Voudouris, C. & Tsang, E.P.K., Guided Local Search and its application to the Travelling Salesman Problem, European Journal of Operational Research, Anbar Publishing, Vol.113, Issue 2, March 1999, 469-499

Title:

Guided Local Search and Its Application to the Traveling Salesman Problem.

Authors:

Christos Voudouris
Intelligent Systems Research Group,
Advanced Research & Technology Dept.,
BT Laboratories,
British Telecommunications plc.
United Kingdom.
e-mail: chrisv@info.bt.co.uk

Edward Tsang
Department of Computer Science,
University of Essex,
United Kingdom.
e-mail: edward@essex.ac.uk

To appear in **European Journal of Operational Research**
(accepted for publication, March 1998)

Correspondence Address:

Christos Voudouris
BT Laboratories
MLB 1/PP 12
Martlesham Heath
IPSWICH
Suffolk IP5 3RE
England

Telephone ++44 1473 605465
Fax ++44 1473 642459

Abstract

The Traveling Salesman Problem (TSP) is one of the most famous problems in combinatorial optimization. In this paper, we are going to examine how the techniques of Guided Local Search (GLS) and Fast Local Search (FLS) can be applied to the problem. Guided Local Search sits on top of local search heuristics and has as a main aim to guide these procedures in exploring efficiently and effectively the vast search spaces of combinatorial optimization problems. Guided Local Search can be combined with the neighborhood reduction scheme of Fast Local Search which significantly speeds up the operations of the algorithm.

The combination of GLS and FLS with TSP local search heuristics of different efficiency and effectiveness is studied in an effort to determine the dependence of GLS on the underlying local search heuristic used. Comparisons are made with some of the best TSP heuristic algorithms and general optimization techniques which demonstrate the advantages of GLS over alternative heuristic approaches suggested for the problem.

Keywords: Heuristics, Combinatorial Optimization, Traveling Salesman, Guided Local Search, Tabu Search.

1. Introduction

The *Traveling Salesman Problem* or TSP for short is one of the most famous combinatorial optimization problems. The problem is known to be *NP-hard* and over the years has been the testing ground for numerous techniques inspired from a variety of sources. Nowadays, TSP plays a very important role in the development, testing and demonstration of new optimization techniques. In this context, we are presenting the application to the TSP of a new metaheuristic approach called *Guided Local Search* (GLS) and its accompanying neighborhood reduction scheme called *Fast Local Search* (FLS).

Guided Local Search originally proposed by Voudouris and Tsang [47] is a general optimization technique suitable for a wide range of combinatorial optimization problems. Successful applications of the technique so far include practical problems such as Frequency Allocation [47], Workforce Scheduling [45] and Vehicle Routing [2, 25] and also classic problems such as the Traveling Salesman Problem (TSP), Quadratic Assignment Problem (QAP) and Global Optimization [48]. In this paper, we present the technique to the wider Operations Research audience by explaining its application to the TSP, a widely known problem in the OR community.

Guided Local Search (GLS) belongs to a class of techniques known as *Metaheuristics* [37, 38, 40]. Prominent members of this class include *Tabu Search* [12-18], *Simulated Annealing* [1, 9, 26, 28], *GRASP* [10], *Genetic Algorithms* [8, 19, 39], Scatter Search [13] and others. Metaheuristics aim at enhancing the performance of heuristic methods in solving large and difficult combinatorial optimization problems.

In the case of GLS, the main focus is on the exploitation of problem and search-related information to effectively guide local search heuristics in the vast search spaces of NP-hard optimization problems. This is achieved by augmenting the objective function of the problem to be minimized with a set of penalty terms which are dynamically manipulated during the search process to steer the heuristic to be guided. Higher goals, such as the distribution of the search effort to the areas of the search space according to the promise of these areas to contain high quality solutions, can be expressed and pursued.

GLS is closely related to the *Frequency-Based Memory* approaches introduced in Tabu Search [14, 18], extending these approaches to take into account the quality of structural parts of the solution and also react to feedback from the local optimization heuristic under guidance.

The paper is structured as follows. We first describe the basics of local search which is the foundation for most metaheuristics. Following that we explain the different components of GLS and how it can be combined with the sister scheme of Fast Local Search particularly suited for speeding up the search of neighborhoods when GLS is used. The rest of the paper is devoted to the application of GLS and FLS to the famous Traveling Salesman Problem when these are combined with commonly used heuristics such as *2-Opt*, *3-Opt* and *Lin-Kernighan*. The benefits from using GLS and FLS with these heuristics are demonstrated and the dependence of GLS on them is investigated. Conclusions are drawn on the relation between GLS and the underlying local search procedures. Finally comparisons are conducted with other well known general or TSP-specific metaheuristic techniques such as Simulated Annealing, Tabu Search, Iterated Lin-Kernighan and Genetic Algorithms. GLS is shown to perform equally well compared with state-of-the-art specialized methods

while outperforming classic variants of well known general optimization techniques. In all cases, publicly available TSP instances are used for which the optimal solutions are known so that the performance of algorithms can be measured with respect to approximating the optimal solutions.

2. Local Search

Local Search, also referred to as Neighborhood Search or Hill Climbing, is the basis of many heuristic methods for combinatorial optimization 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 the following definition of a combinatorial optimization problem.

A combinatorial optimization 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 minimizes the objective function g. The problem is stated as:

$$min\ g(s),\ s \in S.$$

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 neighborhood N for the problem instance (S, g) can be defined as a mapping from S to its powerset:

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

$N(s)$ 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 neighborhood N iff:

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

Local search is the procedure of minimizing 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:

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

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 neighborhood. 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 neighborhood and also the time needed to evaluate a move. In general, the larger the neighborhood, 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. Metaheuristics are trying to remedy that. 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 metaheuristics tend to be much more sophisticated than repeated local search pursuing a range objectives that go beyond simply escaping from local minima. Also, the way they utilize local search may vary and not limited to applying it to a single solution but to a population of solutions as it is the case in some Hybrid Genetic Algorithms.

3. Guided Local Search

Guided Local Search has its root in a Neural Network architecture named *GENET* developed by Wang and Tsang [49]. GENET is applicable to a class of problems known as *Constraint Satisfaction Problems* [46] which are closely related to the class of *SAT* problems. GLS generalizes some of the elements present in the GENET architecture and applies them to the general class of combinatorial optimization problems. For more information on GENET and related techniques for CSP and SAT problems the reader can refer to the following publications [7, 36, 43].

GLS augments the cost function of the problem to include a set of penalty terms and passes this, instead of the original one, for minimization by the local search procedure. 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 minimize the modified cost function.

3.1 Solution Features

GLS employs solution features to characterize 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:

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

The notion of solution features is very similar to the notion *solution attributes* used in Tabu Search. The only difference is that features, as considered in here, are always associated with a binary state given by their indicator function. They also have certain properties such as their *penalty* and *cost*. The indicator functions of features are directly incorporated in the problem's cost function to produce the *augmented cost function*. The augmented cost function replaces the objective function of the problem during the search process and it is dynamically manipulated by GLS to guide the local optimization algorithm used. In the next paragraph, we explain constrains on features and the augmented cost function.

3.2 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:

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

where M is the number of features defined over solutions, p_i is the penalty parameter corresponding to feature f_i and λ (lambda) a parameter for controlling the strength of constraints with respect to the actual solution cost. The penalty parameter p_i gives the degree up to which the solution feature f_i is constrained. The parameter λ represents the relative importance of penalties with respect to the solution cost and it provides a means to control the influence of the information on the search process. We are going to further explain the role of λ later in this paper when we refer to the application of the algorithm on the TSP. For an in depth analysis of the role of the parameter λ the reader is directed to [48].

GLS iteratively uses local search passing it the augmented cost function for minimization and it simply modifies the *penalty vector* $\boldsymbol{p}$ given by:

$$\boldsymbol{p} = (p_1, \ldots, p_M)$$

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 utilized to guide the search process 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* ***c***:

$$\boldsymbol{c} = (c_1, \ldots, c_M)$$

which contains positive or zero elements.

