times over a number of iterations then the term $\frac{c_i}{1+p_i}$ in Eq. 2.5 decreases for the feature, diversifying choices and giving the chance for other features to also be penalised. The policy implemented is that features are penalised with a frequency proportional to their cost. Due to Eq. 2.5, features of high cost are penalised more frequently than those of low cost. The search effort is distributed according to *promise* as it is expressed by the feature costs and the already visited local minima, since only the features of local minima are penalised. Incremental distribution of the search effort according to prior information, though in a probabilistic framework, can be found in a class of methods based on the *optimal search theory* of Koopman [Koo57, Sto83]. Also, counter based schemes for search diversification analogous to that of GLS are used under the name *counter-based exploration* in reinforcement learning [Thr92]. The basic GLS algorithm as described so far is depicted in Figure 2.1.

```
procedure GuidedLocalSeach(S, g, λ, [I1, ...,IM], [c1,...,cM], M)
begin
        k ← 0;
        s0 ← random or heuristically generated solution in S;
        for i ←1 until M do /* set all penalties to 0 */
                pi ← 0;
        while StoppingCriterion do
        begin
                h ← g + λ * ∑pi*Ii ;
                sk+1 ← LocalSearch(sk, h);
                for i ←1 until M do
                        utili ← Ii(sk+1) * ci / (1+pi);
                for each i such that utili is maximum do
                        pi ← pi + 1;
                k ← k+1;
        end
        s* ← best solution found with respect to cost function g;
        return s*;
end
```

where S: search space, g: cost function, h: augmented cost function, λ: regularisation parameter, I_i: indicator function for feature i, c_i: cost for feature i, M: number of features, p_i: penalty for feature i.

Figure 2.1 Guided Local Search in pseudocode

As we will see in the following chapters, this simple algorithm can be applied with simple modifications to a variety of optimisation problems. Applying the algorithm to a problem usually involves defining the features to be used, assigning costs to the them and finally substituting the procedure *LocalSearch* in the GLS loop with a local search algorithm for the problem in hand.

2.7 Regularisation Parameter

Something that has been left out from the analysis so far is the regularisation parameter λ in the augmented cost function Eq. 2.2. This parameter determines the degree up to which constraints on features are going to affect local search. Let us examine how the regularisation parameter is going to affect the moves performed by a local search method. A move alters the solution, adding new features and removing existing features, whilst leaving other features unchanged. In the general case, the difference Δh in the value of the augmented cost function due to a move is given by the following difference equation:

Eq. 2.6

$$\Delta h = \Delta g + \lambda \cdot \sum_{i=1}^{M} p_i \Delta I_i \,.$$

As we can see in Eq. 2.6, if λ is large then the selected moves will solely remove the penalised features from the solution and the information will fully determine the course of local search. This introduces risks because information may be wrong. Conversely, if λ is 0 then local search will not be able to escape from local minima. However, if λ is small and comparable to Δg then the moves selected will aim at the combined objective of improving the solution (taking into account the cost differences) and also removing the penalised features (taking into account the

information). Since the difference Δg is problem dependent, the regularisation parameter is also problem dependent. GLS can be quite tolerant to the choice of the λ, operating well for a wide range of values. In the applications, we are going to elaborate further on the role of this parameter and on how it affects GLS in specific problems.

2.8 Fast Local Search and Other Improvements

There are both minor and major optimisations that significantly improve the basic GLS method. For example, instead of calculating the utilities for all the features, we can restrict ourselves to the local minimum features since for non-local minimum features the utility as given by Eq. 2.5 takes the value 0. Also, the evaluation mechanism for moves needs to be changed to work efficiently on the augmented cost function. Usually, this mechanism is not directly evaluating the cost of the new solution generated by the move but it calculates the difference Δg caused to the cost function. This difference in cost should be combined with the difference in penalty as is shown in Eq. 2.6. This can be easily done and has no significant impact on the time needed to evaluate a move. In particular, we have to take into account only features that change state (being deleted or added). The penalty parameters of the features deleted are summed together. The same is done for the penalty parameters of features added. The change in penalty due to the move is then simply given by the difference:

Eq. 2.7

$$- \sum_{\text{over all features j added}} p_j + \sum_{\text{over all features k deleted}} p_k \quad .$$

Leaving behind the minor improvements, we turn our attention to the major improvements. In fact, these improvements do not directly refer to GLS but to local search. Greedy local search selects the best solution in the whole neighbourhood. This

can be very time-consuming, especially if we are dealing with large instances of problems. Next, we are going to present *Fast Local Search* (FLS), which drastically speeds up the neighbourhood search process by redefining it. The method is a generalisation of the *approximate 2-opt* method proposed in [Ben92] for the Travelling Salesman Problem. The method also relates to *Candidate List Strategies* used in tabu search (see section 1.5.5).

FLS works as follows. The current neighbourhood is broken down into a number of small sub-neighbourhoods and an *activation bit* is attached to each one of them. The idea is to scan continuously the sub-neighbourhoods in a given order, searching only those with the activation bit set to 1. These sub-neighbourhoods are called *active* sub-neighbourhoods. Sub-neighbourhoods with the bit set to 0 are called *inactive* sub-neighbourhoods and they are not being searched. The neighbourhood search process does not restart whenever we find a better solution but it continues with the next sub-neighbourhood in the given order. This order may be static or dynamic (i.e. change as a result of the moves performed).

Initially, all sub-neighbourhoods are active. If a sub-neighbourhood is examined and does not contain any improving moves then it becomes inactive. Otherwise, it remains active and the improving move found is performed. Depending on the move performed, a number of other sub-neighbourhoods are also activated. In particular, we activate all the sub-neighbourhoods where we expect other improving moves to occur as a result of the move just performed. As the solution improves the process dies out with fewer and fewer sub-neighbourhoods being active until all the sub-neighbourhood bits turn to 0. The solution formed up to that point is returned as an approximate local minimum.

The overall procedure could be many times faster than conventional local search. The bit setting scheme encourages chains of moves that improve specific parts of the overall solution. As the solution becomes locally better the process is settling down, examining fewer moves and saving enormous amounts of time which would otherwise be spent on examining predominantly bad moves.

Although fast local search procedures do not generally find very good solutions, when they are combined with GLS they become very powerful optimisation tools. Combining GLS with FLS is straightforward. The key idea is to associate solution features to sub-neighbourhoods. The associations to be made are such that for each feature we know which sub-neighbourhoods contain moves that have an immediate effect upon the state of the feature (i.e. moves that remove the feature from the solution). The combination of the GLS algorithm with a generic FLS algorithm is depicted in Figure 2.2.

The procedure *GuidedFastLocalSearch* in Figure 2.2 works as follows. Initially, all the activation bits are set to 1 and FLS is allowed to reach the first local minimum (i.e. all bits 0). Thereafter, and whenever a feature is penalised, the bits of the associated sub-neighbourhoods and only those are set to 1. In this way, after the first local minimum, fast local search calls examine only a number of sub-neighbourhoods and in particular those which associate to the features just penalised. This dramatically speeds up GLS. Moreover, local search is focusing on removing the penalised features from the solution instead of considering all possible modifications.

```
procedure GuidedFastLocalSearch(S, g, λ, [I_1, ...,I_M], [c_1,...,c_M], M, L)
begin
        k ← 0; s_0 ← random or heuristically generated solution in S;
        for i ←1 until M do p_i ← 0; /* set all penalties to 0 */
        for i ←1 until L do bit_i ← 1; /* set all sub-neighbourhoods to the active state */
        while StoppingCriterion do
        begin
                h ← g + λ * ∑p_i*I_i ;
                s_{k+1} ← FastLocalSearch(s_k, h,[bit_1, …,bit_L], L);
                for i ←1 until M do util_i ← I_i(s_{k+1}) * c_i / (1+p_i);
                for each i such that util_i is maximum do
                begin
                        p_i ← p_i + 1;
                        SetBits ← SubNeighbourhoodsForFeature(i);
                        /* activate sub-neighbourhoods relating to feature i penalised */
                        for each bit b in SetBits do b ← 1;
                end
                k ← k+1;
        end
        s* ← best solution found with respect to cost function g;
        return s*;
end

procedure FastLocalSeach(s, h, [bit_1, …,bit_L], L)
begin
        while ∃bit, bit = l do
                for i ←1 until L do
                begin
                        if bit_i = 1 then /* search sub-neighbourhood for improving moves */
                        begin
                                Moves ← set of moves in sub-neighbourhood i;
                                for each move m in Moves do
                                begin
                                        s′ ← m(s);
                                        /* s′ is the solution generated by move m when applied to s */
                                        if h(s′) < h(s) then /* for minimisation */
                                        begin
                                                bit_i ← 1;
                                                SetBits ← SubNeighbourhoodsForMove(m);
                                                /* spread activation to other sub-neighbourhoods */
                                                for each bit b in SetBits do b ← 1;
                                                s ← s′;
                                                goto ImprovingMoveFound
                                        end
                                end
                                bit_i ← 0; /* no improving move found */
                        end
ImprovingMoveFound:     continue;
                end;
        return s;
end
```

where S: search space, g: cost function, h: augmented cost function, λ: regularisation parameter, I_i: indicator function for feature i, c_i: cost for feature i, M: number of features, L: number of sub-neighbourhoods, p_i: penalty for feature i, bit_i: activation bit for sub-neighbourhood i, SubNeighbourhoodsForFeature(i): procedure which returns the bits of the sub-neighbourhoods corresponding to feature i, and SubNeighbourhoodsForMove(m): procedure which returns the bits of the sub-neighbourhoods to spread activation to when move m is performed.

