8

Travelling Salesperson Problems

The Travelling Salesperson Problem (TSP) is probably the most widely studied combinatorial optimisation problem and has attracted a large number of researchers over the last five decades. Research on the TSP has been a driving force for the emergence of many important fields such as Stochastic Local Search Algorithms and Polyhedral Theory, as well as for the development of Complexity Theory. Apart from its practical importance, the TSP has also become a standard test-bed for new algorithmic ideas.

In this chapter we first give a general overview of aspects for TSP solving and available benchmark problems and present next the most basic local search algorithms for the TSP. Based on these algorithms, several SLS algorithms have been designed which largely improved the ability of finding high quality solutions for large instances. We give a detailed overview of Iterated Local Search algorithms, which are currently among the most successful algorithms for large TSPs, and present several approaches making use of a population of solutions. While most of this chapter focuses on symmetric TSPs, we also discuss aspects that arise for asymmetric TSPs.

8.1 TSP Applications and Benchmark Instances

Given an edge-weighted, completely connected, directed graph $G = (V, E, w)$, where V is the set of $n = |V|$ vertices, E the set of (directed) edges, and $w : E \mapsto \mathbb{R}^+$ a function assigning each edge $e \in E$ a weight $w(e)$, the Travelling Salesperson Problem (TSP) is to find a Hamiltonian cycle in G, *i.e.*, a cyclic path that visits each vertex exactly once, that has minimal total weight (a formal definition was given in Chapter 1, see page 13). Following one of the most intuitive applications of the TSP, namely, finding optimal round trips that visit a number of geographical locations, the vertices of a TSP instance are often called "cities", the paths in G are called "tours", and the edge weights are referred to as "distances". In this chapter we focus mainly on the symmetric TSP, *i.e.*, the class of TSP instances in which for each pair of edges (v_1, v_2) and (v_2, v_1) we have $w((v_1, v_2)) = w((v_2, v_1))$. We will also highlight some of the issues that arise when dealing with the asymmetric TSP (ATSP), where for at least one pairs of vertices the edges (v_1, v_2) and (v_2, v_1) have different weights.

TSP as a Central Problem in Combinatorial Optimisation

The TSP plays a prominent role in research as well as in a number of application areas. The design of increasingly efficient TSP algorithms has provided a constant intellectual challenge and many of the most important techniques for solving combinatorial optimisation problems were developed using the TSP as an example application. This includes cutting planes in integer programming [Dantzig *et al.*, 1954], which later lead to the modern, high performing Branch-and-Cut methods [Grötschel and Holland, 1991; Padberg and Rinaldi, 1991], polyhedral approaches [Grötschel and Padberg, 1985; Padberg and Grötschel, 1985], as well as early local search algorithms [Croes, 1958; Flood, 1956; Lin, 1965; Lin and Kernighan, 1973]. Additionally, many of the general SLS methods presented in Chapter 2, such as Simulated Annealing or Ant Colony Optimisation, were first tested on the TSP. The TSP also played an important role in the development of computational complexity theory [Garey and Johnson, 1979]. In fact, several books are entirely devoted to the TSP [Gutin and Punnen, 2002; Lawler *et al.*, 1985; Reinelt, 1994] and a huge number of research articles was written on aspects related to TSP solving. (For details on the history of TSP solving, we refer toa Schrijver's overview paper on the history of combinatorial optimisation [Schrjiver, 2002], the book chapter by Hoffmann and Wolfe [Hoffman and Wolfe, 1985], and the web-page www.math.princeton.edu/tsp/histmain.html.)

There are various reasons for this central role of the TSP in combinatorial optimisation. First, it is a conceptually simple problem, which is easily explained and understood, but as an $\mathcal{NP}$-hard problem, it is difficult to solve [Garey and Johnson, 1979]. Second, the design and analysis of algorithms for the TSP is not obscured by technicalities that arise from dealing with side constraints which are often difficult to handle in practice. Third, the TSP is now established as a standard testbed for new algorithmic ideas, which are often assessed based on their performance on the TSP. Fourth, given the significant amount of interest in the research community, new contributions to TSP solving or insights into the problem structure are likely to have large impact. Fifth, the TSP is a problem that arises in a variety of applications.

Benchmark Instances

Extensive computational experiments have always played an important role in the history of TSP. These experiments involve several types of TSP instances. In many cases these are predominantly metric TSP instances, *i.e.*, instances in which the vertices correspond to points in a metric space and the edge weights correspond to metric distances between pairs of points, rounded to the nearest integer. Integer distances are used in almost all available TSP benchmark instances, regardless of whether they are metric or not. Metric TSP instances for which the distances are computed by using the standard Euclidean metric, are also called Euclidean.

A well known and widely used collection of TSP instances is available through TSPLIB [Reinelt, 1991], a benchmark library for the TSP, which is accessible at www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95. TSPLIB comprises more than 100 instances with up to $85,900$ cities. For many TSPLIB instances optimal solutions have been determined; as of October 2002, the largest instance solved to optimality has $15,112$ cities (instance `d15112`). Most of the TSPLIB instances stem from influential studies on the TSP; many of them originate from practical applications, such as minimising drill paths in printed circuit board manufacturing, positioning of detectors in X-ray crystallography, or finding the shortest round-trip through all the beer-gardens in Augsburg, Germany.[1] Many of the remaining TSPLIB instances are of geographical nature, where the inter-vertex dis-

[1] Augsburg is close to one of the authors' (T.S.) home town; however, T.S. never managed to do all beer-gardens in one night, although knowing the shortest tour.

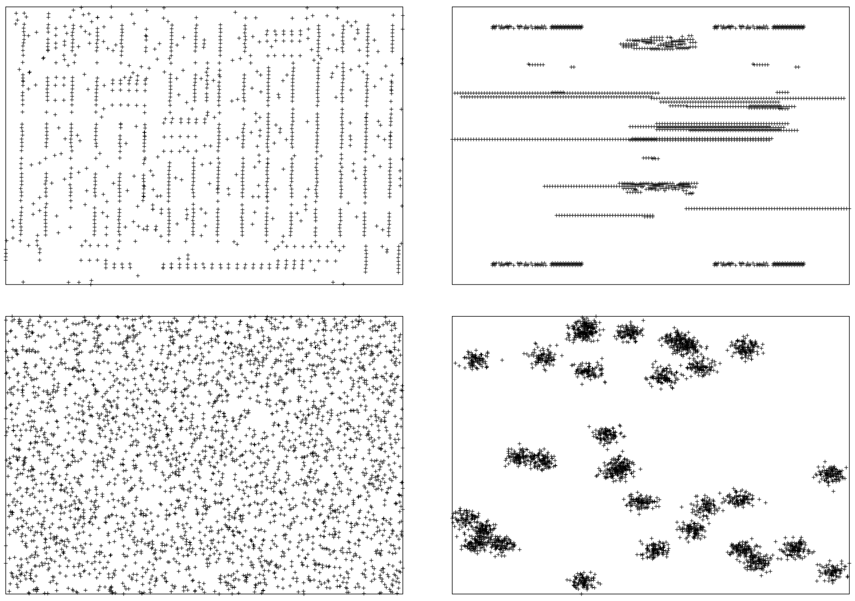


Figure 8.1: Four (Euclidean) TSP benchmark instances. The two upper instances stem from an application in which drill paths in manufacturing printed circuit boards are to be minimised in length (left: TSPLIB instance `pcb1173` with 1173 vertices and right: `fl1577` with 1577 vertices, the latter instance shows a pathological clustering of vertices). The bottom row shows a Random Uniform Euclidean instance (left) and a Random Clustered Euclidean instance (right).

tances are derived from the distances between cities and towns with given coordinates. Two examples of TSPLIB instances are shown in the upper part of Figure 8.1.

A set of TSP instances derived from problems in VLSI design, ranging from 131 to $744,710$ vertices, is available from the web-page on *Solving TSPs* accessible at www.math.princeton.edu/tsp. From the same web-page a TSP instance of potential interest for globetrotter is available, constructed by Applegate, Bixby, Chvátal, and Cook. The *World TSP* instance comprises all $1,904,711$ populated cities or towns registered in the National

Imagery and Mapping Agency database and the Geographic Names Information System; several additional TSP instances comprising the towns and cities of individual countries are available from the same site. An overview of further practical applications of the TSP can be found on the web-page www.math.princeton.edu/tsp/apps/index.html or in the book by Reinelt [Reinelt, 1994].

A large part of the experimental (and also theoretical) research on the TSP has used randomly generated instances with the most widely used classes being Random Euclidean instances (RE) and Random Distance Matrix instances (RDM). In RE instances, the vertices correspond to randomly placed points in a d-dimensional hypercube, and the edge weights are given by the pairwise Euclidean distances between these points. (The Euclidean distance between two points $x = (x_1, \dots, x_d)$ and $y = (y_1, \dots, y_d)$ is defined as $d(x, y) = \sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$.) Commonly, real valued weights are scaled by a constant factor α and subsequently rounded or truncated to obtain integer values; by choosing α sufficiently large, the resulting deviations from true Euclidean distances are rendered insignificant.

Most experimental studies involving RE instances have focused on two dimensional instances (*i.e.*, $d = 2$), in which the points are uniformly distributed in a square; we refer to these as Random Uniform Euclidean instances (RUE). An example for an RUE instance is shown in Figure 8.1 (bottom left). The class of RUE instances has the interesting property that, as the instance size approaches infinity, the ratio of the optimal tour length to $\sqrt{n}$ (where n is the number of vertices) converges towards a constant γ [Beardwood *et al.*, 1959], whose value is now known to be approximately 0.721 [Johnson *et al.*, 1996; Percus and Martin, 1996].

Another type of two dimensional RE instances, which have been used in the 8th DIMACS Implementation Challenge on the TSP [Johnson *et al.*, 2002a], places the points in clusters within a square area. More precisely, these Random Clustered Euclidean instances (RCE) are obtained by first distributing the cluster centres uniformly at random; then, each actual point is placed by choosing a cluster centre uniformly at random, and then adding to each coordinate a displacement that is sampled from a normal distribution. An example of an RCE instance is shown in Figure 8.1 (bottom right). The RCE instances are interesting because it is known that various local search algorithms are negatively affected by a clustering of the vertices. (The same effect can be observed for non-random instances, such as TSPLIB instance `fl1577` shown in Figure 8.1).

RDM instances are symmetric, non-Euclidean instances in which the edge weights are randomly chosen integers from a given interval. This distribution of TSP instances is known to pose a considerable challenge for many SLS algorithms [Johnson and McGeoch, 1997].

Lower Bounds on the Optimal Solution Quality

A large amount of research efforts on the TSP have been dedicated to finding good lower bounds on the optimal solution quality for given instances. Lower bounds are used in complete algorithms like Branch-and-Bound to estimate the minimum cost incurred for completing partial solutions and to prune search trees, if the cost estimation is larger than or equal an already found solution. In this context it is important to have estimates that are close to the real costs, because the better the estimates the larger parts of the search tree can be pruned. When applying SLS algorithms, lower bounds can be used to give a guarantee on the quality of some the solution they return.

A general approach to obtain lower bounds is to solve a relaxation of the original problem. Feasible solutions to the original problem are then a subset of the solutions to the relaxed problem and an optimal solution to the relaxed problem is therefore always a true lower bound for the solution quality of the original problem. The main goal then becomes to find relaxations that result in problems that can quickly be solved but at the same time yield optimal solution values close to the optimum for the original problem.

