Guided Local Search for Combinatorial Optimisation Problems

Christos Voudouris

A thesis submitted for the degree of Ph.D

Department of Computer Science

University of Essex

1997

To my wife, Stella

Acknowledgements

I would like to express my gratitude to my supervisor Edward Tsang for his constant support and guidance throughout my years at Essex University. He introduced me to the fields of constraint satisfaction and combinatorial optimisation and he enormously helped in the evolution and refinement of the ideas presented in this thesis by his constant feedback and supervision. I would like to thank my manager at BT Laboratories Nader Azarmi, without his encouragement and support this thesis may have never been completed. Many thanks to John Ford for his comments on an earlier draft of this thesis and for supervising me during my last months of study.

I would like to thank the Department of Computer Science for the harmonious environment and the Computing Service at the University of Essex for the excellent computer facilities without which it would have been impossible to perform the computational experiments reported here. Many thanks to BT plc which supported me during my final year.

This research has been conducted in the framework of the GENET project funded by the EPSRC grant (GR/H75275).

Abstract

In this thesis, we present the heuristic technique of Guided Local Search for combinatorial optimisation problems. The technique sits on top of local search procedures and has as a main aim to guide these procedures in exploring efficiently and effectively the vast search spaces of combinatorial optimisation problems. This is achieved by exploiting prior information known about the problem in conjunction with historical information gathered during the search process. Information is converted to constraints which take the form of penalties and modify the cost function to be minimised. Local search is guided by these constraints, focusing on solutions of high quality. Guided Local Search can be combined with the neighbourhood reduction scheme of Fast Local Search which significantly speeds up the operations of the algorithm.

In this thesis, Guided Local Search is applied to the Travelling Salesman Problem, Quadratic Assignment Problem, Radio Link Frequency Assignment Problem, Workforce Scheduling and Function Optimisation. Experimental evaluation and comparisons with a variety of other heuristic methods on benchmarks instances of the these problems shows that Guided Local Search compares very favourably to famous general and specialised heuristic algorithms outperforming many of them on the benchmark instances considered.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, we are going to present a technique called Guided Local Search (GLS) which is suitable for a class of difficult computational problems known as *combinatorial optimisation problems*. In this introductory chapter, we will explain the terminology used in the field, examine combinatorial optimisation problems and outline some of the most popular techniques suggested so far for tackling them.

1.1 Combinatorial Optimisation and NP-Hard Problems

Combinatorial optimisation problems appear in many areas such as resource allocation, routing, packing and scheduling. The objective is that of assigning values to a set of decision variables such that a function of these variables is minimised perhaps in the presence of some constraints. A combinatorial optimisation problem can be formulated as follows [Ree96]:

Eq. 1.1

$$\textit{minimise } f(x),\ x \in X \subset R_n$$

$$\textit{subject to } g_i(x) \geq b_i,\ i = 1,\ldots, m.$$

where x is a vector of decision variables and $f(\cdot)$ and $g_i(\cdot)$ are general functions. The condition $x \in X$ is assumed to constrain decision variables to discrete values. Here, we have presumed that the problem is that of minimisation but the modification of Eq. 1.1 for maximisation problems is straightforward.

A class of problems of particular interest in combinatorial optimisation is that of 'hard' combinatorial optimisation problems. This class includes problems famous for their difficulty such as the Travelling Salesman Problem (TSP), the Quadratic Assignment Problem (QAP) and the Vehicle Routing Problem (VRP). Back in the late 60s, many researches recognised the difficulty of such problems and tried to identify whether 'polynomial' algorithms (i.e. algorithms which require a polynomial number of steps) can be devised to solve them. Nobody since then has been able to devise such an algorithm for any of these problems and that despite many man-centuries of research effort invested on the subject by some of the most brilliant researchers. In fact, there seems to be the case that problems such as the TSP are inherently difficult to solve, exhibiting an exponential growth in computing time with the size of the problem.

The hypothesis that no polynomial algorithm exists for solving these problems has been further supported by advances in the field of computational complexity. We briefly describe the findings. The interested reader is referred to classical texts on the subject by Papadimitriou and Steiglitz [PS82] and Garey and Johnson [GJ79] for a more formal and extensive description of these findings.

In brief, problems which have known polynomial algorithms are said to be in the class *P*. A superset of class *P* is the class *NP* where NP stands for "non-deterministic polynomial". NP consists of all problems that can be solved in polynomial time on a *non-deterministic Turing machine*. This includes all problems in P but also 'hard'

problems such as the TSP, QAP, VRP, Satisfiability etc. for which all known algorithms require exponential time.

Hard problems can be transformed one to the other in polynomial time. This property has been used to define a separate sub-class in NP that of *NP-complete* problems. All famous hard combinatorial problems such as the TSP, QAP, Satisfiability, Graph Colouring, Graph Partitioning, Vehicle Routing etc. belong to this class (see [CK95] for a comprehensive list of NP-complete problems). If we were to find a polynomial algorithm for any of these problems, we would have found a polynomial algorithm for all problems in NP. No polynomial algorithm has been found so far and that despite considerable efforts and thus it is widely conjectured that no NP-complete problem is polynomially solvable.