Figure 2.2 Guided Local Search combined with Fast Local Search in pseudocode

Apart from the combination of GLS with fast local search, other variations of GLS to be presented in the applications include:

- features with variable costs where the cost of a feature is calculated during search and in the context of a particular local minimum (see chapter 4)
- penalties with limited duration (see chapter 4)
- multiple feature sets where each feature set is processed in parallel by a different penalty modification procedure (see chapter 4)
- feature set hierarchies where more important features overshadow less important feature sets in the penalty modification procedure (see chapter 5).

Before presenting the applications of GLS, we examine some of the links between GLS and other general optimisation methods also based on local search.

2.9 Connections with Other General Optimisation Techniques

2.9.1 Simulated Annealing

Non-monotonic temperature reduction schemes used in SA (see section 1.4) also referred to as *re-annealing* or *re-heating* schemes are of particular interest in relation to the work presented in this thesis. In these schemes, the temperature is decreased as well as increased in a attempt to remedy the problem that the annealing process eventually settles down failing to continuously explore good solutions. In a typical SA, good solutions are mainly visited during the mid and low parts of the cooling schedule. For resolving this problem, it has been even suggested annealing at a constant temperature high enough to escape local minima but also low enough to visit them [Con90]. It is seems extremely difficult to find such a temperature because it has

to be landscape dependent (i.e. instance dependent) if not dependent of the area of the search space currently searched.

Guided Local Search presented in this thesis can be seen as addressing this problem of visiting local minima but also being able to escape from them. Instead of random up-hill moves, penalties are utilised to force local search out of local minima. The amount of penalty applied is progressively increased in units of appropriate magnitude (i.e. parameter λ) until the method escapes from the local minimum. GLS can be seen adapting to the different parts of the landscape. The algorithm is continuously visiting new solutions rather than converging to any particular solution as SA does.

Another important difference between this work and SA is that GLS is a deterministic algorithm. This is also the case for a wide number of algorithms developed under the tabu search framework.

2.9.2 Tabu Search

GLS has close links with tabu search. Both techniques can be seen as using information (historical in the case of tabu search, prior and historical information in the case of GLS) to impose constraints on local search either by modifying the neighbourhood (tabu search) or by modifying the cost function to be minimised (GLS). Let us consider the neighbourhood graph where each node is a solution to the problem and the arcs are the moves which transform one solution to another. GLS adopts a “solution or node”-centred approach to constrain local search by elevating the cost of specific nodes (i.e. solutions), rather than the “move or arc”-centred approach adopted by many tabu search variants which prevents local search from traversing specific arcs (i.e. executing moves which are tabu). The two approaches can be seen to

be seeking the same goal (i.e. guide local search by using constraints) though they use different means to achieve that.

Solution attributes used in tabu search can been seen as corresponding to the solution features used in GLS. However, constraints on solution attributes by tabu search may take many forms (i.e. tabu lists, frequency-based penalties) while in GLS a single mechanism is used which utilises indicator functions to introduce constraints on solution features.

Rather than elevate selected penalties to drive the search out of a local minimum, as GLS does, the typical tabu search approach seeks a best move to escape from a local minimum based on the current evaluation function, influenced by prior memory and by candidate list strategies. Penalties in tabu search are customarily applied to selected attributes only after the move is made, as a way of preventing a return. Tabu search also typically maintains a recency-based memory to provide a mechanism to avoid reinstating selected attribute combinations found in recently generated solutions. Diversification strategies that make use of frequency-based memory are generally activated periodically, rather than continuously as in GLS.

A more detailed list of the various search elements that are present in both techniques along with the ways they are realised in each individual technique is given in Table 2.1. As we can see in this table, despite the differences between tabu and guided local search, there is common ground in many areas. This common ground may well be utilised in the future to define a more abstract class of methods which one may call *Intelligent Search* methods.

	Tabu Search	**Guided Local Search**
Local search guidance mechanism	modified neighbourhood, intelligent restarts	modified cost function
Information used	mainly the moves executed but also transition & residence frequencies and elite solution sets	feature costs, local minima visited
Constraints	• hard constraints on moves or solution attributes based on moves recently executed, aspiration criteria override the hard constraints • soft constraints on moves or solution attributes based on transition or residence frequencies	soft constraints on solution features based on search plan for distributing search effort taking into account the local gradients
Memory Utilised	• tabu lists recording attributes of moves recently executed • frequency based memory recording the frequency of moves or solution attributes during search	memory of penalty modification actions taken by GLS also used for recording penalties on features
Intervention Period	• every iteration (recency-based memory, some types of diversification strategies) • every N iterations or when local search fails to discover new better solutions (intensification strategies, diversification strategies)	at a local minimum of the augmented cost function
Search Objectives	• avoid getting trapped in local minima and reversing changes created by the moves (proactive approach). • Intensification: restart when slow progress (reactive approach) • Diversification: examine history and penalise moves frequently executed or solution attributes frequently appearing in solutions (reactive approach)	• escape from local minima (reactive approach) • plan and distribute search efforts in the short or long term according to feature costs taking into account the local gradients (proactive approach).
Intensification - Diversification balance	-	The lambda parameter of GLS controls that. • Low lambda leads to intensification (due to cost function term in the augmented cost function). • High lambda leads to diversification (due to penalty function term in the augmented cost).
Neighbourhood Reduction Mechanism	Candidate Lists Strategies	Fast Local Search fully integrated with the diversification - intensification mechanisms of GLS

Table 2.1 Links between Guided Local Search and Tabu Search methods.

2.10 GLS Applications

GLS is a generalisation of GENET and as such it can be applied with the same success as GENET in any of the applications of the latter (i.e. CSP problems). Apart from that, GLS has been successfully applied to a set of seven problems in combinatorial optimisation. This set includes the famous Travelling Salesman and Quadratic Assignment problems, the real-world problems of Radio Link Frequency Assignment, Workforce Scheduling, Bandwidth Packing and Maximum Channel Assignment, and finally a continuous Nonconvex Optimisation problem. FLS has also been applied to all these problems except for the Quadratic Assignment Problem and the NonConvex Optimisation problem. All these applications of GLS and FLS are examined in this thesis except for the Bandwidth Packing and Maximum Channel Assignment problems for which GLS and FLS have been applied in a way similar to that for the Workforce Scheduling problem examined in chapter 6. However, demonstration programs have been developed for both the Bandwidth Packing and Maximum Channel Assignment problems which can be obtained via WWW at http://cswww.essex.ac.uk/CSP/demos.

Chapter 3

Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is one of the most famous problems in combinatorial optimisation. In this chapter, we are going to examine how guided local search and fast local search can be applied to the problem. The combination of GLS and FLS with TSP local search heuristics of different efficiency and effectiveness will be studied in an effort to determine the dependence of GLS on local search. Comparisons will be made with some of the best TSP heuristic algorithms and general optimisation techniques which will demonstrate the advantages of GLS over alternative heuristic approaches suggested so far for this problem.

3.1 The Problem

There are many variations of the Travelling Salesman Problem (TSP). In this work, we examine the classic symmetric TSP. The problem is defined by N cities and a symmetric distance matrix $D=[d_{ij}]$ which gives the distance between any two cities i

and *j*. The goal in TSP is to find a tour (i.e. closed path) which visits each city exactly once and is of minimum length. A tour can be represented as a cyclic permutation π on the *N* cities if we interpret $\pi(i)$ to be the city visited after city *i*, *i* = 1,... ,*N*. The cost of a permutation is defined as:

Eq. 3.1
$$g(\pi) = \sum_{i=1}^{N} d_{i\pi(i)}$$

and gives the cost function of the TSP [PS82].

