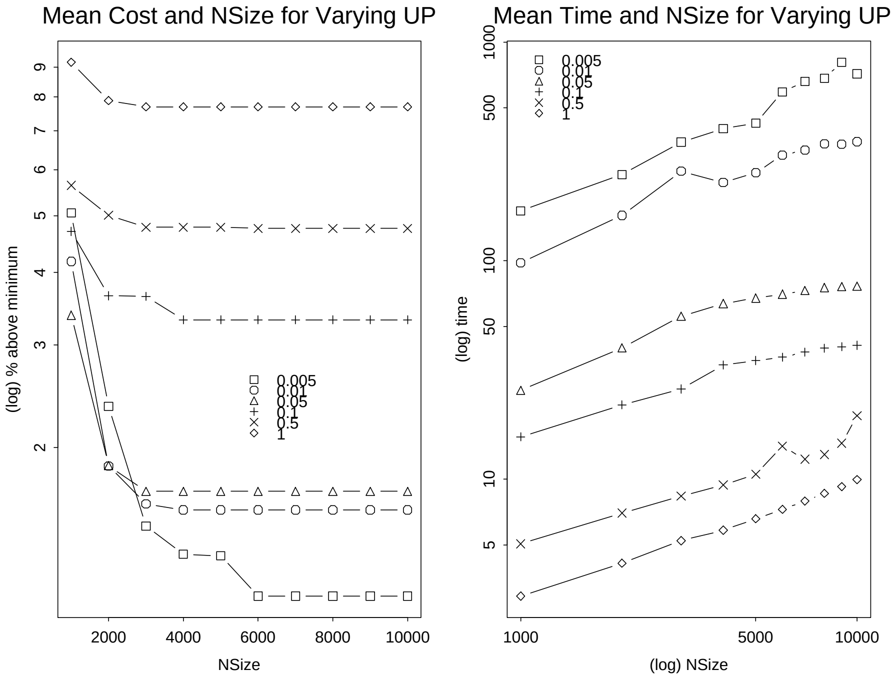

Figure 4.8. GDA results for the eil51 problem for large $NSize$. Tour lengths and run times are shown for $NSize$ values up to 10000.

be UP^C and $NSize^A$ for each equation respectively. Even though the y-intercepts have no real meaning, since $\log(0)$ is undefined, for purposes of the model a further constant term is required to match the projected y-intercept with either the proposed $f(UP)$ or $g(NSize)$. This extra constant is (approximately) equal for varying $NSize$ or UP enabling a model of the run time to be derived.

$$
\begin{aligned}
\log(time) &= A\log(NSize) + UP^C + E \\
\Rightarrow time &= 10^E NSize^C UP^A \\
where &\quad A \approx 0.65258 \\
&\quad C \approx -0.75902 \\
&\quad E \approx -1.503251
\end{aligned}
$$

This predictive model produces the estimates of GDA run time for the eil51 TSP shown in table 4.3.

Comparing the predicted values to the empirical values recorded in table 4.1 shows a reasonable fit. Figure 4.9 shows that the predicted values are generally within $7-8\%$ of the empirical results and at worst differ by 18%.

kroC100

Ten runs of the kroC100 problem were performed for fractional UP values of 0.005, 0.01, 0.05, 0.1, 0.5, and 1 and for $NSize$ values of 250, 500, 750, 1000,

	250	500	750	1000	1250	1500
0.005	64.3	101	132	159	184	207
0.01	38	59.7	77.8	93.9	109	122
0.05	11.2	17.6	22.9	27.7	32	36
0.1	6.62	10.4	13.6	16.3	18.9	21.3
0.5	1.95	3.07	3.99	4.82	5.57	6.28
1	1.15	1.81	2.36	2.85	3.29	3.71

Table 4.3. Predicted run times for the GDA on the eil51 TSP. Running times, in seconds, of the GDA are shown for various combinations of the parameters UP and $NSize$.

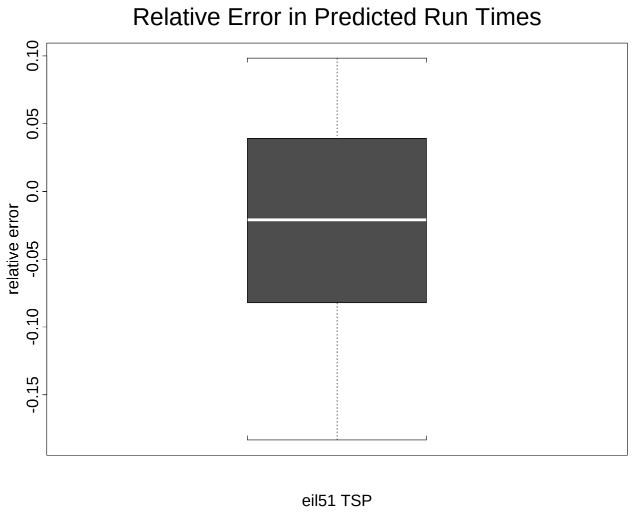

Figure 4.9. Relative differences in predicted run times for the eil51 TSP. Box plot of the relative difference in run times between the empirical and predicted results.

1250, and 1500. The maximum number of iterations was set at 2000000 and, as before, none of the runs came close to this limit, allowing all runs of the algorithm to terminate because of some form of convergence. The GDA was run for each set of parameter values on the same ten seeds as for the eil51 problem.

Results from the runs are shown in table 4.4 and table 4.4 with graphical interpretations of the results given in figure 4.11 and figure 4.10.

UP	NSize	250	500	750	1000	1250	1500
0.005	mean	502	930	1350	1719	1908	2291
	var	8488	13493	17223	57465	92866	55866
0.01	mean	301	557	671	857	988	1192
	var	868	9248	5840	16839	13963	4115
0.05	mean	77.5	132	181	241	262	306
	var	102	816	942	1203	1132	2809
0.1	mean	40.7	80.6	105	132	149	157
	var	86.8	303	473	632	1159	1146
0.5	mean	15.9	29.8	37.8	42	51.5	52.7
	var	9.3	62.4	29.6	48.2	101	122
1	mean	10.9	18.6	25.7	27.5	31.7	37.9
	var	6.46	18.2	22.7	14.9	21.6	83.3

Table 4.4. Running times for the GDA on the kroC100 TSP. Ten runs where made for each parameter setting and the run times, measured in seconds of CPU time were recorded with the means and variances displayed above

UP	NSize	250	500	750	1000	1250	1500
0.005	mean	58.76	27.22	14.97	9.46	8.02	4.87
	var	159.93	34.62	8.32	13.22	12.10	2.12
0.01	mean	47.86	21.52	15.27	10.00	7.71	4.59
	var	37.00	33.65	13.66	8.40	5.96	1.00
0.05	mean	40.39	20.76	11.17	6.19	5.08	4.03
	var	59.61	63.64	10.80	6.24	1.83	2.03
0.1	mean	45.26	17.93	12.07	8.00	6.82	6.61
	var	167.48	44.09	46.72	13.06	11.62	11.82
0.5	mean	38.81	19.43	15.35	14.43	11.38	11.38
	var	93.35	63.38	22.73	31.07	35.95	35.98
1	mean	49.97	25.03	17.10	16.27	14.54	12.52
	var	151.59	42.07	18.73	25.76	19.02	15.94

Table 4.5. GDA solution quality for the kroC100 TSP. Ten runs where made for each parameter setting and the best tour lengths found, measured as percentages above the optimal tour, were recorded with means and variances displayed above.

The results for the kroC100 problem have similar characteristics to those found for the eil51 problem. From figure 4.10 it can be seen that the linear relationships noted for eil51 between log($NSize$) and log($time$) and between log(UP) and log($time$) again hold true.

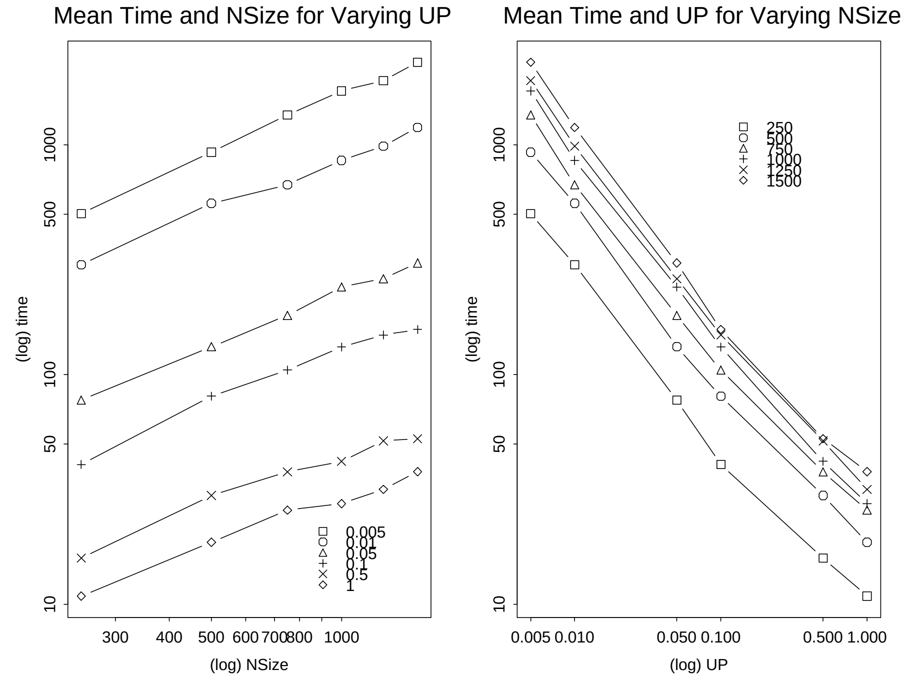

Figure 4.10. Graphical representation of GDA run times on the kroC100 problem. See table 4.4 for greater detail.

Similarly the relative performance of the algorithm for different UP follows the previously established pattern. As $NSize$ increases we find that smaller UP values begin to improve faster than larger UP values, see figure 4.11. This relative improvement in performance seems to happen less quickly than that observed for the eil51 problem. Such behaviour is expected this a 100-city TSP has a neighbourhood size of 9700, defined by the 2-change operation, compared to a size of 2448 for eil51. Observe in figure 4.12 mean times and costs for 5 repetitions of the kroC100 TSP for $NSize$ taking values up to 10000.

Once more better results are produced for smaller UP values only if $NSize$ is sufficiently large and that increasing $NSize$ beyond a certain size has only the effect of producing the same results in more time meaning a *stable* set of solutions has been found. The stability in solution that is reached by increasing $NSize$ for the kroC100 problem differs slightly from that reached by eil51, here an UP parameter setting of 0.005 only just produces solutions with the lowest cost, solution quality generated by $UP = 0.01$ is almost as good and are produced in substantially less time. The difference between the two parameter settings in the mean percentage above the best known solution is slight once stability has been reached, with a value 0.0103 for $UP = 0.005$ and a value 0.0108 for $UP = 0.01$. Whether this small difference is indicative of the particular problem, a statistical anomaly, or reflective of the increase in the problem dimension is unclear.

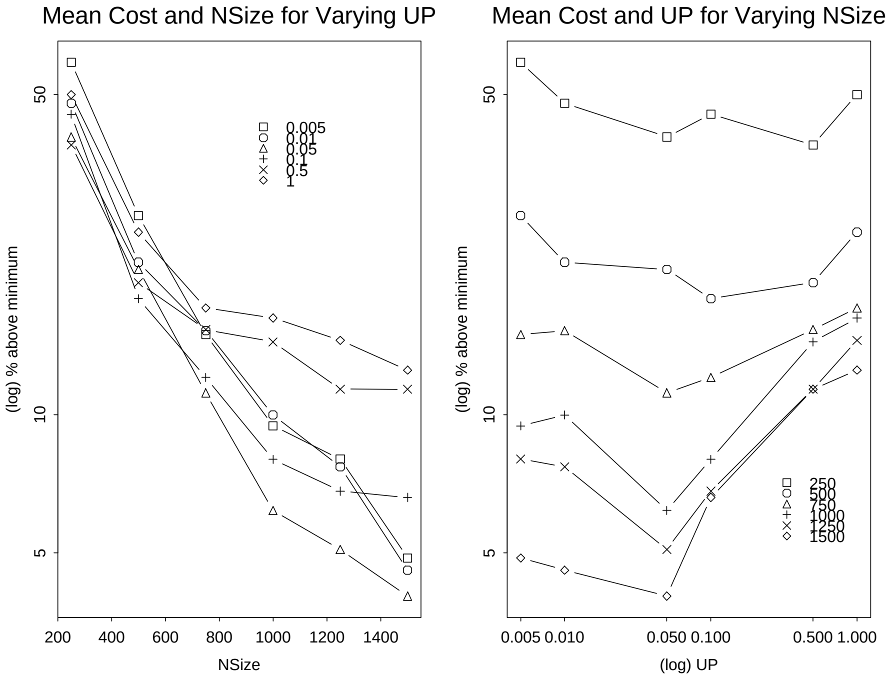

Figure 4.11. Graphical representation of GDA tour lengths on the kroC100 problem. See table 4.5 for greater detail.

At values of $NSize$ close to 10000, the log linear relationships begin to fail largely due, as has been previously noted, to the fact the tour lengths being produced by the algorithm have stabilised for a particular UP value. Nevertheless, for smaller values of $NSize$ an expression to predict run times for the GDA can be formulated in the same manner as for the eil51 problem.

$$\begin{aligned} \log(time) &= A\log(NSize) + UP^C + E \\ \Rightarrow time &= 10^E NSize^C UP^A \\ where \quad & A \approx 0.7501779 \\ & C \approx -0.7562299 \\ & E \approx -0.8494329 \end{aligned}$$

For $NSize$ values up to 1500 figure 4.13 shows that the predicted values are generally within $10-11\%$ of the empirical results and at worst differ by 23%.

kroA150

Ten runs of the kroA150 problem were performed for fractional UP values of 0.005, 0.01, 0.05, 0.1, 0.5, and 1 and for $NSize$ values of 250, 500, 750, 1000, 1250, and 1500. To ensure that the algorithm terminates due to convergence, the maximum number of iterations was set at 2000000. The GDA was run for each set of parameter values on the same ten seeds used as for the two previous problems.

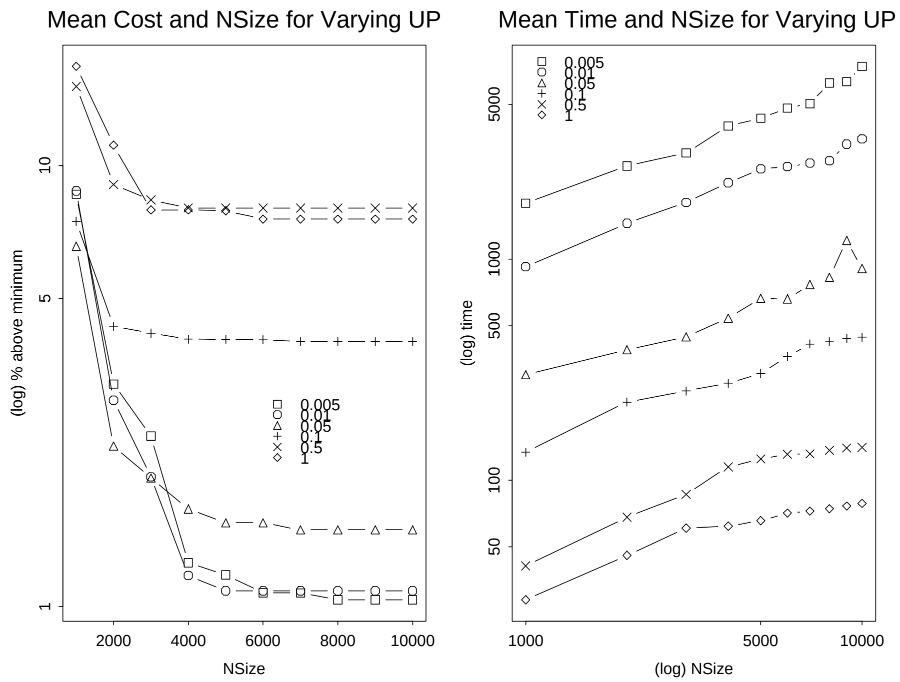

Figure 4.12. GDA results for the kroC100 problem for large $NSize$. Tour lengths and run times are shown for $NSize$ values up to 10000.