There is a subtle difference here. The above results refer to the 'decision' version of combinatorial optimisation problems where the problem is not that of finding the optimal solution but finding an answer to a question such as 'is there a solution with cost less than *C*?'. It is obvious that an algorithm for the decision version can be used to solve the optimisation version by asking a series of questions to this algorithm. Concluding, optimisation problems as such are not in NP. For the optimisation versions of NP-complete problems, we will use the term *NP-hard*[1] adopted by many authors [RB93]. In addition to that and unless otherwise stated, the terms combinatorial problems and combinatorial optimisation problems will be used to refer to NP-hard optimisation problems.

[1] In general, a problem is said to be NP-hard if any problem in NP is polynomialy transformable to it even if the problem itself is not in NP. If the problem also belongs in NP then it is NP-complete. If you could reduce an NP problem to an NP-hard problem and then solve it in polynomial time, you could solve all NP problems in polynomial time.

1.2 Exact and Heuristic Search Techniques

The simplest approach to solve a NP-hard optimisation problem is to list all the feasible solutions, evaluate their objective function values and chose the best. This approach of *complete enumeration,* although widely applicable, is unusable in practice because of the vast number of possible solutions to any problem of reasonable size.

In the early days of combinatorial optimisation, most of the efforts were focused on Linear Programming (LP). The problem was reformulated by using integer variables usually taking the values 0 or 1 to produce an integer programming (IP) formulation. Such a problem can be then solved by variants of a method generally known as "Branch & Bound" (B&B). Branch and Bound is an efficient enumeration scheme which avoids complete enumeration of solutions by building a search tree of the solutions to be evaluated. This tree is pruned during search, so reducing the number of solutions that need to be evaluated before the optimal solution is found and proved to be optimal.

The worst case computational complexity of IP algorithms grows exponentially with the size of the problem. As a result of that, general IP codes usually do not scale well to large instances of problems. Furthermore, for some problems it is difficult to find an IP formulation and even if one is found it sometimes results in a large number of variables and constraints. IP is much more difficult than LP and that because the problems of concern are NP-hard optimisation problems.

Given the difficulty of NP-hard optimisation problems, many researchers have focused on another class of techniques known as *heuristic techniques* or simply *heuristics.* These techniques sacrifice the proof of optimality for solutions and instead focus on finding good near optimal solutions at a reasonable computational cost. In the early days of Operations Research, heuristics were treated with scepticism.

Nowadays, mainly due to the theoretical developments in computational complexity indicating the inherent difficulty of NP-hard problems, heuristics have gained a prominent position amongst optimisation methods. Following Reeves [Ree96], we give the following general definition for a method to qualify as a heuristic:

***Definition 1-1*:**

A heuristic technique is a method which seeks good (i.e. near optimal solutions) at a reasonable computational cost without being able to guarantee optimality, and possibly not feasibility. Unfortunately, it may not even be possible to state how close to optimality a particular heuristic solution is.

Despite the rather pessimistic definition, modern heuristics can find high quality solutions for problems many times larger that those solved to optimality by exact search methods. From a historical perspective, the 'gamble' with heuristics has paid off leading to many real world systems tackling NP-hard optimisation problems in resource allocation, routing, scheduling and many other domains. In the rest of the chapter, we examine some the most famous heuristic techniques starting with *Local Search* [PS82] perhaps the oldest heuristic method.

1.3 Local Search

Local Search, also referred to as Neighbourhood Search or Hill Climbing, is the basis of many heuristic methods for combinatorial optimisation problems. In isolation, it is a simple iterative method for finding good approximate solutions. The idea is that of trial and error. For the purposes of explaining local search, we will consider a slightly different definition of a combinatorial problem to that given in Eq. 1.1.

A combinatorial optimisation problem is defined by a pair *(S, g)*, where *S* is the set of all feasible solutions (i.e. solutions which satisfy the problem constraints) and *g* is the objective function that maps each element *s* in *S* to a real number. The goal is to find the solution *s* in *S* that minimises the objective function *g*. The problem is stated as:

Eq. 1.2 $$min\ \mathrm{g(s)},\ \mathrm{s} \in \mathrm{S}.$$

In the case where constraints difficult to satisfy are also present, penalty terms may be incorporated in *g*(s) to drive toward satisfying these constraints. A neighbourhood *N* for the problem instance *(S, g)* can be defined as a mapping from *S* to its powerset:

Eq. 1.3 $$N: S \rightarrow 2^S.$$

N(s) is called the *neighbourhood* of *s* and contains all the solutions that can be reached from *s* by a single *move*. Here, the meaning of a move is that of an operator which transforms one solution to another with small modifications. A solution *x* is called a *local minimum* of *g* with respect to the neighbourhood *N* iff:

Eq. 1.4 $$g(x) \leq g(y), \forall y \in N(x).$$

Local search is the procedure of minimising the cost function *g* in a number of successive steps in each of which the current solution *x* is being replaced by a solution *y* such that:

Eq. 1.5 $$g(y) < g(x), y \in N(x).$$

A basic local search algorithm begins with an arbitrary solution and ends up in a local minimum where no further improvement is possible. In between these stages, there are many different ways to conduct local search. For example, *best improvement* (*greedy)* local search replaces the current solution with the solution that improves most in cost

after searching the whole neighbourhood. Another example is *first improvement* local search which accepts a better solution when it is found. The computational complexity of a local search procedure depends on the size of the neighbourhood and also the time needed to evaluate a move. In general, the larger the neighbourhood, the more the time one needs to search it and the better the local minima.