Recent and comprehensive surveys of TSP methods are those by Laporte [Lap92], Reinelt [Rei94] and Johnson & McGeoch [JM95]. The reader may also refer to [LLKS85] for a classical text on the TSP. The state of the art is that problems up to 1,000,000 cities are within the reach of specialised approximation algorithms [Ben92]. Moreover, the optimal solutions have been found and proven for non-trivial problems of size up to 7397 cities [JM95]. Nowadays, TSP plays a very important role in the development and testing of new optimisation techniques. In this context, we examine how guided local search and fast local search can be applied to this problem.

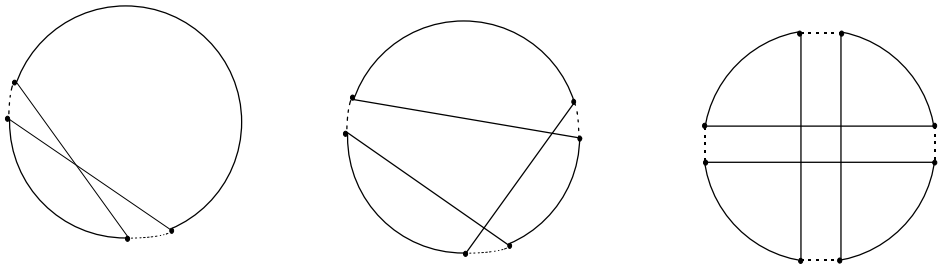
3.2 Local Search Heuristics for the TSP

Local search for the TSP is synonymous with *k*-Opt moves. Using *k*-Opt moves, neighbouring solutions can be obtained by deleting *k* edges from the current tour and reconnecting the resulting paths using *k* new edges. The *k*-Opt moves are the basis of the three most famous local search heuristics for the TSP, namely *2-Opt* [Cro58], *3-Opt* [Lin65] and *Lin-Kernighan (LK)* [LK73]. These heuristics define neighbourhood structures which can be searched by the different neighbourhood search schemes described in sections 1.3 and 2.8, leading to many local optimisation

algorithms for the TSP. The neighbourhood structures defined by 2-Opt, 3-Opt and LK are as follows [Joh90]:

2-Opt. A neighbouring solution is obtained from the current solution by deleting two edges, reversing one of the resulting paths and reconnecting the tour (see Figure 3.1). The worst case complexity for searching the neighbourhood defined by 2-Opt is $O(n^2)$.

3-Opt. In this case, three edges are deleted. The three resulting paths are put together in a new way, possibly reversing one or more of them (see Figure 3.1). 3-Opt is much more effective than 2-Opt, though the size of the neighbourhood (possible 3-Opt moves) is larger and hence more time-consuming to search. The worst case complexity for searching the neighbourhood defined by 3-Opt is $O(n^3)$.



a) 2-Opt move **b)** 3-Opt move **c)** Non-sequential 4-Opt move

Figure 3.1 k-Opt moves for the TSP

Lin-Kernighan (LK). One would expect "4-Opt" to be the next step after 3-Opt but actually that is not the case. The reason is that 4-Opt neighbours can be remotely apart because "non-sequential" exchanges such as that shown in Figure 3.1 are possible for $k \geq 4$. To improve 3-Opt further, Lin and Kernighan developed a sophisticated edge exchange procedure where the number k of edges to be exchanged is variable [LK73]. The algorithm is mentioned in the literature as the *Lin-Kernighan* (LK) algorithm and it was considered for many years to be the "uncontested champion" of local search

heuristics for the TSP. Lin-Kernighan uses a very complex neighbourhood structure which we will briefly describe here.

LK, instead of examining a particular 2-Opt or 3-Opt exchange, is building an exchange of variable size *k* by sequentially deleting and adding edges to the current tour while maintaining tour feasibility. Given node t_1 in tour *T* as a starting point: In step *m* of this sequential building of the exchange: edge (t_1, t_{2m}) is deleted, edge (t_{2m}, t_{2m+1}) is added, and then edge (t_{2m+1}, t_{2m+2}) is picked so that deleting edge (t_{2m+1}, t_{2m+2}) and joining edge (t_{2m+2}, t_1) will close up the tour giving tour T_m. The edge (t_{2m+2}, t_1) is deleted if and when step *m+1* is executed. The first three steps of this mechanism are illustrated in Figure 3.2.

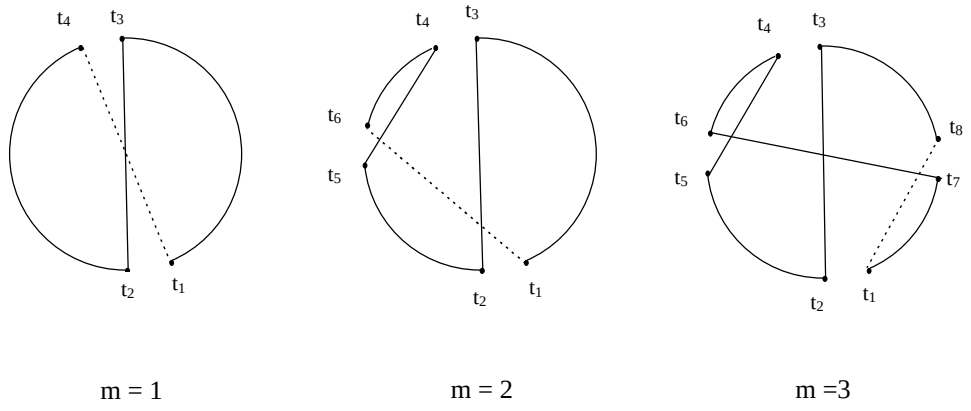


Figure 3.2 The first three steps of the Lin-Kernighan edge exchange mechanism

As we can see in this figure, the method is essentially executing a sequence of 2-Opt moves. The length of these sequences (i.e. depth of search) is controlled by the LK's *gain criterion* which limits the number of the sequences examined. In addition to that, limited backtracking is used to examine the sequences that can be generated if a number of different edges are selected for addition at steps 1 and 2 of the process.

The neighbourhood structure described so far, although it provides the depth needed, is lacking breadth, potentially missing improving 3-Opt moves. To gain breadth, LK

temporarily allows tour infeasibility, examining the so-called “infeasibility” moves which consider various choices for nodes t_4 to t_8 in the sequence generation process, examining all possible 3-Opt moves and more. Figure 3.3 illustrates the infeasibility-move mechanism. The interested reader may refer to the original paper by Lin and Kernighan for a more elaborate description of this mechanism.

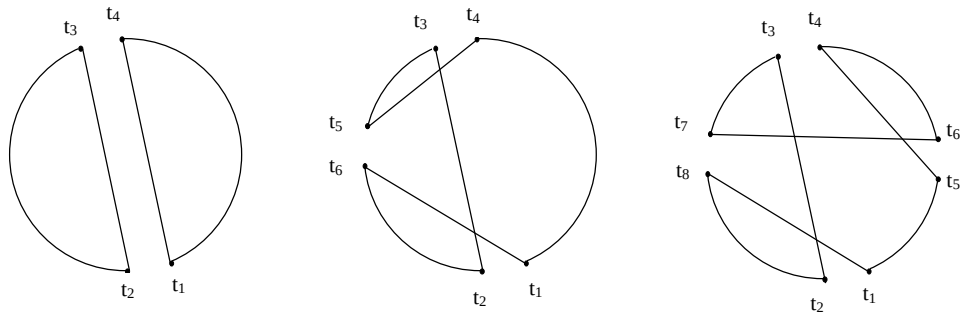


Figure 3.3 Lin-Kerhighan’s infeasibility moves

LK is the standard benchmark against which all heuristic methods are tested. The worst case complexity for searching the LK neighbourhood is $O(n^5)$.

Implementations of 2-Opt, 3-Opt and LK-based local search methods may vary in performance. A very good reference for efficiently implementing local search procedures based on 2-Opt and 3-Opt is that by Bentley [Ben92]. In addition to that, Reinelt [Rei94] and also Johnson and McGeoch [JM95] describe some improvements that are commonly incorporated in local search algorithms for the TSP. We will refer to some of them later in this chapter. The best reference for the LK algorithm is the original paper by Lin and Kernighan [LK73]. In addition to that, Johnson and McGeoch [JM95] provide a good insight into the algorithm and its operations along with information on the many variants of the method. A modified LK version which avoids the complex infeasibility moves without significant impact on performance is described in [MM93].

Fast local search and guided local search can be combined with the neighbourhood structures of 2-Opt, 3-Opt and LK with minimal effort. This will become evident in the next sections where fast local search and guided local search for the TSP are presented and discussed.