One of the simplest lower bounds on the optimal tour length of some TSP instance can be obtained based on the following relaxation: By removing a single edge from an optimal tour s^* with weight $w(s^*)$, a spanning tree t of G with weight $w(t)$ is obtained, where the weight of the spanning tree is defined as the sum of the weights of the edges it contains. Obviously, a minimum weight spanning tree t^*, which can be computed in $\mathcal{O}(m \log n)$ by using Kruskal's or Prim's algorithm [Cormen *et al.*, 2001], where m is the number of edges, has a total edge weight $w(t^*) \leq w(t)$, and hence $w(t^*)$ is a lower bound for $w(s^*)$ (notice that Prim's algorithm can be speeded up using Fibonacci heaps to run in $\mathcal{O}(m + n \log n)$).

This relaxation can be made tighter as follows: Let $G \setminus \{v1\}$ be the graph that is obtained from G by deleting vertex v_1 and all the edges incident to v_1. A one-tree is a spanning tree on the vertices $v_2, v_3, \dots v_n$ plus two edges incident to vertex v_1. We get a minimum weight one-tree for G by computing a minimum spanning tree of $G \setminus \{v1\}$ and adding the two least cost edges incident to v_1. The weight of the resulting one-tree is a lower bound for $w(s^*)$ since every minimum weight tour s^* of G is a one-tree. This lower bound could be improved by choosing several or all vertices to play the role of v_1 and then taking the maximum weight of the corresponding one-trees as a lower bound $w(s^*)$. However, this does not result in significant gains and is quite time consuming [Reinelt, 1994; Cook *et al.*, 1997].

Luckily, there exist other techniques to improve upon the one-tree bound. These are based on the following observation: We can assign a value p_i to each vertex v_i and add these values to the weights of all edges incident to a vertex $v_i \in V$, resulting in a graph $G' = (V, E, w')$ with edge weights $w'((v_i, v_j)) = w((v_i, v_j)) + p_i + p_j$ (recall that the edges are not oriented). This has the effect of increasing the weight of each tour in G by a constant amount of $2 \cdot \sum_{i=1}^{n} p_i$ in G'. Clearly, this transformation preserves the optimality of tours. It may, however, result in different optimal one-trees [Held and Karp, 1971; Cook *et al.*, 1997]; by subtracting $2 \cdot \sum_{i=1}^{n} p_i$ from a minimal weight one-tree of G', a lower bound on the mimimum weight tour in G can be obtained.

The quality of this lower bound depends on the values of the vertex penalties $p_1, \dots, p_n$ and it can be optimised by iteratively modifying the penalties $p_1, \dots, p_n$ [Held and Karp, 1971]. Roughly speaking, these methods iteratively increase the penalties for vertices with degree one in the current optimal one-tree, and decrease the penalties for vertices with degree greater than two; the search is terminated when a minimum weight one-tree is obtained in which all vertices have degree two (this corresponds to a feasible solution of G), or when a maximum number of iterations has passed. In the literature, the bounds as described above are called Held-Karp lower bounds [Reinelt, 1994].

Experimental results suggest that for many types of TSP instances, the Held-Karp bounds are very tight. For RUE instances, the lower bound is typically within less then one percent of the actual optimal solution quality [Johnson *et al.*, 1996]. For TSPLIB instances, the gap between the Held-Karp lower bound and the respective optimum solution quality is often slightly larger, but for almost all instances the lower bound is still within two percent of the optimal solution quality.

State-of-the-Art Methods for TSP Solving

The TSP is probably the best example for the extremely high performance of modern algorithmic techniques for $\mathcal{NP}$-hard optimisation problems. To date, small to medium size symmetric TSP instances ranging from a few hundred to around 1,000 to 3,000 vertices can be solved optimally within a few CPU hours or less using state-of-the-art complete TSP algorithms, particularly Branch-and-Cut methods.

Branch-and-Cut works by solving a series of linear programming relaxations of an integer programming problem [Mitchell, 2002]. Such a relaxation is typically obtained by allowing 0-1 variables, which are typically used in integer programming formulations of the TSP [Nemhauser and Wolsey, 1988], to take arbitrary values from the interval $[0, 1]$ instead of constraining them to integer values from the set $\{0, 1\}$. Cutting plane methods are used to make the relaxation more closely approximate the optimum solution value of the original integer programming problem. This is done by finding linear inequalities that are satisfied by all integer feasible solutions but not by the optimal solution of the current relaxation. These inequalities are then added to obtain the next linear optimisation problem which again is solved to optimality. This process is iterated until finding "good" cuts becomes hard. Then, it may become preferable to branch by splitting the current problem into two subproblems; this is done by forcing one edge to be part of any solution in one subproblem and to not appear in any solution of the other subproblem [Grötschel and Holland, 1991; Padberg and Rinaldi, 1991].

Based on the impressive performance of state-of-the-art complete TSP algorithms for solving large symmetric instances optimally, most studies on symmetric TSP algorithms now focus on solving large instances starting with thousands of vertices. For example, in the context of the 8th DIMACS Implementation Challenge on the TSP [Johnson *et al.*, 2002a], only instances with 1,000 or more vertices were considered. However, complete algorithms have their limitations. Firstly, the computation times become quickly prohibitively large with increasing instance size. For example, the largest TSPLIB instance that has been solved (provably) optimally at the time of this writing (August, 2002) has 15,112 vertices; but this took a total estimated computation time of 22.6 CPU years on a Compaq EV6 Alpha processor running at 500 MHz (this computation was actually performed on a network of 110 workstations). Secondly, the computation times of complete methods vary strongly depending on the particular instance: while TSPLIB instance `pr2392` (with 2392 vertices) was solved within 116 CPU seconds on a 500 MHz Compaq XP1000 workstation, solving the TSPLIB instance `d2103` (with 2103 vertices) took about 129 CPU days on a set of 400 and 500 MHz Alpha 21164 processors.

It may be noted, that the largest part of the computation time required by complete algorithms is not required for finding an optimal solution for a given instance, but rather for proving its optimality; general folklore suggests that around 90% of the computation time are spent for the proof of optimality. Nevertheless, incomplete heuristic search algorithms are the methods of choice for solving very large or hard problem instances as well as for finding high quality solutions very quickly.

Heuristic construction search algorithms are useful for obtaining reasonably good solutions very quickly. The best construction methods, such as the Savings Heuristic or Farthest Insert, can solve symmetric TSP instances with a few thousands vertices within about 11 to 16 percent of the Held-Karp lower bounds (which, as explained above, are known to be close to the optimal solution quality) in fractions of a CPU second on a state-of-the-art processor [Johnson and McGeoch, 2002]. When allowing run-times of a few CPU seconds, the same relative solution quality can be obtained for instances with several hundred thousand vertices [Johnson and McGeoch, 2002].

Better quality solutions can typically be obtained by using hybrid SLS algorithms at the cost of higher computation times. State-of-the-art SLS algorithms for the TSP can find optimal solutions for symmetric instances with around thousand cities within few CPU minutes on a modern workstation; significantly larger instances can be solved optimally within hours of CPU time. For example, the best performing SLS algorithm identified in a recent extensive experimental study found a solution of TSPLIB instance `d15112` whose quality is only 0.0186 percent away from the known optimum in about seven hours of CPU time [Johnson and McGeoch, 2002] (this result was reported for the iterated version of Helsgaun's `LK` heuristic, which is presented in Section 8.3, on a Compaq ES40 machine with

500MHz Alpha processors); some SLS algorithms obtained solution qualities for this instance within one percent of the optimum in less than seven CPU seconds. The impressive performance of SLS algorithms when applied to very large TSP instances is exemplified by the results obtained for a RUE instance comprising 25 million cities, for which after 8 CPU days on a IBM RS6000, Model 43-P 260, a solution quality within 0.3 percent of the estimated optimal solution quality was obtained using a high performing iterated local search algorithm [Applegate *et al.*, 1999b].

Asymmetric TSPs

Empirical results indicate that asymmetric TSP (ATSP) instances, in which a given graph has at least one pair of vertices for which $w((v, v')) \neq w((v', v))$, are typically harder to solve than symmetric TSP instances of comparable size [Johnson *et al.*, 2002b]. TSPLIB includes 27 ATSP instances ranging from 17 to 443 vertices. A large number of additional instances was recently generated for testing various heuristic search algorithms for the ATSP [Cirasella *et al.*, 2001; Johnson *et al.*, 2002b]; these include several classes of randomly generated ATSP instances that model real-world problems such as moving drills along a tilted surface, scheduling read operations on computer disks, collecting coins from pay phones, or finding shortest common super-strings for a set of genomic DNA sequences (a problem that arises in genome reconstruction). There are also some individual instances directly taken from practical applications of the ATSP, such as stacker crane problems, vehicle routing [Fischetti *et al.*, 1994], robot motion planning, scheduling read operations on a tape drive, or code optimisation [Young *et al.*, 1997]. These instances and random instance generators are available online at www.research.att.com/ dsj/chtsp/atsp.html.

Instead of applying a native ATSP algorithm, ATSP instances can be solved by using a transformation into symmetric TSP in conjunction with a symmetric TSP algorithm. One such transformation works as follows [Jonker and Volgenant, 1983]. Given a directed graph $G = (V, E, w)$, with vertex set $V = \{v_1, \ldots, v_n\}$, edge set E, and weight function w, we define an undirected graph $G' = (V', E', w')$ with $V' = V \cup \{v_{n+1}, v_{n+2}, \ldots, v_{n+n}\}$, $E' = V' \times V'$, and w' is defined as

$$w'((v_i, v_{n+j})) := w'((v_{n+j}, v_i)) := w(v_i, v_j) \qquad \text{for} \qquad i, j \in \{1, \ldots, n\} \text{ and } i \neq j$$

$$w'((v_{n+i}, v_i)) := w'((v_i, v_{n+i})) := -M \qquad \text{for} \qquad i \in \{1, \ldots, n\}$$
$$w'(v_i, v_j) := M \quad \text{otherwise} \; ,$$

where M is a sufficiently large number, *e.g.*, $M = \sum\{w(v, v') \mid v, v' \in V\}$.

Although symmetric TSP instances obtained from this transformation have twice as many vertices as the respective original ATSP instances, solving these using algorithms for symmetric TSP is often more effective than solving the original ATSP instances using native ATSP algorithms [Johnson *et al.*, 2002b]. Furthermore, empirical results indicate that the Held-Karp (HK) lower bounds on the optimal solution quality for the symmetric TSP instances obtained from this transformation are often tighter than the widely used Assignment Problem (AP) lower bounds for the respective ATSP instances [Johnson *et al.*, 2002b]. Recent empirical results indicate that ATSP instances with a relatively large gap between the AP and the HK bound, are most efficiently solved by transforming them into symmetric TSP instances and solving these using state-of-the-art symmetric TSP algorithms, such as Helsgaun's Lin-Kernighan variant [Helsgaun, 2000]. However, ATSP algorithms that are guided by information from the AP bound, such as Zhang's heuristic [Zhang, 1993], tend to show better performance for ATSP instances for which both bounds are relatively close to each other [Johnson *et al.*, 2002b]. (Zhang's heuristic is a truncated Branch-and-Bound algorithm which uses the AP lower bound.)

8.2 'Simple' SLS Algorithms for the TSP

Much of the early research on incomplete algorithms for the TSP focused on construction heuristics and iterative improvement algorithms. These techniques are important, because they are at the core of many applications of advanced SLS techniques to the TSP. In this section we give an overview of the most important variants of conceptually simple local search algorithms.

Nearest Neighbour and Insertion Construction Heuristics

There is a large number of constructive search algorithms for the TSP, ranging from extremely fast methods for geometric TSP instances, whose runtime is only slightly larger than the time required for just reading the instance data from hard-disk, to more sophisticated algorithms with non-trivial bounds on the on the solution quality achieved in the worst case. In the context of SLS algorithms, construction heuristics are often used for initialising the search; iterative improvement algorithms for the TSP typically require fewer steps for reaching a local optimum when started from higher quality tours obtained from a good construction heuristic.