Local minima are the main problem with local search. Although these solutions may be of good quality, they are not necessarily optimal. Furthermore if local search gets caught in a local minimum, there is no obvious way to proceed any further toward solutions of better cost. Methods that build on local search to remedy this problem are sometimes referred to as meta-heuristics. One of the first methods in this class is *Repeated Local Search* where local search is restarted from a new arbitrary solution every time it reaches a local minima until a number of restarts is completed. The best local minimum found over the many runs is returned as an approximation of the global minimum. Modern meta-heuristics tend to be much more sophisticated than repeated local search pursuing a range objectives that go beyond simply escaping from local minima. In the following sections, we examine some of the most successful modern meta-heuristic techniques.

1.4 Simulated Annealing (SA)

Simulated Annealing (SA) is perhaps the most widely used meta-heuristic. Mainly because of its simplicity, SA has attracted the interest of many researchers and practitioners from a wide range of disciplines. The technique has its origins in statistical mechanics and it was inspired by the physical process of annealing used for the “cooling” of solids such that they form perfect crystals. Metropolis et al. [MRRTT53] first described an algorithm for simulating the annealing process.

Kirkpatrick et al. [KGV83] proposed the use of this simulation algorithm for searching the solutions of a combinatorial problem.

SA could be described as a randomised scheme which reduces the risk of getting trapped in local minima by allowing moves to inferior solutions. Given the neighbourhood $N(s)$ of a combinatorial problem, moves are randomly selected from this set. A move from a solution s to solution s' is only accepted if:

- s' is better than s or
- s' is worse than s but $e^{\frac{-(g(s)-g(s'))}{T}} > R$,

where T is a control parameter called 'temperature' and $R \in [0,1]$ is a uniform random number. The temperature parameter T is initially set to a high value, allowing many non-improving moves to be accepted and it is gradually reduced to a value where nearly all non-improving moves are rejected. In this way, the algorithm avoids getting trapped in local minima until the final stages of search when the temperature is very low and the algorithm has already settled in a good solution.

There have been many studies on the convergence properties of SA. Research using the theory of Markov chains has proved that if the temperature is lowered slowly enough, SA will eventually converge to a global minimum. Unfortunately, the same research shows that this will, in general, require more iterations than exhaustive search. For detailed information on convergence results for SA, the reader is referred to two excellent books by van Laarhoven and Aarts [LA88] and Aarts and Korst [AK89]. Additionally, Johnson et al. [JAMS89, JAMS91] provide excellent experimental results for SA on a variety of problems which also may be very useful to the interested reader.

1.4.1 Cooling Schedules

In practice, the temperature is lowered according to a scheme referred to as the annealing (or cooling) schedule. A cooling schedule specifies [Osm95]:

- the initial starting value of the temperature parameter T,
- the cooling rate a and the temperature update rule,
- the number of iterations to be performed at each temperature,
- the termination criterion of the algorithm.

The performance of SA strongly depends on the cooling schedule. Not surprisingly, many different types of cooling schedules have been suggested. Osman [Osm95] classifies SA cooling schedules in three categories:

- *Stepwise temperature reduction*. In this case, the temperature remains constant for a number of iterations (i.e. selection of a random move followed by the acceptance test) before it is updated according to the update rule. The update rule commonly used is a geometric reduction function which reduces the temperature to $a(t) = a \cdot t$, where $a < 1$. That is why this type of cooling is often called *geometric cooling*. Best performances are reported in the literature for values of a in the range $0.8 \leq a \leq 0.99$ [Dow93]. The number of iterations at each temperature is related to the size of the neighbourhood but may also vary from temperature to temperature.
- *Continuous temperature reduction* [LM86]. In this type of cooling schedule, the temperature is reduced after every iteration. The reduction of the temperature is very slow and it is conducted according to the rule $a(t) = t/(1+b \cdot t)$ where b is a small value.
- *Non-monotonic temperature reduction* [Dow93, Osm93]. The temperature is reduced after each iteration though occasional increases are also allowed.

The SA algorithm terminates when the number of uphill moves accepted becomes negligible or some other type of stopping criterion is satisfied.

1.5 Tabu Search (TS)

Tabu Search (TS) has been developed by Glover [Glo86] and, independently, by Hansen [Han86]. TS is a meta-heuristic that combines a local search procedure with a number of anti-cycling rules which prevent the search from getting trapped in local minima. Over the years, the method has evolved and incorporated many new elements which further enhance its overall performance.