3.3 Fast Local Search Applied to the TSP

A fast local search procedure for the TSP using 2-Opt has already been suggested by Bentley [Ben92]. Under the name *Don't Look Bits*, the same approach has been used in the context of 2-Opt, 3-Opt and LK by Codenotti et al. [CMMR96] to reduce the running times of these heuristics in very large TSP instances. More recently, Johnson et al. [JBMR96] also use the technique to speed up their LK variant (see [JM95]). In the following, we are going to describe how fast local search variants of 2-Opt, 3-Opt and LK can be developed on the guidelines for fast local search presented in section 2.8.

2-Opt, 3-Opt and LK-based local search procedures are seeking tour improvements by considering for exchange each individual edge in the current tour and trying to extend this exchange to include one (2-Opt), two (3-Opt) or more (LK) other edges from the tour. Usually, each city is visited in tour order and **one** or **both**[3] the edges adjacent to the city are checked if they can lead to an edge exchange which improves the solution.

We can exploit the way local search works on the TSP to partition the neighbourhood in sub-neighbourhoods as required by fast local search. Each city in the problem may be seen as defining a sub-neighbourhood which contains all edge exchanges

[3] In our work, if approximations are used such as nearest neighbour lists or fast local search then both edges adjacent to a city are examined, otherwise only one of the edges adjacent to the city is examined.

originating from either one of the edges adjacent to the city. For a problem with *N* cities, the neighbourhood is partitioned into *N* sub-neighbourhoods, one for each city in the instance. Given the sub-neighbourhoods, fast local search for the TSP works in the following way (see also Figure 2.2).

Initially all sub-neighbourhoods are active. The scanning of the sub-neighbourhoods, defined by the cities, is done in an arbitrary static order (e.g. from 1st to *N*th city). Each time an active sub-neighbourhood is found, it is searched for improving moves. This involves trying either edge adjacent to the city as bases for 2-Opt, 3-Opt or LK edge exchanges, depending on the heuristic used. If a sub-neighbourhood does not contain any improving moves then it becomes inactive (i.e. bit is set to 0). Otherwise, the first improving move found is performed and the cities (corresponding sub-neighbourhoods) at the ends of the edges involved (deleted or added by the move) are activated (i.e. bits are set to 1). This causes the sub-neighbourhood where the move was found to remain active and also a number of other sub-neighbourhoods to be activated. The process always continues with the next sub-neighbourhood in the static order. If ever a full rotation around the static order is completed without making a move, the process terminates and returns the tour found. The tour is declared 2-Optimal, 3-Optimal or LK-Optimal, depending on the type of the *k*-Opt moves used.

3.3.1 Local Search Procedures for the TSP

Apart from fast local search, first improvement and best improvement local search (see section 1.3) can also be applied to the TSP. First improvement local search immediately performs improving moves while best improvement (greedy) local search performs the best move found after searching the complete neighbourhood.

Fast local search for the TSP described above can be easily converted to first improvement local search by searching all sub-neighbourhoods irrespective of their state (active or inactive). The termination criterion remains the same with fast local search: that is, to stop the search when a full rotation of the static order is completed without making a move. The LK algorithm as originally proposed by Lin and Kernighan [LK73] performs first improvement local search.

Fast local search can also be modified to perform best improvement local search. In this case, the best move is selected and performed after all the sub-neighbourhoods have been exhaustively searched. The algorithm stops when a solution is reached where no improving move can be found. The scheme is very time consuming to be combined with the 3-Opt and LK neighbourhood structures and it is mainly intended for use with 2-Opt. Considering the above options, we implemented seven local search variants for the TSP (implementation details will be given later in this chapter). These variants were derived by combining the different search schemes at the neighbourhood level (i.e. fast, first improvement, and best improvement local search) with any of the 2-Opt, 3-Opt, or LK neighbourhood structures. Table 3.1 illustrates the variants and also the names we will use to distinguish them in the rest of the chapter.

Name	**Local Search Type**	**Neighbourhood Type**
BI-2Opt	Best Improvement	2-Opt
FI-2Opt	First Improvement	2-Opt
FLS-2Opt	Fast Local Search	2-Opt
FI-3Opt	First Improvement	3-Opt
FLS-3Opt	Fast Local Search	3-Opt
FI-LK	First Improvement	LK
FLS-LK	Fast Local Search	LK

Table 3.1 Local search procedures implemented for the study of GLS on the TSP.

3.4 Guided Local Search Applied to the TSP

3.4.1 Solution Features and Augmented Cost Function

The first step in the process of applying GLS to a problem is to find a set of solution features that are accountable for part of the overall solution cost. For the TSP, a tour includes a number of edges and the solution cost (tour length) is given by the sum of the lengths of the edges in the tour (see Eq. 3.1). Edges are ideal features for the TSP. First, they can be used to define solution properties (a tour either includes an edge or not) and second, they carry a cost equal to the edge length, as this is given by the distance matrix $D=[d_{ij}]$ of the problem. A set of features can be defined by considering all possible undirected edges e_{ij} ($i = 1..N$, $j = i+1..N$, $i \neq j$) that may appear in a tour with feature costs given by the edge lengths d_{ij}. Each edge e_{ij} connecting cities i and city j is attached a penalty p_{ij} initially set to 0 which is increased by GLS during search. These edge penalties can be arranged in a symmetric penalty matrix $P=[p_{ij}]$. As mentioned in section 2.5, penalties have to be combined with the problem's cost function to form the augmented cost function which is minimised by local search. This can be done by considering the auxiliary distance matrix:

Eq. 3.2
$$D' = D + \lambda \cdot P = [d_{ij} + \lambda \cdot p_{ij}] .$$

Local search must use D' instead of D in move evaluations. GLS modifies P and (through that) D' whenever local search reaches a local minimum. The edges penalised in a local minimum are selected according to the utility function (Eq. 2.5), which for the TSP takes the form:

Eq. 3.3

$$Util\left(tour, e_{ij}\right) = I_{e_{ij}}(tour) \cdot \frac{d_{ij}}{1 + p_{ij}},$$

where

Eq. 3.4

$$I_{e_{ij}}(tour) = \begin{cases} 1, & e_{ij} \in tour \\ 0, & e_{ij} \notin tour \end{cases}.$$

3.4.2 Combining GLS with TSP Local Search Procedures

GLS as depicted in Figure 2.1 makes no assumptions about the internal mechanisms of local search and therefore can be combined with any local search algorithm for the problem, no matter how complex this algorithm is.

The TSP local searches of section 3.3.1 to be integrated with GLS need only to be implemented as procedures which, provided with a starting tour, return a locally optimal tour with respect to the neighbourhood considered. The distance matrix used by local search is the auxiliary matrix D' described in the last section. A reference to the matrix D is still needed to enable the detection of better solutions whenever moves are executed and new solutions are visited. There is no need to keep track of the value of the augmented cost function since local search heuristics make move evaluations using cost differences rather than re-computing the cost function from scratch.

Interfacing GLS with fast local searches for the TSP requires a little more effort (see also Figure 2.2). In particular, each time we penalise an edge in GLS, the sub-neighbourhoods corresponding to the cities at the ends of this edge are activated (i.e. bits set to 1). After the first local minimum, calls to fast local search start by examining only a number of sub-neighbourhoods and in particular those which associate to the edges just penalised. Activation may spread to a limited number of other sub-neighbourhoods because of the moves performed though, in general, local

search quickly settles in a new local minimum. This dramatically speeds up GLS, forcing local search to focus on edge exchanges that remove penalised edges instead of evaluating all possible moves.

3.4.3 How GLS Works on the TSP

Let us now give an overview of the way GLS works on the TSP. Starting from an arbitrary solution, local search is invoked to find a local minimum. GLS penalises one or more of the edges appearing in the local minimum, using the utility function Eq. 3.3 to select them. After the penalties have been increased, local search is restarted from the last local minimum to search for a new local minimum. If we are using fast local search then the sub-neighbourhoods (i.e. cities) at the ends of the edges penalised need also to be activated. When a new local minimum is found or local search cannot escape from the current local minimum, penalties are increased again and so forth.

The GLS algorithm constantly attempts to remove edges appearing in local minima by penalising them. The effort invested by GLS to remove an edge depends on the edge length. The longer the edge, the greater the effort put in by GLS. The effect of this effort depends on the regularisation parameter λ of GLS. A high λ causes GLS decisions to be in full control of local search, overriding any local gradient information while a low λ causes GLS to escape from local minima with great difficulty, requiring many penalty cycles before a move is executed. However, there is always a range of values for λ for which the moves selected aim at the combined objective to improve the solution (taking into account the gradient) and also remove the penalised edges (taking into account the GLS decisions). If longer edges persist in appearing in solutions despite the penalties, the algorithm will diversify its choices, trying to remove shorter edges too.