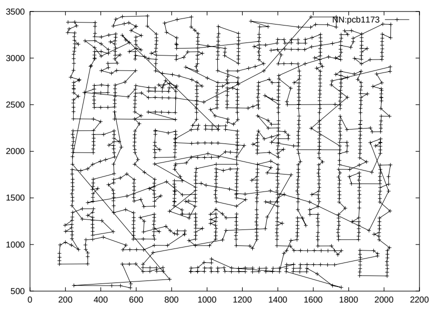
One particularly intuitive and well-known construction heuristic was already discussed in Section 1.4: The *Nearest Neighbour Heuristic (NN)* starts tour construction from some randomly chosen vertex v_1 in the given graph and then iteratively extends the current partial tour $p = (v_1, \ldots, v_k)$ with the unvisited vertex v_{k+1} that is connected to v_k by a minimal weight edge (v_{k+1} is called the nearest neighbour of v_k); when all vertices are visited, a complete tour is obtained by extending p with the initial vertex, v_1. The tours constructed by NN are called *nearest neighbour tours*.

For TSP instances satisfying the triangle inequality, in the worst case the solution quality achieved by NN can be up to a factor $1/3(\lceil \log_2 n \rceil + 4/3)$ worse then the optimum solution quality [Rosenkrantz *et al.*, 1977], and hence, worst case performance of NN cannot be bounded by any constant. In practice, however, NN typically yields tours that are only 20%–35% worse than an optimal tour for metric or TSPLIB instances in terms of solution quality, and are locally similar to optimal solutions, except for some very expensive edges that have to be included towards the end of the construction process in order to complete the tour (see Figure 8.2 for examples). This effect is avoided to some extent by a variant of NN that penalises insertions of such expensive edges [Reinelt, 1994]; compared to standard NN, this variant requires only slightly more computation time, but applied to TSPLIB instances, results in tours that are around 5% better.

Insertion heuristics grow tours in a different way from NN; they extend partial tours by iteratively choosing, according to some heuristic rule, a next vertex that is to be inserted into the current partial tour p, typically at a position where it leads to a minimial increase in cost. Several variants of these heuristics exist, including

(i) *nearest insertion* construction heuristics, where the next vertex to be inserted is a vertex v_i with minimum distance to any vertex v_j in p;

(ii) *cheapest insertion,* which inserts a vertex that leads to the minimum increase of the weight of p for all vertices not yet in p;



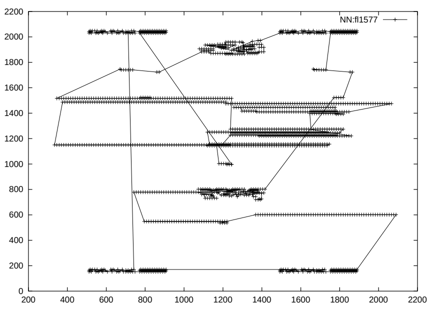


Figure 8.2: Two examples of nearest neighbour tours for TSPLIB instances `pcb1173` with 1173 vertices (left side) and `fl1577` with 1577 vertices (right side). Note the long (*i.e.*, expensive) edges contained in both tours.

(iii) *farthest insertion*, where the next vertex to be inserted is a vertex v_i for which the minimal distance to vertex in p is maximal.

(iv) *random insertion*, where the next vertex to be inserted is chosen randomly.

For TSP instances that satisfy the triangle inequality, the tours constructed by nearest and cheapest insertion are provably at most twice as expensive as an optimal tour [Rosenkrantz *et al.*, 1977]; for random and farthest insertion, the solution quality are only guaranteed to be within a factor $\mathcal{O}(\log n)$ of the optimum [Johnson and McGeoch, 2002]. In practice, however, the farthest and random insertion heuristics perform much better than nearest and cheapest insertion [Johnson and McGeoch, 2002; Reinelt, 1994], averaging between 13 to 15% above the optimum tours for TSPLIB or RUE instances.

The Greedy, Quick-Borůvka, and Savings Heuristics

The construction heuristics discussed so far build a complete tour by iteratively extending a connected partial tour. An alternate approach is to iteratively build several partial tours that are ultimately patched together into a complete tour. One example for a construction heuristic of this type is the

Greedy Heuristic, which works as follows. First all edges in the given graph G are sorted according to increasing weight. Then, this list is scanned, starting from the minimal weight edge, in linear order. An edge is added to the current partial candidate solution p in such a way that at no point in time the subgraph G' of G formed by the edges in p has any vertices of degree greater than two, or any cycles of length less than n edges.

There exist several variants of the Greedy Heuristic that use different criteria for choosing the edge to be added in each construction step. One of these is the *Quick-Borůvka Heuristic* by Applegate *et al.*, which is inspired by the minimum spanning tree (MST) algorithm due to Borůvka [Borůvka, 1926; Applegate *et al.*, 1999a]. First, the vertices in G are sorted arbitrarily (*e.g.*, for geometric TSP instances, the vertices can be sorted according to the first coordinate). Then, the vertices are processed in the given order. For each vertex v_i that has degree less than two, the minimum weight edge incident to v_i, which does not create a cycle of length less than n and which does not make any vertex have a degree larger than two, is added. Hence, at most two scans of the vertices have to be made to generate a tour.

Another construction heuristic that is based on building multiple partial tours, is the *Savings Heuristic*, which was initially proposed for the vehicle routing problem [Clarke and Wright, 1964]. It works by first choosing a base vertex v_b and $n-1$ cyclic paths (v_b, v_i, v_b) that consist of two vertices each. As long as more than one cyclic path is left, at each construction step two cyclic paths p_1 and p_2 are combined by removing one edge incident to v_b in both, p_1 and p_2, and by connecting the two resulting paths into a new cyclic path p_{12} The edges to be removed in this operation are selected such that a maximal reduction in cost of p_{12} compared to the total combined cost of p_1 and p_2 is achieved.

Regarding worst case performance, it can be shown that greedy tours are at most $(1 + \log n)/2$ times as expensive as an optimal solution, while the cost of a savings tour is at most a factor $(1+\log n)$ above the optimum [Ong and Moore, 1984]; no worst case bounds on solution quality are known for Quick-Borůvka tours. Empirically, the Savings Heuristic produces better tours than both Greedy and Quick-Borůvka; for example, for large RUE instances, the cost of savings tours is on average around 12% above the Held-Karp lower bounds, while Greedy and Quick-Borůvka find solutions around 14% and 16% above these lower bounds, respectively [Johnson and McGeoch, 2002]. Computation times are modest, though, ranging for 1 million vertex RUE instances between 22 (for Quick-Borůvka) to around 100 seconds (for Greedy and Savings).

Construction Heuristics based on Minimal Spanning Trees

Yet another class of construction heuristics builds tours based on minimal-weight spanning trees (MSTs). In the simplest case, such an algorithm consists of the following four steps: Firstly, an MST t for the given graph G is computed; secondly, by doubling each edge in t, a new graph G' is obtained. In the third step, an Eulerian tour p of G', *i.e.*, a cyclic path that uses each edge in G' exactly once, is generated; an Eulerian tour can be found in $\mathcal{O}(e)$, where e is the number of egdes in the graph [Cormen *et al.*, 2001]. Finally, p is converted into a Hamiltonian cycle in G by iteratively short-cutting subpaths of p (see Chapter 6 in [Reinelt, 1994] for an algorithm for this step). This last step, however, does not increase the weight of a tour, if an instance satisfies the triangle inequality; hence, in this case, the final tour is at most twice as expensive as an optimal solution. However, empirically this construction heuristic performs rather poorly, with solution qualities that are on average around 40% above the optimal tours for TSPLIB and RUE instances [Reinelt, 1994; Johnson and McGeoch, 2002].

Much better performance is obtained by the *Christofides Heuristic*. The central idea behind this heuristic is to compute a minimum weight perfect matching of the odd degree vertices of the MST (there must be an even number of such vertices), which can be done in time $\mathcal{O}(k^3)$, where k is the number of odd degree vertices. (A minimum perfect matching of a vertex set is a set of edges such that each vertex is incident to exactly one of these edges; the weight of the matching is the sum of the weights of its edges). This is sufficient for converting the MST into an Eulerian graph, *i.e.*, a graph containing an Eulerian tour. As described above, in a final step this Eulerian tour is converted into a Hamiltonian cycle. Interestingly, for TSP instances that satisfy the triangle inequality, the resulting tours are guaranteed to be at most a factor 1.5 above the optimum solution quality.

While the standard version of the Christofides Heuristic appears to perform worse than both Savings and Greedy [Reinelt, 1994; Johnson and McGeoch, 2002], its performance can be substantially improved by additionally using greedy heuristics in the conversion of the Eulerian tour into a Hamiltonian cycle. This variant of the Christofides Heuristic appears to be the best performing construction heuristic for the TSP in terms of the solution qualities achieved; however, its run-times are higher by a factor 1.3 to 4 (depending on the implementation details) than those of Savings or Greedy [Johnson and McGeoch, 2002].

K-Exchange Iterative Improvement Methods

Most iterative improvement algorithms for the TSP are based on the k-exchange neighbourhood, in which candidate solutions s and s' are direct neighbours if s' can be obtained from s by deleting a set of k edges and rewiring the resulting fragments into a complete tour by inserting a different set of k edges. For iterative improvement algorithms for the TSP that use a fixed k-exchange neighbourhood, $k = 2$ and $k = 3$ are the most common choices; the respective TSP algorithms are known as `2-opt` and `3-opt`. Current knowledge suggests that the slight improvement in solution quality obtained by increasing k to four and beyond is not amortised by the substantial increase in computation time [Lin, 1965].

The most straightforward implementation of a k-exchange iterative improvement algorithm, considers in each step all possible combinations for the k edges to be deleted and replaced. After deleting k edges from a given candidate solution s, the number of different ways of reconnecting the resulting fragments into a candidate solution different from s depends on k; for $k = 2$, after deleting two edges (v_i, v_j) and (v_k, v_l), the only way to rewire the two partial tours into a complete tour is by introducing the edges (v_i, v_k) and (v_l, v_j) (a 2-exchange move is illustrated in Figure 1.4 on page 35.) Note that after a 2-exchange move, one of the two partial tours is reversed.

For $k = 3$, there are several ways of reconnecting the three tour fragment obtained after deleting three edges, and in an iterative improvement algorithm based on this neighbourhood, all of these need to be checked for possible improvements. Figure 8.3 shows two of the four ways of completing a 3-exchange move after removing a given set of three edges. Furthermore, 2-exchange moves can be seen as special cases of 3-exchange moves in which the set of removed and added edges have one edge in common. Allowing overlap between these two sets has the advantage that any tour that is locally optimal w.r.t. a k-exchange neighbourhood is also locally optimal w.r.t. to all k'-exchange neighbourhoods with $k' < k$.