Before explaining the penalty modification scheme in detail, we would like to draw the reader's attention to the meaning of local minima in the context of GLS. The local minima encountered by local search when GLS is used are with respect to the augmented cost function and may be different from the local minima with respect to the original cost function of the problem. Hereafter and whenever we refer to a local minimum in the context of GLS, we mean the former and not the later. Before any penalties are applied, the two are identical but as the search progresses the local minima with respect to the original cost function may not be local minima with respect to the augmented cost function. This allows local search to escape from the local minima of the original cost function since GLS is altering their local minimum status under the augmented cost function using the penalty modification mechanism to be explained next.

3.3 Penalty Modifications

The penalty modification mechanism is responsible for manipulating the augmented cost function when local search is trapped in a local minimum. A particular local minimum solution s_* exhibits a number of features and the indicators of the features f_i exhibited take the value 1 (i.e. $I_i(s_*) = 1$). When local search is trapped in s_*, the penalty parameters are incremented by one for all features f_i that maximize the utility expression:

$$util(s_*, f_i) = I_i(s_*) \cdot \frac{c_i}{1 + p_i} \,. \tag{2}$$

In other words, incrementing the penalty parameter of the feature f_i is considered an *action* with utility given by (2). In a local minimum, the actions with *maximum* utility are selected and then performed. The penalty parameter p_i is incorporated in (2) to prevent the scheme from being totally biased towards penalizing features of high cost. The role of the penalty parameter in (2) is that of a counter which counts how many times a feature has been penalized. If a feature is penalized many times over a number of iterations then the term $\frac{c_i}{1 + p_i}$ in (2) decreases for the feature, diversifying choices and giving the chance for other features to also be penalized. The policy implemented is that features are penalized with a frequency proportional to their cost. Due to (2), features of high cost are penalized 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 penalized.

Depending on the value of λ (i.e. strength of penalties) in (1) one or more penalty modification iterations as described above may be required before a move is made out of the local minimum. High values for λ make the algorithm more

aggressive escaping quickly out of the local minima encountered while low values for λ make the algorithm more cautious requiring more penalty increases before an escape is achieved. Low values, although slow down the method in terms of escaping from local minima, lead to a more careful exploration of the search space putting less weight on the penalty part of the augmented cost function *h(s)* as given by (1).