As the penalties build up for both bad and good edges frequently appearing in local minima, the algorithm starts exploring new regions in the search space, incorporating edges not previously seen and therefore not penalised. The speed of this "continuous" diversification of search is controlled by the parameter λ. A low λ slows down the diversification process, allowing the algorithm to spend more time in the current area before it is forced by the penalties to explore other areas. Conversely, a high λ speeds up diversification, at the expense of intensification.

From another viewpoint, GLS realises a "selective" diversification which pursues many more choices for long edges than short edges by penalising the former many more times than the later. This selective diversification achieves the goal of distributing the search effort according to prior information as expressed by the edge lengths. Selective diversification is smoothly combined with the goal of intensifying search by setting λ to a value low enough to allow the local search gradients to influence the course of local search. Escaping from local minima comes at no expense because of the penalties but alone without the goal of distributing the search effort, as implemented by the selective penalty modification mechanism, is not enough to produce high quality solutions.

3.5 Evaluation of GLS in the TSP

To investigate the behaviour of GLS on the TSP, we conducted a series of experiments. The results presented in subsequent sections attempt to provide a comprehensive picture of the performance of GLS on the TSP. First, we examine the combination of GLS with 2-Opt, the simplest of the TSP heuristics. The benefits from using fast local search instead of best improvement local search are clearly demonstrated, along with the ability of GLS to find high quality solutions in small to

medium size problems. These results for GLS are compared with results for Simulated Annealing and Tabu Search when these techniques use the 2-Opt heuristic.

From there on, we focus on efficient techniques for the TSP based on GLS. The different combinations of GLS with the local search procedures of Table 3.1 are examined and conclusions are drawn on the relation between GLS and local search. Efficient GLS variants are compared with methods based on the Lin-Kernighan algorithm (known to be the best heuristic techniques for the TSP).

3.5.1 Experimental Setting

In the experiments conducted, we used problems from the publicly available library of TSP problems, TSPLIB [Rei91]. Most of the instances included in TSPLIB have already been solved to optimality and they have been used in many papers in the TSP literature.

For each algorithm evaluated, ten runs from different **random** initial solutions were performed and the various performance measures (solution quality, running time etc.) were averaged. The solution quality was measured by the percentage *excess* above the best known solution (or optimal solution if known), as given by the formula:

Eq. 3.5
$$excess = \frac{\text{solution cost - best known solution cost}}{\text{best known solution cost}} \times 100.$$

Unless otherwise stated, all experiments were conducted on DEC Alpha 3000/600 machines (175 MHz) with algorithms implemented in GNU C++.

3.5.2 Regularisation Parameter λ

The only parameter of GLS which requires tuning is the regularisation parameter λ. The GLS algorithm performed well for a relatively wide range of values when we

tested it on problems from TSPLIB with either one of the 2-Opt, 3-Opt or LK heuristics. Experiments showed that GLS is quite tolerant to the choice of λ as long as λ is equal to a fraction of the average edge length in good solutions (e.g. local minima). These findings were expressed by the following equation for calculating λ:

Eq. 3.6
$$\lambda = a \cdot \frac{g(\text{local minimum})}{N} \ ,$$

where *g(local minimum)* is the cost of a local minimum tour produced by local search (e.g. first local minimum before penalties are applied) and *N* the number of cities in the instance. Eq. 3.6 introduces a parameter *a* which, although instance-dependent, results in good GLS performance for values in the more manageable range (0,1]. Experimenting with *a*, we found that it depends not only on the instance but also on the local search heuristic used. In general, there is an inverse relation between *a* and local search effectiveness. Not-so-effective local search heuristics such as 2-Opt require higher *a* values than more effective heuristics such as 3-Opt and LK. This is because the amount of penalty needed to escape from local minima decreases as the effectiveness of the heuristic increases and therefore lower values for *a* have to be used to allow the local gradients to affect the GLS decisions. For 2-Opt, 3-Opt and LK, the following ranges for *a* generated high quality solutions in the TSPLIB problems.

Heuristic	**Suggested range for *a***
2-Opt	$1/8 \leq a \leq ½$
3-Opt	$1/10 \leq a \leq ¼$
LK	$1/12 \leq a \leq 1/6$

Table 3.2 Suggested ranges for parameter a when GLS is combined with different TSP heuristics.

The lower bounds of these intervals represent typical values for *a* that enable GLS to escape from local minima at a *tolerable* rate. If values less than the lower bounds are used, then GLS requires too many penalty cycles to escape from local minima. In

general, the lower bounds depend on the local search heuristic used and also the structure of the landscape (i.e. depth of local minima). On the other hand, the upper bounds give a good indication of the maximum values for *a* that can still produce good solutions. If values greater than the upper bounds are used then the algorithm is exhibiting excessive bias towards removing long edges and failing to reach high quality local minima. In general, the upper bounds also depend on the local search heuristic used but they are mainly affected by the quality of the information contained in the feature costs (i.e. how accurate is the assumption that long edges are preferable over short edges in the particular instance).

3.6 Guided Local Search and 2-Opt

In this section, we look into the combination of GLS with the simple 2-Opt heuristic. More specifically, we present results for GLS with best improvement 2-Opt local search (BI-2Opt) and fast 2-Opt local search (FLS-2Opt). The set of problems used in the experiments consisted of 28 small to medium size TSPs from 48 to 318 cities all from TSPLIB. The stopping criterion used was a limit on the number of iterations not to be exceeded. An iteration for GLS with BI-2Opt was considered one local search iteration (i.e. complete search of the neighbourhood) and for GLS with FLS-2Opt, a call to fast local search as in Figure 2.2. The iteration limit for both algorithms was set to 200,000 iterations. In both cases, we tried to provide the GLS variants with plenty of resources in order to reach the maximum of their performance.

The exact value of λ used in the runs was manually determined by running a number of test runs and observing the sequence of solutions generated by the algorithm. A well-tuned algorithm generates a smooth sequence of gradually improving solutions. A not so well tuned algorithm either progresses very slowly (λ is lower than it should

Problem	GLS with BI-2Opt			GLS with FLS-2Opt		
	optimal runs out of 10	Mean Excess (%)	Mean CPU Time (sec)	optimal runs out of 10	Mean Excess(%)	Mean CPU Time (sec)
att48	10	0.0	0.77	10	0.0	0.4
eil51	10	0.0	1.62	10	0.0	0.46
st70	10	0.0	7.68	10	0.0	1.2
eil76	10	0.0	3.83	10	0.0	0.97
pr76	10	0.0	15.1	10	0.0	3.01
gr96	10	0.0	16.48	10	0.0	2.26
kroA100	10	0.0	11.27	10	0.0	1.25
kroB100	10	0.0	16.36	10	0.0	2.46
kroC100	10	0.0	12.2	10	0.0	0.74
kroD100	10	0.0	12.94	10	0.0	1.78
kroE100	10	0.0	35.68	10	0.0	2.46
rd100	10	0.0	10.75	10	0.0	2.74
eil101	10	0.0	19.49	10	0.0	2.37
lin105	10	0.0	17.46	10	0.0	2.06
pr107	10	0.0	150.28	10	0.0	5.41
pr124	10	0.0	22.47	10	0.0	1.56
bier127	10	0.0	254.36	10	0.0	24.67
pr136	9	0.0009	416.78	10	0.0	32.16
gr137	10	0.0	66.54	10	0.0	7.82
pr144	10	0.0	52.84	10	0.0	6.95
kroA150	10	0.0	257.06	10	0.0	7.03
kroB150	10	0.0	289.02	10	0.0	44.85
u159	10	0.0	74.35	10	0.0	6.9
rat195	8	0.01	525.48	10	0.0	55.15
d198	0	0.08	1998.37	0	0.05	353.97
kroA200	10	0.0	614.6	10	0.0	50.16
kroB200	10	0.0	665.3	10	0.0	61.79
lin318	8	0.01	4484.4	9	0.005	346.44

Table 3.3 Performance of 2-Opt based variants of GLS on small to medium size TSP instances.

be) or very quickly finds no more than a handful of good local minima (λ is higher than it should be). The values for λ determined in this way were corresponding to values for *a* around 0.3. Ten runs from different random solutions were performed on each instance included in the set of problems and the various performance measures (excess, running time to reach the best solution etc.) were averaged. The results obtained are presented in Table 3.3.