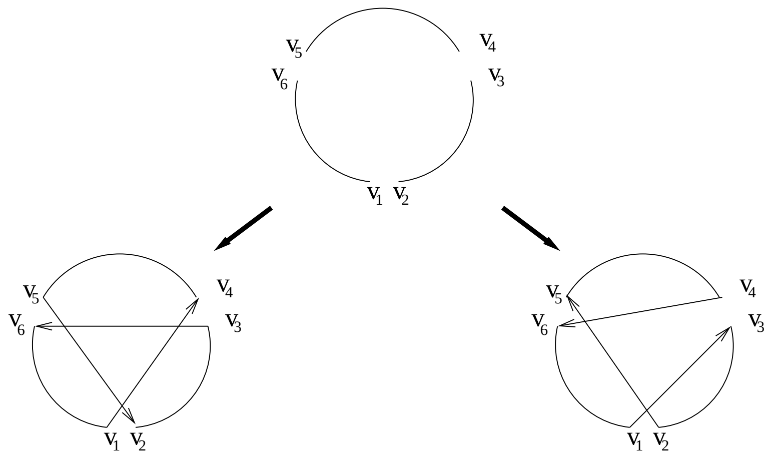


Figure 8.3: Two possible ways of reconnecting partial tours in a 3-exchange move after edges (v_1, v_2), (v_3, v_4), and (v_5, v_6) have been removed from a complete tour. Note that in the left result, the direction of all three tour fragments is preserved.

Based on the 2-exchange and 3-exchange neighbourhoods, various iterative improvement algorithms for the TSP can be defined in a straightforward way; these are generally known as `2-opt` and `3-opt` algorithms, since they produce tours that are locally optimal w.r.t. the 2-exchange and 3-exchange neighbourhoods, respectively. In particular, different pivoting rules can be used (these determine the exact mechanism used for selecting an improving neighbouring candidate solution; see also Chapter **??**, Section 2.1). In general, first-improvement algorithms for the TSP can be implemented in such a way that the time complexity of each search step is substantially lower than for best-improvement algorithms. But even first-improvement `2-opt` and `3-opt` algorithms need to examine up to $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ neighbouring candidate solutions in each step, which leads to a significant amount of CPU time per search step when applied to TSP instances with several hundreds or thousands of vertices. Fortunately, there exist a number of speed-up techniques for the TSP that allow to significantly improve the time-complexity of local search steps [Bentley, 1992; Johnson and McGeoch, 1997; Martin *et al.*, 1991; Reinelt, 1994]. By using these techniques, which will be described in detail in the following, `2-opt` and `3-opt` implementations can find local optima for instances with several hundreds or thousands of vertices within fractions of a second on current PCs.

Fixed Radius Search

For any improving 2-exchange move from a tour s to a neighbouring tour s', there is at least one vertex that is incident to an edge e in s that is replaced by a different edge e' with lower weight than e. This observation can be exploited for speeding up the search for an improving 2-exchange move from a given tour s. For a vertex v_i, we consider both of its tour neighbours as v_j. Then, we search around v_i for vertices v_k that are closer than $w(v_i, v_j)$, the radius of the near neighbour search, to v_i. This fixed radius near neighbour search for vertex v_i can be performed efficiently by using appropriately defined candidate lists, which will be discussed in more detail in the next section. For each vertex v_k found in this search, removing one of its two incident edges in s leads to a feasible 2-exchange move. The first such 2-exchange move that leads to an improvement in solution quality is applied to s and the iterative improvement search is continued from the resulting tour s' by performing a fixed radius near neighbour search for another vertex. If fixed radius near neighbour searches for all vertices do not result in any improving 2-exchange move, the current tour is `2-opt`.

The idea of fixed radius search can be extended to `3-opt` algorithms [Bentley, 1992]. In this case, for each search step, two fixed radius near neighbour searches are required, one for a vertex v_i as in the case of `2-opt` (see above), resulting in a vertex v_k, and the other for the tour neighbour v_l of v_k with radius $w(v_i, v_j) + w(v_k, v_l) - w(v_i, v_k)$.

Candidate Lists

In the context of identifying candidates for k-exchange moves, it is useful to be able to efficiently access the vertices in the given graph G that are connected to a given vertex v_i by edges with low weight, *e.g.*, in the form of a list of neighbouring vertices v_k that is sorted according to edge weight $w((v_i, v_k))$ in ascending order. By using such candidate lists for all vertices in G, typical fixed radius near neighbour searches can be performed very efficiently; this is exemplified by the empirical results reported in Example 8.1 on page 319. Interestingly, the use of candidate lists within iterative first-improvement algorithms, such as `2-opt`, often leads to improvements in the quality of the local optima found by these algorithms. This suggests that the highly localised local search steps that are evaluated first when using candidate lists are more effective than other k-exchange moves.

Full candidate lists comprising all $n - 1$ other vertices require $\mathcal{O}(n^2)$ memory and take $\mathcal{O}(n^2 \log n)$ time to construct. Therefore, especially to reduce the memory requirements, it is often preferable to use bounded-length candidate lists. When using bounded-length candidate lists, a fixed radius near neighbour search for a given vertex v_i is aborted when the candidate list for v_i has been completely examined, if the radius criterion did not stop the search earlier. As a consequence, the tour obtained by an iterative improvement algorithm based on this mechanism is no longer guaranteed to be locally optimal, because some improving moves may be missed.

Typically, candidate lists of lengths between 10 and 40 are used, although lower or lower bounds are sometimes applied. Simply using that short candidate lists comprising the vertices connected by the k lowest weight edges incident to a given vertex can be problematic, especially for clustered instances, like those shown on the right side of Figure 8.1 on page 302. For geometric TSP instances, an alternative approach to constructing bounded-length candidate lists include so-called quadrant-nearest neighbour lists [Miller and Pekny, 1991; Johnson and McGeoch, 1997] and candidate lists based on Delaunay triangulations [Reinelt, 1994].

Helsgaun proposed a more complex mechanism for constructing candidate lists that is based on an approximation on the Held-Karp lower bounds (see Section 8.1) [Helsgaun, 2000]. This mechanism works as follows: Based on the modified edge weights $w'((v_i, v_j))$ obtained from the approximation of the Held-Karp lower bounds, so-called α-values are computed for each edge (v_i, v_j) as $\alpha((v_i, v_j)) := w'(t^+(v_i, v_j)) - w'(t)$, where $w(t)$ is the weight of a minimal weight one-tree t and $w'(t^+(v_i, v_j))$ is the weight of a minimal weight one-tree $t^+(v_i, v_j)$ that is forced to contain edge (v_i, v_j). For each edge $\alpha(v_i, v_j) \geq 0$, and $\alpha(v_i, v_j) = 0$ if the edge (v_i, v_j) is contained in some minimal weight one-tree. A candidate list for a vertex v_i can now be obtained by sorting the edges incident to v_i according to their α-values in ascending order and bounding the length of the list to a fixed value k or by accepting only edges with α-values that are below some given upper bound. The vertices contained in these candidate lists are called α-nearest neighbours.

Empirically it has been shown that compared to the candidate lists obtained by the other methods mentioned above, candidate lists based on α-values can be much smaller and still cover all edges contained in an optimal solution. For example, for TSPLIB instance `att532`, candidate lists consisting of 5 α-nearest neighbours cover an optimal solution, while list length 22 is required when using standard candidate lists based on the given edge weights [Helsgaun, 2000].

Don't Look Bits

Another widely used mechanism for speeding up iterative improvement search for TSP is based on the following observation. If in a given search step, no improving k-exchange move can be found for a given vertex v_i (*e.g.*, in a fixed radius near neighbour search), it is unlikely that an improving move involving v_i will be found in future search steps, unless at least one of the edges incident to v_i in the current tour has changed.

This can be exploited for speeding up the search process by associating a single *Don't Look Bit (DLB)* with each vertex; at the start of the iterative improvement search, all DLBs are turned off, *i.e.*, set to zero. If in a search step no improving move can be found for a given vertex, the respective DLB is turned on, *i.e.*, set to one. After each local search step, the DLBs of all vertices incident to edges that were modified (*i.e.*, deleted from or added to the current tour) in this step are turned off again. The search for improving moves is generally restricted to only those vertices whose DLB is turned off. In practice, the DLB mechanism significantly reduces the time complexity of first–improvement search, since after a few neighbourhood scans, most of the DLBs will be turned on. The obtainable speed improvment is illustrated by the empirical results for various variants of `2-opt` shown in Example 8.1.

The DLB mechanism can easily be integrated with more complex SLS algorithms such as Iterated Local Search or Genetic Local Search: One possibility is to set only the DLBs of those vertices to zero which are deleted by the application of a tour perturbation or a recombination operator; this approach is followed in various algorithms described in Sections 8.3 and 8.4 and typically leads to a further substantial reduction of computation time

	`2-opt`-**std**		`2-opt`-**fr+cl**		`2-opt`-**fr+cl+dlb**		`3-opt`-**fr+cl+dlb**	
instance	Δavg	t_{avg}	Δavg	t_{avg}	Δavg	t_{avg}	Δavg	t
`kroA100`	8.9	1.6	6.4	0.5	6.6	0.4	2.4	4.3
`d198`	5.7	6.4	4.2	1.2	4.3	0.8	1.4	30.1
`lin318`	10.6	22.1	7.5	2.1	7.9	1.5	3.4	65.5
`pcb442`	12.7	55.7	7.1	2.9	7.6	2.2	3.8	63.4
`rat783`	13.0	239.7	7.5	7.5	8.0	5.8	4.2	213.8
`pr1002`	12.8	419.5	8.4	13.2	9.2	9.7	4.6	357.6
`pcb1173`	14.5	603.1	8.5	16.7	9.3	12.4	5.2	372.3
`d1291`	16.8	770.3	10.1	16.9	11.1	12.4	5.5	377.6
`fl1577`	13.6	1251.1	7.9	25.8	9.0	19.2	4.0	506.8
`pr2392`	15.0	2962.8	8.8	65.5	10.1	49.1	5.3	878.1

Table 8.1: Computational results for different variants of `2-opt` and `3-opt`. Δ_{avg} denotes the average percentage deviation from the optimum solution quality over 1000 runs per instance, and t_{avg} is the average run-time for 1000 local searches in CPU milliseconds on a Pentium III 500 MHz CPU. (For further details, see text.)

when compared to re-setting all DLBs to zero. Furthermore, DLBs can be used to speed up first-improvement local search algorithms for combinatorial problems other than TSP.

Example 8.1: Effects of Speed-Up Techniques for `2-opt` __________

To illustrate the effectivenes of the speed-up techniques, we empirically evaluated three variants of `2-opt`: a straight-forward implementation that in each search steps evaluates every possible 2-exchange move (std); a fixed radius near neighbour search that uses candidate lists of unbounded length (fr+cl); and a fixed radius near neighbour search that uses candidate lists of unbounded length as well as DLBs (fr+cl+dlb). For all variants, the search process was initialised at a random permutation of the vertices and it was terminated when no improving search step could be found within a given iteration.

These algorithms were run 1000 times on each of a number of standard benchmark instances from TSPLIB; the machine used is a Pentium III 500MHz with 128 MB RAM running Suse Linux 7.3. (The `2-opt` implementation used for these experiments is available from the SLS webpage.)

The results reported in Table 8.1 show that the speed-up techniques achieve substantial decreases in run-time; furthermore, the speed-up strongly increases with instance size. As noted before, candidate lists also result in a significant improvement in the solution quality obtained by `2-opt`; the additional use of DLBs diminishes this effect only slightly.

Using bounded length candidate lists resulted in very similar results for most instances (not shown here); only on the pathologically clustered instance `fl1577` the solution quality decreased to an average of almost 60% above the optimum, while the computation was reduced by about 10%.

`3-opt` achieves much better quality solutions than the `2-opt` variants at the cost of substantially higher computations times; this is illustrated by the results shown for `3-opt` with a fixed radius search using candidate lists and DLBs in the last column of Table 8.1. Interestingly, using bounded-length candidate lists leads to significant speedups for `3-opt`: Using candidate lists of length 40, the computation time of our `3-opt` implementation is reduced by a factor between five to seven, depending on the particular instance.