Another issue to consider is the always increasing penalties for features and what is the impact of that. Actually, as soon as penalties reach the same value for all features in a vicinity of the search space, they tend to cancel out each other. For example, if all the features have their penalties set to 1 this has the same effect as all the features have their penalties set to 0. This is because moves look at the cost differences from exchanging certain features with others rather than the actual costs incurred. The basic GLS algorithm as described so far is depicted in Figure 1.

```
procedure GuidedLocalSeach(S, g, λ, [I_1, ...,I_M], [c_1,...,c_M], M)
begin
        k ← 0;
        s_0 ← random or heuristically generated solution in S;
        for i ←1 until M do /* set all penalties to 0 */
                p_i ← 0;
        while StoppingCriterion do
        begin
                h ← g + λ * ∑p_i*I_i ;
                s_k+1 ← LocalSearch(s_k, h);
                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
                        p_i ← p_i + 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: lambda 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 1. Guided Local Search in pseudocode.

Applying the GLS 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.

3.4 Fast Local Search and Other Improvements

There are both minor and major optimizations 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 (2) 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. 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:

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

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 neighborhood. 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 neighborhood search process by redefining it. The method is a generalization of the *approximate 2-opt* method proposed in [3] for the Traveling Salesman Problem. The method also relates to *Candidate List Strategies* used in tabu search [14].

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

Initially, all sub-neighborhoods are active. If a sub-neighborhood 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-neighborhoods are also activated. In particular, we activate all the sub-neighborhoods 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-neighborhoods being active until all the sub-neighborhood 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 optimization tools. Combining GLS with FLS is straightforward. The key idea is to associate solution features to sub-neighborhoods. The associations to be made are such that for each feature we know which sub-neighborhoods 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.

The procedure *GuidedFastLocalSearch* in Figure 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 penalized, the bits of the associated sub-neighborhoods 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-neighborhoods and in particular those which associate to the features just penalized. This dramatically speeds up GLS. Moreover, local search is focusing on removing the penalized 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-neighborhoods 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-neighborhoods relating to feature i penalized */
                        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-neighborhood for improving moves */
                        begin
                                Moves ← set of moves in sub-neighborhood 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 minimization */
                                        begin
                                                bit_i ← 1;
                                                SetBits ← SubNeighbourhoodsForMove(m);
                                                /* spread activation to other sub-neighborhoods */
                                                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: GLS parameter, I_i: indicator function for feature i, c_i: cost for feature i, M: number of features, L: number of sub-neighborhoods, p_i: penalty for feature i, bit_i: activation bit for sub-neighborhood i, SubNeighbourhoodsForFeature(i): procedure which returns the bits of the sub-neighborhoods corresponding to feature i, and SubNeighbourhoodsForMove(m): procedure which returns the bits of the sub-neighborhoods to spread activation to when move m is performed.

Figure 2. Guided Local Search combined with Fast Local Search in pseudocode.

Apart from the combination of GLS with fast local search, other useful variations of GLS include:

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

More information about these variations can be found in [48]. Also for a combination of GLS with Tabu Search the reader may refer to the work by Backer et. al [2].

4. Connections with Other General Optimisation Techniques

4.1 Simulated Annealing

Non-monotonic temperature reduction schemes used in Simulated Annealing (SA) also referred to as *re-annealing* or *re-heating* schemes are of interest in relation to the work presented in this paper. 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 [5]. 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 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 utilized 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.

4.2 Tabu Search

GLS is directly related to Tabu Search and to some extent can be considered a Tabu Search variant. Solution features are very similar to solution attributes used in Tabu Search. Both Tabu Search and GLS impose constrains on them to guide the underlying local search heuristics.

Tabu Search in its *Short-Term Memory* form of *Recency-Based Memory* is imposing hard constraints on solutions attributes of recently visited solutions or recently performed moves [14, 18]. This prevents local search from returning to recently visited solutions. Local search is not getting trapped in a local minimum given the duration of these constraints is long enough to lead to an area outside the local minimum basin. Variable duration of these constraints is sometimes advantageous allowing Tabu Search to adapt better to the varying radius of the

numerous local minimum basins that could be encountered during the search [44]. Nonetheless, there is always the risk of cycling if all the escaping routes require constraint duration longer than those prescribed in the beginning of the search.

The approach taken by GLS is not to impose hard constraints but instead to leave local search to settle in a local minimum (of the augmented cost function) before any of the guidance mechanisms are triggered. The purpose of doing that is to allow GLS to explore a number of alternative escape routes from the local minimum basin by first allowing local search to settle in that and consequently applying one or more penalty modification cycles which depending on the structure of the landscape may or may not result in a escaping move. Furthermore the continuous penalization procedure has the effect of progressively "filling up" the local minimum basin present in the original cost function. The risks of cycling are minimized since penalties are not retracted but are permanently marking substantially big areas of the search space that incorporate the specific features penalized. Local minima for the original cost function may not have a local minimum status under the augmented cost function after a number of penalty increases is performed. This allows local search to leave them and start exploring other areas of the search space.

Long-Term Memory strategies for diversification used in Tabu Search such as *Frequency-Based Memory* have many similarities to the GLS penalty modification scheme. Frequency-Based Memory based on solution attributes is increasing the penalties for attributes incorporated in a solution every time this is solution is visited [14, 18]. This leads to a diversification function which guides local search towards attributes not incorporated frequently in solutions.