Both GLS variants found solutions with cost equal to the optimal cost in the majority of runs. GLS with BI-2Opt failed to find the optimal solutions (as reported by Reinelt in [Rei91] and also [Rei94]) in only 15 out of the total 280 runs. From another

viewpoint, the algorithm was successful in finding the optimal solution in 94.6% of the runs. Ten out of the 14 failures referred to a single instance namely *d198*. However, the solutions found for *d198* were of high quality and on average within 0.08% of optimality.

GLS with FLS-2Opt found the optimal solutions in 3 more runs than GLS with BI-2Opt, missing the optimal solution in only 11 out of the 280 runs (96.07% success rate). In particular, the algorithm missed only once the optimal solution for *lin318* but still found no optimal solution for *d198* which proved to be a relatively 'hard' problem for both variants. GLS using fast local search was on average ten times faster than GLS using best improvement local search and that without compromising on solution quality. In the worst case (*att48*), it was two times faster while in the best case (*kroA150*) it was thirty seven times faster. Remarkably, GLS with fast local search was able in most problems to find a solution with cost equal to the optimum (already known) in less than 10 seconds of CPU time on the DEC Alpha 3000/600 machines used.

The results presented in this section clearly demonstrate the ability of GLS even when combined with 2-Opt the simplest of TSP heuristics to find consistently the optimal solutions for small to medium size TSPs. The use of fast local search introduces substantial savings in running times without compromising in solution quality.

3.6.1 Comparison with General Methods for the TSP

The above performance of GLS is remarkable considering that GLS is not an exact method and that in this case it only used the short-sighted 2-Opt heuristic. Searching the related TSP literature, we could not find any other approximation methods that use only the simple 2-Opt move and consistently find optimal solutions for problems up to

318 cities. Only the Iterated Lin-Kernighan algorithm and its variants [Joh90, JM95, JBMR96] share the same consistency in reaching the optimal solutions. These algorithms will be considered later in this chapter.

A meaningful comparison that can be made is between GLS using 2-Opt and other general methods that also use the same heuristic. For that purpose, we implemented simulated annealing [KGV83] and a tabu search variant for the TSP suggested by Knox [Kno94].

3.6.2 Simulated Annealing

The Simulated Annealing (SA) algorithm implemented for the TSP was the one described by Johnson in [Joh90] and uses geometric cooling schedules (see section 1.4.1). The algorithm generates random 2-Opt moves. If a move improves the cost of the current solution then it is always accepted. Moves that do not improve the cost of the current solution are accepted with probability:

$$e^{\frac{-\Delta}{T}}$$

where Δ is the difference in cost due to the move and T is the current temperature. In the final runs, we started the algorithm from a relatively high temperature (around 50% of moves were accepted). At each temperature level the algorithm was allowed to perform a constant number of trials to reach equilibrium. After reaching equilibrium, the temperature was multiplied by the cooling rate a which was set to a high value (a = 0.9) . To stop the algorithm, we used the scheme with the counter described in [JAMS89].

3.6.3 Tabu Search

The tabu search variant implemented was the one proposed by Knox [Kno94] using a combination of *tabu restrictions* and *aspiration level criteria.* The method is briefly described in here.

Tabu search performs best improvement local search selecting the best move in the neighbourhood but only amongst those not characterised as *tabu*. Determining the tabu status of a move is very important in tabu search and holds the key for the development of efficient recency-based memory (see section 1.5).

In this tabu search variant for the TSP, a 2-Opt move is classified as tabu only if both added edges of the exchange are on the tabu list. If one or both of the added edges are not on the tabu list, then the candidate move is not classified as tabu. Updating the tabu list involves placing the deleted edges of the 2-Opt exchanges performed on the list. If the list is full, the oldest elements of the list are replaced by the new deleted edge information.

In order for a 2-Opt exchange to override tabu status, both added edges of the exchange must pass the aspiration test. An individual edge passes the aspiration test if the new tour resulting from the candidate exchange is better than the aspiration values associated with the edge. The aspiration values of edges are the tour cost which exists prior to making the candidate 2-Opt move. Only edges deleted by the exchanges performed have their values updated.

For the experiments reported here, the tabu list size was set to $3N$ (where N is the number of cities in the problem) as suggested by Knox [Kno94]. Tabu search was allowed to run for 200,000 iterations which is equivalent in terms of number of moves evaluated to the number of iterations GLS with BI-2Opt was given on the same instances.

Problem Name	GLS with FLS-2Opt		Simulated Annealing		Tabu Search		Repeated BI-2Opt (200,000 iterations)	
	Mean Excess (%)	Mean CPU Time (sec)	Mean Excess (%)	Mean CPU Time (sec)	Mean Excess (%)	Mean CPU Time (sec)	Mean Excess (%)	Mean CPU Time (sec)
eil51	0.0	0.46	0.73	6.34	0.0	1.14	0.23	42.4
eil76	0.0	0.97	1.21	18.0	0.0	5.24	1.85	153.45
eil101	0.0	2.37	1.76	33.29	0.0	61.41	3.97	319.15
kroA100	0.0	1.25	0.42	37.36	0.0	21.34	0.34	706.35
kroC100	0.0	0.74	0.80	36.58	0.25	4.80	0.33	1301.98
kroA150	0.0	7.03	1.86	103.32	0.03	413.06	1.41	3290.95
kroA200	0.0	50.16	1.04	229.38	0.72	776.93	1.7	731.1
lin318	0.005	346.44	1.34	829.46	1.31	2672.80	3.11	9771.28

Table 3.4 GLS, Simulated Annealing, and Tabu Search performance on TSPLIB instances.

3.6.4 Simulated Annealing and Tabu Search Compared with GLS

Simulated annealing and tabu search were tested on 8 instances from the greater set of 28 instances mentioned above. The results were averaged as with GLS. Table 3.4 illustrates the results for simulated annealing and tabu search compared with those for GLS with FLS-2Opt on the same instances. Results are also contrasted with the best solution found by repeating BI-2Opt starting from random tours until a total of 200,000 local search iterations were completed.

As we can see in Table 3.4, the superiority of GLS over the tabu search variant and simulated annealing is evident. The tabu search variant found easily the optimal solutions for small problems and it scaled well for larger problems. However, it was many times slower than GLS and moreover failed to reach the solution quality of GLS in the larger problems. Simulated annealing had a consistent behaviour finding good solutions for all problems but failed to reach the optimal solutions in all but 3 runs. All three meta-heuristics significantly improved over the performance of repeated 2-Opt.

3.7 Efficient GLS Variants for the TSP

In order to study the combinations of GLS with higher order heuristics such as 3-Opt and LK, a library of TSP local search procedures was developed in C++. The library comprises all local search procedures of Table 3.1 and allows combinations of GLS with any one of these procedures. Furthermore, a number of approximations (not used in the GLS of section 3.6) are adopted which further reduce the computation times of local search and GLS as reported in section 3.6. In the rest of the chapter, we will examine and report results for these efficient variants of GLS.

The most significant approximation introduced is the use of a pre-processing stage which finds and sorts by distance the 20 nearest neighbours of each city in the instance. 2-Opt, 3-Opt and LK were considering in exchanges only edges to these 20 nearest neighbours (see also [Rei94, JM95]). Each time the penalty was increased for an edge, the nearest neighbour lists of the cities at the ends of the edge were reordered though no new neighbours were introduced.

To reduce the computation times required by 3-Opt, 3-Opt was implemented as two locality searches each of which looks for a "short enough" edge to extend further the exchange (see [Ben92] for details). The LK implementation was exactly as proposed by Lin and Kernighan [LK73] incorporating their lookahead and backtracking suggestions (i.e. backtracking at the first two levels of the sequence generation, considering at each step only the five smallest and available candidate edges that can be added to the tour and taking into account in the selection of the edges to be added the length of the edges to be deleted by these additions).

The library is portable to most UNIX machines though experiments reported in here were solely performed on DEC Alpha workstations 3000/600 (175 MHz) using a library executable generated by the GNU C++ compiler.

The set of problems used in the evaluation of the GLS variants included 20 problems from 48 to 1002 cities all from TSPLIB. For each variant tested, 10 runs were performed and 5 minutes of CPU time were allocated to each algorithm in each run. To measure the success of the variants, we considered the percentage excess above the optimal solution as in Eq. 3.5. The normalised lambda parameter *a* was provided as input to the program and λ was determined after the first local minimum using Eq. 3.6. For GLS variants using 2-Opt, *a* was set to $a = 1/6$ while the GLS variants based on 3-Opt used the slightly lower value $a = 1/8$ and the LK variants the even lower value $a = 1/10$. The full set of results for the various combinations of GLS with local search can be found in Appendix A. Next, we focus on selected results from this set.