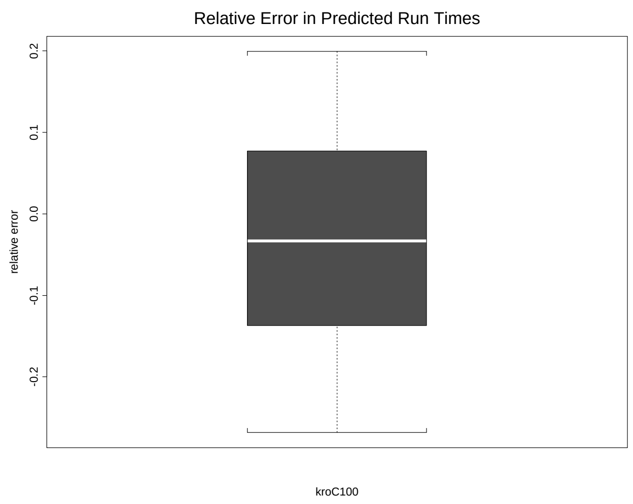

Figure 4.13. Relative differences in predicted run times for the kroC100 TSP. Box plot of the relative difference in run times between the empirical and predicted results.

Results from the runs are shown in table 4.6 and table 4.6 with graphical interpretations of the results given in figure 4.15 and figure 4.14.

Nsize	UP	0.005	0.01	0.05	0.1	0.5	1
250	mean	1524	855	234	140	52	31
	var	42955	31125	2162	474	53	52
500	mean	2962	1608	417	255	94	57
	var	231234	71203	2833	2439	198	98
750	mean	4003	2351	643	342	130	77
	var	246858	262780	5698	4795	777	356
1000	mean	5165	3063	856	462	148	89
	var	788923	89719	15056	6698	996	362
1250	mean	6554	3518	990	521	162	109
	var	1129611	224598	9422	12381	1250	188
1500	mean	7442	4048	1144	561	193	116
	var	989115	170257	25810	8062	2529	392

Table 4.6. Running times for the GDA on the kroA150 TSP. Ten runs where made for each parameter setting and the run times, measured in seconds of CPU time were recorded with the means and variances displayed above

UP	NSize	250	500	750	1000	1250	1500
0.005	mean	87.23	46.23	31.69	23.14	16.19	12.66
	var	93.27	91.79	25.70	33.13	35.47	11.18
0.01	mean	80.48	42.80	27.92	17.65	14.26	11.03
	var	236.49	60.93	136.04	9.20	13.65	5.87
0.05	mean	65.79	34.65	19.62	12.50	9.76	7.64
	var	120.77	32.20	14.37	17.42	4.13	4.83
0.1	mean	61.45	31.69	22.34	14.67	12.60	11.41
	var	139.17	57.52	40.72	20.66	15.21	8.84
0.5	mean	55.01	28.49	20.34	17.83	16.66	14.20
	var	140.16	45.78	53.07	31.48	30.97	12.90
1	mean	75.64	41.84	30.21	25.13	19.77	19.05
	var	206.64	160.83	135.30	66.28	3.50	6.13

Table 4.7. GDA solution quality for the kroA150 TSP. Ten runs where made for each parameter setting and the best tour lengths found, measured as percentages above the optimal tour, were recorded with means and variances displayed above.

The results for the kroA150 problem have similar characteristics to those found for both the eil51 and kroC100 problems. From figure 4.14 it can be seen that similar linear relationships noted for the previous problems between log($NSize$) and log($time$) and also between log(UP) and log($time$) again hold true.

Also the relative performance of the algorithm for different UP follows the previously established pattern. As $NSize$ increases we find that smaller UP values begin to improve faster than larger UP values, see figure 4.15.

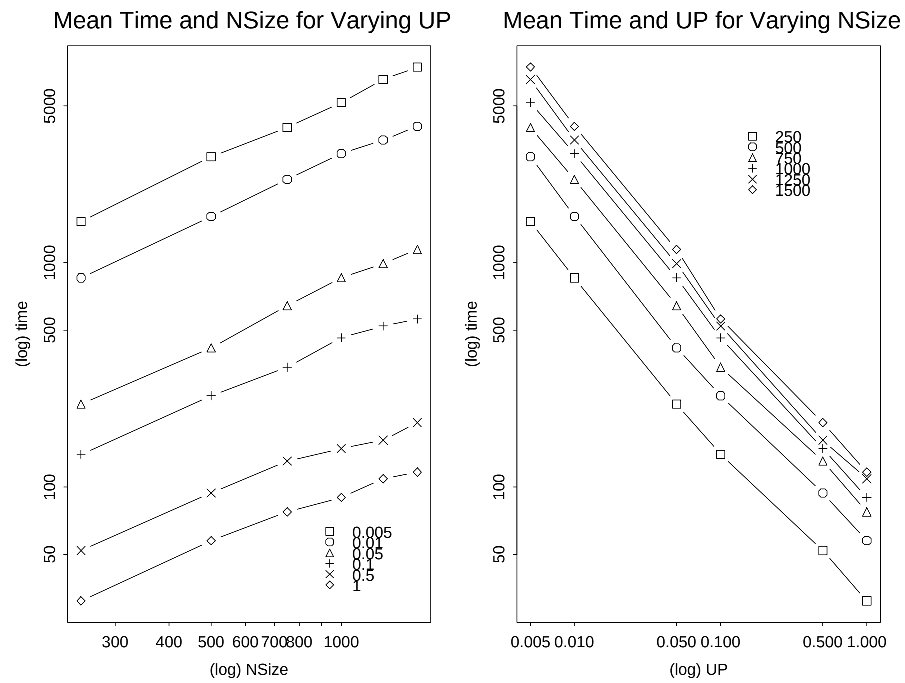

Figure 4.14. Graphical representation of GDA run times on the kroA150 problem. See table 4.6 for greater detail.

This relative improvement in performance seems to happen less quickly than that observed for either the eil51 problem or the kroC100 problem. Again this is to expected since the algorithm is dependent on $NSize$ which is related to the neighbourhood size, which for a 150-city TSP is 22050 as defined by the 2-change operation.

Although the time required to run the algorithm for repeated trials of $NSize$ values up to 10000 becomes prohibitive for TSPs this size we would expect comparable behaviour to the two smaller problems considered, i.e. once $NSize$ is large enough smaller UP values should indeed produce the best (in terms of solution quality) results.

As before, an expression to predict run times for the GDA for smaller values of $NSize$ can be formulated where the linear relationship still appears to hold true:

$$\begin{aligned}\log(time) &= A\log(NSize) + UP^C + E\\ \Rightarrow time &= 10^E NSize^C UP^A\\ where \quad & A \approx 0.8141832\\ & C \approx -0.7576995\\ & E \approx -0.5178442\end{aligned}$$

The accuracy of these predictions for $NSize$ values up to 1500 can be seen from figure 4.16,which are largely within 12% of the empirical results and at worst differ by no more than 20%.

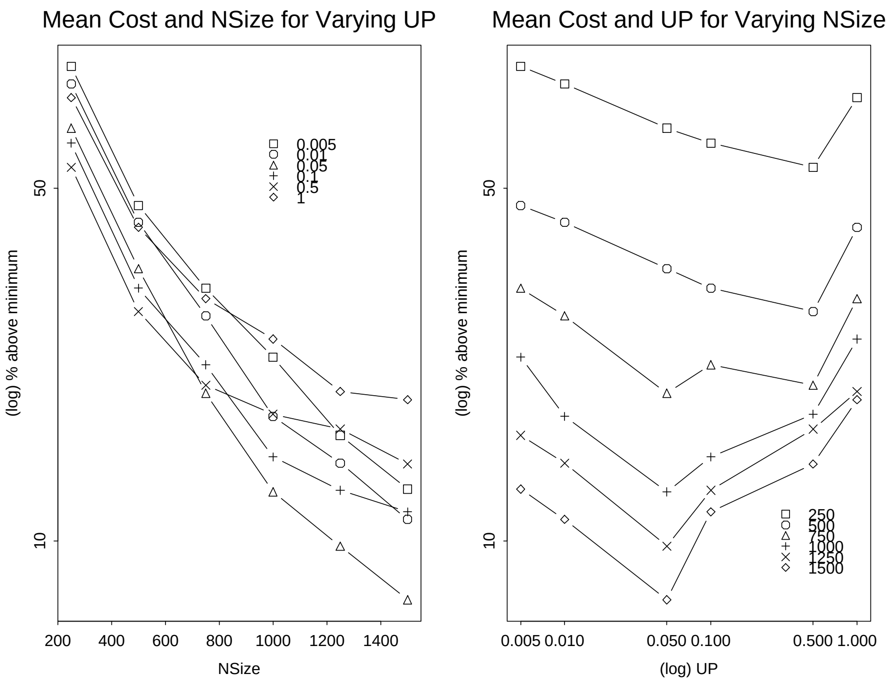

Figure 4.15. Graphical representation of GDA tour lengths on the kroA150 problem. See table 4.7 for greater detail.

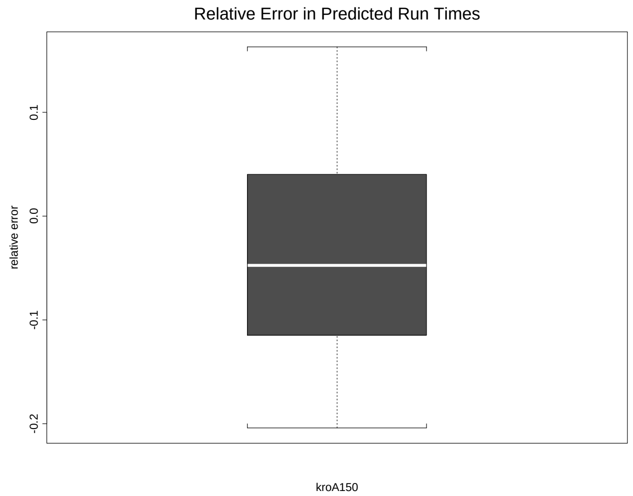

Figure 4.16. Relative differences in predicted run times for the kroA150 TSP. Box plot of the relative difference in run times between the empirical and predicted results.

4.6 Conclusions

The main concern of the empirical study, given in section 4.5, is to analyse the claim made by Dueck [11] that the TA algorithms are dependent only on a single parameter, the rain speed for the GDA and the allowed deviation for the RTRT algorithm. The empirical study has produced one clear result, that the GDA is not dependent on a single parameter, other factors can and do influence the quality of the solutions produced. Now consider some of the issues raised about parameter selection and interaction in section 4.4.

Concerns about the initial configuration and initial water levels for the GDA, when starting near a local optimum, are perhaps justified but are seldom likely to influence the result of the algorithm to any significant degree. Since the vast majority of solutions to a TSP instance are of poor quality the chance of randomly hitting a local (or near local) optimum to begin the algorithm are slim. Certainly for all of the trials performed in empirical analysis of the GDA this problem was never encountered.

The need for a failsafe stopping condition specifying a maximum number of iterations seems unnecessary. If the algorithm converges to a local optimum the failsafe is redundant, it simply introduces the possibility of terminating the search too soon. In all of the 2000 or so trials performed with various parameter settings, never once did the GDA terminate due to the maximum number of iterations being reached, all converged to a final configuration. Since the maximum number of iterations needs increasing as the problem dimension increases, to prevent an early halt in the algorithm, it would seem far simpler to abandon this stopping condition altogether and rely instead on the convergence of the GDA.

The question remains as to whether the GDA is guaranteed to converge. By appealing to the SA analogy made in section 4.4, a method that has the property of theoretical asymptotic convergence, I claim that the GDA is guaranteed to converge. In SA, when minimising, as the temperature approaches zero there is a smaller chance of an uphill move being accepted. Thus the algorithm is forced to intensify the search by accepting a greater proportion of solutions that improve the current position, leading to convergence to a particular solution. The GDA exhibits similar behaviour. As the water level decreases there are fewer allowable uphill moves to choose and the algorithm is more likely, in some sense, to make moves improving the current position rather than choosing moves that lead away from the local minimum. This forces the search to intensify, which inevitably leads to convergence.

As has been mentioned above, lower rain speed on its own is not enough to guarantee better quality results. Indeed in some cases, having low rain speed produces worse results than rain speeds 10 times as fast, while taking much longer to do so! The quality of the solutions produced by the algorithm seem to depend on two main factors, the rain speed, UP and $NSize$, the amount of the current neighbourhood looked at before admitting a better solution cannot be found. By increasing $NSize$ sufficiently, smaller UP values begin to give the best quality results. Why the relative performance of smaller UP values should change as $NSize$ increases, irrespective of the problem dimension, is unclear. Two potential reasons to explain this behaviour, whose precise interaction with the algorithm is not fully understood, are given below:

- The main termination criterion used in the present algorithm is to stop if no new solution is found for $NSize$ iterations. This is stricter than the specified criterion of stopping when no *improving* moves have been made in the last $NSize$ iterations. Certainly the weaker condition should result in the algorithm terminating sooner, but is the unusual relationship between UP and $NSize$ maintained?

- The water level increase is linked to the present distance between the current position and the water level. When a general 'good' area is entered the rain speed increases. If the rain is fast enough the algorithm will become trapped in this good region and is subsequently forced to intensify the search. With a slower rain speed the algorithm has time to 'wander off' from the good area and eventually become trapped in an area of worse solution quality.

However making modifications to the GDA to investigate these two explanations produced results with similar behaviour to that mentioned above. It remains an open question as to why the interaction between UP and $NSize$ should produce the results that it does.

The matter of which parameter settings should be used to balance the conflicting goals of good quality solutions and reasonable run time is dependent on the particular problem involved. Is any good solution satisfactory or does the solution really need to be near optimal? How is the algorithm to be used? Is a single run acceptable or will many runs be required? Such questions must be considered before the selection of suitable parameters. Nevertheless, general guidelines about the parameters can be made. In agreement with Dueck and Scheuer [10] a fractional UP value of around 0.01 produces good quality solution in reasonable time. The appropriate value of $NSize$ is inextricably linked to the problem dimension through the size of the neighbourhood. Roughly speaking, an appropriate $NSize$ value appears to be somewhere over half the size of the neighbourhood.

In conclusion, the GDA is (and in fact all the TA algorithms are) very simple to implement, producing good quality solutions in a reasonable amount of time. Since the only problem specific areas of the algorithm are the definition of a cost structure and a neighbourhood, GDA is easily applicable to many problems. Comparisons to several other generally applicable heuristics are made in chapter 5.

Chapter 5

Comparisons of Methods

5.1 Heuristic Evaluation

Evaluation of any heuristic method, either exact or generally applicable, is subject to the comparison of a number of criteria that relate various aspects of algorithm performance. Ball and Magazine [2] list several evaluation criteria:

- quality of solution,
- running time,
- ease of implementation,
- simplicity,
- flexibility and robustness.

An algorithm that can be stated simply has much appeal and can also lead to formal analysis of the method. A theoretical study of a complicated technique is difficult, so simple algorithms that lend themselves to theoretical analysis have a certain amount of æsthetic charm.

Linked to algorithm simplicity is its ease of implementation. Generally, a simply stated algorithm should be easier to code into a computer language than a more complicated algorithm. However this is not necessarily the case, since a simple idea may have subtle coding requirements. Also, data structures required for an algorithmic technique may be complicated to implement. An algorithm that is comparatively easy to code has an advantage over a technique with intricate implementation details, although, as with algorithm simplicity, the advantage is difficult to quantify.

Since heuristic methods are ultimately designed to solve real world problems, flexibility is an important consideration. The algorithm should be able to easily handle changes to the model, the constraints, and the objective function. Furthermore, a robust algorithm is still able to produce results under difficult circumstances such as when a problem instance is pathological. Taken to extremes, an algorithm that has a great amount of flexibility and robustness can be generally applicable and thus able to solve many different types of problems.

The time an heuristic takes to produce good quality solutions can be crucial when choosing between different techniques. Similarly the quality of the final solution, as measured by the objective function, is important. How close the solution is to the optimal solution is a standard measure of quality or if the heuristic is designed to simply produce feasible solutions then the ability of the heuristic to give such solutions is important.