The Lin-Kernighan (LK) Algorithm

Empirical evidence suggests that iterative improvement algorithms based on k-exchange neighbourhoods with $k > 3$ return better tours, but the computation times required for searching these large neighbourhoods render this approach ineffective. Variable-depth search algorithms overcome this problem by partially exploring larger neighbourhoods (see also Chapter 2).

The best-known variable-depth search method for the TSP is the Lin-Kernighan algorithm (LK) described in Chapter 2, Section 2.1 (page **??***ff.*); it is an iterative improvement method that uses complex search steps obtained by iteratively concatenating a variable number of elementary 1-exchange moves. In each complex step, a set of edges $X = \{x_1, \ldots, x_r\}$ is deleted from the current tour, and another set of edges $Y = \{y_1, \ldots, y_r\}$ is added to it. The number of edges that are exchanged, r, is determined dynamically and can vary for each complex search step. (This is explained in more detail below.)

The two sets X and Y are constructed iteratively, element by element, such that edges x_i and y_i as well as y_i and x_{i+1} must share an endpoint, respectively; a complex step that satisfies this criterion is called *sequential*. Based on this criterion, the edges in X and Y can be represented as $x_i = (v_{2i-1}, v_{2i})$ and $y_i = (v_{2i}, v_{2i+1})$, respectively. Furthermore, at any point during the iterative construction of a complex step, *i.e.*, for any $X = \{x_1, \ldots, x_i\}$ and $Y = \{y_1, \ldots, y_i\}$, there needs to be an alternate edge y'_i such that the complex step defined by $X = \{x_1, \ldots, x_i\}$ and $Y' = \{y_1, \ldots, y'_i\}$ applied to the current tour yields a valid tour, (*i.e.*, a Hamiltonian cycle in the given graph G); there is only one exception to this rule for the case $i = 2$, which is treated in a special way [Lin and Kernighan, 1973].

To bound the length of the search for an improving complex step, the sets X and Y are required to be disjoint in the original Lin-Kernighan algorithm; this means that an edge that has been selected for removal cannot be added back later in the same complex search step and vice versa. Finally, complex steps are only executed if they result in a net improvement of the current tour.

The LK algorithm initialises the search process at a randomly chosen Hamiltonian cycle (*i.e.*, vertex permutation) of the given graph G. The search for each improving (complex) LK step starts with selecting a vertex v_1; next, an edge $x_1 = (v_1, v_2)$ is selected for removal, then an edge $y_1 = (v_2, v_3)$ is chosen to be added, *etc.* At each stage of this construction process, the cost $w(p_i)$ of the tour p_i obtained by applying the constructive search step defined by $X = \{x_1, \ldots, x_i\}$ and $Y' = \{y_1, \ldots, y'_i\}$ (as defined above) is computed as well as $g_i = \sum_{j=1}^{i} w(y_i) - w(x_i)$, the total gain for $X = \{x_1, \ldots, x_i\}$ and $Y = \{y_1, \ldots, y_i\}$. The construction process is terminated whenever the total gain g_i is smaller than $w(p) - w(p_{i^*})$, where p is the current tour and p_{i^*} is the best tour encountered during the construction, *i.e.*, $i^* = argmin\{i \mid w(p_i)\}$. At this point, if the complex step corresponding to $X = \{x_1, \ldots, x_{i^*}\}$ and $Y' = \{y_1, \ldots, y'_{i^*}\}$ leads to an improvement in solution quality, this step is executed and p_{i^*} becomes the current tour.

A limited amount of backtracking is allowed if a sequence of elementary moves does not yield an improved tour. In the LK, backtracking is triggered, when during the construction of an LK step no improving complex search step could be found and it is only applied at the first two levels, *i.e.*, for the choices of x_1, y_1, x_2, and y_2. During backtracking alternatives for edge y_2 are considered in order of increasing (or equal) weight $w(y_2)$. If this is unsuccessful, the alternate choice for x_2 is considered; since this leads to a temporary violation of the sequentiality criterion (see above), it needs to be handled in a specific way. If none of these alternatives for x_2 and y_2 can be extended into an improving complex step, backtracking is applied to the choice of y_1. When all alternatives for y_1 are exhausted without finding an improving complex step, the other edge incident to the starting vertex v_1 is considered as a choice for x_1. Only after all these attempts at finding an improving step by a search centred at vertex v_1 have failed, an alternate choice for v_1 is considered. This backtracking mechanism ensures that all 2- and 3-exchange moves are checked when searching for improving search steps; consequently, the tours obtained by LK (when run to completion) are locally optimal w.r.t. to the 2- and 3-exchange neighbourhoods.

In addition to the complex LK steps, Lin and Kernighan also proposed to consider some specially structured, non-sequential 4-exchange moves as candidates for improving search steps. (An example of a non-sequential 4-exchange move is the double-bridge move which is illustrated in Figure 2.7 on page 79.) However, Lin and Kernighan noted that the improvement obtained by the additional check of these moves depends strongly on the particular instance.

The LK algorithm uses several techniques for pruning the search. Firstly, the search for edges (v, v') to be added to Y is limited to the five cheapest edges incident to vertex v. Secondly, for $k \geq 4$ no edge in the current tour can be removed if it was contained in a collection previously found high quality tours. Furthermore, several mechanisms are provided for guiding the search. These include a rule that the edges to be added to Y are chosen such that $w(x_{i+1}) - w(y_i)$ is maximised (a limited form of lookahead) and a preference for the more expensive of two alternative edges in the context of choosing edge x_4, one of the edges that is removed from the current tour.

Lin and Kernighan applied LK to various TSP instances ranging from 20 to 110 vertices. For all of these instances, LK found optimal solutions; however, the success probability, *i.e.*, the probability that one run of LK finds an optimal solution, dropped from 1 for small instances with approx. 20 cities to 0.2–0.3 for instances with around 100 cities.

Variants of the LK Algorithm

The details of the original LK algorithm can be varied in many ways, and the design choices made by Lin and Kernighan do not necessarily lead to optimal performance. These design choices include the depth and the width of backtracking, the rules used for guiding the search (look-ahead *etc.*), the use of a bound on the length of complex LK steps, and the choice of 2-exchange moves as elementary search steps. Additional room for variation exists w.r.t. algorithmic details that are not specific to LK, such as the type and length of neighbourhood list, or the search intialisation procedure.

Some of these design choices are realised in the four well-known LK variants by Johnson and McGeoch [Johnson and McGeoch, 1997; 2002], Applegate *et al.* [Applegate *et al.*, 1999a], Neto [Neto, 1999], and Helsgaun [Helsgaun, 2000]. For a detailed discussion of these LK algorithms and their performance we refer to the original papers.

A particularly noteworthy LK variant is Helsgaun's LK (HLK), which differs from the original LK algorithm in several key features and typically performs substantially better. In HLK, the complex moves correspond to sequences of sequential 5-exchange moves; these are iteratively built using candidate lists based on α-values (see page 317). If at any point during the construction of a complex step a tour improvement can be achieved, the corresponding search step is executed immediately. In some sense this corresponds to a first-improvement search mechanism within the construction sequence for a single complex search step, while the original LK algorithm uses a best-improvement strategy in this context. Finally, HLK uses only backtracking on the choice of the first edge, x_1, to be removed from the current tour.

The performance of LK algorithms varies substantially between different variants and depends strongly on the efficiency of the respective implementation. The LK algorithm by Johnson and McGeoch (JM-LK) is an example of a very efficient implementation of an LK variant that differs only in few (high-level) details from the original algorithm by Lin and Kernighan. On RUE, RCE and TSPLIB instances, JM-LK typically obtains tours within 1% to 2.5% from the optimum (see [Johnson and McGeoch, 1997; 2002] and [Johnson *et al.*, 2002a]) in relatively short computation times; for example, on 100,000 vertex RUE instances JM-LK takes about 23 seconds on a Compac ES40 machine with a 500 MHz Alpha processor and 2GB RAM, however, the computation times increase for the same size RCE instances to about 173 seconds, which is due to the appearance of clusters in that latter type of instances. HLK finds even higher-quality tours: For RUE instances of 1,000 and 3,162 vertices it was shown to find tours within 0.03% to 0.5% from the optimum, and on TSPLIB instances with $n > 1,000$ it reaches solution qualities within less than 1% of the optimum, except for three highly clustered instances [Johnson *et al.*, 2002a]. However, this comes at the price of substantially higher run-times than, for example JM-LK, especially for large instances.

In Depth: Efficiently Implementing SLS Algorithms for TSP

In order to obtain the performance results reported in most empirical studies on SLS algorithms for the TSP, efficient implementations are required that make use of fairly sophisticated data structures. This is particularly true for state-of-the-art LK variants. In general, data structures used within SLS algorithms for the TSP need to support the following operations:

(i) determine the successor and the predecessor of a vertex within a given tour;

(ii) check whether a vertex v_k is visited between vertices v_i and v_j for a given tour orientation; and

(iii) execute a k-exchange move, which includes swaps and inversions of tour segments.

For TSP instances between 1,000 and about 10,000 vertices, the standard array representation for tours appears to be most efficient [Applegate *et al.*, 1999a; M.L. Fredman and Ostheimer, 1995]. It uses two arrays for representing a cyclic path $(v_{\pi(1)}, \ldots, v_{\pi(n)}, v_{\pi(1)})$ in the given graph G to represent the permutation of vertex indices $\pi = (\pi(1), \ldots, \pi(n))$ and its inverse, $\rho = (\rho(i), \ldots, \rho(n))$, where $\rho(n)$ is the position of vertex index i in π. Clearly, the predecessor, successor, and between queries can be answered in constant time. The time-complexity of the move operation, however, has been empirically determined as $\mathcal{O}(n^{0.7})$ (where n is the number of vertices in the given graph G); this operation is therefore a bottleneck for large instances, and more advanced data structures are required to reduce its time complexity.

One widely used alternative to the array representation is based on two-level trees [Chrobak *et al.*, 1990; M.L. Fredman and Ostheimer, 1995]. In this representation, a tour is divided into roughly $\sqrt{n}$ segments of length between $\sqrt{n/2}$ and $2\sqrt{n}$ each; these segments are represented by vertices at the first level of a tree whose root corresponds to the entire tour, while the leaves are the vertices of G. (For details on the implementation of this data structures we refer to [M.L. Fredman and Ostheimer, 1995; Applegate *et al.*, 1999a].) When using two-level trees, the successor and predecessor of a vertex can be determined in constant time and the same holds for answering between queries; the respective constants, however, are slightly larger than for the array representation. The worst-case complexity of the move operation, on the other hand, is only $\mathcal{O}(\sqrt{n})$. Based on extensive computational experiments, Fredman *et al.* [M.L. Fredman and Ostheimer, 1995] recommend the use of the two-level tree representation when solving TSP instances with up to a few million vertices. For even larger instances, they recommend to use a tour representation based on splay-trees [Sleator and Tarjan, 1985], which allow each operation to be performed in $\mathcal{O}(\log n)$ in the worst case.

It should be noted that LK algorithms are not easy to implement efficiently. Neto estimated that the development of a high performing LK implementation that uses most of the techniques described here requires around eight man-months [Neto, 1999]; this estimate has been confirmed by other researchers [Merz, 2002]. Fortunately, at least three very efficient implementations of LK variants are publically available. These are the LK implementation by Applegate, Bixby, Chvatál, and Cook, which is a part of the Concorde library accessible at http://www.math.princeton.edu/tsp/concorde.html, Helsgaun's LK variant, which can be downloaded from http://www.dat.ruc.dk/keld/research/LKH, and Neto's LK implementation, which is accessible at http://www.cs.toronto.edu/ neto/research/lk/index.html.