GLS is also increasing the penalties for features though not in every iteration but only in a local minimum. Furthermore not all features have their penalties

increased but a selective penalization is implemented which bases its decisions on the quality of the features (i.e. cost), decisions made by the algorithm in previous iterations (i.e. penalties already applied) and also the current landscape of the problem which may force more than one penalization cycles before a move to a new solution is achieved. If GLS is used in conjunction with FLS, the different escaping directions from the local minimum can be quickly evaluated allowing the selective diversification of GLS to also direct local search through the moves evaluated and not only through the augmented cost function.

In general, GLS can alone perform similar functions to those achieved by the simultaneous use of both Recency-Based and Frequency-Based memory as this is the case in many Tabu Search variants. Other elements like *intensification* based on elite solution sets may well be incorporated in GLS as in Tabu Search.

Concluding, Tabu Search and GLS share a lot of common ground in both taking the approach of constraining solution attributes (features) to guide a local search procedure. Tabu Search mechanisms are usually triggered in every iteration and local search is not allowed to settle in a local minimum. GLS mechanism are triggered when local search settles in a local minimum and thereafter until it escapes. Usually, Tabu Search uses a Short-Term Memory and a Long-Term Memory component, GLS is not using separate components and it is trying to perform similar functions using a single penalty modification mechanism. There is a lot of promise in investigating hybrids that combine elements from both GLS and Tabu Search in a single scheme. For an example, the reader can refer to the work by Backer et. al on the Vehicle Routing Problem [2].

5. The Traveling Salesman Problem

In the previous sections, we examined the method of GLS and its generic framework. We are now going to examine the application of the method to the well-known Traveling Salesman Problem. There are many variations of the 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, $i = 1,\ldots,N$. The cost of a permutation is defined as:

$$g(\pi) = \sum_{i=1}^{N} d_{i\pi(i)} \tag{3}$$

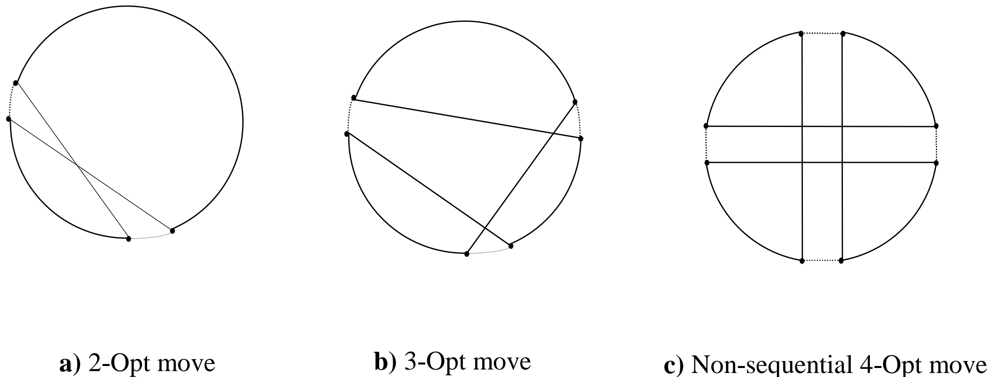
and gives the cost function of the TSP.

Recent and comprehensive surveys of TSP methods are those by Laporte [29], Reinelt [42] and Johnson & McGeoch [21]. The reader may also refer to [30] 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 specialized approximation algorithms [3]. Moreover, the optimal solutions have been found and proven for non-trivial problems of size up to 7397 cities [21]. Nowadays, TSP plays a very important role in the development and testing of new optimization techniques. In this context, we examine how guided local search and fast local search can be applied to this problem.

6. Local Search Heuristics for the TSP

Local search for the TSP is synonymous with *k*-Opt moves. Using *k*-Opt moves, neighboring 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* [6], *3-Opt* [31] and *Lin-Kernighan (LK)* [32]. These heuristics define neighborhood structures which can be searched by the different neighborhood search schemes described in sections 2 and 3.4, leading to many local optimization algorithms for the TSP. The neighborhood structures defined by 2-Opt, 3-Opt and LK are as follows [20]:

2-Opt. A neighboring solution is obtained from the current solution by deleting two edges, reversing one of the resulting paths and reconnecting the tour (see Figure 3). The worst case complexity for searching the neighborhood defined by 2-Opt is $O(n^2)$.



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

Figure 3. k-Opt moves for the TSP.

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). 3-Opt is much more effective than 2-Opt, though the size of the neighborhood (possible 3-Opt

moves) is larger and hence more time-consuming to search. The worst case complexity for searching the neighborhood defined by 3-Opt is $O(n^3)$.

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 neighbors can be remotely apart because "non-sequential" exchanges such as that shown in Figure 3 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 [32]. 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 neighborhood 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 4.

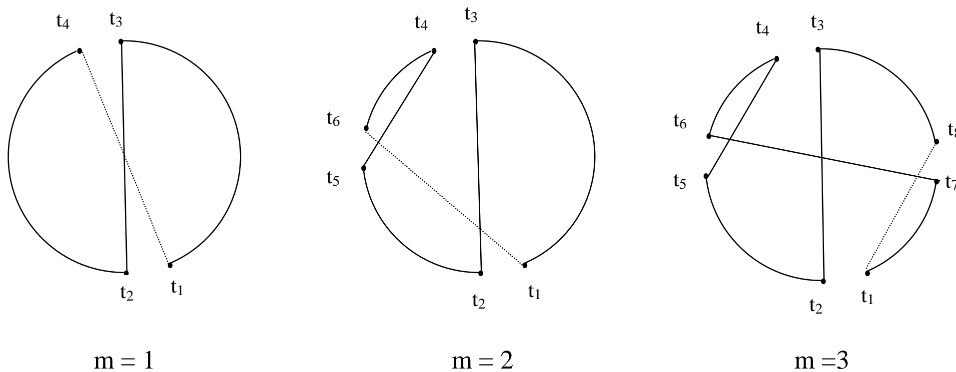


Figure 4. The first three steps of the Lin-Kernighan edge exchange mechanism.

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 neighborhood 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 5 illustrates the infeasibility-move mechanism.

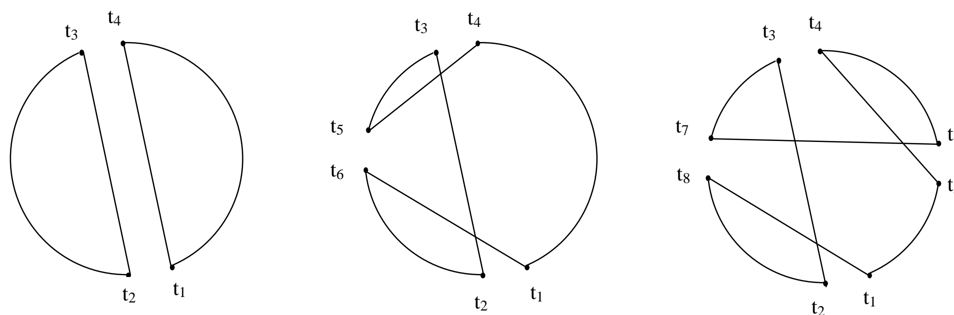


Figure 5. Lin-Kerhighan's infeasibility moves.

The interested reader may refer to the original paper by Lin and Kernighan [32] for a more elaborate description of this mechanism. LK is the standard benchmark against which all heuristic methods are tested. The worst case complexity for searching the LK neighborhood 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 [3]. In addition to that, Reinelt [42] and also Johnson and McGeoch [21] describe some improvements that

are commonly incorporated in local search algorithms for the TSP. We will refer to some of them later in this paper. The best reference for the LK algorithm is the original paper by Lin and Kernighan [32]. In addition to that, Johnson and McGeoch [21] 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 [33].

Fast local search and guided local search can be combined with the neighborhood 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.

6.1 Fast Local Search Applied to the TSP

A fast local search procedure for the TSP using 2-Opt has already been suggested by Bentley [3]. 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. [4] to reduce the running times of these heuristics in very large TSP instances. More recently, Johnson et al. [24] also use the technique to speed up their LK variant (see [21]). 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 3.4.

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**[1] 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 neighborhood in sub-neighborhoods as required by fast local search. Each city in the problem may be seen as defining a sub-neighborhood which contains all edge exchanges originating from either one of the edges adjacent to the city. For a problem with N cities, the neighborhood is partitioned into N sub-neighborhoods, one for each city in the instance. Given the sub-neighborhoods, fast local search for the TSP works in the following way (see also 3.4).

Initially all sub-neighborhoods are active. The scanning of the sub-neighborhoods, defined by the cities, is done in an arbitrary static order (e.g. from 1st to Nth city). Each time an active sub-neighborhood 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-neighborhood 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-neighborhoods) 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-neighborhood where the move was found to remain active and also a number of other sub-neighborhoods to be activated. The process always continues with the next sub-neighborhood 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.

[1] In our work, if approximations are used such as nearest neighbor 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.

The tour is declared 2-Optimal, 3-Optimal or LK-Optimal, depending on the type of the *k*-Opt moves used.

6.2 Local Search Procedures for the TSP

Apart from fast local search, first improvement and best improvement local search (see section 2) 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 neighborhood.