In this section, we will present the most important elements of TS. The interested reader may refer to one or more of the excellent survey papers available on the technique [Glo89, Glo90, GTW93, GL93, Glo94, Glo95, Glo96]. These survey papers examine in more detail the elements of TS described in this chapter and also outline some less frequently used elements not examined here.

1.5.1 The Basis for Tabu Search

The basis for tabu search is described by Glover in [Glo95] as follows. Given a function $f(x)$ to be optimised over a set X, TS begins in the same way as ordinary local search, proceeding iteratively from one point (solution) to another until a chosen termination criterion is satisfied. Each $x \in X$ has an associated neighbourhood $N(x) \subset X$ and each solution $x' \in N(x)$ is reached from x by an operation called a *move*.

TS goes beyond local search by employing a strategy of modifying $N(x)$ as the search progresses, efficiently replacing it by another neighbourhood $N^*(x)$. A key aspect of tabu search is the use of special memory structures which serve to determine $N^*(x)$, and hence to organise the way in which the space is explored.

We will start our brief account of TS by examining the so-called *recency-based* memory which can be used as a stand-alone device or as the basis for more advanced TS schemes.

1.5.2 Recency-Based Memory

Recency-based memory is utilising information pertaining to the moves executed by local search to avoid reversing changes created by these moves. The information used is the "attributes" of solutions (i.e. solution properties) that change state (i.e. deleted or added) when a move is executed. These attributes are used to define the "tabu status" of moves at future iterations, that is, moves which are forbidden to be executed. For example, if a move *m* changes the value of a 0-1 variable x_j from 0 to 1 then the solution attribute $x_j = 0$ can be used to prevent the reversal of the changes created by the move. After move *m* is executed the solution attribute $x_j = 0$ becomes tabu-active rendering tabu (i.e. forbidden) all moves that reinstate this attribute in the solution. These restrictions are temporary and they last only for a small number of iterations. For that purpose, tabu-active attributes are assigned appropriate *tabu-tenures* which determine for how many iterations local search is prevented from reinstating these attributes. This mechanism is sometimes implemented using a data structure called a *tabu list* [Glo89].

A move may change the state of more than one solution attribute. In such cases, tabu restrictions on moves can be defined by rendering a move tabu only if all (or some number) of its component solution attributes are tabu-active [Glo95]. By deciding on the combinations of attributes that render a move tabu, we have the flexibility to strengthen or weaken the tabu restrictions. The choices may vary from a disjunction between the attributes (more restrictive) to a conjunction (less restrictive). Another

way of controlling tabu restrictions is to assign different tabu-tenures to different types of attributes. Furthermore, tabu-tenures may vary during search leading to a dynamic and robust form of search [Tai91, GTW93].

Another part of recency-based memory are the so-called *aspiration criteria* which mainly aim at adding flexibility to compensate for the hard nature of constraints in recency-based memory. Aspiration criteria are sets of conditions which if satisfied overrule the tabu restrictions. The most commonly used aspiration criterion is to accept a move which is classified as tabu if the move generates a solution better than any previously seen. The interested reader may refer to [GL93] where a more extensive account is given on the different types of aspiration criteria.

In many applications, recency-based memory is sufficient to produce high quality solutions. However, this type of memory is of a short-term nature and therefore insufficient to support a long-term strategy necessary for a more systematic exploration of the search space. For that purpose, a set of additional tabu search elements have been developed which are known as long-term memory components. The two main goals for these components are the intensification and diversification of search.

1.5.3 Intensification Strategies

The purpose of intensification strategies is to concentrate the search on good regions of the search space or good solution features. This usually manifests itself in a solution recording mechanism which keeps a copy of high quality solutions found during the search. These solutions, often referred to as *elite solutions*, are used each time the search progresses slowly to restart it from the good regions which lie around these elite solutions. The state of the recency-based memory (when the elite solution

was recorded) may also be saved and partially or fully restored when starting from this elite solution. Such approaches have been successfully used in vehicle routing [XK96] and telecommunications network design problems [XCG95].

Another form of intensification is based on identifying "consistent and strongly determined" variables. A strongly determined variable is one whose value cannot be changed except by inducing a disruptive effect on the objective function value or the values of the other variables. On the other hand, a consistent variable is one that is frequently assigned the same value in good solutions. The idea is to identify the most consistent and strongly determined variables and assign to these variables their "preferred" values by reference to a set of elite solutions. This is usually done in the framework of a multi-start approach where new starting points are generated by assigning consistent and strongly determined variables to their "preferred" values. This approach has been successfully applied to the vehicle routing problem [RT95].

1.5.4 Diversification Strategies

Diversification strategies are designed to drive the search into new regions. Often they are based on modifying choice rules to bring attributes into the solutions that are infrequently used. More of these schemes are based on type of memory called *frequency-based memory*. In short, frequency-based memory is a long-term memory which either

- records the frequency at which solution attributes occur in the solutions generated (*residence* frequencies)
- or records the frequency different moves are executed (*transition* frequencies).