There is generally a trade-off between run time and solution quality, the longer a heuristic is run the better the quality of the final solution. A compromise is needed so that 'good quality' solutions are produced in a 'reasonable' amount of time. More often than not it is the final solution quality that is used as a measure of the success of an heuristic. As noted by Reeves [36], there are three general methods to compare heuristic solution quality:

- analytical methods,
- empirical methods,
- statistical inference.

Analytical analysis of an heuristic can be achieved a number of ways, the most common of which are worst-case analysis and average-case analysis. Worst-case analysis, as its name suggests, involves finding a theoretical guarantee by placing a bound on algorithm performance (in terms of a generalised problem instance). However this often gives bounds for the heuristic that are too loose to be of any practical use when determining how well the algorithm has done. Average-case analysis seeks to determine how well the heuristic performs in terms of expectation. However as with any probabilistic analysis, a mean without any measure of variability is not of much use. Also, this form of analysis often associates an unrealistic probability space with the solution space, giving meaningless results. Both these measures of performance are seldom useful in a practical sense, but do lead to a better understanding of the heuristic method and the problem at hand.

Statistical inference, the method of estimating a parameter of a statistical population from a sample has only been applied to the estimation of heuristic solution quality recently. Reeves [36] has details of such an approach using results from the statistical theory of extreme values. The method is promising but has received little attention in the evaluation of algorithm performance.

By far the most common (and most trusted) method of evaluating the solution quality of a heuristic algorithm is empirical analysis. In general, empirical analysis involves testing the heuristic across a wide range of problem instances to give an idea of overall performance. To give conclusions that have meaning in a statistical sense, experimental design should really be used on different levels of the various algorithm parameters and the results compared by ANOVA[1] or by a nonparametric test[2]. Lawler *et al.* [27] devotes a chapter to the empirical analysis of heuristics specific to the Traveling Salesman Problem (TSP), but their discussion is valid for all heuristics.

The analysis of algorithm performance carried out here is via simple empirical analysis of the run time and final solution quality produced from several heuristics applied to the TSP. The means and variances for the above performance measures from repeated runs of several heuristics for a number of test problems are compared to determine which heuristic is 'best' at solving combinatorial optimisation problems. The emphasis is on generally applicable heuristics and the methodologies investigated here are the Great Deluge Algorithm (GDA), Genetic Algorithms (GAs), and repeated local search (RLS), described in chapter 4, chapter 3, and section 2.6 respectively. For a full comparison, the heuristics should be compared across a range of combinatorial optimisation problems, but due to time constraints the TSP is used as the single combinatorial problem investigated and it is hoped that the TSP is sufficiently indicative of other combinatorial optimisation problems to give some insight into the relative performance of the three heuristics when applied to combinatorial problems. For completeness, results from the three generally applicable heuristics are compared to those produced by a tailored TSP heuristic, the variable k-opt algorithm, discussed in section 2.5. A description of the four TSP instances used in the comparison of the heuristics is given in appendix B. The problems are of various dimensions, 29 cities, 51 cities, 100 cities, and 150 cities, hereafter referred to as the bays29 TSP, the eil51 TSP, the kroC100 TSP, and the kroA150 TSP respectively.

5.2 A Preliminary Analysis

The three generally applicable heuristic algorithms compared in this chapter all have a similar purpose when applied to combinatorial optimisation problems. All three methods seek to find a 'good' solution to a large and complicated search space by escaping the trap of local optimality in a controlled fashion. Since GAs, the GDA, and RLS can easily be applied to many different problem types with little need for recoding, they require the definition of a cost function to evaluate the success of a solution and also require a problem structure to determine how to move about the search space. The heuristic methods differ in the manner in which they move about the search space and in the way local optima are avoided.

Although the three heuristics mentioned above have similar intent and use similar ideas to achieve their goal, there are many differences between them.

[1] ANOVA is an acronym for ANalysis Of VAriance, a statistical test for homogeneity across several groups of data.

[2] The usual tests are the Wilcoxon signed rank test or an extension of this test, a nonparametric version of ANOVA, the Friedman test.

The GDA and GAs "converge" in some sense, to a final solution (or set of solutions) whereas RLS has an arbitrary stopping condition dependent on the number of times local search is restarted. The GDA and GAs both 'explore' the search space in a general manner then 'exploit' information by refining the search to examine a particular good area to ultimately find a good local optimum. RLS applies this process in reverse, immediately searching for a local optimum then restarting the procedure in another area of the search space.

RLS and the GDA move around the solution space by way of the 2-change operator described in appendix A, GAs move around the solution space by means of the reproductive plan, especially the crossover and mutation operators described in chapter 3. The GA operators are much more computationally intensive than the 2-change operator and are certainly more complicated (at least for the TSP GA operators). Furthermore, when moving between solutions in the search space, the 2-change operator allows for incremental cost calculation since the costs of the old and new solutions are closely linked, with no need to recalculate the entire cost of the new solution. GAs have no such ability and must recalculate the cost of a new solution every time one is produced.

GAs maintain a population of solutions to overcome the trap of local optimality. The GDA works with a single solution allowing uphill moves in a controlled manner to climb out of a poor local optimum. RLS also works with a single solution, simply restarting itself in a random location in the search space once a local optimum has been found.

GAs and the GDA are randomised algorithms that use a stochastic element to help guide the search for optimality and so are not strongly dependent on the initial solution. Local search (when using steepest descent) has no probabilistic element and so is strongly dependent on the initial solution selected. If the initial local search solution is in a good area then a good local optimum will result, if started in a poor area then the algorithm will produce a poor quality solution. This dependence is overcome to a certain extent by restarting the algorithm in a random location once a local optimum has been found, but the ability of RLS to give a good final solution is dependent on one of the initial solutions being in a good area, which may be a unlikely event.

RLS has the advantage that it does not require any parameters to change, given an initial solution the heuristic will continue until told to stop, producing a series of local optima, of which the best is selected. The GDA and GAs have many parameters that can be adjusted to try and produce better results, and the algorithm can be heavily dependent on the exact choice of parameters used, meaning in many situations a preliminary trial must be carried out to select suitable parameter values. There may also be unforeseen interaction between the parameters, so a careful watch must be kept on how altering parameters affects solution quality.

It can be of use in examining heuristic performance to view how solution quality varies over time as the algorithm proceeds. To this end, solutions from a typical run of each of the three generally applicable heuristics, applied

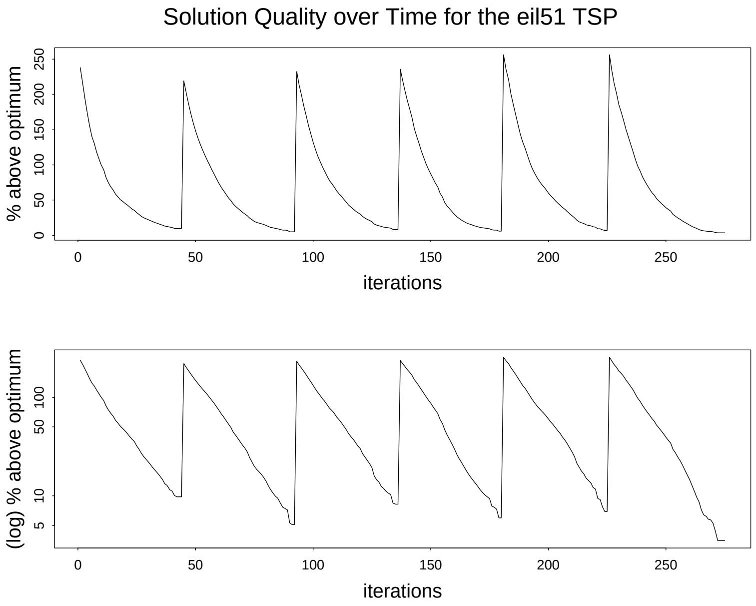

Figure 5.1. Solution quality over time for RLS. Beginning with an initial solution the algorithm finds the steepest path and follows it downhill to the local optimum. The heuristic is then restarted several times to generate a number of local optima. The solution quality over time for the eil51 TSP is shown in the top graph, with the log of the solution quality shown in the bottom graph.

to the eil51 TSP, have been collected and are displayed in figure 5.1, figure 5.2, and figure 5.3.

Looking at the three graphs mentioned above it is apparent that the GDA spends a lot of time searching the solution space at levels of quality that are close to optimal. GAs do not appear to spend as much time at quality levels close to optimality. The question this poses is whether the extra effort spent at fine-tuning the search produces solutions of substantially better quality.

5.3 Empirical Results

A tailored TSP heuristic, k-opt, and three generally applicable heuristics, the GDA, GAs (looking at the best solution found rather than the average generation solution quality), and RLS (steepest descent), discussed in detail in chapter 4, chapter 3, and section 2.6 respectively, have been run several times on four TSP instances, bays29, eil51, kroC100, and kroA150. The solution qualities, measured as a percentage above optimal, and the run times, measured in seconds of CPU time, were recorded are displayed in two formats. The means and variances of the solution qualities and of the run times are given in a tabular form while boxplots of the results are given as a graphical portrayal of the relative performance of the heuristics. For each TSP instance different heuristic parameters and various numbers of repetitions have been used.

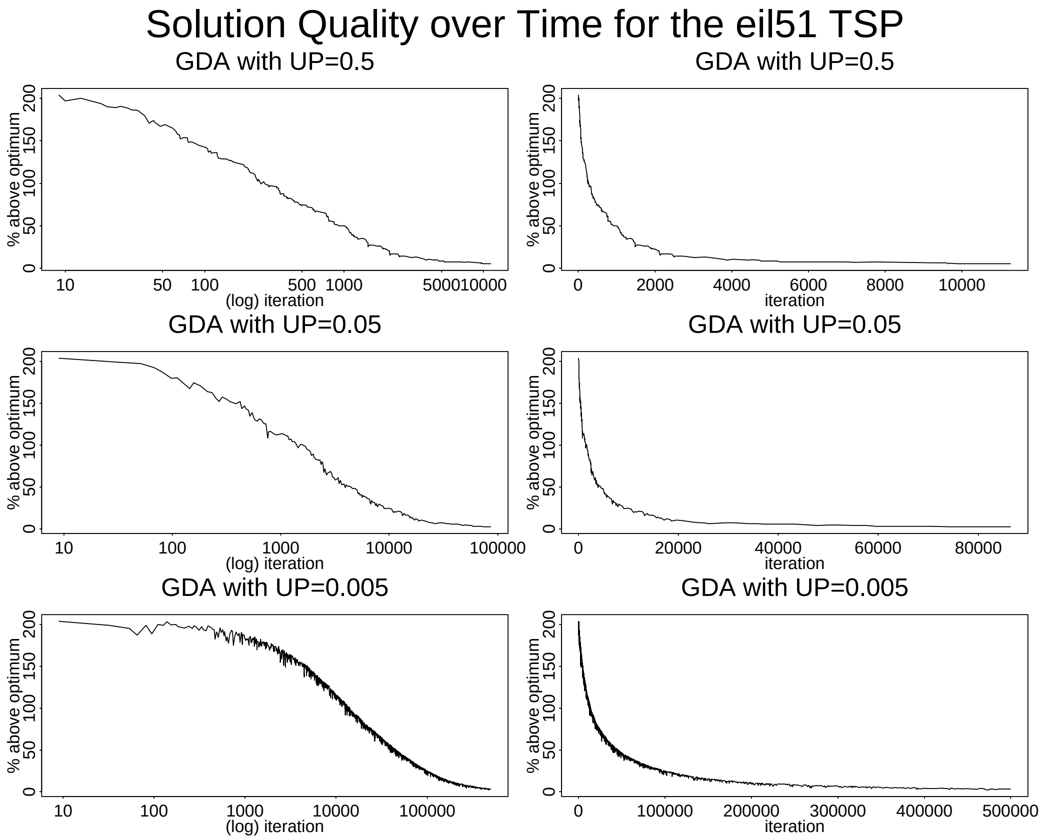

Figure 5.2. Solution quality over time for the GDA. The solution quality over time for the eil51 TSP on the right and the solution quality over the log of time on the left, for several UP values with $NSize$ and the maximum number of iterations both large, are shown above.

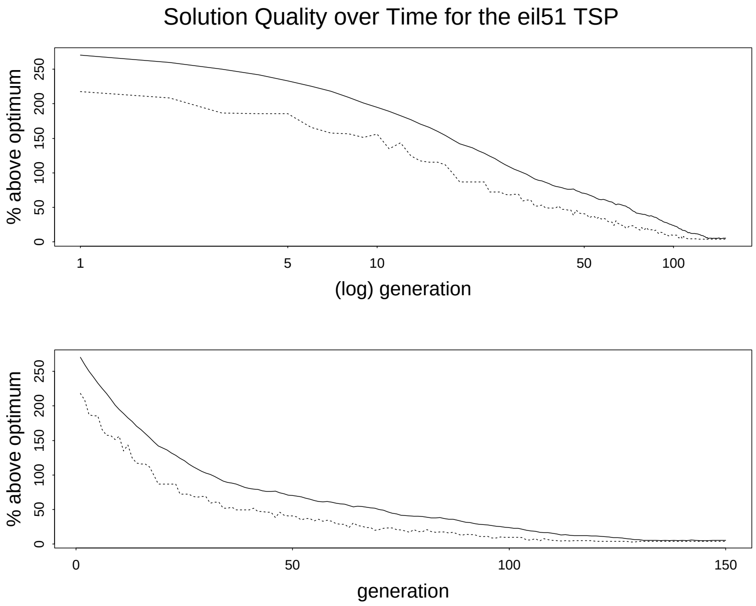

Figure 5.3. Solution quality over time for a GA. The solution qualities over time for the eil51 TSP on the bottom and over log time on the top are displayed above. The best solution quality in a generation is shown by the dashed line and the average solution quality from a generation is shown by the solid line. Parameter settings: n = 500, p_c = 0.6, p_m = 0.05, ϵ = 0.0005, N = 10.

bays29:

- k-opt was run for ten repetitions.
- The GA was run for ten repetitions with parameter settings: n = 400, p_c = 0.6, p_m = 0.05 ϵ = 0.05, and N = 10
- RLS_1 was run for ten repetitions for a length of time approximately equal to the mean run time required by the GA on the bays29 TSP.
- The GDA was run for ten repetitions with parameter settings: UP = 0.01, $NSize$ = 750, and the maximum number of iterations set at 5 million.
- RLS_2 was run for ten repetitions for a length of time approximately equal to the mean run time required by the GDA on the bays29 TSP.

eil51:

- k-opt was run for ten repetitions.
- The GA was run for ten repetitions with parameter settings: n = 600, p_c = 0.6, p_m = 0.05 ϵ = 0.0005, and N = 10
- RLS_1 was run for ten repetitions for a length of time approximately equal to the mean run time required by the GA on the eil51 TSP.
- The GDA was run for ten repetitions with parameter settings: UP = 0.01, $NSize$ = 2500, and the maximum number of iterations set at 5 million.

- RLS_2 was run for ten repetitions for a length of time approximately equal to the mean run time required by the GDA on the eil51 TSP.

kroC100:

- k-opt was run for ten repetitions.
- The GA was run for five repetitions with parameter settings: $n =$ 1500, $p_c = 0.6$, $p_m = 0.05$ $\epsilon = 0.000001$, and $N = 10$
- RLS_1 was run for five repetitions for a length of time approximately equal to the mean run time required by the GA on the kroC100 TSP.
- The GDA was run for ten repetitions with parameter settings: $UP = 0.01$, $NSize = 5000$, and the maximum number of iterations set at 5 million.
- RLS_2 was run for ten repetitions for a length of time approximately equal to the mean run time required by the GDA on the kroC100 TSP.

kroA150:

- k-opt was run for ten repetitions.
- The GA was run for five repetitions with parameter settings: $n =$ 1500, $p_c = 0.6$, $p_m = 0.05$ $\epsilon = 0.0000001$, and $N = 10$ but no convergence was achieved for reasons that have been discussed in chapter 3.
- RLS_1 was not run since no results from the GA were available for comparison.
- The GDA was run for five repetitions with parameter settings: $UP = 0.01$, $NSize = 7500$, and the maximum number of iterations set at 5 million.
- RLS_2 was run for five repetitions for a length of time approximately equal to the mean run time required by the GDA on the kroA150TSP.

Tabular results are given in table 5.1 and table 5.2 while a graphical representation is displayed in figure 5.4 and figure 5.5.