Fast local search for the TSP described above can be easily converted to first improvement local search by searching all sub-neighborhoods 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 [32] 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-neighborhoods 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 neighborhood 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). These variants were derived by combining the different search schemes at the neighborhood level (i.e. fast, first improvement, and best improvement local search) with any of the 2-Opt, 3-Opt, or LK neighborhood structures. Table 1 illustrates the variants and also the names we will use to distinguish them in the rest of the paper.

Name	Local Search Type	Neighborhood 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 1. Local search procedures implemented for the study of GLS on the TSP.

7. Guided Local Search Applied to the TSP

7.1 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 (3)). 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 3.2, penalties have to be combined with the problem's cost function to form the augmented cost function which is minimized by local search. This can be done by considering the auxiliary distance matrix:

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

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 penalized in a local minimum are selected according to the utility function (2), which for the TSP takes the form:

$$Util\left(tour, e_{ij}\right) = I_{e_{ij}}(tour) \cdot \frac{d_{ij}}{1 + p_{ij}}, \tag{4}$$

where

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

7.2 Combining GLS with TSP Local Search Procedures

GLS as depicted in Figure 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 6.2 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 neighborhood 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 3.4). In particular, each time we penalize an edge in GLS, the

sub-neighborhoods 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-neighborhoods and in particular those which associate to the edges just penalized. Activation may spread to a limited number of other sub-neighborhoods 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 penalized edges instead of evaluating all possible moves.

7.3 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 penalizes one or more of the edges appearing in the local minimum, using the utility function (4) 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-neighborhoods (i.e. cities) at the ends of the edges penalized 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 penalizing 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 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 penalized 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 penalized. 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 realizes a "selective" diversification which pursues many more choices for long edges than short edges by penalizing 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.

8. Evaluation of GLS in the TSP

To investigate the behavior 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 6.2 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).

8.1 Experimental Setting

In the experiments conducted, we used problems from the publicly available library of TSP problems, TSPLIB [41]. 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:

$$excess = \frac{\text{solution cost - best known solution cost}}{\text{best known solution cost}} \times 100. \qquad (5)$$

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

8.2 Parameter λ

The only parameter of GLS which requires tuning is the 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:

$$\lambda = a \cdot \frac{g(\text{local minimum})}{N} , \qquad (6)$$

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. (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	$1/8 \le a \le ½$
3-Opt	$1/10 \le a \le ¼$
LK	$1/12 \le a \le 1/6$

Table 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).

8.3 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 neighborhood) and for GLS with FLS-2Opt, a call to fast local search as in Figure 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 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.

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 [41] and also [42]) 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.

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

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.

8.4 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 [20, 21, 24] 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 [1, 9, 22, 23, 26, 28] and a tabu search variant for the TSP suggested by Knox [27].

8.4.1 Simulated Annealing

The Simulated Annealing (SA) algorithm implemented for the TSP was the one described by Johnson in [20] and uses geometric cooling schedules. 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 [22].

8.4.2 Tabu Search

The tabu search variant implemented was the one proposed by Knox [27] 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 neighborhood but only amongst those not characterized 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.

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 3*N* (where *N* is the number of cities in the problem) as suggested by Knox [27]. 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.

8.4.3 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 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.

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

As we can see in Table 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 behavior 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.

8.5 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 6.2 and allows combinations of GLS with any one of these procedures. Furthermore, a number of approximations (not used in the GLS of section 8.3) are adopted which further reduce the computation times of local search and GLS as reported in section 8.3. 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 neighbors of each city in the instance. 2-Opt, 3-Opt and LK were considering in exchanges only edges to these 20 nearest neighbors (see also [21, 42]). Each time the penalty was increased for an edge, the nearest neighbor lists of the cities at the ends of the edge were reordered though no new neighbors 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 [3] for details). The LK implementation was exactly as proposed by Lin and Kernighan [32] 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 (5). The normalized lambda parameter a was provided as input to the program and λ was determined after the first local minimum using (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 the Appendix. Next, we focus on selected results from this set.

8.5.1 Results for GLS with First Improvement Local Search

Figure 6 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 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.

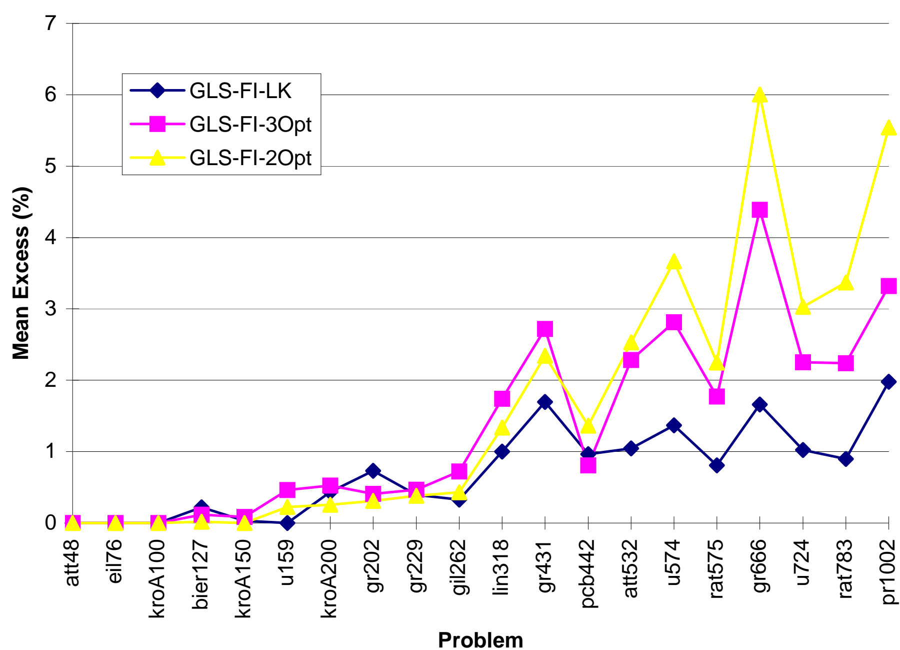


Figure 6. Performance of GLS variants using first improvement local search procedures.

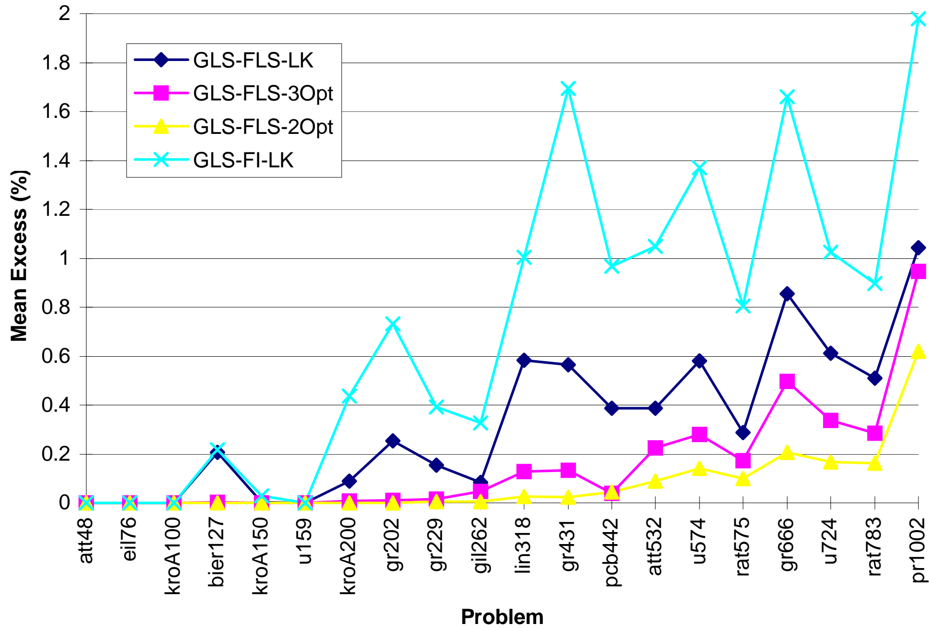


Figure 7. Performance of GLS variants using fast local search procedures.

8.5.2 Results for GLS with Fast Local Search

Figure 7 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 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 8 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. The Appendix includes the full set of results for repeated local search. In Figure 8, 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.

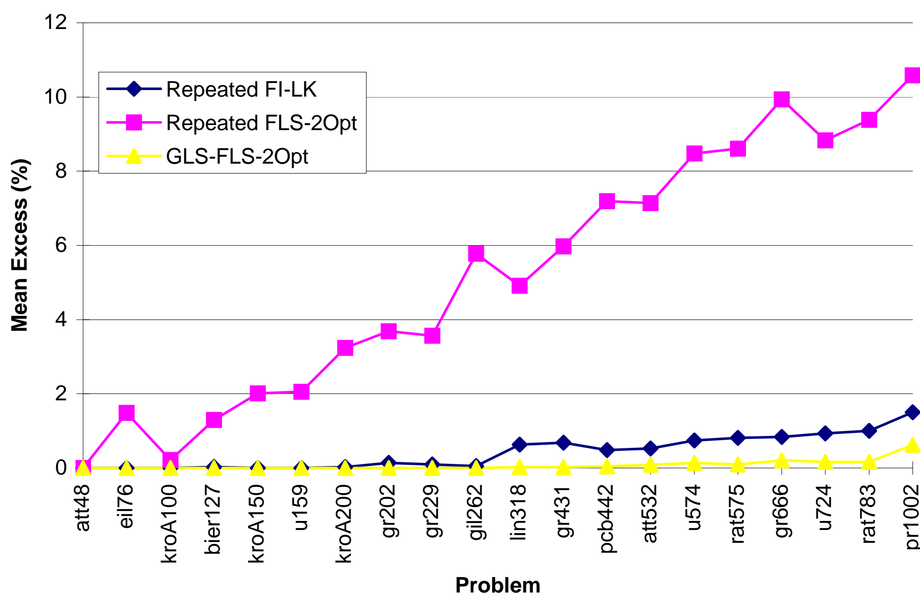


Figure 8. Improvements introduced by the application of GLS to the simple FLS-2Opt.

8.6 Comparison with Specialised TSP algorithms

8.6.1 Iterated Lin-Kernighan

The *Iterated Lin-Kernighan* algorithm (not to be confused with Repeated LK) has been proposed by Johnson [20] and it is considered to be one of the best if not the best heuristic algorithm for the TSP [21]. 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) to the current local minimum and then optimizes this 4-Opt neighbor 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 neighbor 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 9 contains the original description of the algorithm as given in [20].