3.7.1 Results for GLS with First Improvement Local Search

Figure 3.4 graphically illustrates the results for the first improvement versions of 2-Opt, 3-Opt and LK when combined with GLS. In this figure, we see that the

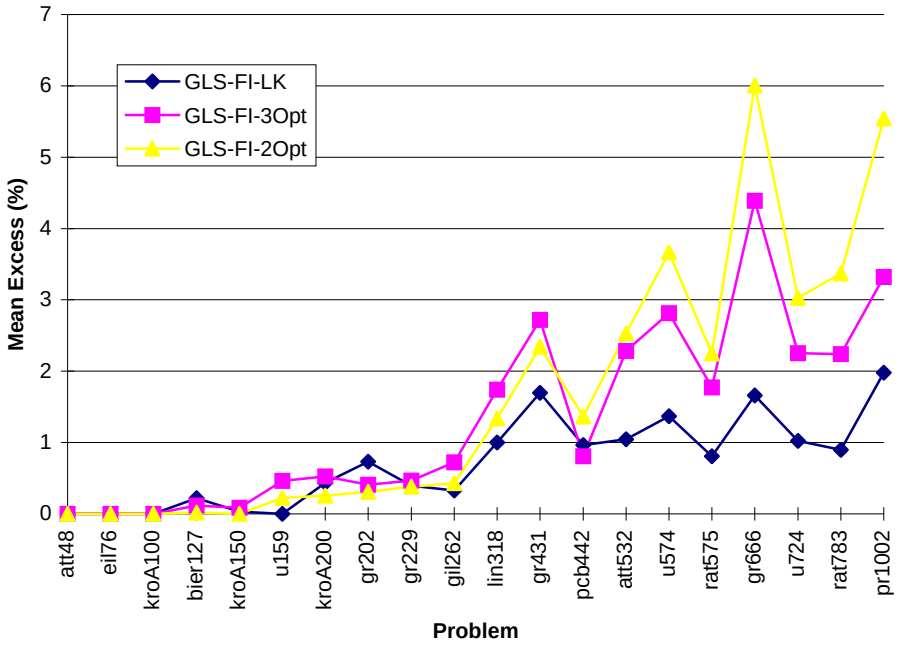


Figure 3.4 Performance of GLS variants using first improvement local search procedures

combination of GLS with FI-3Opt and FI-LK significantly improves over the performance of GLS with FI-2Opt especially when applied to large problems. FI-LK combined with GLS achieved the best performance amongst the three methods tested.

3.7.2 Results for GLS with Fast Local Search

Figure 3.5 graphically illustrates the results obtained for GLS when combined with the fast local search variants of 2-Opt, 3-Opt and LK. GLS with FI-LK (found to be best amongst the first improvement versions of GLS) is also displayed in the figure as a point of reference. In this figure, we can see that the fast local search variants of GLS are much better than the best of the first improvement local search variants (i.e. GLS-FI-LK). Another far more important observation is that for fast local search the 2-Opt variant is better than the 3-Opt variant which in turn is better than the LK variant. This is exactly the opposite order than one would have expected. One possible explanation can be derived by considering the strength of GLS. More specifically, FLS-2Opt allows GLS to perform many more penalty cycles in the time given than its

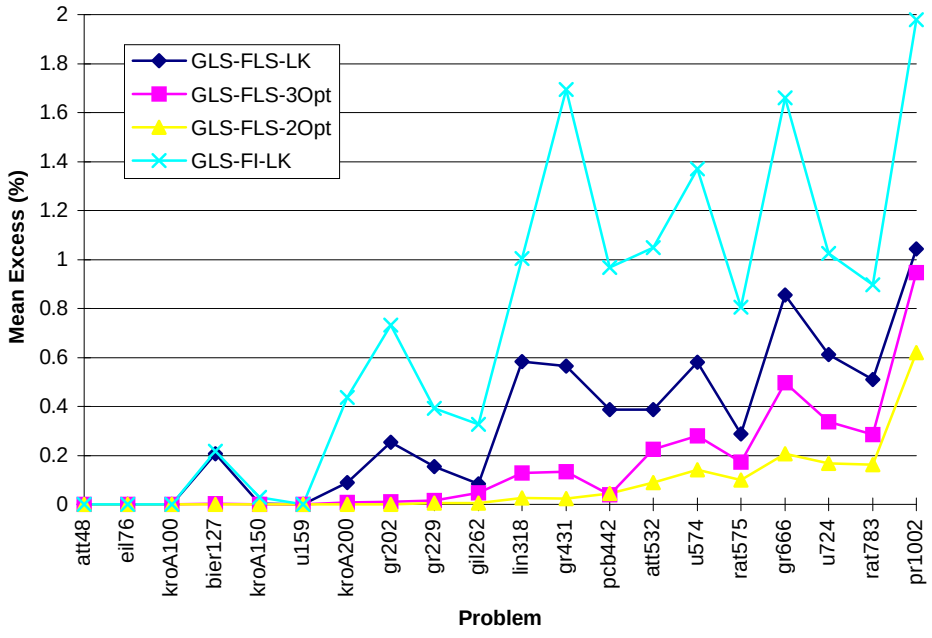


Figure 3.5 Performance of GLS variants using fast local search procedures

FLS-3Opt or FLS-LK counterparts. More GLS penalty cycles seem to increase efficiency at a level which outweighs the benefits from using a more sophisticated local search procedure such as 3-Opt or LK.

The remarkable effects of GLS on local search are further demonstrated in Figure 3.6 where GLS with FLS-2Opt is compared against Repeated FLS-2Opt and Repeated FI-LK. In Repeated FLS-2Opt and Repeated FI-LK, local search is simply restarted from a random solution after a local minimum and the best solution found over the many runs is returned. These two algorithms along with other versions of repeated local search were tested under the same settings with the GLS variants. Appendix A includes the full set of results for repeated local search. In Figure 3.6, we can see the huge improvement in the basic 2-Opt heuristic when this is combined with GLS. GLS is the only technique known to us which when applied to 2-Opt can outperform the Repeated LK algorithm (and that without requiring excessive amounts of CPU time) as illustrated in the same figure.

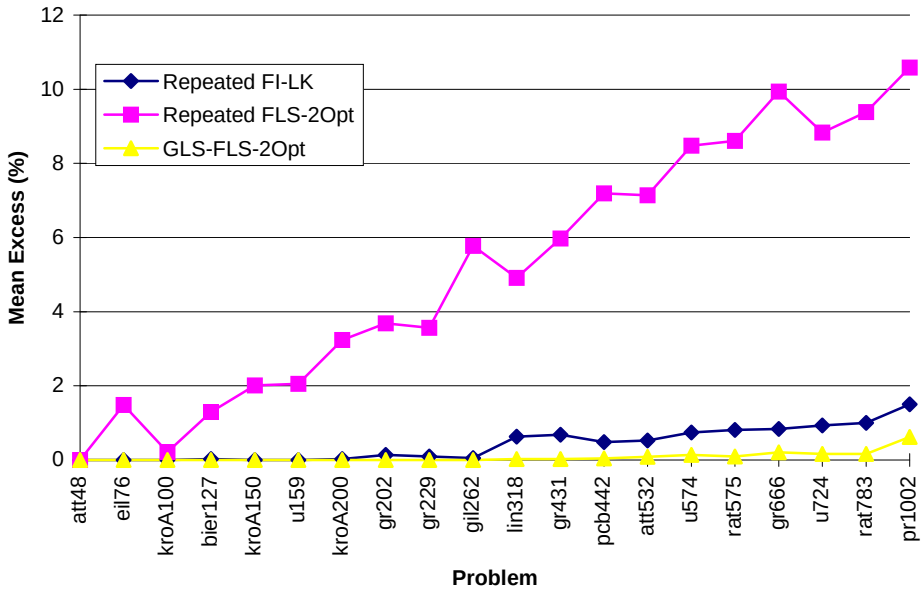


Figure 3.6 Improvements introduced by the application of GLS to the simple FLS-2Opt

3.8 Comparison with Specialised TSP algorithms

3.8.1 Iterated Lin-Kernighan

The *Iterated Lin-Kernighan* algorithm (not to be confused with Repeated LK) has been proposed by Johnson [Joh90] and it is considered to be one of the best if not the best heuristic algorithm for the TSP [JM95]. Iterated LK uses LK to obtain a first local minimum. To improve this local minimum, the algorithm examines other local minimum tours "near" the current local minimum. To generate these tours, Iterated LK first applies a random and unbiased non-sequential 4-Opt exchange (see Figure 3.1) to the current local minimum and then optimises this 4-Opt neighbour using the LK algorithm. If the tour obtained by the process (i.e. random 4-Opt followed by LK) is better than the current local minimum then Iterated LK makes this tour the current local minimum and continues from there using the same neighbour generation process. Otherwise, the current local minimum remains as it is and further random 4-Opt moves are tried. The algorithm stops when a stopping criterion based either on the number of iterations or computation time is satisfied. Figure 3.7 contains the original description of the algorithm as given in [Joh90].