From figure 5.4 it can be seen that k-opt and the GDA consistently give the best quality solutions. For problems of smaller dimensions the tailored heuristic produces results with almost negligible variation, but as the problem size increases the variation becomes comparable with that produced by the GDA and it is difficult to say which heuristic is 'better' in terms of solution quality. For all TSP instances investigated the GDA outperforms RLS (for a comparable length of time) quite noticeably. The GA seldom produces solutions of a quality as good as that produced by the GDA or k-opt and appears to have a relatively large variation associated with the solutions. Furthermore, the heuristic is often outperformed by RLS.

The run times of the various heuristics can be seen in figure 5.5. As expected the tailored heuristic produces results very quickly in comparison to the general heuristics, while the GDA is substantially faster than the GA. In comparing run times it should be noted that the three general heuristic

		bays29	eil51	kroC100	kroA150
k-opt	mean	0.0281	0.8440	2.5386	1.2465
	var	$9.7600 \cdot 10^{-5}$	0.5046	2.8910	0.2898
GA	mean	1.8330	3.1904	8.8648	NA
	var	1.0645	1.9723	23.1528	NA
RLS_1	mean	0.2242	3.4017	2.7059	NA
	var	0.0286	1.4943	0.9644	NA
TA	mean	0.4670	1.6011	1.5895	1.5256
	var	0.7748	2.1429	1.2341	0.9248
RLS_2	mean	0.5187	3.6865	6.2957	6.8512
	var	0.1354	1.6610	4.5712	6.6225

Table 5.1. Heuristic solution quality for the TSP. A tailored TSP heuristic, k-opt, and three generally applicable heuristics, the GDA, GAs, and RLS have been run several times on four TSP instances, bays29, eil51, kroC100, and kroA150. The solution qualities, measured as a percentage above optimal, have been recorded and their means and variances are given above.

		bays29	eil51	kroC100	kroA150
k-opt	mean	0.0367	0.0683	0.1533	0.2883
	var	$1.7284 \cdot 10^{-4}$	$3.3642 \cdot 10^{-4}$	$6.0494 \cdot 10^{-4}$	$1.1142 \cdot 10^{-3}$
GA	mean	47.1050	275.089	12534.1	NA
	var	9.9756	996.985	$3.0795 \cdot 10^{7}$	NA
RLS_1	mean	48.4433	294.155	13149.9	NA
	var	0.6009	115.098	234461.4	NA
TA	mean	18.3417	185.315	2259.61	10912.8
	var	15.0959	2379.79	123872.9	1741921
RLS_2	mean	19.5850	199.915	3110.61	15764.4
	var	0.6208	59.8037	4424.50	227156.1

Table 5.2. Heuristic run time for the TSP. A tailored TSP heuristic, k-opt, and three generally applicable heuristics, the GDA, GAs, and RLS have been run several times on four TSP instances, bays29, eil51, kroC100, and kroA150. The run times, measured in seconds of CPU time, have been recorded and their means and variances are given above.

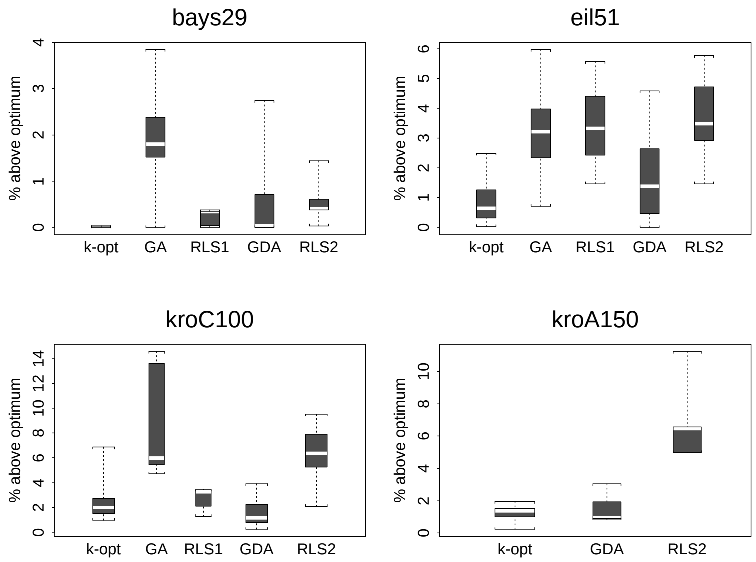

Figure 5.4. Comparison of heuristic solution quality. Four heuristic algorithms were run repeatedly for four TSP instances and solution qualities were measured as a % above the optimal tour length with boxplots given above. The heuristics are k-opt, a GA, RLS and the GDA. The four TSP instances are bays29, eil51, kroC100, and kroA150 as described in appendix B.

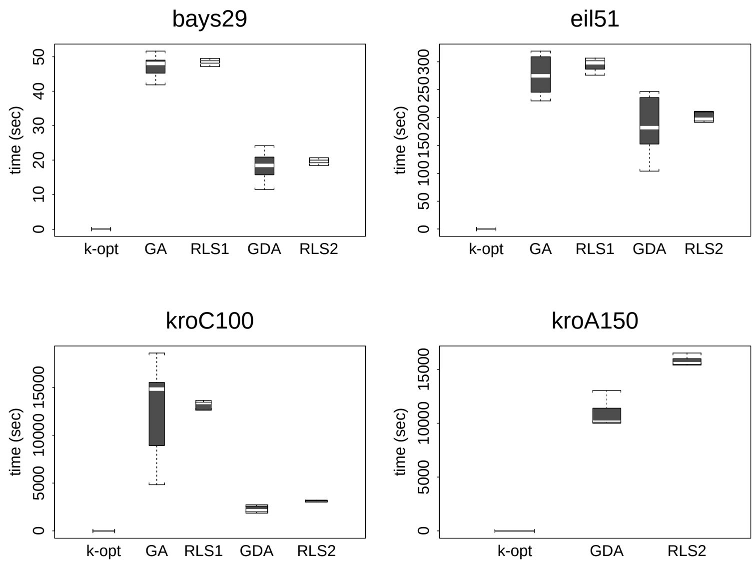

Figure 5.5. Comparison of heuristic run time. Four heuristic algorithms were run repeatedly for four TSP instances and algorithm run times were measured in seconds of CPU time with boxplots given above. The heuristics are k-opt, a GA, RLS and the GDA. The four TSP instances are bays29, eil51, kroC100, and kroA150 as described in appendix B.

are coded in the same language, C++ (see appendix C), whereas the k-opt[3] algorithm is written in a different language, FORTRAN, so a direct comparison of run times is not relevant although the general relationship is still valid.

5.4 Conclusions

From the above empirical analysis it may be possible to determine which heuristic is the best at solving combinatorial optimisation problems, especially the TSP. In terms of solution quality there is not much difference between the GDA and the k-opt algorithm, and these two algorithms certainly produce solutions closest to optimality with the greatest regularity. However the time taken by the k-opt algorithm is orders of magnitude smaller than that required for all three general heuristics considered, which was expected from a tailored algorithm. Of the general heuristics it can be said that GDA outperforms RLS (when comparative run times are used) with regularity while GA is not much better than if at all better than RLS when considering solution quality. The GDA is better than the GA, both in solution quality and in run time, and, for the TSP at least, is therefore a 'better' heuristic.

Although solution quality and algorithm run time are the main comparative measures used in this analysis, other evaluation criterion can be examined. Comparing the ease of heuristic implementation, subject to personal opinions, RLS requires the least effort to code, then the GDA and lastly the GA. The simplicity of algorithm (again subject to my own prejudices) mirrors the ease of implementation, RLS is most simple, then the GDA (as is essentially an extended local search) and lastly the GA. All three general heuristics are flexible and robust so can easily handle changes to the problem structure, constraints and objective function. The k-opt cannot easily accommodate such changes and so although k-opt ties for solution quality with the GDA and wins hands down for run time over all the other heuristics, as with any tailored algorithm it will need substantial reworking if difficult side constraints or a change to the problem are required.

Considering only the performance of the three general heuristics, the reasons for the difference in solution quality needs examining. The difference in the manner by which the GA moves about the solution space together with the lack of incremental cost improvement for the GA should lead to slower running times for the same amount of exploration. Also, whereas RLS and the GDA work with a single solution, by design GAs work with a population of solutions and so require a lot more work to maintain than the the local search style algorithms, which should lead to a slower algorithm unless convergence occurs more rapidly than for the GDA.

The fact that the GDA spends a larger fraction of total run time search-

[3] D Slowinski (`slow@runkel.cray.com`) kindly provided the code for the k-opt heuristic via the goodwill of G Grimwood (`geoff@isor.vuw.ac.nz`). It is in FORTRAN and was compiled using the Sun FORTRAN compiler.

ing solutions that are close to optimal as compared to GAs which spend a relatively small fraction of run time fine-tuning the search seems to result in solutions of better quality. Even though the GDA spends more time fine-tuning it is still faster than the GA which is perhaps attributable to the fact that the algorithm has less work to do, and with an incremental cost structure the GDA can sift through a comparable amount of the search space with less effort. RLS seems to outperform GAs when run in comparable amounts of time and is also a much simpler algorithm, which questions the validity of the GA paradigm when applied to combinatorial optimisation. GAs, as has been mentioned, are good at finding a general area of optimality but are poor at fine-tuning the search, this is reflected both in chapter 3 and in the results given in this chapter. Although RLS does outperform the GA implemented here, it does not outperform the GDA which appears better at fine-tuning the search than the GA. GDA is not as dependent on the initial solution as RLS, so can find a good solution with greater regularity. RLS will give a good solution only if an initial solution in a good area is found when restarting the algorithm.

GAs have only been applied to function optimisation and combinatorial optimisation relatively recently and do have a broader application than the GDA and RLS which are essentially restricted to the combinatorial optimisation domain. At present it seems GAs are best suited to areas other than combinatorial optimisation, especially Artificial Intelligence and other related areas. Since general heuristics such as the GDA already exist for combinatorial optimisation and do a better job than GAs, both in solution quality and in run time, there seems little need to appeal to the GA paradigm to solve such problems. Having said this, GAs have more parameters to 'optimise' than for the GDA when considering a combinatorial optimisation problem, and it is conceivable that further refinement of the parameters will produce better results for the GA than those displayed here. However a similar argument can be applied to the GDA.

Direct comparisons of generally applicable heuristic seem scarce in the literature, although the following papers tend to support the conclusions made here.

Nurmela [32] investigates the use of general heuristics when applied to covering design, packing design and t-designs. The conclusions made at the end of the report are that GAs were well outperformed by local search style algorithms, and of the local search algorithms SA performed consistently well although GDA and RTRT often outperformed SA in solution quality and run time.

Sinclair [38] investigated several general heuristics when applied to a real-world problem, hydraulic turbine runner balancing (a special case of the quadratic assignment problem) and found that GAs performed poorly, Tabu Search gave the best quality solutions but took a long time, annealing gave good quality solutions and the TA methods gave the best balance between solution quality and algorithm run time.

Appendix A

The 2-Change Neighbourhood Generation Operation

The 2-change operation was first proposed by Lin [29] and is thus sometimes referred to as the Lin-2-Opt change. Given a solution x to an instance of the Traveling Salesman Problem (TSP) the 2-change operation generates a new tour in the following manner. Two edges are removed from the tour representing x and are replaced so that a tour is maintained. There is just one way in which to do this for the 2-change and it amounts to the reversal of the order in which the cities are traversed inbetween two arbitrary cities, p and q say, in the tour x. A 2-change is demonstrated in figure A.1.

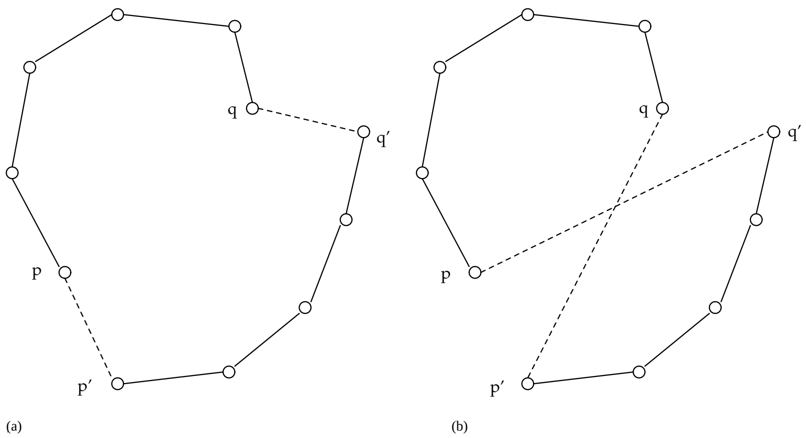

Figure A.1. Example of a interchange operation on a instance of the TSP between the initial tour (a) and the tour resulting from a 2-change (b).If p and q are the cities selected by the operation then the links pp' and qq' are broken and replaced by pq' and qp'.

By considering all the possible tours that can be reached from x by the two change, a neighbourhood for x, $N(x, \sigma)$, is defined, where σ is the 2-change operation. As mentioned above the operation is fully described by choosing two cities p and q and reversing the order in which the cities are traversed between them. Let y be the solution resulting from the 2-change operation on x with specific values for p and q. Then the *cyclic permutations* π_x and π_y representing the two tours are related.

A cyclic permutation is a term used in *algebra* , a field of mathematical study. A *permutation* is a *one-to-one mapping* from a set S onto itself. When S is the set of integers up to and including n, $\{1, 2, \cdots, n\}$, the associated *group* of permutations is called S_n. An element of S_n, π, can be represented in a *two-row form*. Write $1 \cdots n$ on the first line and below each number k write its image $\pi(k)$. Thus

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 1 & 3 \end{pmatrix}$$

represents the permutation in S_4 defined by $1 \mapsto 2$, $2 \mapsto 4$, $3 \mapsto 1$, and $4 \mapsto 3$. *Cycle notation* can be used to represent elements of S_n. A *k-cycle* is a cycle $(a_1, a_2, \cdots, a_k)$, where $a_1 \cdots a_k \in S$, representing the permutation in S defined by $a_1 \mapsto a_2, a_2 \mapsto a_3, \cdots, a_{k-1} \mapsto a_k$. Thus an n-cycle represents a tour if $1 \cdots n$ are "cities". So the *cyclic permutation* representing an n-cycle is a one-to-one mapping from each city k in the tour to its successor $\pi(k)$, and

$$\begin{pmatrix} 1 & 2 & \cdots & n \\ \pi(1) & \pi(2) & \cdots & \pi(n) \end{pmatrix}$$

fully represents a tour of the n cities. In any permutation π we let $\pi^m(i)$ be the composition of m images beginning at i, i.e $\pi^m(i) = \pi(\cdots(\pi(\pi(i))))$. In the above example then $\pi^2(2) = \pi(\pi(2)) = \pi(4) = 3$.

Let π_x and π_y be the permutations representing the two tours x and y, and p and q the two "cities" representing the 2-change operation.. Let k equal the integer value for which $\pi_x^k(p) = q$ $1 < k < n$. The two permutations are then related as follows:

$$\pi_y(p) = \pi_x^{-1}(q) = \pi_x^{k-1}(p)$$

$$\pi_y(\pi_x(p)) = q = \pi_x^k(p)$$

$$\pi_y(\pi_x^r(p)) = \pi_x^{r-1}(p) \ r = 2, \cdots, k-1$$

$$\pi_y(s) = \pi_x(s) \ otherwise$$

The size of the neighbourhoods generated by this operation is theoretically *the number of ways of choosing p* $\times$ *the number of ways of choosing q* , which is n^2 in an n-city TSP. However we exclude all tours from the neighbourhood which are equal to the current tour, since they are not genuine permutations. By excluding cases where $|p - q| < 3$ any tour in the neighbourhood is ensured to be different from the present solution, giving a neighbourhood size of $n(n-3)$.

Using the 2-change operation to generate neighbours has the implication that any solution x can be reached from any other solution y in $n - 2$ moves, since for all tours x and y there exists a sequence $l_0, \cdots, l_{n-2}$ such that

$$\pi_{l_0} = \pi_x \ and \ \pi_{l_{n-2}} = \pi_y$$

and $\pi_{l_k} \mapsto \pi_{l_{k+1}}$ $k = 0, \cdots, n-3$ through a 2-change with p and q such that

$$p = \pi_y^k(1) \ and \ q = \pi_{l_k}(\pi_y^{k+1}(1))$$

This reachability is of great theoretical importance in many of the generally applicable heuristics.