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'') $\leq$ length (T'), set $T = T''$.

3. Return T'.

Figure 9. Iterated Lin-Kernighan as described by Johnson in [20].

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. [35] 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 [34] 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 [21] 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 optimization, 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 [34]. More recent experiments show that even sophisticated tabu search variants of LK cannot improve over Iterated LK [50] 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 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 optimized by either one of the procedures of Table 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-neighborhoods corresponding to the cities at the ends of the edges involved in the Double-Bridge move (see also [4]). The above extensions to the library made available a general meta-heuristic method applicable to all the local search procedures of Table 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 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 similar to the original method proposed by Johnson [20]). The results for the various combinations of DB with local search are included in the Appendix.

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

Table 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 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. 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 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 6, the method is able to compete and even outperform highly specialized heuristic methods for the TSP.

Problem	**GLS with FLS-2Opt**	**DB with FLS-LK**	**DB with FI-LK**	**Problem**	**GLS with FLS-2Opt**	**DB with FLS-LK**	**DB with FI-LK**
	Excess (%) in one run per instance				**Excess (%) in one run per instance**		
att48	0	0	0	pr264	0	0	0
eil51	0	0	0	pr299	0	0	0
st70	0	0	0	lin318	0	0.27124	0
eil76	0	0	0	fl417	0.00843	0.00843	0.42998
pr76	0	0	0	gr431	0	0	0.01458
gr96	0	0	0	pr439	0.00653	0.04104	0
rat99	0	0	0	pcb442	0.01182	0	0
kroA100	0	0	0	d493	0.02	0.00857	0.09142
kroB100	0	0	0	att532	0.06501	0	0.04696
kroC100	0	0	0	ali535	0.02323	0.01433	0.01433
kroD100	0	0	0	u574	0	0.08129	0.10568
kroE100	0	0	0	rat575	0.04429	0.08859	0.05906
rd100	0	0	0	p654	2.04659	2.27174	0.04619
eil101	0	0	0	d657	0.0184	0.0368	0.13289
lin105	0	0	0	gr666	0.00612	0.09988	0.20315
pr107	0	0	0	u724	0.05727	0.09783	0.04534
pr124	0	0	0	rat783	0	0.06814	0.01136
bier127	0	0	0	dsj1000	0.31222	0.40289	0.88742
pr136	0	0	0	pr1002	0.12315	0.07566	0.11658
gr137	0	0	0	u1060	0.05132	0.15663	0.43285
pr144	0	0	0	pcb1173	0.14765	0.02461	0.43767
kroA150	0	0	0	d1291	0.22244	0.63581	1.16139
kroB150	0	0	0	rl1304	0.20241	0	0.50366
pr152	0.18458	0	0	rl1323	0.18542	0.14027	0.22909
u159	0	0	0	fl1400	1.56009	2.58359	3.11025
rat195	0	0	0	u1432	0.05295	0.27783	0.30464
d198	0	0	0	d1655	0.40722	0.27846	1.19753
kroA200	0	0	0	vm1748	0.33219	0.32387	0.75678
kroB200	0	0	0	u1817	0.57517	0.3916	1.02096
gr202	0	0	0	rl1889	0.37279	0.90953	0.52443
pr226	0	0	0	u2152	0.61476	0.46379	0.75327
gr229	0	0	0	u2319	0.00726	0.25229	0.28729
gil262	0	0	0	pr2392	0.35209	0.27458	0.90019
				Mean	**0.12138**	**0.15575**	**0.20947**
				Standard Deviation	0.33047	0.43627	0.47296

Table 6. GLS with FLS-2Opt compared with variants of Iterated Lin-Kernighan (long runs).

The relative gains from the GLS and also DB meta-heuristic are further illustrated in Figure 10. 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.

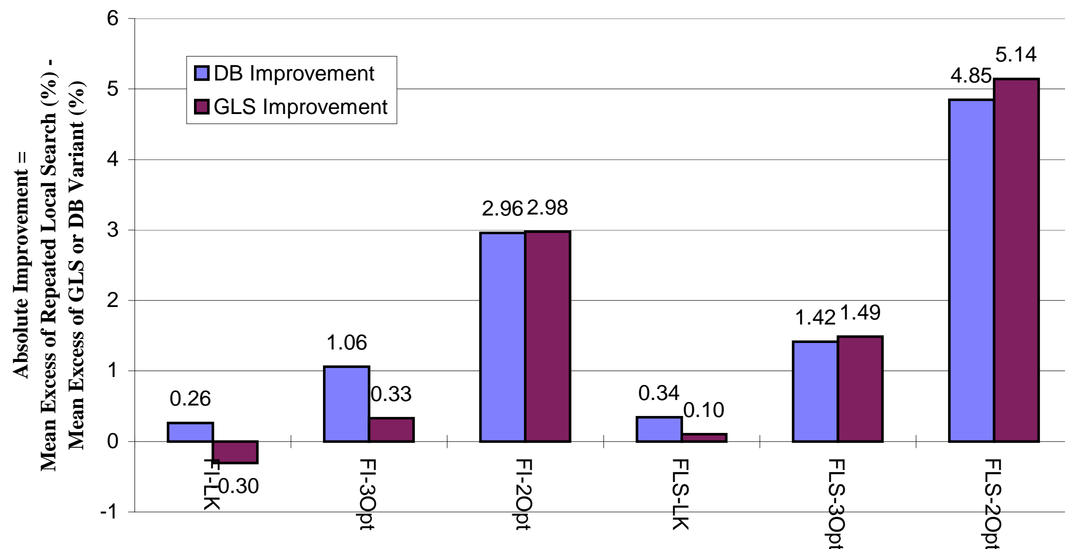


Figure 10. Improvements in solution quality by the GLS and DB meta-heuristics in a set of 20 TSPLIB problems.

As shown in Figure 10, the DB meta-heuristic is more effective than GLS when combined with LK. In fact, GLS when combined with FI-LK is even worse than Repeated FI-LK. This situation dramatically changes for fast local search variants where GLS is better than DB when combined with the FLS-3Opt or FLS-2Opt local searches improving the solution quality over repeated local search up to 5.14% in the case of FLS-2Opt. The overall ranking of all the variants developed in terms of average excess in the set of 20 TSPLIB problems is given Figure 11. GLS with FLS-2Opt was found to be best amongst the 18 algorithms tested.

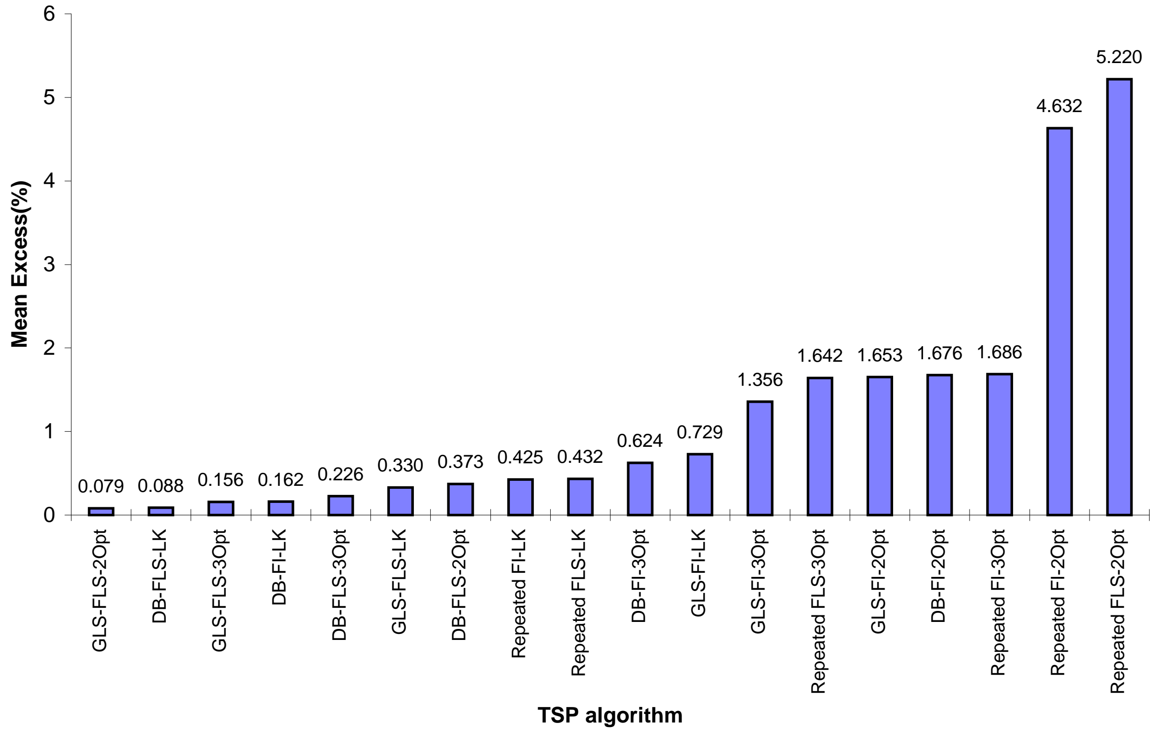


Figure 11. Overall ranking of the algorithms in terms of solution quality when tested on a set of 20 TSPLIB problems.

8.6.2 Genetic Local Search

In an effort to further improve the LK heuristic, Genetic Algorithms recently appeared which internally use LK for improving offspring solutions generated by crossover operations. An example of such a technique is the Genetic Local Search algorithm proposed by Freisleben and Merz [11]. This method, in addition to using LK for improving offspring solutions, uses a mutation operator which performs first an 4-Opt exchange on a population solution and then runs LK to convert this solution to a local minimum. Iterated LK mentioned above can be seen as a special case of this method. In [11], results are reported for Genetic Local Search on TSPLIB instances. The authors consider the results produced by the technique as superior to those published

for any GA approaches known to them and comparable to top quality non-GA heuristic techniques. Fortunately, the experiments in [11] were also conducted on a DEC Alpha workstation running at 175 MHz. This permits a meaningful comparison between this GA variant and GLS. We ran GLS-FLS-2Opt on the same instances with a = 1/6 and for an equal number of times as the GA approach. In Table 7, the results from [11] are compared with those we obtained for GLS using FLS-2Opt.

Problem	**GLS with FLS-2Opt**		**Genetic Local Search**	
	Mean Excess	**Mean CPU time (sec)**	**Mean Excess**	**Mean CPU time (sec)**
eil51 (20 runs)	0%	1.2	0%	6
kroA100 (20 runs)	0%	1.59	0%	11
d198(20 runs)	0%	435	0%	253
att532 (10 runs)	0%	3526	0.05%	6076
rat783 (10 runs)	0%	5232	0.04%	14925

Table 7. GLS with FLS-2Opt compared with Genetic Local Search on five TSPLIB instances.

Except for d198 which is a hard instance for GLS (see results in section 8.3), GLS was better than the GA approach finding solutions of better quality for att532 and rat783 while running faster between 1.7 to 6.9 times. Note here that the GA is using the best heuristic for the TSP (i.e. DB followed by LK) while GLS the worst (i.e. 2-Opt). Another remarkable result which emerged from these experiments was that GLS with FLS-2Opt can consistently find the optimal solutions for problems att532 and rat783. As far as we know, optimal solutions to such large problems can be consistently found only by heuristic methods that are using LK (e.g. Iterated LK or its variant Large-Step Markov Chains method).

In fact, GLS was able to find the optimal solution in even larger problems. For example, GLS with FLS-3Opt found the optimal solution for a 2319-city problem from TSPLIB (u2319) in less than 20 minutes while GLS with FLS-2Opt found the optimal solution to a 1002-city problem from TSPLIB (pr1002) in 14 hours of CPU

time despite running on Sparcstation 5 workstation which is much slower than the DEC Alpha machines used in the rest of the experiments.

9. Summary and Conclusions