Local Search for the asymmetric TSP

ATSP algorithms are generally much less studied than algorithms for symmetric TSP instances; in particular, this is true for construction heuristics as well as for "simple" local search methods. Although most construction heuristics are directly applicable to the ATSP, few computational results are available [Johnson *et al.*, 2002b]. Empirical results show that for the ATSP, different from what is observed on symmetric TSP instances, the Nearest-Neighbour Heuristic typically performs much better than the Greedy Heuristic. However, for many classes of ATSP instances, even better results (in terms of solution quality) are obtained by construction heuristics that are based on the Assignment Problem lower bound for the ATSP; these constructive search methods obtain a tour by iteratively merging a set of vertex-disjoint simple directed cycles forming a minim cost vertex cover of the given graph G that is obtained as a side product of the computation of the AP lower bound. An early heuristic for the merging step by Karp [Karp, 1979]; a variant of this approach has been recently proposed by Glover *et al.* [Glover *et al.*, 2001].

When applying iterative improvement algorithms to the ATSP, a slight complication arises from the fact that sub-tour reversals lead to changes in solution quality. While 2-exchange moves always involve sub-tour reversals, there are 3-exchange moves that preserve the direction of all partial tours. The iterative improvement methods based on these moves are called `reduced 3-opt` algorithms; these are amongst the simplest iterative improvement algorithms for the ATSP. The speed-up techniques described above for symmetric TSP algorithms can be directly applied to `reduced 3-opt`.

A variable-depth search algorithm for the ATSP has been developped by Kanellakis and Papadimitriou (KP) [Kanellakis and Papadimitriou, 1980]. The KP algorithm can be seen an adaptation of the `LK` algorithm to the ATSP case; it makes use of double-bridge moves, a special type of non-sequential 4-exchange moves. An implementation of this method by Cirasella *et al.* [Cirasella *et al.*, 2001] was shown to yield significantly better solution qualities than `reduced 3-opt`, although at the cost of substantially increased run-times [Johnson *et al.*, 2002b].

8.3 Iterated Local Search Algorithms for the TSP

Iterated Local Search (ILS), as introduced in Chapter 2, Section 2.3, offers a straight-forward, yet flexible way of extending simple local search algorithms (see also the algorithm outline on page 2.6 and the GLSM model on page 3.16). Some of the hybrid SLS algorithms thus obtained are amongst the best performing TSP algorithms currently known.

Iterated Descent

Historically, the *Iterated Descent* algorithm by Baum [Baum, 1986a; 1986b] was the first ILS algorithm for the TSP. Within this algorithm, several different first-improvement methods were used as the underlying local search procedure, including `2-opt` local search, a limited form of `3-opt` local search that examines only a part of the 3-exchange neighbourhood, and a first improvement search based on a 2-exchange neighbourhood on vertices, under which two candidate solutions are direct neighbours if and only if the corresponding vertex permutations differ in exactly two positions. The perturbation phase of Iterated Descent consists of a random 2-exchange step, and its acceptance criterion always selects the candidate solution with the better solution quality.

From today's perspective, the performance of these first ILS algorithms is not impressive; however, improved performance when compared to a pure `2-opt` or `3-opt` local search were obtained. Most likely, the most substantial weakness of Iterated Descent is its perturbation mechanism. It is also now known that the underlying `2-opt` and `3-opt` local search procedures used in Iterated Descent perform poorly on the RDM instances used in Baum's empirical evaluation.

Large-Step Markov Chains (LSMC)

The *Large-Step Markov Chains* (LSMC) algorithm by Martin, Otto, and Felten is the first well performing ILS algorithm for the TSP [Martin *et al.*, 1991; 1992]. The name of this approach reflects the fact that the behaviour of LSMC (and that of many other ILS algorithms) can be modelled as a Markov chain on the locally minimal candidate solutions obtained at the end of each local search phase, where the segment of the search trajectory between any two such subsequent local minima corresponds to a "large step".

Probably the most important contribution of the LSMC approach is the introduction of a particular 4–exchange step, the so-called double-bridge move, for the perturbation phase. A double-bridge move first removes four edges from the tour, resulting in a decomposition into four segments A, B, C, D. Then, these segments are reconnected in the order A, D, C, B by adding four new edges (for a graphical illustration, see Figure 2.7 on page 79). In LSMC, only double-bridge moves are considered in which the combined weight of the four new edges is shorter than a constant k times the average edge weight in the current locally optimal candidate solution. Originally, a value of $k = 10$ was used, but experimental results suggest that the performance of the algorithm is not very sensitive to the value of k, as long as it is not too small [Martin *et al.*, 1992].

As the underlying local search procedure, LSMC initially used a `3-opt` first-improvement search, which was later replaced by the more powerful `LK` algorithm. By using several speed-up techniques, each step of the `3-opt` local search procedure is performed in sub-quadratic time (w.r.t. the number of vertices in the given problem instance) [Martin *et al.*, 1991]. These speed-up techniques include (i) a type of fixed radius search, which uses the minimal and the maximal weight edge in the current candidate solution for pruning the search for improving 3-exchange steps; (ii) a so-called *change list*, an idea that is equivalent to the use of don't look bits; and (iii) a hash table for storing `3-opt` candidate solutions, which is consulted for checking whether a tour was previoulsy identified to be locally optimal.

The acceptance criterion used in LSMC is taken from Simulated Annealing: If the candidate solution that is returned after a perturbation and the subsequent local search, s'', improves over the incumbent solution, s, s'' is always accepted; otherwise, it is accepted with a probability $\exp(f(s) - f(s''))/T)$, where T is a parameter called temperature, which may change at run time. Later, it was found that for several TSP instances best performance is obtained by accepting only better quality candidate solutions s''. This zero-temperature LSMC algorithm is also known as *Chained Local Optimisation* (CLO) [Martin and Otto, 1996].

LSMC with a subsidiary `3-opt` local search procedure was shown to solve small random Euclidean TSP instances with up to 200 cities in less than one hour on a SUN SPARC 1 workstation (a very slow machine compared to nowadays PCs). Relatively good performance was also observed on several TSPLIB instances. For example, LSMC could find an optimal solution of instance `lin318` in about four CPU hours on the SUN SPARC 1; by using an `LK` local search algorithm as the subsidiary local search procedure, the time required for solving this instance optimally was reduced by a factor of about four. In this particular case, however, it was shown to be essential to use non–zero temperatures in the LSMC acceptance criterion. LSMC with the `LK` subsidiary local search procedure also solved several larger TSPLIB instances optimally, including `att532` and `rat783`.

Iterated Lin-Kernighan

Following preprints of Martin, Otto and Felten's work on LSMC, Johnson developed his Iterated Lin-Kernighan (ILK) algorithm [Johnson, 1990; Johnson and McGeoch, 1997]. There are a few differences between the details of the LSMC algorithm and Johnson's ILK. Firstly, the acceptance criterion used in ILK always selects the better of the two locally optimal candidate solutions. Secondly, the perturbation phase does not make use of the limiting condition on the edge weight of a double-bridge move imposed in the LSMC approach, but it applies random double-bridge moves instead, *i.e.*, the four cut-points are randomly chosen according to a uniform distribution. Thirdly, the local search is initialised with a randomised greedy construction heuristic; in the randomized version instead of selecting deterministically the least weight feasible edge, among the two cheapest edges, the one with less weight is chosen with a probability of 2/3. Early results for ILK were quite promising: Applied to sample TSPLIB instances with 318 to 2392 vertices, optimal solutions were obtained (for the 2392 vertex city in about 55 hours on a Sequent computer).

The ILK algorithm was further fine-tuned and extensively tested in a 1997 overview article on the state-of-the-art in approximate TSP algorithms [Johnson and McGeoch, 1997]. The main differences between the 1997 variant and the earlier ILK implementation appears to be the exploitation of don't look bits after the double-bridge move and the use of a bound on the depth of the LK search. This "production–mode ILK" was shown to achieve almost optimal solutions on a variety of random Euclidean instances and a few TSPLIB instances. Major difficulties in finding solutions within less than one percent of the optimum solution quality were only reported on the strongly clustered TSPLIB instance fl3795. Running times were modest, for example on 10,000 vertex RUE instances ILK took approx. 1,570 seconds on a SGI Challenge 196 MHz machine [Johnson and McGeoch, 1997].

Based on its excellent performance, ILK has also been used for the empirical analysis of the quality of the Held-Karp lower bound for several classes of TSP instances [Johnson *et al.*, 1996]. In this context, ILK was run for a large number of iterations to obtain upper bound estimates of the optimal solution quality for TSP instances ranging in size from 100 to 100,000 vertices.

Chained Lin-Kernighan

Like ILK, the Chained Lin-Kernighan (CLK) algorithm, developped by Applegate, Bixby, Chvatál, and Cook [Applegate *et al.*, 1999a], uses the LK algorithm as its subsidiary local search procedure. CLK differs from ILK in various implementation aspects of the LK local search, including its use of smaller candidate sets (by default it uses quadrant nearest neighbour sets of size 12); it also uses a different perturbation mechanism that affects only a locally restricted part of a candidate solution, and initialises the search using the Quick-Borůvka construction heuristic (see Section 8.2). For details on the implementation of the LK local search used in CLK we refer to [Applegate *et al.*, 1999a] and to the original CLK code that is available at www.math.princeton.edu/tsp. In the following, we focus on some of the other algorithmic features of CLK, particularly the perturbation mechanism.

The standard CLK algorithm uses so-called *geometric double-bridge moves* as perturbation steps; these are based on the following method for selecting the four edges to be removed from the current candidate tour s. In a first step, a set U of $\min\{0.001 \cdot n, 10\}$ vertices are randomly sampled from the given graph $G = (V, E, w)$. Then, among the edges (u, u') contained in s with $u \in U$, the one with the maximal difference $w((u, u')) - w((u, u^*))$, where u^* is the nearest neighbour of u (*i.e.* the vertex in V that minimises $w((u, u^*))$), is removed from s. In a second step, the other three edges to be removed from s are then chosen uniformly at random from the edges connecting vertex u with its k nearest neighbours in V. The four edges thus selected uniquely determine the double-bridge move used for perturbation. The value of k controls the locality of the perturbation: For small k, a geometric double bridge move results in a localised perturbation that only affects edges close to one specific vertex, while for large k, less localised perturbations are obtained.

As said in Section 8.2, one reasonable strategy is to set only the DLBs of vertices to zero that are incident to edges changed by the perturbation. In fact, such a strategy is followed in the LSMC and the ILK algorithm. Then, only these vertices are considered as start vertices of the search for an improving move. In CLK, several other resetting strategies for the DLBs have been studied, including randomised schemes as well as mechanisms which additionally reset the DLBs of vertices that are at most 10 away from the end-points of the changed edges in the current tour as well as the end-points' neighbour sets. Experimental results suggest that this latter approach leads to the best performance of CLK. (Similar observations were also made independently by Stützle in the context of Iterated `2-opt` and `3-opt` algorithms for the TSP [Stützle, 1998].)

The original CLK and a CLK variant by Applegate, Cook, and Rohe (CLK-ACR) that uses a different mechanism for selecting the double bridge moves used as perturbation steps and slightly different implementation choices for the LK were tested on a large number of TSP instances [Applegate *et al.*, 1999a; 1999b]. A comparison of the two CLK algorithms in the context of the 8th DIMACS Implementation Challenge on the TSP revealed better performance for CLK-ACR on most of the instances tested (see [Johnson and McGeoch, 2002] and the challenge web-pages at http://www.research.att.com/ dsj/chtsp). When compared to ILK, no fully conclusive results can be drawn from the Challenge results: When running both algorithms for n iterations, on most instances ILK returns better quality solutions than CLK-ACR, however at run-times that are several times larger than those of CLK-ACR (for RUE instances a factor between 2 and 5; for TSPLIB and RCE the run-time penalty of ILK is several times larger). The CLK-ACR code is also capable of handling extremely large TSP instances and has been applied to instances up to 25 million vertices, where it reached a solution within 1% of the estimated optimum in 24 CPU hours on a IBM RS6000, Model 43-P 260 workstation with 4 GB RAM and a solution within an estimated 0.3% of the optimum in 8 CPU days.