Using the notation of cyclic permutations makes aspects of the TSP easy to express. The solution space of a TSP instance is represented by the set of all cyclic permutations $\pi = (\pi(1), \cdots, \pi(n))$, where $\pi(i)$ represents the successor of city i $i = 1 \cdots n$, in the tour represented by π.

Given a cost matrix $D = [d_{ij}]$ representing the distances between all cities i and j, the cost function (the total length of the tour) is given by:

$$f(\pi) = \sum_{i=1}^{n} d_{i,\pi(i)}$$

By performing the 2-change operation, reversing the tour order between cities p and q, a new tour results with a cost equal to

$$f(\pi) + \triangle f$$

where

$$\triangle f = -d_{p,\pi(p)} - d_{\pi^{-1}(q),q} + d_{p,\pi^{-1}(q)} + d_{\pi(p),q}$$

The cost of the entire tour need not be reevaluated after each 2-change, only the above incremental difference must be calculated.

Appendix B

TSP Test Problems

To investigate the performance of specific heuristics and how different heuristics compare, four TSP instances are used as common test problems. Since relative performances of different heuristics may vary with increasing problem size, four problems used are bays29.tsp, eil51.tsp, kroC100.tsp, and kroA150.tsp of 29, 51, 100, and 150 cities respectively. These problems form part of a standard TSP library, tsplib-1.2, that can be accessed electronically using anonymous FTP at *ftp elib.zib-berlin.de* , in the directory *pub/mp-testdata/tsp/TSPLIB*.

B.1 bays29.tsp

NAME: bays29

COMMENT: 29-city problem in Bavaria

TYPE: TSP

DIMENSION: 29

TYPE: Euclidean, 2-dimensions

Node Number:x-coordinate y-coordinate			
1:1150.0 1760.0	2:630.0 1660.0	3:40.0 2090.0	4:750.0 1100.0
5:750.0 2030.0	6:1030.0 2070.0	7:1650.0 650.0	8:1490.0 1630.0
9:790.0 2260.0	10:710.0 1310.0	11:840.0 550.0	12:1170.0 2300.0
13:970.0 1340.0	14:510.0 700.0	15:750.0 900.0	16:1280.0 1200.0
17:230.0 590.0	18:460.0 860.0	19:1040.0 950.0	20:590.0 1390.0
21:830.0 1770.0	22:490.0 500.0	23:1840.0 1240.0	24:1260.0 1500.0
25:1280.0 790.0	26:490.0 2130.0	27:1460.0 1420.0	28:1260.0 1910.0
29:360.0 1980.0			

NAME: bays29.opt.tour

COMMENT: Optimum tour for bays29.tsp[1]

TYPE: TOUR

DIMENSION: 29

LENGTH: 9074.15

1-24-13-16-27-8-23-7-25-19-11-22-17-14-18-15-4-10-20-2-21-5-29-3-26-9-12-6-28

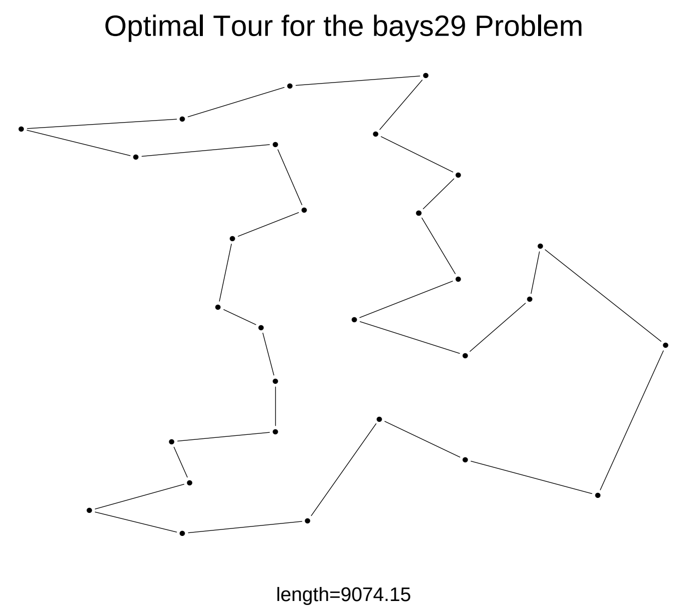

Figure B.1. Optimal tour for the bays29 TSP instance.

[1]The optimum tour given in the TSP library is incorrect. The tour given is the best tour produced by many runs of the Lin and Kernighan variable k-opt algorithm and is not guaranteed to be optimal. See chapter 2.

B.2 eil51.tsp

NAME: eil51

COMMENT: 51-city problem (Christofides/Eilon)

TYPE: TSP

DIMENSION: 51

TYPE: Euclidean, 2-dimensions

Node Number:x-coordinate y-coordinate				
1:37 52	2:49 49	3:52 64	4:20 26	5:40 30
6:21 47	7:17 63	8:31 62	9:52 33	10:51 21
11:42 41	12:31 32	13:5 25	14:12 42	15:36 16
16:52 41	17:27 23	18:17 33	19:13 13	20:57 58
21:62 42	22:42 57	23:16 57	24:8 52	25:7 38
26:27 68	27:30 48	28:43 67	29:58 48	30:58 27
31:37 69	32:38 46	33:46 10	34:61 33	35:62 63
36:63 69	37:32 22	38:45 35	39:59 15	40:5 6
41:10 17	42:21 10	43:5 64	44:30 15	45:39 10
46:32 39	47:25 32	48:25 55	49:48 28	50:56 37
51:30 40				

NAME: eil51.opt.tour

COMMENT: Optimum tour for eil51.tsp[2]

TYPE: TOUR

DIMENSION: 51

LENGTH: 428.872

1-32-11-38-5-37-17-4-18-47-12-46-51-27-6-48-23-7-43-24-14-25-11-40-19-42-44-15-45-33-39-10-49-9-30-34-21-50-16-2-29-20-35-36-3-28-31-6-8-22

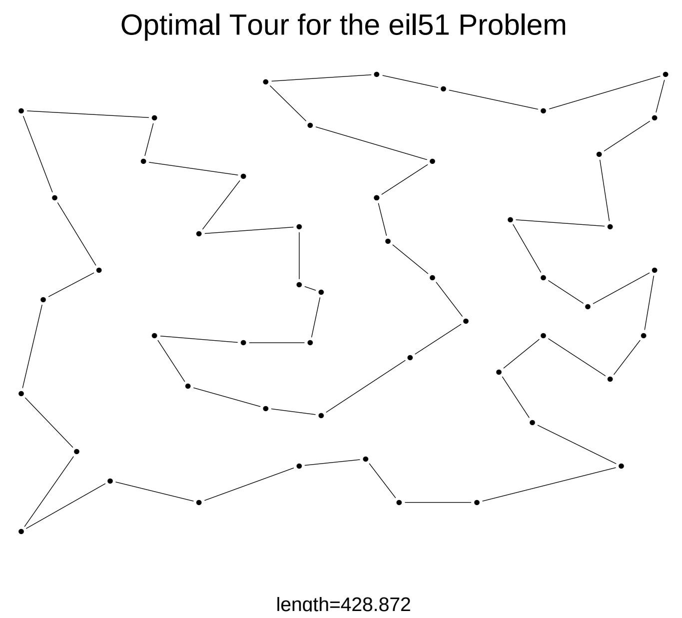

Figure B.2. Optimal tour for the eil51 TSP instance.

[2]The optimum tour given in the TSP library is incorrect. The tour given is the best tour produced by many runs of the Lin and Kernighan variable k-opt algorithm and is not guaranteed to be optimal. See chapter 2.

B.3 kroC100.tsp

NAME: kroC100

TYPE: TSP

COMMENT: 100-city problem C (Krolak/Felts/Nelson)

DIMENSION: 100

TYPE: Euclidean

Node Number:x-coordinate y-coordinate				
1:1357 1905	2:2650 802	3:1774 107	4:1307 964	5:3806 746
6:2687 1353	7:43 1957	8:3092 1668	9:185 1542	10:834 629
11:40 462	12:1183 1391	13:2048 1628	14:1097 643	15:1838 1732
16:234 1118	17:3314 1881	18:737 1285	19:779 777	20:2312 1949
21:2576 189	22:3078 1541	23:2781 478	24:705 1812	25:3409 1917
26:323 1714	27:1660 1556	28:3729 1188	29:693 1383	30:2361 640
31:2433 1538	32:554 1825	33:913 317	34:3586 1909	35:2636 727
36:1000 457	37:482 1337	38:3704 1082	39:3635 1174	40:1362 1526
41:2049 417	42:2552 1909	43:3939 640	44:219 898	45:812 351
46:901 1552	47:2513 1572	48:242 584	49:826 1226	50:3278 799
51:86 1065	52:14 454	53:1327 1893	54:2773 1286	55:2469 1838
56:3835 963	57:1031 428	58:3853 1712	59:1868 197	60:1544 863
61:457 1607	62:3174 1064	63:192 1004	64:2318 1925	65:2232 1374
66:396 828	67:2365 1649	68:2499 658	69:1410 307	70:2990 214
71:3646 1018	72:3394 1028	73:1779 90	74:1058 372	75:2933 1459
76:3099 173	77:2178 978	78:138 1610	79:2082 1753	80:2302 1127
81:805 272	82:22 1617	83:3213 1085	84:99 536	85:1533 1780
86:3564 676	87:29 6	88:3808 1375	89:2221 291	90:3499 1885
91:3124 408	92:781 671	93:1027 1041	94:3249 378	95:3297 491
96:213 220	97:721 186	98:3736 1542	99:868 731	100:960 303

NAME: kroC100.opt.tour

COMMENT: Optimum tour for kroC100.tsp

TYPE: TOUR

DIMENSION: 100

LENGTH: 20750.76

1-85-27-15-13-79-64-20-42-55-67-47-31-65-80-77-30-68-35-2-54-6-75-22-8-17-25-90-34-58-98-88-28-39-38-71-56-43-5-86-72-83-62-50-95-94-91-76-70-23-21-89-41-59-73-3-69-60-4-93-99-19-92-10-14-36-57-74-100-33-45-81-97-96-87-52-11-84-48-66-44-63-51-16-37-9-78-82-7-26-61-32-24-46-29-18-49-12-40-53

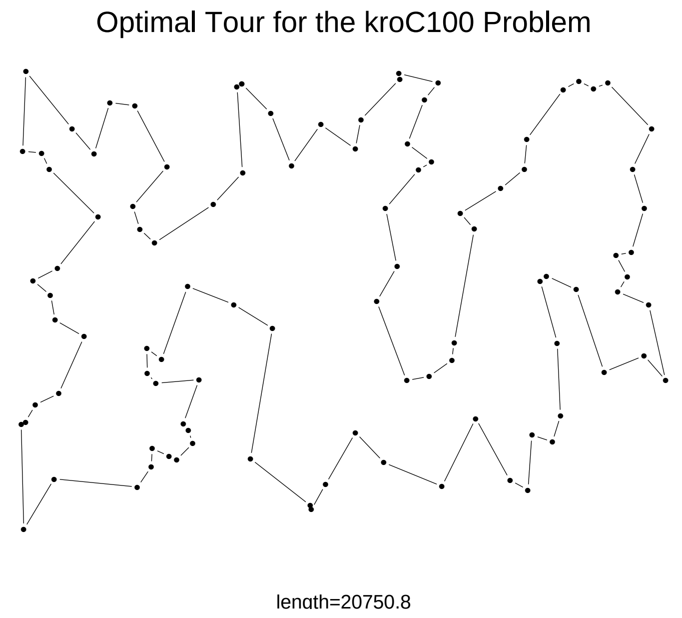

Figure B.3. Optimal tour for the kroC100 TSP instance.

B.4 kroA150.tsp

NAME: kroA150

TYPE: TSP

COMMENT: 150-city problem C (Krolak/Felts/Nelson)

DIMENSION: 150

TYPE: Euclidean

Node Number:x-coordinate y-coordinate

1:1380 939	2:2848 96	3:3510 1671	4:457 334	5:3888 666
6:984 965	7:2721 1482	8:1286 525	9:2716 1432	10:738 1325
11:1251 1832	12:2728 1698	13:3815 169	14:3683 1533	15:1247 1945
16:123 862	17:1234 1946	18:252 1240	19:611 673	20:2576 1676
21:928 1700	22:53 857	23:1807 1711	24:274 1420	25:2574 946
26:178 24	27:2678 1825	28:1795 962	29:3384 1498	30:3520 1079
31:1256 61	32:1424 1728	33:3913 192	34:3085 1528	35:2573 1969
36:463 1670	37:3875 598	38:298 1513	39:3479 821	40:2542 236
41:3955 1743	42:1323 280	43:3447 1830	44:2936 337	45:1621 1830
46:3373 1646	47:1393 1368	48:3874 1318	49:938 955	50:3022 474
51:2482 1183	52:3854 923	53:376 825	54:2519 135	55:2945 1622
56:953 268	57:2628 1479	58:2097 981	59:890 1846	60:2139 1806
61:2421 1007	62:2290 1810	63:1115 1052	64:2588 302	65:327 265
66:241 341	67:1917 687	68:2991 792	69:2573 599	70:19 674
71:3911 1673	72:872 1559	73:2863 558	74:929 1766	75:839 620
76:3893 102	77:2178 1619	78:3822 899	79:378 1048	80:1178 100
81:2599 901	82:3416 143	83:2961 1605	84:611 1384	85:3113 885
86:2597 1830	87:2586 1286	88:161 906	89:1429 134	90:742 1025
91:1625 1651	92:1187 706	93:1787 1009	94:22 987	95:3640 43
96:3756 882	97:776 392	98:1724 1642	99:198 1810	100:3950 1558
101:3477 949	102:91 1732	103:3972 329	104:198 1632	105:1806 733
106:538 1023	107:3430 1088	108:2186 766	109:1513 1646	110:2143 1611
111:53 1657	112:3404 1307	113:1034 1344	114:2823 376	115:3104 1931
116:3232 324	117:2790 1457	118:374 9	119:741 146	120:3083 1938
121:3502 1067	122:1280 237	123:3326 1846	124:217 38	125:2503 1172
126:3527 41	127:739 1850	128:3548 1999	129:48 154	130:1419 872
131:1689 1223	132:3468 1404	133:1628 253	134:382 872	135:3029 1242
136:3646 1758	137:285 1029	138:1782 93	139:1067 371	140:2849 1214
141:920 1835	142:1741 712	143:876 220	144:2753 283	145:2609 1286
146:3941 258	147:3613 523	148:1754 559	149:2916 1724	150:2445 1820

NAME: kroA150.opt.tour

COMMENT: Optimum tour for kroA150.tsp [3]

TYPE: TOUR

DIMENSION: 150

LENGTH: 26524.86

1-130-28-93-131-47-113-63-6-49-90-106-134-53-19-75-97-139-56-143-119-4-65-118-124-26-129-66-70-22-94-16-88-137-79-18-24-38-104-111-102-99-36-84-10-72-21-74-127-59-141-17-15-11-32-109-45-91-98-23-110-77-60-62-150-35-86-27-12-20-57-7-9-117-34-83-55-149-120-115-123-43-128-136-41-71-100-48-14-3-46-29-132-112-107-30-121-101-39-96-78-52-5-37-147-103-146-33-76-13-95-126-82-116-50-44-114-144-2-54-40-64-69-73-68-85-135-140-145-87-51-125-25-81-61-58-108-67-105-142-148-133-138-89-31-80-122-42-8-92

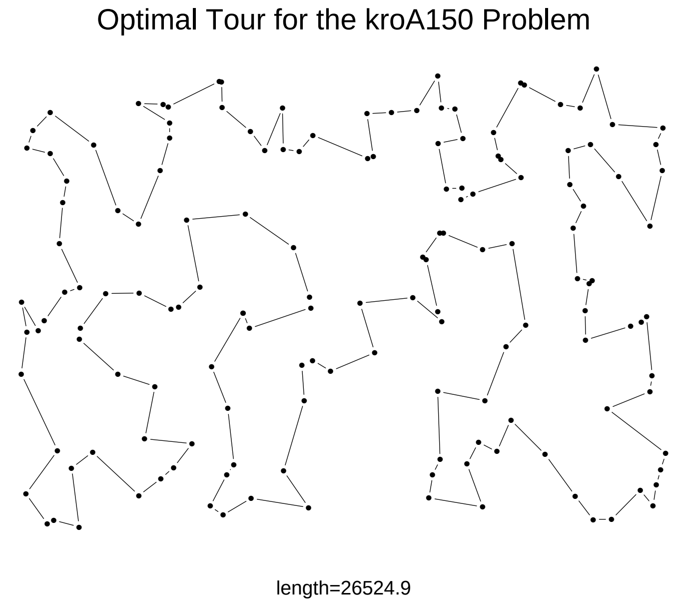

Figure B.4. Optimal tour for the kroA150 TSP instance.