Residence frequencies are used to encourage the incorporation in the solution of infrequently used attributes while transition frequencies are used to encourage the execution of less frequently performed moves.

One way to use the information recorded in residence frequencies is to periodically restart the search from solutions that incorporate the less frequently used solution attributes. More often, residence frequencies are used in a continuous fashion by being directly incorporated in the cost function multiplied by a penalty factor. Attributes with high frequencies have higher penalties than those with lower frequencies. Thus the use of the later is encouraged while the use of the former is discouraged.

Transition frequencies are used in a similar way to residence frequencies and penalties are usually introduced that discourage the execution of frequently executed moves while encouraging the execution of less frequently executed moves.

Residence or transition frequencies have been successfully used in problems such as maximum clique [SG96], bin packing [LG93], network design [XCG95], quadratic assignment [Sko90], machine scheduling [LG93b], vehicle routing [GHL94, TBGGP95, XK96] and others.

1.5.5 Candidate List Strategies

For many problems, the amount of computational effort required to search the complete neighbourhood in every iteration is prohibitive. Candidate list strategies are aiming at reducing this effort by restricting the number of solutions examined on a given iteration. The different types of candidate list strategies are the following [Glo95]:

- *Random Strategy***.** The neighbourhood is randomly sampled until enough moves are evaluated to give some assurance that some good choices were examined.
- *Subdivision Strategy*. Moves that involve more than one component are decomposed and moves which incorporate good components only are examined.
- *Aspiration Plus Strategy*. This approach establishes a threshold based on the search history for the quality of moves to be selected and examines moves until finding one which satisfies this threshold.
- *Elite Candidate Lists*. A list of elite moves is constructed after searching a large part of the neighbourhood. At subsequent iterations only solutions from this elite list are examined until the quality of moves drops below a specified threshold. At this point a new list is constructed and the process is repeated.
- *Sequential Fan Strategy*. The idea is to generate some *p* best alternative moves at a given step and then to create a fan of solution streams, one for each alternative. The best available moves for each stream are again examined and only the *p* best moves overall provide the *p* new streams at the next step. This technique is very much oriented towards parallel processing.

Candidate list strategies conclude our account of Tabu Search. Other elements not examined here include strategic oscillation, path re-linking, ejection chains, vocabulary building and probabilistic tabu search. The reader may refer to [GL93, Glo95, Glo96] for information on these variants. Additionally, the reader may also refer to [XCG96] on the use of statistical tests to determine the many parameter values that need to be specified when various elements of TS are integrated together to solve a combinatorial problem.

1.6 Genetic Algorithms

Genetic Algorithms (GAs) are a class of methods based on a highly abstract model of natural evolution. They were developed by Holland in the 70s [Hol75, Gol89, Dav91] and since then they have been applied to numerous domains. Only recently their potential application to combinatorial optimisation problems has been investigated. We first examine some of the terminology used in the GA literature.

A solution to a combinatorial optimisation problem is often called a *chromosome*, *string* or *vector*. Variables of the problem are called *genes* and their possible values *alleles*. The position of a variable in a chromosome is called its *locus*. Each chromosome encodes a solution to the optimisation problem and it is evaluated according to some *fitness function*. The fitness function is related to the cost function of the combinatorial optimisation problem. The *fitness value* given to a chromosome by this function represents the suitability of this chromosome (after decoding) as a solution to the combinatorial problem. For a review of Genetic Algorithm techniques in the context of combinatorial optimisation the reader may refer to [Ree93].

1.6.1 A Basic GA Algorithm

A basic GA algorithm for a combinatorial problem functions in the following way. Initially, a finite population of solutions is generated randomly or by other means. After that, an iterative process is applied to the population which at each step transforms the current population to a new population. This involves selecting pairs of parent solutions from the population according to a selection scheme which takes into account their fitness values and combining them to generate offspring solutions. The combination of the parents is performed by a special type of operator called the *crossover* or *recombination* operator. After the generation of the 'children' random

changes are inflicted upon them by a second type of operator called the *mutation* operator. The children are finally inserted in the population by either replacing their parents (Canonical GAs) or the weakest individuals in the population (Steady State GAs[2]). This completes one iteration of the GA which transforms one generation of solutions to the next. The algorithm iterates until a termination criterion is satisfied based either on computational resources, the convergence of the population (high similarity between the solutions contained in the population) or both. In the following, we examine more closely the various elements of a GA.

1.6.1.1 Initial Population

The initial population is normally generated at random. Yet in most of the successful GAs for combinatorial optimisation, solutions in the initial population are heuristically generated (by a construction heuristic, local search, or sometimes by local search applied to a solution generated by a construction heuristic) and they are already of good quality. Particular attention must be paid that the size of this initial population is not too small to avoid premature convergence of the GA.

1.6.1.2 Genetic Operators

As mentioned above, the crossover operator is used to combine two parent solutions. There are many versions of this operator. The simplest case is that of the *1-point* crossover. A cut-point X is selected at random and each offspring consists of the pre-X section from one parent followed by the post-X section from the other. The 1-point crossover can be extended to *2-point* crossover, *3-point* crossover or even *k-point* crossover. Another useful crossover operator is the *uniform* crossover where the value of each variable in each parent is equally likely to be passed to the offspring.