In this paper, we described the technique of GLS in detail and examined its application to the Traveling Salesman Problem. Eight combinations of GLS with commonly used TSP heuristics were described and evaluated on publicly available instances of the TSP. GLS with FLS-2Opt was found to be the best GLS variant for the TSP. The variant was compared and found to be superior to commonly used variants of general search methods such as simulated annealing and tabu search. Furthermore, we demonstrated that GLS with FLS-2Opt is highly competitive (if not better) than some of the best specialized algorithms for the TSP such as Iterated Lin-Kernighan and Genetic Local Search. In total sixteen alternative TSP algorithms were compared against the GLS variants and many of the GLS variants were found to outperform or perform equally well to all these techniques. In total 24 algorithms for the TSP considered and extensive results were presented on publicly available instances of the problem.

Experimental results should be treated with care. Experimentation no matter how elaborate and extensive it may be, it can only give indications of which algorithms are better than others and that because of the many parameters involved in the algorithms, differences in implementation, and the limited number of instances used in experiments.

We can safely conclude that the evidence provided in this paper is enough to place GLS amongst what somebody will characterize as efficient and effective methods for the TSP. Given the simplicity of the algorithm and the ease of tuning (i.e.

single parameter), GLS with FLS-2Opt could be considered as an ideal practical method for the TSP especially when no programming effort can be devoted in implementing one of the complex specialized TSP algorithms.

More generally, GLS is applicable not only to TSP but to a range of other problems in combinatorial optimization. Open research issues include the use of incentives implemented as negative penalties which encourage the use of specific solution features is one promising direction to be explored. Other potentially interesting research directions include automated tuning of the parameter lambda, definition of effective termination criteria, and different utility functions for selecting the features penalized. GLS could also be used to distribute the search effort in other techniques such as Genetic Algorithms.

Finally, from our experience on the TSP and other domains we found it very easy to adapt GLS and FLS to problem in hand something which suggests that it may be possible to built a generic software platform for combinatorial optimization based on GLS. Although local search is problem dependent, most of the other structures of GLS and also FLS are problem independent. Furthermore, a step by step procedure is usually followed when GLS is applied to a new problem (i.e. identify features, assign costs, etc.) something which makes easier the use of the technique by OR practitioners and Optimization Systems engineers.

References

[1] Aarts, E.H.L., and Korst, J.H.M., *Simulated Annealing and Boltzmann machines*, Wiley, Chichester, 1989.

[2] Backer, B.D., Furnon, V., Prosser, P., Kilby, P., and Shaw, P., "Solving vehicle routing problems using constraint programming and metaheuristics", Submitted to the *Journal of Heuristics* special issue on Constraint Programming, July 1997.

[3] Bentley, J.L., "Fast Algorithms for Geometric Traveling Salesman Problems", *ORSA Journal on Computing*, Vol. 4, 387-411, 1992.

[4] Codenotti, B., Manzini, G., Margara, L., and Resta, G., "Pertrubation: An Efficient Technique for the Solution of Very Large Instances of the Euclidean TSP", *ORSA Journal on Computing*, Vol. 8, No. 2, 125-133, 1996.

[5] Connoly, D.T., "An improved annealing scheme for the QAP", *European Journal of Operational Research*, 46, 93-100, 1990.

[6] Croes, A., "A Method for Solving Traveling-Salesman Problems", *Operations Research*, 5, 791-812, 1958.

[7] Davenport, A., Tsang, E., Wang, C.J., and Zhu, K., "GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement", *Proceedings of AAAI-94*, 325-330, 1994.

[8] Davis, L., *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991.

[9] Dowsland, A., "Simulated Annealing", in: Reeves, C. R. (ed.), *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific Publishing, 20-69, 1993.

[10] Feo, T.A., and Resende, M.G.C., "Greedy Randomized Adaptive Search Procedures", *Journal of Global Optimization*, vol. 6, pp. 109-133, 1995.

[11] Freisleben, B., and Merz, P., "A Genetic Local Search Algorithm for Solving the Symmetric and Asymmetric TSP", *Proceedings of IEEE International Conference on Evolutionary Computation*, Nagoya, Japan, 616-621, 1996.

[12] Glover, F., "Future paths for integer programming and links to artificial intelligence", *Computers Ops Res.*, 5, 533-549, 1986.

[13] Glover, F., "Tabu Search and Adaptive Memory Programming - Advances, Applications and Challenges", in: *Interfaces in Computer Science and Operations Research*, Barr, Helgason and Kennington eds., Kluwer Academic Publishers, 1996.

[14] Glover, F., "Tabu Search Fundamentals and Uses", Graduate School of Business, University of Colorado, Boulder, 1995.

[15] Glover, F., "Tabu search Part I", *ORSA Journal on Computing*, Vol. 1, 190-206, 1989

[16] Glover, F., "Tabu search Part II", *ORSA Journal on Computing*, Vol. 2, 4-32, 1990.

[17] Glover, F., "Tabu Search: improved solution alternatives for real world problems", in: Birge & Murty (ed.), *Mathematical Programming: State of the Art*, 64-92, 1994.

[18] Glover, F., and Laguna, M., "Tabu Search", in: Reeves, C. R. (ed.), *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific Publishing, 71-141, 1993.

[19] Goldberg, D.E., *Genetic algorithms in search, optimization, and machine learning*, Addison-Wesley, 1989.

[20] Johnson, D., "Local Optimization and the Traveling Salesman Problem", *Proceedings of the 17th Colloquium on Automata Languages and Programming*, Lecture Notes in Computer Science, 443, 446-461, Springer-Verlag, 1990.

[21] Johnson, D., and McGeoch, L., "The Traveling Salesman Problem: A Case Study in Local Optimization", Manuscript, November, 1995 (revised version appeared in *Local Search in Optimization*, Ed. E. H. L. Aarts and J. K. Lenstra, Wiley, Chichester, pp. 215-310, 1997).

[22] Johnson, D., Aragon, C., McGeoch, L., and Schevon, C., "Optimization by simulated annealing: an experimental evaluation, part I, graph partitioning", *Operations Research*, 37, 865-892, 1989.

[23] Johnson, D., Aragon, C., McGeoch, L., and Schevon, C., "Optimization by simulated annealing: an experimental evaluation, part II, graph coloring and number partitioning", *Operations Research*, 39, 378-406, 1991.

[24] Johnson, D., Bentley, J., McGeoch, L., and Rothberg, E., "Near-optimal solutions to very large traveling salesman problems", in preparation. (for experimental results from this work see also [21])

[25] Kilby, P., Prosser, P., and Shaw, P., "Guided local search for the vehicle routing problem", *Proceedings of the 2nd International Conference on Metaheuristics*, July 1997.

[26] Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P., "Optimization by Simulated Annealing", *Science*, Vol. 220, 671-680, 1983.

[27] Knox, J., "Tabu Search Performance on the Symmetric Traveling Salesman Problem", *Computers Ops Res.*, Vol. 21, No. 8, pp. 867-876, 1994.

[28] Laarhoven, P.J.M.V., and Aarts, E.H.L., *Simulated Annealing: Theory and Applications*, Kluwer, Dordrecht, 1988.

[29] Laporte, G., "The Traveling Salesman Problem: An overview of exact and approximate algorithms", *European Journal of Operational Research*, 59, 231-247, 1992.

[30] Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., and Shmoys, D.B., (eds.), *The Traveling Salesman Problem: A guided tour in combinatorial optimization*, John Wiley & Sons, 1985.

[31] Lin, S., "Computer Solutions of the Traveling-Salesman Problem", *Bell Systems Technical Journal,* 44, 2245-2269, 1965.

[32] Lin, S., and Kernighan, B.W., "An effective heuristic algorithm for the traveling salesman problem", *Operations Research*, 21, 498-516, 1973.

[33] Mak, K., and Morton, A.J., "A modified Lin-Kernighan traveling-salesman heuristic", *Operations Research Letters*, 13, 127-132, 1993.