[3]No optimum tour is given in the TSP library. The tour given is the best tour produced by many runs of the Lin and Kernighan variable k-opt algorithm and is not guaranteed to be optimal. See chapter 2.

Appendix C

Program Listings

This appendix is a listing of the C++ programs for the Great Deluge Algorithm, a Genetic Algorithm, and repeated local search algorithm, implemented for the Traveling Salesman Problem. The LEDA library[31] is used with GNU g++ 2.2.3 on a SPARCstation running SunOS 4.1.3 with OpenWindows 3.0 and X11 Release 5. One particular machine is used for running programs: a SPARCstation 10/512 (dual 50MHz processor) with 160MB main memory and 400MB swap space. The programs are run directly from the Unix prompt, with output redirected to relevant collection files.

C.1 The Genetic Algorithms

C.1.1 Makefile

```
#This Makefile takes the .cc files defined by SOURCES and compiles and links
#then to give the program with name defined by PROGRAM.

LEDA = /u/grads/geoff/LEDA

#LEDA is a handy library of graph, list, array and geometrical objects.

#This Makefile is not setup to compile and link source files that use the
#LEDA X11 window objects and programs.

X11 = /usr/openwin/lib

TARGET = GA
SOURCES = TSPprob popn GA

OBJECTS = $(SOURCES:%=%.o)

#OBJECTS is a list of .o filenames corresponding to SOURCES.

.SILENT:

#To see all that is happening comment out the .SILENT line. The horrific
#commands that do the compiling and linking will be shown.

.SUFFIXES: .cc .o
```

```
.KEEP_STATE:

CCC = g++

#g++ is the GNU C++ compiler and linker.

INCLUDES= -I$(LEDA)/incl

#INCLUDES is the directory in which to search for LEDA .h files.

CCFLAGS = -O2

#-O2 is the optimisation flag for the g++ compiler and linker.

CPPFLAGS=
LDFLAGS =
LEDALIBS= -L$(LEDA) -lP -lG -lL

#LEDALIBS is the directory and list of library archive files in which to
#for LEDA objects and programs. -lP means libP.a etc.

X11LIBS =

COMPILE = $(CCC) $(CCFLAGS) $(INCLUDES) $(CPPFLAGS)-c

#COMPILE is the compile command without linking.

LINK = $(CCC) $(CCFLAGS) $(CPPFLAGS) $(LDFLAGS)

#LINK is the link command

.cc.o:
	@echo Compiling -- $*
	$(COMPILE) $<

#.cc.o means for each .cc with an old .o or no .o perform the compile command.

$(TARGET): $(OBJECTS)
	@echo Linking --- $@
	$(LINK)-o $(TARGET) $(OBJECTS) $(LEDALIBS) $(X11LIBS)-lm

#(TARGET): $(OBJECTS) means link any new .o files to give a new PROGRAM file
#or link all .o files to give a PROGRAM file if no PROGRAM file exists.
```

C.1.2 GA.cc

```
//GA.cc
//Implements a Genetic Algorithm
#include "popn.h"

void WriteParams(int popsize,double converge,double cprob,double mprob,
                 int sqsize,int gen,float time)
{
  cout<<popsize;newline;
  cout<<converge;newline;
  cout<<cprob;newline;
```

```
  cout≪mprob;newline;
  cout≪sqsize;newline;
  cout≪gen;newline;
  cout≪time;newline;
  newline;
}

void GArun(int popsize,double converge,double cprob,double mprob,
           int sqsize,AseedList* sListptr,CostStruct* cstructptr)
//Given popsize, converge, cprob, mprob, and sqsize performs a
//single GA run.
{
  int gen;
  float time=used_time();
  Apopn popn(popsize,converge,cprob,mprob,sqsize,sListptr);
  popn.GenInitPop(cstructptr);
  gen=1;
  do {
    popn.GenMatePool();
    popn.GenNewPop(cstructptr);
    gen++;
  } while(!popn.Converge(gen));
  time=used_time()-time;
  popn.PrintGen();
  WriteParams(popsize,converge,cprob,mprob,sqsize,gen,time);
}

main()
//Reads in the GA parameters and allows for repeated runs of
//varying parameters to be made.
{
  int i,j,k,l,m,n,o;
  int PopSizeNum=read_int();
  array<int> PopSizes(1,PopSizeNum);
  PopSizes.read();
  int CvgNum=read_int();
  array<double> Cvgs(1,CvgNum);
  Cvgs.read();
  int CrossProbNum=read_int();
  array<double> CrossProbs(1,CrossProbNum);
  CrossProbs.read();
  int MuteProbNum=read_int();
  array<double> MuteProbs(1,MuteProbNum);
  MuteProbs.read();
  int QSizeNum=read_int();
  array<int> QSizes(1,QSizeNum);
  QSizes.read();
  int SeedNum=read_int();
  array<int> Seeds(1,(6*SeedNum));
  Seeds.read();
  ProbSize probsize;
  CostStruct cstruct;
  AseedList sList(1,6);
  loop(i,1,PopSizeNum) {
    loop(j,1,CvgNum) {
      loop(k,1,CrossProbNum) {
        loop(l,1,MuteProbNum) {
```

```
        loop(m,1,QSizeNum) {
          loop(n,1,SeedNum) {
            loop(o,1,6) {
              sList[o]=Seeds[o+6*(n-1)];
            }
            GArun(PopSizes[i],Cvgs[j],CrossProbs[k],MuteProbs[l],
                  QSizes[m],&sList,&cstruct);
          }
        }
      }
    }
  }
 }
}
```

C.1.3 popn.h

```
//popn.h
//Defines the essentials needed for the GA process, a method of
//generating the initial population, a method of implementing the
//generational replacement process, a method of selecting individuals
//for reproduction, and a method of determining convergence.
#ifndef popn_H
#define popn_H
#include "TSPprob.h"
#include<LEDA/list.h>

typedef array<int> AMatePool;
typedef array<Asoln> AGen;
typedef list<double> AnAvgQ;

class CvgStats {
  int qsize;
  double avg;
  double cvg;
  AnAvgQ newq;
  AnAvgQ oldq;
  double newsum;
  double oldsum;
 public:
  CvgStats(int,double);
  void NewAvg(double);
  void UpdateStats();
  bool DiffMeanTest();
  double GetAvg();
  int GetQsize();
};

class Apopn {
  AGen members;
  AGen newgen;
  Asoln best;
  double crossprob;
  double mutieprob;
  AMatePool pool;
```

```
  CvgStats* cstatsptr;
  RandNumGen* rngptr;
 public:
  Apopn(int,double,double,double,int,AseedList*);
  ~Apopn();
  void GenInitPop(CostStruct*);
  void GenMatePool();
  void GenNewPop(CostStruct*);
  bool Converge(int);
  void PrintGen();
};

#endif
```

C.1.4 popn.cc

```
//popn.cc
//Defines the essentials needed for the GA process, a method of
//generating the initial population, a method of implementing the
//generational replacement process, a method of selecting individuals
//for reproduction, and a method of determining convergence.
#include "popn.h"
#include <math.h>

CvgStats::CvgStats(int sqsize,double converge)
//Maintains statistics needed to determine convergence.
{
  qsize=sqsize;
  cvg=converge;
  int i;
  loop(i,1,qsize) {
    newq.append(0.0);
    oldq.append(0.0);
  }
  newsum=0;
  oldsum=0;
}

void CvgStats::NewAvg(double newavg)
//Resets the average.
{
  avg=newavg;
}

void CvgStats::UpdateStats()
//Updates the convergence statistics/
{
  double temp,rubbish;
  newq.append(avg);
  temp=newq.pop();
  newsum=newsum+avg-temp;
  oldq.append(temp);
  rubbish=oldq.pop();
  oldsum=oldsum+temp-rubbish;
}
```

```
bool CvgStats::DiffMeanTest()
//Determines if convergence has occurred.
{
  if ((fabs(newsum-oldsum)/newsum)<cvg) {return true;}
  else {return false;}
}

double CvgStats::GetAvg()
//Returns the average.
{
  return avg;
}

int CvgStats::GetQsize()
//Returns the convergence set size.
{
  return qsize;
}

Apopn::Apopn(int popsize,double converge,double cprob,double mprob,
             int sqsize,AseedList* sListptr):
members(1,popsize),
newgen(1,popsize),
pool(1,2*popsize),
cstatsptr(new CvgStats(sqsize,converge)),
rngptr(new RandNumGen(sListptr,popsize))
//Constructs a new population.
{
  crossprob=cprob;
  mutieprob=mprob;
}

Apopn::~Apopn()
//A destructor.
{
  delete cstatsptr;
  delete rngptr;
}

void Apopn::GenInitPop(CostStruct* cstructptr) {
//Generates the initial population.
  int i;
  double totcost=0;
  loop(i,1,members.high()) {
    Asoln* newsolnptr=new Asoln((*rngptr).dunifptr,cstructptr);
    members[i]=(*newsolnptr);
    if (i==1) { best=(*newsolnptr); }
    if ((*newsolnptr).cost<best.cost) {
      best=(*newsolnptr);
    }
    totcost=totcost+(*newsolnptr).cost;
    delete newsolnptr;
  }
  (*cstatsptr).NewAvg(totcost/(members.high()));
}

void Apopn::GenMatePool() {
```

```
//Fills the mating pool using a binary tournament process.
  int i,p,q;
  loop(i,1,pool.high()) {
    p=(*(*rngptr).DUmateptr)();
    q=(*(*rngptr).DUmateptr)();
    if (members[p].cost≤members[q].cost) pool[i]=p;
    else pool[i]=q;
  }
}

void Apopn::GenNewPop(CostStruct* cstructptr) {
//Generates a new population oncer the mating pool is full, by
//crossover and mutation.
  int i;
  double longlife;
  for(i=1;i≤(pool.high()-1);i+=2) {
    longlife=(*((*rngptr).CUcrossptr))();
    if (longlife<crossprob) {
      Asoln* offspringptr=new Asoln;
      members[pool[i]].Reproduce(&members[pool[i+1]],cstructptr,offspringptr,rngptr);
      newgen[(i+1)/2]=*(offspringptr);
      delete offspringptr; }
    else if (longlife<(0.5*(crossprob+1)))
      { newgen[(i+1)/2]=members[pool[i]]; }
    else { newgen[(i+1)/2]=members[pool[i+1]]; }
    if ((*(*rngptr).CUmutieptr)()<mutieprob) { newgen[(i+1)/2].Mutate(cstructptr,rngptr);
}
  }
  double totcost=0;
  loop(i,1,members.high()) {
    members[i]=newgen[i];
    if (i==1) { best=newgen[1]; }
    else if (newgen[i].cost<best.cost) {
      best=newgen[i];
    }
    totcost=totcost+newgen[i].cost;
  }
  (*cstatsptr).NewAvg(totcost/(members.high()));
  (*cstatsptr).UpdateStats();
}

bool Apopn::Converge(int genNum) {
//Determines if convergence has occurred.
  if (genNum<(2*(*cstatsptr).GetQsize())) {return false;}
  else {return (*cstatsptr).DiffMeanTest();}
}

void Apopn::PrintGen() {
//Outputs details of a generation to standard output.
  cout≪best.GetFit();newline;
  cout≪(*cstatsptr).GetAvg();newline;
}
```

C.1.5 TSPprob.h

```
//TSPprob.h
//Details specific GA operators needed in the implementation
//of the TSP.
#ifndef TSPprob_H
#define TSPprob_H
#include<LEDA/basic.h>
#include<LEDA/array.h>
#include<LEDA/plane.h>
#include "Generator.h"

typedef array<int> Aperm;
typedef array2<double> Amatrix;
typedef array<point> AposList;
typedef array<int> AseedList;

int NumCities;

class ProbSize {
 public:
  int NumberCities;
  ProbSize();
};

class RandNumGen {
 public:
  Generator* DUmateptr;
  CGenerator* CUcrossptr;
  CGenerator* CUmutieptr;
  Generator* dunifptr;
  CGenerator* unifptr;
  Generator* DUmuteptr;
  RandNumGen(AseedList*,int);
  ~RandNumGen();
};

class CostStruct {
  Amatrix* distMatrixptr;
  AposList* posListptr;
 public:
  CostStruct();
  ~CostStruct();
  double CalcCost(Aperm*);
  double GetDist(int,int);
  void PrintDist();
  void PrintPos();
};

class Asoln {
 public:
  Aperm* permptr;
  double cost;
  Asoln():permptr(new Aperm(1,NumCities)){
    cost=0;
};
  Asoln(Generator*,CostStruct*);
```

```
  Asoln(const Asoln& s):permptr(new Aperm(1,NumCities)){
    Aperm& perm=(*permptr);
    perm=(*s.permptr);
    cost=s.cost;
};
  ~Asoln() { delete permptr;};
  Asoln& operator=(const Asoln&);
  void Reproduce(Asoln*,CostStruct*,Asoln*,RandNumGen*);
  void Mutate(CostStruct*,RandNumGen*);
  double GetFit();
  friend void Print(const Asoln& s,ostream& os=cout);
  friend void Read(Asoln& s,istream& is=cin);
  friend int compare(const Asoln& s,const Asoln& t);
  LEDA_MEMORY(Asoln)
};
inline void Print(const Asoln& s, ostream& os=cout)
{
  (*s.permptr).print();newline;
  os<<s.cost;newline;
}
inline void Read(Asoln& s, istream& is=cin)
{
  (*s.permptr).read();
  is>>s.cost;
}
inline int compare(const Asoln& s, const Asoln& t)
{
  return 0;
}
LEDA_TYPE_PARAMETER(Asoln)

#endif
```