1. Generate a random tour T.

2. Do the following for some prespecified number M of iterations:

    2.1. Perform an (unbiased) random 4-Opt move on T, obtaining T'.

    2.2. Run Lin-Kernighan on T', obtaining T''.

    2.3. If length(T'') ≤ length (T'), set $T = T''$.

3. Return T'.

Figure 3.7 Iterated Lin-Kernighan as described by Johnson in [Joh90]

The random 4-Opt exchange performed by Iterated LK is mentioned in the literature as the “double-bridge” move and plays a diversification role for the search process, trying to propel the algorithm to a different area of the search space preserving at the same time large parts of the structure of the current local minimum. Martin et al. [MOF92] describe this action as a “kick” and show that can be also used with 3-Opt in the place of LK. The same authors also suggest the combination of the method with Simulated Annealing (Long Markov Chains method). Martin and Otto [MO96] further demonstrate the efficiency of this last algorithm on the TSP and also the Graph Partitioning problem though they admit that simulated annealing does not significantly improve the method for TSP problems up to 783 cities. Finally, Johnson and McGeoch [JM95] review Iterated LK and its variants and provide results for both structured and random TSP instances.

Iterated LK or Iterated 3-Opt share some of the principles of GLS in the sense that they produce a sequence of diversified local minima though this is conducted in a random rather than a systematic way. Furthermore, iterated local search accepts the new solution, produced by the 4-Opt exchange and the subsequent LK or 3-Opt optimisation, only if it improves over the current local minimum (or it is slightly worse in the case of Large Markov Chains Method which uses simulated annealing) .

Iterated LK outperforms Repeated LK previously thought to be the “champion” of TSP heuristics and also long simulated annealing runs [MO96]. More recent experiments show that even sophisticated tabu search variants of LK cannot improve over Iterated LK [ZD95] which rightly deserves the title of the “champion” of TSP meta-heuristics.

To compare Iterated LK and its other variants such as Iterated 3-Opt with GLS, we extended our C++ library mentioned above to allow the iterated local search scheme

to be combined with the local search procedures of Table 3.1 included in the library. In particular, a random and unbiased Double-Bridge (DB) move was performed in a local minimum. The solution obtained was optimised by either one of the procedures of Table 3.1 before compared against the current local minimum. The new solution was accepted only if it improved over the current local minimum. To combine iterated local search with fast local search procedures, we activated the sub-neighbourhoods corresponding to the cities at the ends of the edges involved in the Double-Bridge move (see also [CMMR96]). The above extensions to the library made available a general meta-heuristic method applicable to all the local search procedures of Table 3.1. We will refer to this method as the Double-Bridge (DB) meta-heuristic.

We tested all the possible combinations of the DB meta-heuristic with the local searches of Table 3.1 (except for BI-2Opt) on the set of 20 problems used to test the GLS combinations. The same time limit (5 minutes of CPU time on DEC Alpha 3000/600 machines) was used and ten runs were performed on each instance in the set. The percentage excess was averaged in each problem for each DB variant. The best combination proved to be that of the DB heuristic with FLS-LK which outperformed DB with FI-LK (this last algorithm is roughly the same with the original method proposed by Johnson [Joh90]). The results for the various combinations of DB with local search are included in Appendix A.

Problem	Mean Excess (%) over 10 runs			
	GLS with FLS-2Opt	**DB with FLS-LK**	**DB with FI-LK**	**Repeated FI-LK**
att48	0	0	0	0
eil76	0	0	0	0
kroA100	0	0	0	0
bier127	0	0	0	0.0301
kroA150	0	0	0	0.00226
u159	0	0	0	0
kroA200	0	0	0	0.02452
gr202	0	0	0.00921	0.14143
gr229	0.00431	0.00475	0.01412	0.0977
gil262	0.00421	0	0.01682	0.05467
lin318	0.02641	0.24079	0.25578	0.62957
gr431	0.02392	0.22239	0.3327	0.67964
pcb442	0.04431	0.08173	0.06637	0.48525
att532	0.08994	0.08163	0.22502	0.53023
u574	0.14144	0.0924	0.11435	0.73838
rat575	0.09892	0.09745	0.13731	0.80762
gr666	0.20628	0.17587	0.41888	0.83762
u724	0.16822	0.16655	0.35696	0.93367
rat783	0.16125	0.15331	0.24075	1.00045
pr1002	0.62063	0.44633	1.04742	1.5046
Average Excess	**0.07949**	**0.08816**	**0.16178**	**0.42488**

Table 3.5 GLS with FLS-2Opt compared with variants of Iterated Lin-Kernighan.

Table 3.5 presents the results obtained for DB with FLS-LK and DB with FI-LK compared with those for GLS with FLS-2Opt found to be the best GLS variant. As a point of reference, we also provide results for FI-LK when repeated from random starting points and for the same amount of time. As we can see in Table 3.5, GLS with FLS-2Opt is better on average than both DB with FLS-LK and DB with FI-LK. The solution quality improvement over these methods although small it is very significant given that these methods are amongst the best heuristic techniques for the TSP. Note here that GLS with FLS-2Opt is by far a simpler method requiring only a fraction of the programming effort required to develop the DB variants based on LK.

To further test GLS against the DB variants of LK, we used a set of 66 TSPLIB problems from 48 to 2392 cities but this time we performed longer runs lasting 30 minutes of CPU time each. This amount of time on the DEC Alpha machines used translates to many hours of CPU time on an average PC where most of these

algorithms are most likely to be utilised. Because of the large number of instances used and the long time the algorithms were allowed to run, one run was performed on each instance. The results from the experiments are presented in Table 3.6.

Even in these longer runs, GLS with FLS-2Opt still finds better solutions than the DB variants of LK. This result is of great significance since it further supports our claim that the application of GLS on FLS-2Opt successfully converted the method to a powerful algorithm. As we can see in Table 3.6, the method is able to compete and even outperform highly specialised heuristic methods for the TSP.

The relative gains from the GLS and also DB meta-heuristic are further illustrated in Figure 3.8. In this figure, we give the absolute improvement in average solution quality (i.e. excess above the optimal solution) by the GLS and DB variants over the corresponding repeated local search variants in the set of 20 problems from TSPLIB.

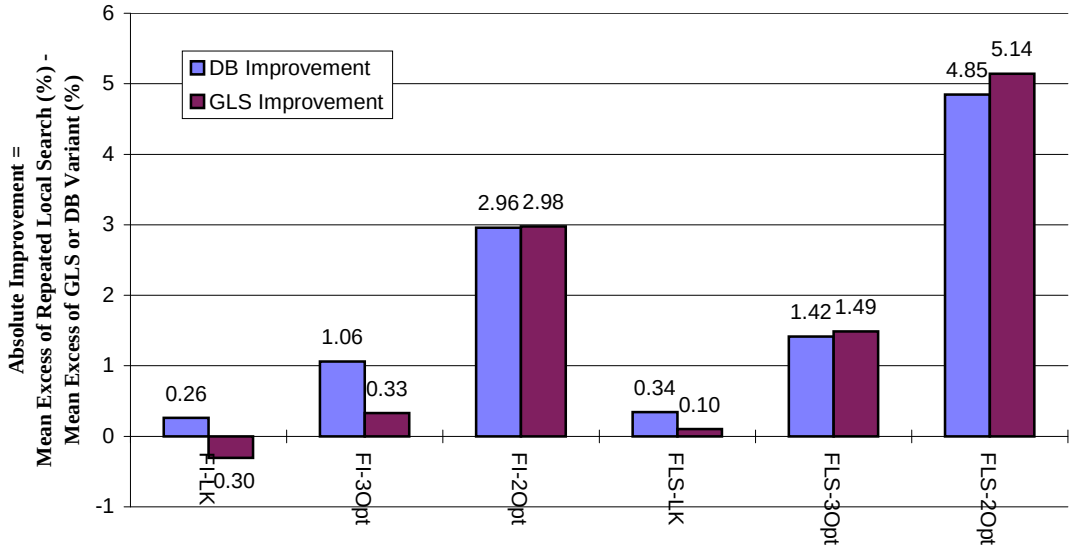


Figure 3.8 Improvements in solution quality by the GLS and DB meta-heuristics in a set of 20 TSPLIB problems