---

[2] Steady State GAs also generate one child instead of two children as in Canonical GAs.

Many combinatorial optimisation problems require special types of operators which can combine sequences or permutations (often used to represent solutions) to produce feasible offspring solutions. An example of such an operator for the TSP is the PMX operator (Partially Mapped Crossover). Many other special operators exist for different types of combinatorial optimisation problems (see [Ree93] for some examples).

In addition to crossover and after the generation of the children, a mutation operator is employed to modify the population of solutions by introducing small random modifications to solutions randomly selected from the population. If bit vectors are used for representing the solutions, this frequently means flipping the bits of some of the solutions. In general, the probability of mutation is very low.

1.6.2 Hybrid GAs.

As Davis states in the Handbook of Genetic Algorithms [Dav91], “Traditional genetic algorithms, although robust, are generally not the most successful optimisation algorithm on any particular domain”. For that reason, Davis and many others have argued that hybridising GAs with the most successful optimisation methods for particular problems gives one the best of both worlds.

The idea of including in the initial population solutions constructed by a problem-specific heuristic, mentioned above, can be viewed as a primitive form of hybridisation.

Several GA approaches which have produced very good results for famous combinatorial optimisation problems go one step further, utilising local search algorithms to optimise the solution generated by crossover or mutation operators (see [MGK88, FF94, FM96]). These GA algorithms essentially work on local minima

constructed by local search trying to recombine them to produce new and hopefully better local minima. The rationale is that local minima solutions consist of good solution fragments which if properly combined by crossover type operators will lead to solutions where these fragments are combined even better and therefore be of higher quality. This leads to a type of search intensification around the areas of good solutions. Diversification of search is also important and is performed by the particular mutation operator used. From another viewpoint, Hybrid GAs can be seen as a type of local search which explores the space of good solution fragment combinations. There are similarities there with tabu search variants which also try to identify and recombine good solution fragments [RT95]. These tabu search variants are sometimes seen as part of a wider framework of techniques called *Adaptive Memory Programming* [Glo96].

1.7 GENET and Other Weighting Methods for CSPs

Guided Local Search (GLS) studied in this thesis is a meta-heuristic which guides local search in exploring the vast search spaces of combinatorial problems. The technique extends to general optimisation problems methods applied with considerable success to Constraint Satisfaction [Tsa93]. In this section, we will briefly refer to these methods and in particular to the GENET neural network [WT91, Tsa93, DTWZ94] which is a direct predecessor of GLS.

The Constraint Satisfaction Problem (CSP) is that of assigning values to a number of variables with finite domains such that a set of linear or non-linear constraints involving one or more variables are satisfied. CSP is in general NP-Hard and it is closely related to the propositional satisfiability or SAT problem [GJ79]. In contrast to most combinatorial optimisation problems, the goal in CSPs is to find one or all

feasible solutions. Real world CSPs usually involve difficult non-linear constraints spanning two or more variables of the problem. Amongst other techniques, local search has been considered for solving CSPs.

A local search approach to constraint satisfaction treats a CSP as an optimisation problem. The objective function, which is to be minimised, is the number of constraints being violated. A typical local search method assigns an arbitrary value to each variable in the CSP. Then it proceeds iteratively to reduce the number of constraint violations by re-assigning values to variables, using a heuristic known as the min-conflict heuristic [MJPL92]. This iterative improvement of the number of unsatisfied constraints leads either to a solution to the CSP or to a local minimum where some constraints are still being violated but no further improvement is possible by changing the value of any of the variables. Local minima are of little use in CSPs since they violate hard problem constraints.

A successful approach to escape local minima, proposed in the context of CSPs, is to assign weights to the problem constraints (clauses for SAT) and increase these weights in a local minima for the violated constraints (unsatisfied clauses for SAT) in a effort to ‘fill up’ the local minimum until local search escapes from it.

Various algorithms based on this scheme have been developed in the last few years and applied either to the CSP or the SAT problem. Amongst them GENET [WT91, Tsa93, DTWZ94], Weighted GSAT [SK93, Fra96], and also the Breakout Method [Mor93]. Here, we briefly examine GENET which was the point of origin for this work.

1.7.1 The GENET Neural Network

GENET is a connectionist approach to constraint satisfaction with a basic operation that resembles the min-conflicts heuristic. Basically a CSP is represented by a network in which the nodes represent possible assignments to the variables and the edges represent constraints. One of the innovations in GENET was the use of and manipulation of weights assigned to the edges (constraints). All edges are inhibitory connections which have weights initialised to -1. GENET will continuously select assignments which receive the least inhibitory input (which roughly means violating the least number of constraints). The operation of the network is designed in such a way that will ensure its convergence to some states, which could be solutions or local minima (in terms of number of constraints violated). Each time the network converges to a local minimum, the weights associated with the violated constraints are decreased, and the network is then allowed to converge again. Since GENET always makes moves which improve the number of constraint violations, decreasing the weights allows it to escape from the local minimum to states which have lower cost. Such convergence-learning cycles continue until a solution is found or a stopping condition is satisfied.