C.1.6 TSPprob.cc

```
//TSPprob.cc
//Details specific GA operators needed in the implementation
//of the TSP.
#include "TSPprob.h"
#include<LEDA/set.h>

typedef set<int> intset;
typedef array<intset*> EdgeMap;

ProbSize::ProbSize()
//Defines the problem dimension for the TSP.
{
  NumberCities=read_int();
  NumCities=NumberCities;
}

RandNumGen::RandNumGen(AseedList* sList,int popsize):
DUmateptr(new Generator(1,popsize,(*sList)[3])),
CUcrossptr(new CGenerator(0,1,(*sList)[4])),
CUmutieptr(new CGenerator(0,1,(*sList)[5])),
dunifptr(new Generator(1,NumCities,(*sList)[1])),
```

```
unifptr(new CGenerator(0,1,(*sList)[2])),
DUmuteptr(new Generator(1,NumCities,(*sList)[6]))
//Constructs a random number generator.
{
}

RandNumGen::~RandNumGen()
//A destructor.
{
  delete DUmateptr;
  delete CUcrossptr;
  delete CUmutieptr;
  delete dunifptr;
  delete unifptr;
  delete DUmuteptr;
}

double Asoln::GetFit()
//Returns the length of a tour.
{
  return cost;
}

Asoln::Asoln(Generator* dunifptr,CostStruct* cstructptr)
:permptr(new Aperm(1,NumCities))
//Constructs a TSP tour and determines its cost.
{
  Aperm& perm=(*permptr);
  int i,j,p,q,temp;
  loop(i,1,NumCities) {
    perm[i]=i;
  }
  loop(j,1,NumCities) {
    p=(*dunifptr)();
    q=(*dunifptr)();
    temp=perm[p];
    perm[p]=perm[q];
    perm[q]=temp;
  }
  cost=(*cstructptr).CalcCost(&perm);
}

Asoln& Asoln::operator=(const Asoln& copysoln)
//Copies a TSP solution.
{
  Aperm& perm=(*permptr);
  int i;
  loop(i,1,NumCities) {
    perm[i]=(*copysoln.permptr)[i];}
  cost=copysoln.cost;
  return *this;
}

class AnEdgeMap {
//The edge map required for the edge recombination technique.
public:
  EdgeMap emap;
```

```
  AnEdgeMap():emap(1,NumCities) {
    //A constructor.
    int i;
    loop(i,1,NumCities) {
      emap[i]=new intset;
    }
  };
  ~AnEdgeMap() {
    //A destructor.
    int i;
    loop(i,1,NumCities) {
      delete emap[i];
    }
  };
  void MakeEdgeMap(Aperm* P1ptr,Aperm* P2ptr);
};

void AnEdgeMap::MakeEdgeMap(Aperm* P1ptr,Aperm* P2ptr)
//Makes an edge map given two TSP tours.
{
  int fromP1,fromP2,i;
  Aperm& P1=*P1ptr;
  Aperm& P2=*P2ptr;
  fromP1=P1[1];
  fromP2=P2[1];
  (*emap[fromP1]).insert(P1[2]);
  (*emap[fromP1]).insert(P1[NumCities]);
  (*emap[fromP2]).insert(P2[2]);
  (*emap[fromP2]).insert(P2[NumCities]);
  loop(i,2,NumCities-1) {
    fromP1=P1[i];
    fromP2=P2[i];
    (*emap[fromP1]).insert(P1[i-1]);
    (*emap[fromP1]).insert(P1[i+1]);
    (*emap[fromP2]).insert(P2[i-1]);
    (*emap[fromP2]).insert(P2[i+1]);
  }
  fromP1=P1[NumCities];
  fromP2=P2[NumCities];
  (*emap[fromP1]).insert(P1[NumCities-1]);
  (*emap[fromP1]).insert(P1[1]);
  (*emap[fromP2]).insert(P2[NumCities-1]);
  (*emap[fromP2]).insert(P2[1]);
}

int GetNext(intset* mysetptr,RandNumGen* rngptr)
//Returns a random number from a set of arbitrary size.
{
  RandNumGen& rng=*rngptr;
  intset& myset=*mysetptr;
  int intselection=int((*rng.unifptr)()*myset.size())+1;
  int anint,theint,i;
  i=1;
  theint=myset.choose();
  forall(anint,myset) {
    if (i==intselection) {theint=anint;}
    i++;
```

```
  }
  return theint;
}

void Asoln::Reproduce(Asoln* mateptr,CostStruct* cstructptr,
            Asoln* childptr,RandNumGen* rngptr)
//Implements the edge recombination crossover operator.
{
  Aperm& perm=(*permptr);
  AnEdgeMap map;
  Aperm& P1=perm;
  Aperm& P2=(*(*mateptr).permptr);
  CostStruct& cstruct=*cstructptr;
  RandNumGen& rng=*rngptr;
  Aperm* newpermptr=new Aperm(1,NumCities);
  map.MakeEdgeMap(permptr,(*mateptr).permptr);
  int i,msize,tsize,city;
  intset minsize;
  minsize.insert(P1[1]);
  minsize.insert(P2[1]);
  (*newpermptr)[1]=GetNext(&minsize,rngptr);
  loop(i,1,NumCities-1) {
    if ((*map.emap[(*newpermptr)[i]]).empty()) {
      intset remcities;
      int x;
      loop(x,1,NumCities) { remcities.insert(x); }
      loop(x,1,i) { remcities.del((*newpermptr)[x]); }
      (*newpermptr)[i+1]=GetNext(&remcities,rngptr);
    }
    else {
      minsize.clear();
      msize=NumCities;
      int j;
      forall(city,(*map.emap[(*newpermptr)[i]])) {
        (*map.emap[city]).del((*newpermptr)[i]);
        tsize=(*map.emap[city]).size();
        if (tsize<msize) {
          msize=tsize;
          minsize.clear();
          minsize.insert(city);
        }
        else if (tsize==msize) {
          minsize.insert(city);
        }
      }
      (*newpermptr)[i+1]=GetNext(&minsize,rngptr);
    }
  }
  double newcost=cstruct.CalcCost(newpermptr);
  (*(*childptr).permptr)=(*newpermptr);
  (*childptr).cost=newcost;
  delete newpermptr;
}

void Asoln::Mutate(CostStruct* cstructptr,RandNumGen* rngptr)
//Implements a swap mutation operator.
{
```

```
  int p,q,temp;
  Aperm& perm=(*permptr);
  CostStruct& cstruct=(*cstructptr);
  RandNumGen& rng=(*rngptr);
  do {
    p=(*rng.DUmuteptr)();
    q=(*rng.DUmuteptr)();
  } while (p==q);
  temp=perm[p];
  perm[p]=perm[q];
  perm[q]=temp;
  cost=cstruct.CalcCost(&perm);
}

CostStruct::CostStruct():
distMatrixptr(new Amatrix(1,NumCities,1,NumCities)),
posListptr(new AposList(1,NumCities))
//Reads in city coordinates and constructs an associated distance
//matirx.
{
  AposList& posList=*posListptr;
  Amatrix& distMatrix=*distMatrixptr;
  int x,y,i;
  posList.read();
  int row,col;
  loop(row,1,NumCities) {
    loop(col,1,NumCities) {
      distMatrix(row,col)=posList[row].distance(posList[col]);
    }
  }
}

CostStruct::~CostStruct()
//A destructor.
{
  delete distMatrixptr;
  delete posListptr;
}

double CostStruct::GetDist(int city1,int city2)
//Returns the distance between two cities.
{
  return (*distMatrixptr)(city1,city2);
}

void CostStruct::PrintDist()
//Prints the distance matrix to standard output.
{
  int i,j;
  loop(i,1,NumCities) {
    loop(j,1,NumCities) {
      cout<<(*distMatrixptr)(i,j)<<"  ";
    }
    newline;
  }
}
```

```
void CostStruct::PrintPos()
//Prints the city coordinates to standard output.
{
  int i;
  loop(i,1,NumCities) {
    cout<<(*posListptr)[i]<<" ";
    newline;
  }
}

double CostStruct::CalcCost(Aperm* permptr)
//Calculates the length of a tour.
{
  int i;
  Aperm& perm=*permptr;
  Amatrix& dMatrix=*distMatrixptr;
  double cost=0;
  loop(i,1,NumCities-1) {
    cost=dMatrix(perm[i],perm[i+1])+cost;
  }
  cost=dMatrix(perm[NumCities],perm[1])+cost;
  return cost;
}
```

C.1.7 Generator.h

```
//Generator.h
#ifndef GENERATOR_H
#define GENERATOR_H
//Constructs a random number generator for discrete uniform numbers
//and continuous random numbers on the interval [0,1]
#include <ACG.h>
#include <DiscUnif.h>
#include <Uniform.h>

class Generator{
  ACG* RNGptr;
  DiscreteUniform* DUptr;
public:
  Generator(int low=0,int high=1,int seed=1){
    RNGptr = new ACG(seed, 30);
    DUptr = new DiscreteUniform(low,high,RNGptr);
  }
  ~Generator(){
    delete RNGptr;
    delete DUptr;
  }
  int operator()(){
    return int((*DUptr)());
  }
};

class CGenerator{
  ACG* RNGptr;
  Uniform* Uptr;
public:
```

```
  CGenerator(double low=0,double high=1,int seed=1){
    RNGptr = new ACG(seed, 30);
    Uptr = new Uniform(low,high,RNGptr);
  }
  ~CGenerator(){
    delete RNGptr;
    delete Uptr;
  }
  double operator()(){
    return (*Uptr)();
  }
};

#endif
```

C.2 The Great Deluge Algorithm

The Makefile used in the compilation and linking of the Great Deluge Programs is the same as that used for Genetic Algorithms, with the source and target names changed accordingly.

C.2.1 GDA.cc

```
//GDA.cc
//Great Deluge Algorithm implemented for minimisation
#include<LEDA/basic.h>
#include "TSP.h"

double InitLevel(SolnObj* solnptr,double LEVEL) {
  return (*solnptr).GetCost();
}

void GDArun(double UP,int MaxIter,int QualityLim,double LEVEL,
            CostStruct* cstructptr,Generator* dunifptr)
//Given a value of UP, MaxIter, QualityLim, and LEVEL peroms a
//single run of the GDA algorithm for a generalised problem.
{
  float time=used_time();
  int iteration;
  int increase;
  SolnObj* solnptr=new SolnObj(cstructptr);
  SolnObj* bestptr=new SolnObj(solnptr);
  SolnObj& soln=*solnptr;
  SolnObj& best=*bestptr;
  double bestcost=best.GetCost();
  double newsolncost;
  LEVEL=InitLevel(solnptr,LEVEL);
  iteration=0;
  increase=0;
  do {
    iteration++;
    SolnObj* newsolnptr=new SolnObj(solnptr);
    SolnObj& newsoln=*newsolnptr;
    newsoln.PerturbSoln(cstructptr,dunifptr);
```

```
    newsolncost=newsoln.GetCost();
    if (newsolncost<LEVEL) {
      soln.CopySoln(newsolnptr);
      LEVEL=LEVEL-UP*(LEVEL-newsolncost);
      if (newsolncost<bestcost) {
        best.CopySoln(newsolnptr);
        bestcost=best.GetCost();
      }
      increase=iteration;
    }
    delete newsolnptr;
  } while ((iteration<MaxIter)&&((iteration-increase)<QualityLim));
  time=used_time()-time;
  cout<<iteration;newline;
  cout<<bestcost;newline;
  cout<<time;newline;
  cout<<UP;newline;
  cout<<MaxIter;newline;
  cout<<QualityLim;newline;
  cout<<LEVEL;newline;
}

main()
//Reads in the GDA parameters allowing for several runs with
//various parameter settings.
{
  int i,j,k;
  int MaxIter;
  double LEVEL;
  int numUp=read_int();
  array<double> UPS(1,numUp);
  UPS.read();
  int numReps=read_int();
  array<int> seeds(1,numReps);
  seeds.read();
  MaxIter=read_int();
  int numQual=read_int();
  array<int> QualityLim(1,numQual);
  QualityLim.read();
  LEVEL=0.0;
  ProbSize* psptr=new ProbSize;
  CostStruct* cstructptr=new CostStruct(psptr);
  loop(i,1,numUp) {
    loop(j,1,numReps) {
      loop(k,1,numQual) {
        Generator* dunifptr=new Generator(1,(*psptr).NumberCities,seeds[j]);
        GDArun(UPS[i],MaxIter,QualityLim[k],LEVEL,cstructptr,dunifptr);
        delete dunifptr;
      }
    }
  }
}
```

C.2.2 TSP.h

```
//TSP.h
//A Traveling Salesman module implementing a neighbourhood move and
//a cost function.
#ifndef TSP_H
#define TSP_H
#include<LEDA/basic.h>
#include<LEDA/array.h>
#include<LEDA/plane.h>
#include "Generator.h"

typedef array<int> Atourperm;
typedef array2<double> Amatrix;
typedef array<point> AposList;

class ProbSize {
 public:
  int NumberCities;
  ProbSize();
};

class CostStruct {
  Amatrix* distMatrixptr;
  AposList* posListptr;
 public:
  CostStruct(ProbSize*);
  ~CostStruct();
  double GetCost(int,int);
  void PrintDist();
  void PrintPos();
};

class SolnObj {
  double dist;
  Atourperm* tourptr;
 public:
  int Succ(Atourperm,int,int);
  SolnObj(CostStruct*);
  SolnObj(SolnObj*);
  ~SolnObj();
  void ShowSoln();
  double GetCost();
  void CopySoln(SolnObj*);
  int Succ(Atourperm*,int,int);
  void PerturbSoln(CostStruct*,Generator*);
  double CalcCost(CostStruct*);
};


#endif
```

C.2.3 TSP.cc

```
//TSP.cc
//A Traveling Salesman module implementing a neighbourhood move and
```

```
//a cost function. The neighbourhood move implemened is the 2-opt
//operation with an incremental cost update.
#include "TSP.h"
#include<math.h>

int NumCities;

ProbSize::ProbSize()
//Determines the dimension of the TSP.
{
  NumberCities=read_int();
}


CostStruct::CostStruct(ProbSize* psptr):
distMatrixptr(new Amatrix(1,(*psptr).NumberCities,1,(*psptr).NumberCities)),
posListptr(new AposList(1,(*psptr).NumberCities))
//Reads in the 2D coordinates of the TSP abd constructs an associated
//cost matrix.
{
  NumCities=(*psptr).NumberCities;
  AposList& posList=*posListptr;
  Amatrix& distMatrix=*distMatrixptr;
  int x,y,i;
  posList.read();
  int row,col;
  loop(row,1,NumCities) {
    loop(col,1,NumCities) {
      distMatrix(row,col)=posList[row].distance(posList[col]);
    }
  }
}

CostStruct::~CostStruct()
//A destructor.
{
  delete distMatrixptr;
  delete posListptr;
}

double CostStruct::GetCost(int x,int y)
//Returns the distance between two cities.
{
  return (*distMatrixptr)(x,y);
}

void CostStruct::PrintDist()
//Prints the distance matrix to standard output.
{
  int i,j;
  loop(i,1,NumCities) {
    loop(j,1,NumCities) {
      cout<<(*distMatrixptr)(i,j)<<"  ";
    }
    newline;
  }
}
```

```
void CostStruct::PrintPos()
//Prints the city coordinates to standard output.
{
  int i;
  loop(i,1,NumCities) {
    cout<<(*posListptr)[i]<<" ";
    newline;
  }
}

SolnObj::SolnObj(CostStruct* dmatrixptr)
:tourptr(new Atourperm(1,NumCities))
//Constructs a tour of the cities and determines the cost.
{
  int i;
  loop(i,1,NumCities) {
    (*tourptr)[i]=int(fmod(i,NumCities)+1);
  }
  dist=CalcCost(dmatrixptr);
}

SolnObj::SolnObj(SolnObj* copyptr)
:tourptr(new Atourperm(1,NumCities))
//Copys a solution to the TSP.
{
  CopySoln(copyptr);
}

SolnObj::~SolnObj()
//A destructor.
{
  delete tourptr;
}

void SolnObj::ShowSoln()
//Outputs a TSP solution to standard output.
{
  Atourperm& tour=*tourptr;
  int i,pos;
  cout<<"Tour:  1 ";
  pos=tour[1];
  loop(i,1,NumCities) {
    cout<<pos<<" ";
    pos=tour[pos];
  }
  newline;
  cout<<"With distance:  "<<dist;newline;
}

double SolnObj::GetCost()
//Returns the tour length.
{
  return dist;
}

void SolnObj::CopySoln(SolnObj* copyptr)
```

```
//Copies a TSP solution.
{
  Atourperm& tour=*tourptr;
  SolnObj& copy=*copyptr;
  tour=(*(copy.tourptr));
  dist=copy.dist;
}

int SolnObj::Succ(Atourperm* permptr,int p,int k)
//Determines the city that is k steps away from the city p in the
//current tour.
{
  Atourperm& perm=*permptr;
  if (k==1) { return perm[p]; }
  else {
    int i,pos,newpos;
    pos=perm[p];
    loop(i,1,k-1) {
      newpos=perm[pos];
      pos=newpos;
    }
    return newpos;
  }
}

void SolnObj::PerturbSoln(CostStruct* cstructptr,Generator* dunifptr)
//Performs the 2-opt neighbourhood move on the current tour to form a
//new tour and evaluates its cost incrementally.
{
  if (NumCities>3) {
    Atourperm& tour=*tourptr;
    CostStruct& cstruct=*cstructptr;
    int p,q,r,k,succ;
    do {
      p=(*dunifptr)();
      q=(*dunifptr)();
      k=0;
      do {
        k=k+1;
        succ=Succ(&tour,p,k);
        } while (succ≠q);
    } while ((k≤2)||(k==NumCities));
    Atourperm temp(tour);
    tour[p]=Succ(&temp,p,k-1);
    tour[Succ(&temp,p,1)]=Succ(&temp,p,k);
    loop(r,2,k-1) {
      tour[Succ(&temp,p,r)]=Succ(&temp,p,r-1);
    }
    int predq=Succ(&temp,q,NumCities-1);
    int succp=temp[p];
    double dist1=cstruct.GetCost(p,succp);
    double dist2=cstruct.GetCost(predq,q);
    double dist3=cstruct.GetCost(p,predq);
    double dist4=cstruct.GetCost(succp,q);
    double change=dist3+dist4-dist1-dist2;
    dist=dist+change;
  }
```

```
}

double SolnObj::CalcCost(CostStruct* cstructptr)
//Calculates the length of a tour.
{
  int i,pos;
  Atourperm& tour=*tourptr;
  CostStruct& cstruct=*cstructptr;
  pos=1;
  double cost=0;
  loop(i,1,NumCities) {
    cost=cstruct.GetCost(pos,tour[pos])+cost;
    pos=tour[pos];
  }
  return cost;
}
```

C.2.4 Generator.h

```
//Generator.h
#ifndef GENERATOR_H
#define GENERATOR_H
//Constructs a random number genertor that can be queried for
//discrete uniform numbers.
#include <ACG.h>
#include <DiscUnif.h>

class Generator{
  ACG* RNGptr;
  DiscreteUniform* DUptr;
public:
  Generator(int low=0,int high=1,int seed=1){
    RNGptr = new ACG(seed, 30);
    DUptr = new DiscreteUniform(low,high,RNGptr);
  }
  ~Generator(){
    delete RNGptr;
    delete DUptr;
  }
  int operator()(){
    return int((*DUptr)());
  }
};

#endif
```

C.3 Repeated Local Search

The Makefile used in the compilation and linking of the repeated local search programs is the same as that used for Genetic Algorithms, with the source and target names changed accordingly. Also the Generator module is the same as that used for Genetic Algorithms, so is not given below.