[34] Martin, O., and Otto, S.W., "Combining Simulated Annealing with Local Search Heuristics", in: G. Laporte and I. H. Osman (eds.), *Metaheuristics in Combinatorial Optimization*, *Annals of Operations Research*, Vol. 63, 1996.

[35] Martin, O., Otto, S.W., and Felten, E.W., "Large-step Markov chains for the TSP incorporating local search heuristics", *Operations Research Letters*, 11, 219-224, 1992.

[36] Morris, P., "The breakout method for escaping from local minima", *Proceedings of AAAI-93*, 40-45, 1993.

[37] Osman, I.H., "An Introduction to Meta-Heuristics", M. Lawrence and C. Wilson (eds.), in: *Operational Research Tutorial Papers*, Operational Research Society Press, Birmingham, UK, 92-122, 1995.

[38] Reeves, C.R. (ed.), *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific Publishing, 1993.

[39] Reeves, C.R., "Genetic Algorithms", in: Reeves, C. (ed.), *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific Publishing, 151-196, 1993.

[40] Reeves, C.R., "Modern Heuristic Techniques", in: V. J. Rayward-Smith, I. H. Osman, C. R. Reeves and G. D. Smith (ed.), John Wiley & Sons, *Modern Heuristic Search Methods*, 1-25, 1996.

[41] Reinelt, G., "A Traveling Salesman Problem Library", *ORSA Journal on Computing*, 3, 376-384, 1991.

[42] Reinelt, G., *The Traveling Salesman: Computational Solutions for TSP Applications*, Lecture Notes in Computer Science, 840, Springer-Verlag, 1994.

[43] Selman, B., and Kautz, H., "Domain independent versions of GSAT solving large structured satisfiability problems", *Proceedings of IJCAI-93*, 290-295, 1993.

[44] Taillard, E., Robust taboo search for the QAP. *Parallel Computing*, 17, 443-455, 1991.

[45] Tsang, E., and Voudouris, C., "Fast local search and guided local search and their application to British Telecom's workforce scheduling problem", *Operations Research Letters*, Vol. 20, No. 3, 119-127, 1997.

[46] Tsang, E., *Foundations of Constraint Satisfaction*, Academic Press, 1993.

[47] Voudouris, C., and Tsang, E., "Partial Constraint Satisfaction Problems and Guided Local Search", *Proceedings of 2nd Int. Conf. on Practical Application of Constraint Technology* (PACT'96), London, April, 337-356, 1996.

[48] Voudouris, C., *Guided Local Search for Combinatorial Optimization Problems*, PhD Thesis, Department of Computer Science, University of Essex, Colchester, UK, July, 1997.

[49] Wang, C.J., and Tsang, E., "Solving constraint satisfaction problems using neural-networks", *Proceedings of IEE Second International Conference on Artificial Neural Networks*, 295-299, 1991.

[50] Zachariasen, M., and Dam, M., "Tabu Search on the Geometric Traveling Salesman Problem", in: I. H. Osman and J. P. Kelly (ed.), *Meta-heuristics: Theory and Applications*, Kluwer, Boston, 571-587, 1996.

Appendix

The set of problems used in the evaluation of the Repeated Local Search, Guided Local Search and Iterated Local Search (using the Double Bridge move) variants on the TSP included 20 problems from 48 to 1002 cities all from TSPLIB. For each variant tested, 10 runs were performed from random solutions and 5 minutes of CPU time were allocated to each algorithm in each run on a DEC Alpha 3000/600 (175MHz) machine. To measure the success of the variants, we considered the percentage excess above the optimal solution as in (5). For GLS variants, the normalized lambda parameter a was provided as input and λ was determined after the first local minimum using (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 $a = 1/10$. Results for GLS are shown in Table A.1.

Iterated Local Search was using the Double Bridge move. No simulated annealing was used which is roughly equivalent to the Large-Markov Chains Methods with temperature T set to 0. Results for Iterated Local Search are shown in Table A.2.

Finally, Repeated Local Search was restarting from a random solution whenever local search was reaching a local minimum. Results for Repeated Local Search are shown in Table A.3. The names of the variants were formed according to the following convention:

<meta-heuristic>-<local search type>-<neighbourhood type>.

Problem	No.Cities	Mean Excess (%) over 10 runs					
		GLS-FI-LK	GLS-FI-3Opt	GLS-FI-2Opt	GLS-FLS-LK	GLS-FLS-3Opt	GLS-FLS-2Opt
att48	48	0	0	0	0	0	0
eil76	76	0	0	0	0	0	0
kroA100	100	0	0	0	0	0	0
bier127	127	0.218207	0.116586	0.019699	0.206625	0.002198	0
kroA150	150	0.029784	0.084075	0.000754	0.001508	0.001131	0
u159	159	0	0.460551	0.225285	0	0	0
kroA200	200	0.436189	0.526083	0.257083	0.088872	0.00681	0
gr202	202	0.732321	0.406375	0.309512	0.252988	0.011703	0
gr229	229	0.392788	0.468195	0.381644	0.152969	0.015007	0.004309
gil262	262	0.328007	0.723297	0.428932	0.084104	0.046257	0.004205
lin318	318	1.00264	1.74284	1.33884	0.583407	0.129197	0.02641
gr431	431	1.69438	2.71862	2.34071	0.563665	0.134003	0.023919
pcb442	442	0.966363	0.80783	1.36634	0.38816	0.038403	0.044311
att532	532	1.04746	2.28599	2.52871	0.386116	0.224662	0.089937
u574	574	1.36892	2.81263	3.66807	0.580951	0.278824	0.141444
rat575	575	0.806142	1.77174	2.25011	0.287908	0.171268	0.098922
gr666	666	1.66056	4.38707	6.00476	0.855251	0.497863	0.206279
u724	724	1.02505	2.25101	3.03054	0.61298	0.336674	0.168218
rat783	783	0.897116	2.24052	3.36929	0.511015	0.285033	0.161254
pr1002	1002	1.97877	3.31969	5.54336	1.04229	0.945357	0.620626
Average Excess		**0.729235**	**1.356155**	**1.653182**	**0.32994**	**0.15622**	**0.079492**

Table A.1 Results for GLS on the TSP.

Problem	No.Cities	Mean Excess (%) over 10 runs					
		DB-FI-LK	DB-FI-3Opt	DB-FI-2Opt	DB-FLS-LK	DB-FLS-3Opt	DB-FLS-2Opt
att48	48	0	0	0	0	0	0
eil76	76	0	0	0	0	0	0
kroA100	100	0	0	0	0	0	0
bier127	127	0	0	0	0	0	0
kroA150	150	0	0.001508	0.003393	0	0	0
u159	159	0	0	0	0	0	0
kroA200	200	0	0.077295	0.10113	0	0.004767	0.075252
gr202	202	0.009213	0.088396	0.457171	0	0.155129	0.257719
gr229	229	0.014116	0.157576	0.382387	0.004755	0.064115	0.124515
gil262	262	0.016821	0.20185	0.626577	0	0.075694	0.475189
lin318	318	0.255776	0.719027	1.14588	0.240786	0.279093	0.3519
gr431	431	0.332703	0.94403	2.13495	0.222386	0.394192	0.615294
pcb442	442	0.066367	0.368861	1.8961	0.081728	0.309977	0.684745
att532	532	0.225023	1.03554	2.64971	0.08163	0.270534	0.422957
u574	574	0.114348	1.20038	2.94269	0.092399	0.404823	0.553042
rat575	575	0.13731	1.15016	3.75904	0.097446	0.445888	0.649638
gr666	666	0.418878	1.25178	3.27054	0.175874	0.359528	0.816489
u724	724	0.356955	1.43617	3.94106	0.166547	0.367693	0.627535
rat783	783	0.240745	1.79764	5.00454	0.153305	0.516693	0.744947
pr1002	1002	1.04742	2.05625	5.19902	0.446332	0.872049	1.05727
Average Excess		**0.161784**	**0.624323**	**1.675709**	**0.088159**	**0.226009**	**0.372825**

Table A.2 Results for Iterated Local Search on the TSP.

Problem	No.Cities	**Mean Excess (%) over 10 runs**					
		REP-FI-LK	REP-FI-3Opt	REP-FI-2Opt	REP-FLS-LK	REP-FLS-3Opt	REP-FLS-2Opt
att48	48	0	0	0	0	0	0
eil76	76	0	0	1.35688	0	0	1.48699
kroA100	100	0	0.39564	0.222254	0	0.225543	0.215205
bier127	127	0.030098	0.403696	1.19629	0.027899	0.370386	1.29513
kroA150	150	0.002262	0.8317	2.00912	0.002262	0.8038	2.01553
u159	159	0	0.30038	1.62619	0	0.265447	2.05894
kroA200	200	0.024517	1.00688	3.30768	0.004767	0.922092	3.23583
gr202	202	0.141434	1.22958	3.58591	0.129731	1.19995	3.68352
gr229	229	0.097695	1.36774	3.40129	0.094427	1.27301	3.56443
gil262	262	0.054668	1.3709	5.12195	0.054668	1.2868	5.77796
lin318	318	0.629565	2.17992	4.37936	0.636703	2.022676	4.9128
gr431	431	0.679641	2.07801	5.33877	0.665232	2.20915	5.97495
pcb442	442	0.48525	1.77636	6.65012	0.516956	1.72417	7.19544
att532	532	0.530232	2.29033	6.28368	0.579354	2.29141	7.13899
u574	574	0.738382	2.91397	7.46674	0.703157	2.6934	8.4788
rat575	575	0.807618	2.69895	7.69231	0.887347	2.70781	8.61066
gr666	666	0.837619	3.18259	8.14712	0.847811	2.97203	9.94096
u724	724	0.933667	2.90551	7.76903	1.0241	2.87473	8.83202
rat783	783	1.00045	3.2864	8.46468	1.06518	3.39882	9.38792
pr1002	1002	1.5046	3.50511	8.62028	1.39138	3.59138	10.5847
Average Excess		**0.424885**	**1.686183**	**4.631983**	**0.431549**	**1.64163**	**5.219539**

Table A.3 Results for Repeated Local Search on the TSP.