Iterated Helsgaun (IHLK)

Given the excellent performance of HLK, Helsgaun's variant of the LK algorithm (see also Section 8.2), using this local search procedure as the core of an ILS algorithm is a fairly obvious idea. This leads to the Iterated Helsgaun algorithm (IHLK), which is one of the best SLS algorithms for TSP currently known, particularly w.r.t. to the solution quality obtained for large run-times [Helsgaun, 2000; Johnson and McGeoch, 2002].

Since HLK potentially uses double-bridge moves within its search, a perturbation mechanism based on this move can be expected to be insufficient. Instead, the perturbation mechanism used in IHLK is based on a construction heuristic that is strongly biased by the incumbent candidate solution. This constructive search procedure, shown in Figure 8.4, iteratively builds a candidate solution for a given TSP instance in a manner similar to the nearest neighbour construction heuristic: Starting from a randomly selected vertex, in each step the partial tour is extended with a vertex v_j that is not contained in the current incomplete path p and that is connected by an edge to v_i, the current endpoint of p.

In this process, a vertex v_j is always chosen for extending the partial tour p if (i) the edge (v_i, v_j) is contained in the incumbent candidate solution, (ii)

```
procedure $ConstrHLK(\pi')$
    input problem instance $\pi' \in \Pi'$, candidate solution $\hat{s}$
    output candidate solution $s \in S(\pi')$
    $v_i := chooseRandomVertex$
    $p := []$
    while $p$ is not a tour do
        $\mathcal{C} := \{v_j \mid (v_i, v_j)$ is a candidate edge $\wedge \alpha((v_i, v_j)) = 0 \wedge (v_i, v_j) \in \hat{s}\}$
        if $\mathcal{C} = \emptyset$
            $\mathcal{C} := \{v_j \mid (v_i, v_j)$ is a candidate edge$\}$
        end
        if $\mathcal{C} = \emptyset$
            $\mathcal{C} := \{v_j \mid v_j$ not chosen yet$\}$
        end
        $v_j$ := chooseRandomVertex($\mathcal{C}$);
        $p := [p|v_j]$;
        $v_i := v_j$;
    end
    return $\hat{s}$
end $ConstrHLK$
```

Figure 8.4: The construction procedure used in the perturbation phase of the IHLK algorithm. (For details, see text.)

(v_i, v_j) is contained in the candidate list for vertex v_i, and (iii) $\alpha((v_i, v_j)) = 0$ (see Section 8.2 on page **??** for the definition of $\alpha((v, v')))$. If at any stage of the search process no such vertex exists, a vertex v_j contained in v_i's candidate list is chosen, if feasible. This is done by traversing the candidate list until a vertex is found that is not contained in the current partial tour p. If no such vertex can be found, a list of all vertices is traversed until a vertex v_j is found that is not contained in p.

The acceptance criterion used in IHLK only accepts better tours than the best seen so far as the incumbent solution. In addition to standard speedup techniques such as don't look bits, IHLK uses hashing techniques (originally described by Lin and Kernighan [Lin and Kernighan, 1973]) to check whether a solution was already earlier found to be a local optimum and to, thus, reduce the check-out time. Further details of IHLK can also be directly checked at the source code, which is available from http://www.dat.ruc.dk/keld/research/LKH.

IHLK finds optimal solutions for many TSPLIB instances with up to a few thousands of vertices within relatively short CPU times of some minutes on modern PCs as of 2002. Longer runs of IHLK found new best upper bounds on the largest, still unsolved TSPLIB instances and for the World TSP instance mentioned in Section 8.1; in the latter case, lower bound computations have shown that the solution found by IHLK deviates at most 0.17% from the optimal solution quality (see www.math.princeton.edu/tsp/world/index.html). However, the running times of IHLK increase very strongly with instance size.

Other Perturbation Mechanisms

The perturbation mechanism and its relation to the subsidiary local search procedure can have a significant impact on the performance of an ILS algorithm; consequently, a wide range of perturbation mechanisms have been proposed and studied in the context of ILS algorithms for TSP.

Hong, Kahng, and Moon studied ILS algorithms based on `2-opt`, `3-opt`, and `LK` local search that use single random k-exchange steps with fixed values of k between 2 and 50 for perturbation [Hong *et al.*, 1997]. More specifically, the k-exchange moves they use are determined by disconnecting the current tour at k randomly chosen positions. The resulting sub-tours are then reconnected according to a specific, fixed template; only for $k = 3$ one of the several possible ways of reconnecting the sub-tours is chosen randomly and for $k = 4$ a random double-bridge move is applied. The resulting algorithms were empirically evaluated on TSPLIB instances `lin318` and `att532` as well as on a RDM instance with 800 vertices. Their results suggest that on the TSPLIB instancs, using their random k-exchange perturbation with $k > 4$ results in better solution qualities after a fixed number of local search steps than a perturbation based on a random double-bridge move. On the RDM instance, the best results were obtained for $k = 3$, independent of the subsidiary local search procedure used. It is not clear whether these results also hold when measuring time complexity in terms of CPU time rather than local search steps, or whether these observations generalise to other TSP instances.

Perturbations can be more complex than simple (random) k-exchange steps. One example for a complex perturbation is the mechanism proposed by Codenotti *et al.* which involves the modification of the instance data [Codenotti *et al.*, 1996]. Their perturbation procedure works as follows: First, the given a geometric TSP instance G is slightly modified by introducing small perturbations in the edge weights. (For Euclidean TSP instances this is achieved by changing the coordinates of the vertices.) The current locally optimal tour s is not necessarily a local minimum w.r.t. to this modified instance G'. Thus, the subsidiary local search procedure is run on G' until a local minimum s' is found. At this point the modified instance G' is discarded and the candidate solution s' is returned as the overall result of the perturbation, which provides the starting point of the subsequent local search phase. Codenotti et al. gave some indication that this perturbation despite its relatively high time-complexity can achieve slightly better performance than a more standard ILK implementation using double-bridge perturbations. However, state-of-the-art ILS algorithms for the TSP, such as CLK typically achieve much better performance [Applegate *et al.*, 1999b]. (It should be noted that this general perturbation approach has been proposed and successfully applied in the context of a very early ILS algorithm for a location problem [Baxter, 1981].)

Another interesting perturbation mechanism is the *genetic transformation* (GT) introduced by Katayama and Narisha, which introduces ideas from evolutionary algorithms into ILS [Katayama and Narihisa, 1999] (see also Section 2.3). The GT procedure is based on the intuition that sub-tours that are common between the best tour found so far, $\hat{s}$, and the current locally optimal tour, t, should be preserved, and works as follows: First, all common sub-tours between $\hat{s}$ and t are determined; this can be achieved in time $\mathcal{O}(n)$, where n is the number of vertices in the given TSP instance. Then, the perturbation result is obtained by connecting these sub-tours using a procedure that is closely related to the nearest neighbour construction heuristic. The overall *Genetic Iterated Local Search* (GILS) algorithm for the TSP is outlined in Figure 8.5, where the function GT implements the GT perturbation mechanism. Computational experiments with an iterated LK algorithm that uses the GT perturbation instead of the standard double-bridge move have shown that the approach is very effective [Katayama and Narihisa, 1999].

```
procedure $GILS(\pi')$
    input problem instance $\pi' \in \Pi'$, objective function $f(\pi)$
    output solution $\hat{s} \in S(\pi')$ or $\emptyset$
    $s := init(\pi'), t := init(\pi')$
    $s := localSearch(\pi', s), t := localSearch(\pi', t)$
    if $(f(s) < f(t))$
        $\hat{s} := s$
    else
        $\hat{s} := t$
    end
    while not $terminate(\pi', \hat{s})$ do
        $t' := GT(\pi', \hat{s}, t)$
        $t'' := localSearch(\pi', t')$
        if $(f(t'') < f(\hat{s}))$
            $\hat{s} := t''$
        end
        $t = t''$
    end
    return $\hat{s}$
end $GILS$
```

Figure 8.5: Algorithmic outline of the Genetic Iterated Local Search (GILS) for the TSP. (For details, see text.)

Other Acceptance Criteria

While various possibilities for the local search or the perturbation step are well examined, the acceptance criterion was largely neglected, although it is known that it can have a strong impact on the balance between diversification and intensification of the search. In fact, most ILS implementations accept only better quality solutions. However, some few exceptions exist. A first one is the LSMC algorithms of Martin, Otto, and Felten, where a simulated annealing (SA) type acceptance criterion is used (see Equation 2.1 on page 64 for a possible SA acceptance criterion). Using non-zero temperatures in the acceptance criterion actually improved the performance of their algorithms on a few instances. Simulated annealing type acceptance criteria were also examined by Rohe [Rohe, 1997]. For one TSPLIB instance (`d18512`) he found that with a carefully tuned annealing schedule for long computation times slight improvements over a standard ILK were possible.

Hong, Kahng and Moon studied variants of ILS algorithms using what they called an *hierarchical* LSMC algorithm. The hierarchical LSMC accepts by default only better quality solutions. However, if it is deemed that the ILS is stuck (this is the case after $2n$ iterations without accepting an improved solution when using a `2-opt` or a `3-opt` local search, after 100 iterations when using `LK` local search), they set the temperature in the SA acceptance criterion to $f(s_i)/200$ for 100, *i.e.*, a deterioration of the tour length by 0.5 percent is accepted with a probability of $1/e$; after this 100 iterations, again only better quality solutions are accepted. This diversification is invoked, if for i_r no improved solution is found by the ILS; when using `2-opt` or `3-opt` local search they set $i_r = 2n$, when applying `LK`, they set $i_r = 100$. Some limited experiments with that approach on three TSP instances and a `2-opt`, `3-opt`, and a `LK` local search showed that on the two instances `lin318` and `att532` improvements were possible; however, from the paper it is not clear whether these improvements are statistically significant.

A more detailed study of different acceptance criteria was undertaken by Stützle [Stützle, 1998; Stützle and Hoos, 2001]. He compared the standard better acceptance criterion for ILS with two acceptance criteria, which introduce diversification features in the ILS. The first is a simple *soft restart criterion* (see also Section 4.4), which restarts ILS from a new initial solution, if no improved solution was found for i_r iterations. The restart of the algorithm can easily be modelled by the acceptance criterion *Restart*$(s, s'',$ *history*$)$, where the *history* component captures, e.g., the very simple use of search history underlying the soft restart criterion. Let i_{last} be the most recent iteration in which a better solution has been found since the last restart and i be the iteration counter. Then *Restart*$(s, s'',$ *history*$)$ returns s'', if $w(s'') < w(s)$, s_0, if $w(s'') > w(s)$ and $i - i_{last} > i_r$, where s_0 is some new initial solution, or s, otherwise.

Fitness-distance-based Diversification

A disadvantage of restarting ILS from new initial solutions is that previously obtained high quality solutions are lost. Additionally, ILS algorithms need some initial, instance dependent time t_{init} before effectively very high quality solutions can be identified, a time which is "wasted" with each restart. To avoid these disadvantages, a less radical and more directed diversification is used. The basic idea of the method is to attempt to *find a good quality solution beyond a certain minimal distance from the current search point without using restart* and it is implemented as follows. Let s_c^* be the current candidate solution from which the search should escape, and let $d(s, s')$ be the distance between two tours s and s' measured as the number of edges that differ between s and s'. Then the following steps are repeated until we obtain a solution beyond a minimal distance $d_{\min}$ from s_c^*:

(1) Generate p copies of s_c^*.