C.3.1 BigHillClimb.cc

```
//BigHillClimb.cc
//Repeated Hill Climb for minimisation.
#include<LEDA/basic.h>
#include "TSP.h"

void BigHillClimbRun(int seed,double runtime,CostStruct* cstructptr,
                     int num)
//Given a runtime performs a single run of repeated local search
//by restarting the local search procedure (steepest descent variation)
//until the specified time interval has ellapsed.
{
  Generator* dunifptr=new Generator(1,num,seed);
  double totaltime=used_time();
  SolnObj thebest;
  do {
    SolnObj soln(cstructptr,dunifptr);
    SolnObj best(soln);
    int p,q;
    double newsolncost,bestcost;
    bool optimum=false;
    do {
      int p,q;
      best=soln;
      bestcost=best.GetCost();
      loop(p,1,num-1) {
        loop(q,p+1,num) {
          SolnObj newsoln(soln);
          newsoln.PerturbSoln(cstructptr,p,q);
          newsolncost=newsoln.GetCost();
          if(newsolncost<bestcost) {
            best=newsoln;
            bestcost=best.GetCost();
          }
        }
      }
      if(best==soln) { optimum=true; }
      soln=best;
    } while(!optimum);
    double solncost=soln.GetCost();
    double thebestcost=thebest.GetCost();
    if((solncost<thebestcost)|(thebestcost==0)) { thebest=soln; };
  } while((used_time()-totaltime)<runtime);
  cout<<(used_time()-totaltime);newline;
  thebest.ShowSoln();
}

main()
//Reads in several run times and allows for multiple runs of the hill
//climbing procedure.
{
  int runtimeNum=read_int();
  array<double> RunTimes(1,runtimeNum);
  RunTimes.read();
  int seedNum=read_int();
  array<int> SeedNums(1,seedNum);
```

```
  SeedNums.read();
  ProbSize* psptr=new ProbSize;
  int num=(*psptr).NumberCities;
  CostStruct* cstructptr=new CostStruct(psptr);
  int i,j;
  loop(i,1,runtimeNum) {
    loop(j,1,seedNum) {
      BigHillClimbRun(SeedNums[j],RunTimes[i],cstructptr,num);
    }
  }
}
```

C.3.2 TSP.h

```
//TSP.h
//Defines a neighbourhood structure and a cost structure for the TSP.
#ifndef TSP_H
#define TSP_H
#include<LEDA/basic.h>
#include<LEDA/array.h>
#include<LEDA/plane.h>
#include "Generator.h"

typedef array<int> Atourperm;
typedef array2<double> Amatrix;
typedef array<point> AposList;

int NumCities;

class ProbSize {
 public:
  int NumberCities;
  ProbSize();
};

class CostStruct {
  Amatrix* distMatrixptr;
  AposList* posListptr;
 public:
  CostStruct(ProbSize*);
  ~CostStruct();
  double GetCost(int,int);
  void PrintDist();
  void PrintPos();
};

class SolnObj {
public:
  double dist;
  Atourperm* tourptr;
  SolnObj():tourptr(new Atourperm(1,NumCities)){
    int i;
    loop(i,1,NumCities) {
      (*tourptr)[i]=int(fmod(i,NumCities)+1);
    }
```

```
    dist=0;
  };
  SolnObj(CostStruct*,Generator*);
  SolnObj(const SolnObj& s):tourptr(new Atourperm(1,NumCities)){
    Atourperm& tour=(*tourptr);
    tour=(*s.tourptr);
    dist=s.dist;
  };
  ~SolnObj() { delete tourptr; };
  SolnObj& operator=(const SolnObj&);
  int operator==(const SolnObj&);
  void ShowSoln();
  double GetCost();
  int Succ(Atourperm*,int,int);
  void PerturbSoln(CostStruct*,int,int);
  double CalcCost(CostStruct*);
  friend void Print(const SolnObj& s,ostream& os=cout);
  friend void Read(SolnObj& s,istream& is=cin);
  friend int compare(const SolnObj& s,const SolnObj& t);
  LEDA_MEMORY(SolnObj)
};
inline void Print(const SolnObj& s, ostream& os=cout)
{
  (*s.tourptr).print();newline;
  os<<s.dist;newline;
}
inline void Read(SolnObj& s, istream& is=cin)
{
  (*s.tourptr).read();
  is>>s.dist;
}
inline int compare(const SolnObj& s, const SolnObj& t)
{
  return 0;
}
LEDA_TYPE_PARAMETER(SolnObj)

#endif
```

C.3.3 TSP.cc

```
//TSP.cc
//Defines a neighbourhood structure and a cost structure for the TSP.
#include "TSP.h"
#include<math.h>

ProbSize::ProbSize()
//Determines the problem dimension.
{
  NumberCities=read_int();
  NumCities=NumberCities;
}


CostStruct::CostStruct(ProbSize* psptr):
```

```
distMatrixptr(new Amatrix(1,(*psptr).NumberCities,1,(*psptr).NumberCities)),
posListptr(new AposList(1,(*psptr).NumberCities))
//Reads in city coordinates and constructs a corresponding distance
//matrix.
{
  NumCities=(*psptr).NumberCities;
  AposList& posList=*posListptr;
  Amatrix& distMatrix=*distMatrixptr;
  int x,y,i;
  posList.read();
  int row,col;
  loop(row,1,NumCities) {
    loop(col,1,NumCities) {
      distMatrix(row,col)=posList[row].distance(posList[col]);
    }
  }
}

CostStruct::~CostStruct()
//A destructor.
{
  delete distMatrixptr;
  delete posListptr;
}

double CostStruct::GetCost(int x,int y)
//Returns the distance between two cities.
{
  return (*distMatrixptr)(x,y);
}

void CostStruct::PrintDist()
//Writes the distance matrix to standard output.
{
  int i,j;
  loop(i,1,NumCities) {
    loop(j,1,NumCities) {
      cout<<(*distMatrixptr)(i,j)<<"  ";
    }
    newline;
  }
}

void CostStruct::PrintPos()
//Writes the city coordinates to standard output.
{
  int i;
  loop(i,1,NumCities) {
    cout<<(*posListptr)[i]<<"  ";
    newline;
  }
}

SolnObj::SolnObj(CostStruct* dmatrixptr,Generator* dunifptr)
:tourptr(new Atourperm(1,NumCities))
//Constructs a random initial tour and determines its length.
{
```

```
  int i,j,p,q,temp;
  Atourperm& tour=(*tourptr);
  loop(i,1,NumCities) {
    tour[i]=int(fmod(i,NumCities)+1);
  }
  dist=0;
  loop(j,1,NumCities) {
    p=(*dunifptr)();
    q=(*dunifptr)();
    (*this).PerturbSoln(dmatrixptr,p,q);
  }
  dist=CalcCost(dmatrixptr);
}

SolnObj& SolnObj::operator=(const SolnObj& copysoln)
//Copies a TSP solution.
{
  Atourperm& tour=(*tourptr);
  int i;
  loop(i,1,NumCities) {
    tour[i]=(*copysoln.tourptr)[i];}
  dist=copysoln.dist;
  return *this;
}

int SolnObj::operator==(const SolnObj& copysoln)
//Copies a TSP solution.
{
  Atourperm& tour=(*tourptr);
  Atourperm& copy=(*copysoln.tourptr);
  int equal=1;
  int i;
  loop(i,1,NumCities) {
    if(tour[i]≠copy[i]) { equal=0; }
  }
  return equal;
}

void SolnObj::ShowSoln()
{
//Writes the tour length to standard output.
  cout≪dist;newline;
}

double SolnObj::GetCost()
//Returns the tour length.
{
  return dist;
}

int SolnObj::Succ(Atourperm* permptr,int p,int k)
//Returns the k'th city from p in the current tour.
{
  Atourperm& perm=*permptr;
  if (k==1) { return perm[p]; }
  else {
    int i,pos,newpos;
```

```
    pos=perm[p];
    loop(i,1,k-1) {
      newpos=perm[pos];
      pos=newpos;
    }
    return newpos;
  }
}

void SolnObj::PerturbSoln(CostStruct* cstructptr,int p,int q)
//Performs the 2-opt neighbourhood move given a p and a q and
//determines the new cost incrementally.
{
  Atourperm& tour=*tourptr;
  CostStruct& cstruct=*cstructptr;
  Atourperm temp(tour);
  int k,r,succ;
  k=0;
  do {
    k=k+1;
    succ=Succ(&tour,p,k);
  } while (succ≠q);
  if(k>2) {
    tour[p]=Succ(&temp,p,k-1);
    tour[Succ(&temp,p,1)]=Succ(&temp,p,k);
    loop(r,2,k-1) {
      tour[Succ(&temp,p,r)]=Succ(&temp,p,r-1);
    }
    int predq=Succ(&temp,q,NumCities-1);
    int succp=temp[p];
    double dist1=cstruct.GetCost(p,succp);
    double dist2=cstruct.GetCost(predq,q);
    double dist3=cstruct.GetCost(p,predq);
    double dist4=cstruct.GetCost(succp,q);
    double change=dist3+dist4-dist1-dist2;
    dist=dist+change;
  }
}

double SolnObj::CalcCost(CostStruct* cstructptr)
//Calculates the cost of a TSP tour.
{
  int i,pos;
  Atourperm& tour=*tourptr;
  CostStruct& cstruct=*cstructptr;
  pos=1;
  double cost=0;
  loop(i,1,NumCities) {
    cost=cstruct.GetCost(pos,tour[pos])+cost;
    pos=tour[pos];
  }
  return cost;
}
```

References

[1] J. Antonisse. A New Interpretation of Schema Notation That Overturns the Binary Encoding Constraint. In *[37]*.

[2] M. Ball and M. Magazine. The Design and Analysis of Heuristics. *Networks*, 11:215–219, 1981.

[3] David Beasley, David R. Bull, and Ralph R. Martin. An Overview of Genetic Algorithms: Part 1. Fundamentals. *University Computing*, 15(2):58–69, 1993.

[4] L. B. Booker. Improving Search in Genetic Algorithms. In L. Davis, editor, *[7]*. Morgan Kauffmann.

[5] V. Černy. Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, 1985.

[6] Dantzig, Fulkerson, and Johnson. Solution of a Large–Scale Traveling–Salesman Problem. *Journal of the Operations Research Society of America*, 1954.

[7] L. Davis, editor. *Genetic Algorithms and Simulated Annealing*. Morgan Kauffmann, 1987.

[8] L. Davis. Job Shop Scheduling With Genetic Algorithms. In J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 136–140. Lawrence Erlbaum Associates, 1988. (original proceedings 1985).

[9] L. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.

[10] Gunter Dueck and Tobias Scheuer. Threshold Accepting: A General Purpose Optimization Algorithm Superior to Simulated Annealing. *Journal of Computational Physics*, 90:161–175, 1990.

[11] Gunter Dueck. New Optimization Heuristics: The Great Deluge Algorithm and the Record-to-Record Travel. Technical report, IBM Germany, Heidelberg Scientific Center, 1990.

[12] L. J. Eshelman, R. A. Caruana, and J. D. Schaffer. Biases in the Crossover Landscape. In *[37]*.

[13] A. Fairley. Comparison of Methods of Choosing the Crossover Point in the Genetic Crossover Operation. Technical report, Department of Computer Science, University of Liverpool, 1991.

[14] Michael R. Garey and David S. Johnson. *Computers and Intractability:A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[15] Fred Glover and Manuel Laguna. Tabu Search. In *[36]*, pages 70–150. John Wiley & Sons, Inc.

[16] Fred Glover. Future paths for Integer Programming and Links to Artificial Intelligence. *Computers and Operations Research*, 5:533–549, 1986.

[17] Fred Glover. Tabu Search: A Tutorial. *Interfaces*, 20(4):74–94, 1990.

[18] D. E. Goldberg and R. Lingle. Alleles, Loci, and the Traveling Salesman Problem. In *[21]*, 1985.

[19] David E. Goldberg. *Genetic Algorithms in Search, Optimzation, and Machine Learning*. Addison-Wesley Publishing Company, Inc, 1989.

[20] J. J. Grefenstette. Incorporating Problem Specific Knowledge into Genetic Algorithms. In *[7]*. Morgan Kauffmann.

[21] J. J. Grefenstette, editor. *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*. Lawrence Erlbaum Associates, 1985.

[22] J. J. Grefenstette, editor. *Proceedings of the Second International Conference on Genetic Algorithms*. Lawrence Erlbaum Associates, 1987.

[23] M. Hobbs. Private correspondence. 1994.

[24] J.H. Holland. *Adaption in Natural and Artificial Systems*. MIT Press, 1975.

[25] D. S. Johnson and C. H. Papadimitriou. Computational Complexity. In *[27]*. John Wiley & Sons.

[26] S. Kirkpatrick, Jr. C.D. Gelatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–679, 1983.

[27] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons, 1985.

[28] S. Lin and B. W. Kernighan. An Effective Heuristic Algorithm for the Traveling Salesman Problem. *Operations Research*, 21:498–516, 1973.

[29] S. Lin. Computer Solutions of the Traveling Salesman Problem. *Bell System Technical Journal*, 44:2245–2269, 1965.

[30] Nicholas Metropolis, Arianna W. RosenBluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1955.

[31] S. Näher. *LEDA User Manual:Version 3.0*. Max-Planck-Institut für Informatik, Im Stadtwald, D-6600 Saarbrücken. Available by anonymous ftp from **ftp.cs.uni-sb.de** in /pub/LEDA.

[32] K.J. Nurmela. Constructing Combinatorial Designs by Local Search. Technical report, Helsinki University of Technology: Digital Systems Laboratory, 1993.

[33] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A Study of Permutation Crossover Operators on the Traveling Salesman Problem. In *[22]*.

[34] N. J. Radcliffe. Equivalence Class Analysis of Genetic Algorithms. *Complex Systems*, 5:183–205, 1991.

[35] C. R. Reeves. Genetic algorithms. In *[36]*, pages 151–196. John Wiley & Sons, Inc.

[36] Colin R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems*. John Wiley & Sons, Inc, 1993.

[37] J.D. Schaffer, editor. *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[38] M. Sinclair. Comparison of the Peformance of Modern Heuristics for Combinatorial Optimisation on Real Datasmmisat. *Computers and Operations Research*, 20(7):687–695, 1993.

[39] D. J. Sirag and P. T. Weisser. Towards a Unified Thermodynamic Genetic Operator. In *[22]*, pages 116–122.

[40] G. S. Syswedra. Uniform Crossover in Genetic Algorithms. In *[37]*, pages 2–9.

[41] D. Whitley. The GENITOR Algorithm and Selective Pressure: Why Rank-Based Allocation of Reproductive Trials is Best. In *[37]*, pages 116–121.

[42] D. Whitley, T. Starkweather, and D. Shaner. Scheduling Problems and the Traveling Salesman: The Genetic Edge Recombination Operator. In *[37]*, 1989.

[43] D. Whitley, T. Starkweather, and D. Shaner. The Traveling Salesman and Sequence Scheduling: Qulaity Solutions Using Genetic Edge Recombination. In *[9]*. 1991.