GENET's mechanism for escaping from local minima resembles reinforcement learning [BSA83]. Basically, patterns in a local minimum are stored in the constraint weights and are discouraged to appear thereafter. For this reason, the mechanism was named "learning". GENET's learning scheme can be viewed as a method to transform the objective function (i.e. the number of constraint violations) so that a local minimum gains an artificially higher value. Consequently, local search will be able to leave the local minimum state and search other parts of the space.

In the CSP context, modifying the weights for unsatisfied constraints in local minima modifies the cost function of the problem though that does not affect the cost of an optimal solution which if exists satisfies all the constraints by definition and therefore always has zero cost.

Guided Local Search (GLS) presented in this thesis utilises a similar approach to tackle famous combinatorial problems. In these problems, modifications to the cost function, although they may affect the cost of many solutions of a combinatorial problem including the optimal can effectively guide local search in the search space. Apart from escaping local minima in a way similar to GENET and other techniques for CSP and SAT problems, GLS introduces additional functionality for distributing the search efforts over the various areas of the search space, taking into account the promise of these areas to contain the optimal solution. Furthermore, it uses sophisticated neighbourhood reduction techniques which can speed up the algorithm many times.

1.8 Overview of the Thesis

In this thesis, we describe the technique of GLS and examine its application to a comprehensive set of traditional and modern real world combinatorial optimisation problems. The performance of the technique is experimentally evaluated on benchmark instances of these problems. Extensive comparisons are conducted with general and specialised heuristic algorithms including all the general heuristic methods examined in this chapter. The thesis is structured as follows. In the next chapter, we present GLS and discuss various extensions to the method. Following that, five applications of the algorithm are examined in the following order:

- Travelling Salesman Problem (chapter 3),
- Quadratic Assignment Problem (chapter 4),
- Radio Link Frequency Assignment Problem (chapter 5),
- Workforce Scheduling (chapter 6),
- Non-convex Optimisation (chapter 7).

The thesis concludes with chapter 8 where the work on GLS is summarised and future research directions are suggested.

Most of the findings in chapter 5 have appeared in the Proceedings of the 2nd International Conference on Practical Application of Constraint Technology [VT96] while the results in chapter 6 have appeared in the journal of *Operations Research Letters* [TV97]. Earlier results for GLS on the Travelling Salesman Problem, Quadratic Assignment Problem and Nonconvex Optimisation have been reported in two Essex University technical reports [VT95a, VT95b].

Chapter 2

Guided Local Search

Embarking on this research almost three years ago, the main objective was to extend GENET for Constraint Satisfaction Problems (CSPs) to a more general class of problems known as Partial Constraint Satisfaction Problems (PCSPs). Through the process of trying to apply GENET to PCSPs, we soon realised that a more general optimisation technique was hidden under GENET's neural network architecture. This technique, namely Guided Local Search, is the subject of this chapter and the core of the thesis.

2.1 History of Guided Local Search

Partial CSPs are CSPs where no solution satisfies all the constraints and one is interested in solutions which minimise the number of constraint violations and possibly other application dependent criteria (see section 5.1 for a formal definition).

The RLFAP problem described in chapter 5 was one of the first problems we tried to solve using extensions of GENET. The problem is a PCSP and requires the minimisation of constraint violations combined with domain specific optimisation criteria. Minimising constraint violations was within the capabilities of the GENET neural network but minimising the other RLFAP optimisation criteria seemed difficult and required extra complexity in the neural network architecture. Because of that, we decided at the time to convert GENET to a pure algorithm, abandoning any efforts to solve the problem by extending the model of the neural network. This resulted in the Tunnelling Algorithm [VT94] which was very successful in the RLFAP instances and moreover preserved the good performance of GENET on classic CSPs. While experimenting with the tunnelling algorithm, we had the idea to apply the method to the Travelling Salesman Problem (TSP), utilising some of the work we did on the modelling of RLFAP's optimisation criteria. To our surprise, the method worked extremely well on the TSP and some preliminary results on that were included in the paper on the tunnelling algorithm [VT94].

The success on the TSP convinced us of the great potential of the algorithm. We generalised the Tunnelling Algorithm even further, so that it could be applied to the bulk of combinatorial optimisation. The result of this generalisation was Guided Local Search. Guided Local Search exceeded all our expectation. We applied the method to seven Combinatorial Optimisation problems and obtained very good results both in terms of solution quality and running times. The method and five of its applications will be presented in this thesis. We start by introducing the principles of Guided Local Search.

2.2 Guided Local Search Principles

Guided Local Search is a general and compact optimisation technique suitable for a wide range of combinatorial optimisation problems. Guided Local Search takes advantage of problem and search-related information to guide *local search* in a search space. This is made possible by augmenting the cost function of the problem to include a set of penalty terms. Local search is confined by the penalty terms and focuses attention on promising regions of the search space. Iterative calls are made to local search. Each time local search gets caught in a local minimum, the penalties are modified and local search is called again to minimise the modified cost function.

Penalty modifications *regularise* the solutions generated by local search to be in accordance with prior information or information gathered during search. The approach taken by GLS is analogous to that of regularisation methods for 'ill-posed' problems [TAJ77, Hay94]. The idea behind regularisation methods and GLS, to an extent, is the use of prior information to help us solve an approximation problem. Prior information translates to constraints which further define our problem, so reducing the number of candidate solutions to be considered. GLS also exploits information learnt during search by imposing extra constraints on the basis of this information. GLS is essentially a *meta-heuristic* based on local search. In the following sections, we examine the various components of GLS.

2.3 Local Search

Local search is the basis of many heuristic methods for combinatorial optimisation problems. In section 1.3, we presented an overview of local search. A variety of moves and local search procedures have been used for the problems in this study. For

the purpose of describing GLS in the general case, local search is considered a general procedure of the form:

$$s_2 \leftarrow \textit{procedure}\ \textbf{LocalSearch}(s_1, g),$$

where s_1 is the initial solution, s_2 the final solution (local minimum) and g the cost function to be minimised.

In contrast to other general meta-heuristics such as SA and tabu search, GLS is not modifying the internal mechanisms of local search. Instead, it makes iterative calls to a local search procedure modifying the cost function between successive calls. Before that, the cost function of the problem is augmented to include a set of penalty terms which enable us to constrain solutions dynamically. This augmentation of the cost function with penalty terms is explained in the next section.

2.4 Solution Features

GLS employs solution features to characterise solutions. A *solution feature* can be any solution property that satisfies the simple constraint that is a non-trivial one. What it is meant by that is that not all solutions have this property. Some solutions have the property while others do not. Solution features are problem dependent and serve as the interface between the algorithm and a particular application.

Constraints on features are introduced or strengthened on the basis of information about the problem and also the course of local search. Information pertaining to the problem is the cost of features. The cost of features represents the direct or indirect impact of the corresponding solution properties on the solution cost. Feature costs may be constant or variable. Information about the search process pertains to the solutions visited by local search and in particular local minima. A feature f_i is represented by an indicator function in the following way:

Eq. 2.1

$$I_i(s) = \begin{cases} 1, & \text{solution } s \text{ has property } i \\ 0, & \text{otherwise} \end{cases}, s \in S.$$

2.5 Augmented Cost Function

Constraints on features are made possible by augmenting the cost function *g* of the problem to include a set of penalty terms. The new cost function formed is called the *augmented cost function* and it is defined as follows:

Eq. 2.2

$$h(s) = g(s) + \lambda \cdot \sum_{i=1}^{M} p_i \cdot I_i(s),$$

where *M* is the number of features defined over solutions, p_i is the penalty parameter corresponding to feature f_i and λ (lambda) is the *regularisation parameter*. The penalty parameter p_i gives the degree up to which the solution feature f_i is constrained. The regularisation parameter λ represents the relative importance of penalties with respect to the solution cost and is of great significance because it provides a means to control the influence of the information on the search process. GLS iteratively uses local search and it simply modifies the *penalty vector* $\mathbf{p}$ given by:

Eq. 2.3

$$\mathbf{p} = (p_1, ..., p_M)$$

each time local search settles in a local minimum. Modifications are made on the basis of information. Initially, all the penalty parameters are set to 0 (i.e. no features are constrained) and a call is made to local search to find a local minimum of the augmented cost function. After the first local minimum and every other local minimum, the algorithm takes a modification *action* on the augmented cost function and re-applies local search, starting from the previously found local minimum. The modification action is that of simply incrementing by **one** the penalty parameter of one

or more of the local minimum features. Prior and historical information is gradually inserted into the augmented cost function by selecting which penalty parameters to increment.

Sources of information are the cost of features and the local minimum itself. Let us assume that each feature f_i defined over the solutions is assigned a cost c_i. This cost may be constant or variable. In order to simplify our analysis, we consider feature costs to be constant and given by the *cost vector* $\mathbf{c}$:

Eq. 2.4 $$\mathbf{c} = (c_1, ..., c_M)$$

which contains positive or zero elements. A particular local minimum solution s_* exhibits a number of features. Indicators of the features f_i exhibited take the value 1 (i.e. $I_i(s_*) = 1$).

2.6 Penalty Modifications

In a local minimum s_*, the penalty parameters are incremented by one for all features f_i that maximise the utility expression:

Eq. 2.5 $$util(s_*, f_i) = I_i(s_*) \cdot \frac{c_i}{1 + p_i} .$$

In other words, incrementing the penalty parameter of the feature f_i is considered an *action* with utility given by Eq. 2.5. In a local minimum, the actions with *maximum* utility are selected and then performed. The penalty parameter p_i is incorporated in Eq. 2.5 to prevent the scheme from being totally biased towards penalising features of high cost. The role of the penalty parameter in Eq. 2.5 is that of a counter which counts how many times a feature has been penalised. If a feature is penalised many