(2) To each of the p solutions, apply Perturbation followed by LocalSearch.

(3) Choose the best q solutions, $1 \leq q \leq p$, as candidate solutions.

(4) Let s be the candidate solution with maximal distance to s_c^*.
If $d(s, s_c^*) \leq d_{\min}$, then goto (2); otherwise return s.

The goal of step (3) is to obtain good quality solutions, while the goal of step (4) is to choose a candidate solution at a maximal distance from the current solution. The steps (2) to (4) are then iterated until a solution s^* is found for which the requirement in step (4) is satisfied. To avoid getting stuck in infinite loops, this iterative process is stopped if after a maximal number of iterations no solution s^* beyond $d_{\min}$ is found. In [Stützle and Hoos, 2001], the parameter $d_{\min}$ is estimated by first computing the average distance Δ_{avg} between an number of local optima w.r.t. the local search algorithm applied in the ILS. Then $d_{\min}$ is set alternatingly to $0.25\Delta_{avg}$ and $0.5\Delta_{avg}$; s_c^* is always taken as the best solution found since the start of the algorithm.

Example 8.2: Effectiveness of acceptance criteria

We exemplify influence of acceptance criteria on the computational results when applying an iterated `3-opt` algorithm. The computational results are given in Table 8.2, more complete results including the use of `2-opt` and a `LK` local search can be found in [Stützle and Hoos, 2001]. We refer with

ILS-Descent to the ILS accepting only improved solutions, ILS-Restart to the ILS plus soft restart and ILS-FDD to the ILS plus distance-based diversification; the maximum computation times ranged from 120 seconds on the smallest instance `d198` to 7200 seconds for the largest instance `fl3795` on a Pentium II 266MHz CPU. Generally, the experimental results show that ILS-Restart and ILS-FDD can improve significantly over ILS-Descent: Both find optimal solutions much more frequently and on average achieve much better solution quality. Only on instance `rat783` the frequency of finding the optimum is lower with ILS-Restart than with ILS-Descent. In addition, the advantage of the most complex algorithm, ILS-FDD over ILS-Descent but also over ILS-Restart becomes more and more noticeable with increasing instance size.

It should be noted that the performance of ILS-FDD is particularly good on instances which are known to be difficult for several other algorithms. This is the case for `fl1577` and `fl3795` which show a pathological clustering of cities. On these instances, also ILS-restart performs surprisingly well; this is probably due to the fact that they contain deep local minima from which the standard ILS algorithm has strong difficulties to escape. Also note that on these instances it often appears preferable to run `3-opt` instead of `LK`, because the run-time of most `LK` variants increases very strongly on highly clustered instances (see also the discussion in [Johnson and McGeoch, 1997]).

Tour Merging

The tour merging approach by Applegate, Bixby, Chvátal, and Cook uses an ILS algorithm as a subroutine of a more complex, hybrid search algorithm [Applegate *et al.*, 1999a]. It is based on the observation that high quality tours typically have many edges in common; furthermore, the number of shared edges increases with solution quality, and close-to-optimal tours share many edges with optimal tours (see also Chapter 5). Hence, given a TSP instance G, the edges from a few close-to-optimal tours, which can be found using a high performance ILS algorithm, induce a subgraph G' with the following properties: (i) G' is a sparse graph with low branch-

Table 8.2: Comparison of ILS-Descent, ILS-Restart, and ILS-FDD on symmetric TSPs. For each instance (the number in the instance identifier is the problem size), we report the frequency of finding the known optimal solution (f_{opt}), the average percentage deviation from the optimum (Δ_{avg}), the average CPU-time t_{avg} to find the best solution in a run, and the maximally allowed computation time t_{max}. The algorithms were run on a 266MHz Pentium II CPU.

Instance	ILS-Descent			ILS-Restart			ILS-FDD		
	f_{opt}	Δ_{avg}	t_{avg}	f_{opt}	Δ_{avg}	t_{avg}	f_{opt}	Δ_{avg}	t_{avg}
d198	1.0	0.0	1.1	1.0	0.0	0.8	1.0	0.0	1.5
lin318	0.65	0.10	13.9	1.0	0.0	7.1	1.0	0.0	13.7
pcb442	0.56	0.12	34.9	1.0	0.0	46.5	1.0	0.0	30.8
att532	0.22	0.055	91.6	0.74	0.0096	214.6	0.96	0.002	202.3
rat783	0.71	0.029	238.8	0.51	0.018	384.9	1.0	0.0	159.5
pr1002	0.56	0.11	389.7	1.0	0.0	578.2	1.0	0.0	207.2
pcb1173	0.0	0.26	461.4	0.0	0.040	680.2	0.56	0.011	652.9
d1291	0.08	0.29	191.2	0.68	0.012	410.8	1.0	0.0	245.4
fl577	0.12	0.52	494.6	0.92	0.00008	477.6	1.0	0.0	294.1
pr2392	0.0	0.23	1538.5	0.0	0.22	2008.5	0.3	0.027	2909.1
pcb3038	0.0	0.22	3687.6	0.0	0.20	4824.3	0.0	0.099	5535.9
fl3795	0.2	0.36	3601.9	0.2	0.0035	3080.9	0.9	0.0003	3506.7

width [Robertson and Seymour, 1991] and (ii) there is a reasonably high chance that G' contains at least one optimal tour.

Based on these considerations, the tour merging procedure works as follows: Given a TSP instance $G = (V, E, w)$, first a set S of close-to-optimal tours is generated. This can be done by applying any of the very effective ILS algorithms like ILK, CLK or IHLK. From these tours, a subgraph $G' = (V, E', w')$ of G is formed, where E' is the set of all edges which are contained in at least one of the tours in S, and w' is the original weight function w restricted to those edges. In a second step, the objective is then to find an optimal tour w.r.t. the TSP instance defined by G'. This can be achieved by means of a complete TSP algorithm. Alternatively, Applegate *et al.* have proposed a dynamic programming algorithm that determines an optimal solution for a (heuristically determined) branch-width decomposition of G'.

The tour merging approach was empirically evaluated on a large number of TSPLIB instances [Applegate *et al.*, 1999a]. For medium size instances with up to 5,000 vertices, it found optimal solutions for each of the tested instances, on the largest (`fnl4461`) taking about 90,000 CPU seconds on a 500MHz Alpha 21164a processor; most of this computation time was used by the optimisation algorithm applied in the second step. Further experiments indicated the following tradeoffs between the two steps of the tour merging approach: As can be expected, given a fixed number of CLK runs for the first step, the computation time needed by the second step decreases when increasing the length of each CLK run; this is mainly caused by the fact that for increased tour quality, the graph G' contains fewer edges. Furthermore, if the length of the individual CLK runs is kept constant, the overall solution quality improves when the number CLK runs (and hence the number of tours generated) is increased. This, however, also leads to an increase in overall run-time because of the higher time-consumption in step one and due to the increased number of edges in G', which leads to higher computation times in step two.

Iterated Local Search for the Asymmetric TSP

Most of the research efforts on designing efficient ILS algorithms has focussed on the symmetric TSP and much less work has been done for the asymmetric TSP (ATSP). algorithms for the ATSP are typically very similar to ILS algorithms for symmetric TSP instances; the main difference lies in the use of an ATSP specific subsidiary local search procedures, such as `reduced 3-opt` or the Kanellakis–Papadimitriou variable depth heuristic (KP), in order to avoid the high overhead involved in computing the effects of sub-tour reversals (see Section 8.2). Since double-bridge moves do not involve any sub-tour reversals, they can be used for perturbation exactly as in ILS algorithms for symmetric TSP.

Only few ILS algorithm have been developed for the ATSP. These include iterated `reduced 3-opt` (`Ired3-opt`) algorithms by Johnson *et al.* [Johnson *et al.*, 2002b] and Stützle & Hoos [Stützle, 1998; Stützle and Hoos, 2001] as well as an iterated KP (IKP) algorithm by Johnson *et al.* [Johnson *et al.*, 2002b]. Experimental results suggest that IKP finds better quality solutions than the Johnson version of `Ired3-opt`, however at the cost of substantially higher run-times on certain classes of instances [Johnson *et al.*, 2002b]. In fact, for some types of large ATSP instances, n iterations of IKP take up to 100 times more CPU time $10n$

iterations of `Ired3-opt`. Published results for an ILS algorithms using HyperOpt local search by Burke *et al.* [Burke *et al.*, 2001] suggest that in most cases the solution qualities reached by this algorithm are worse than for `Ired3-opt` while requiring larger computation times [Burke *et al.*, 2001; Johnson *et al.*, 2002b].

Somewhat surprisingly, in a recent comparative study of several ATSP algorithms the best quality tours were obtained by first transforming ATSP instances into symmetric TSP instances (see also Section 8.1) and then to apply state-of-the-art SLS algorithms for the symmetric TSP, such as IHLK [Johnson *et al.*, 2002b]. In fact, for various classes of ATSP instances this approach produced the best solutions known todate; however, the computation times required in this context are in the range of several hours for the largest ATSP instances with 3,162 vertices (which corresponds to an symmetric TSP instance with 6,324 vertices).

8.4 Population-based SLS Algorithms for the TSP

Maintaining a population of candidate solutions within an SLS algorithm provides natural means of search diversification, which can help to avoid or overcome search stagnation and improve the robustness of SLS performance. In the following, we present several population-based SLS methods for the TSP, ranging from simple extentions of Iterated Local Search to inherently population-based approaches, such as Genetic Local Search and Ant Colony Optimisation (see also Chapter 2. Given the large number of TSP algorithms based on the latter two approaches, we focus on some prominent algorithms that illustrate the main considerations arising in the context of applying these methods to efficiently solve TSP instances.

Population-based ILS

Iterated Local Search can be easily extended into a population-based SLS method by independently applying a standard ILS algorithm to to population, *i.e.*, a set of candidate solutions, and allowing some limited interaction between the population elements. Such extensions have strong similarities to well-known population-based search metaphors such as Evolutionary Algorithms [Bäck, 1996; Mitchell, 1996] and, in particular, Genetic Local

Search (see Chapter 2, page **??***ff.*). Both approaches make use of selection mechanisms in order to focus the search on promising regions of the search space. Furthermore, the perturbation mechanism used in ILS can be seen as a (sometimes very complex) mutation operator. Different from Evolutionary Algorithms, however, population-based extensions of ILS do not use recombination mechanisms to generate new candidate solutions from two or more elements of the population.

Generally, the interaction between population elements is fairly limited in population-based ILS algorithms, which facilitates rather straightforward and efficient parallel implementations. Martin and Otto described a parallel implementation of a population-based extension of their Chained Local Optimisation (CLO) algorithm for the TSP, in which a single ILS process (which runs the CLO algorithm on an individual candidate solution) runs on each processor, and a simple selection mechanism is used to guide the search [Martin and Otto, 1996]. In particular, infrequently, *i.e.*, after around 10 to 100 CLO steps per processor, the best tour within the current population is broadcast to all processore and replaces all other population elements the solutions kept on the other processors. Martin and Otto reported that this "winner-take-all" strategy [Aldous and Vazirani, 1994] achieved good solution qualities for high run-times; however, they did not present empirical results for this population-based CLO algorithm.

Hong, Kahng, and Moon introduced a similar population-based ILS algorithm that uses a slightly more complex selection mechanism [Hong *et al.*, 1997]. After every I ILS iterations, where I is a parameter of the algorithm, one candidate solution s_j is probabilistically selected based on its solution quality; in particular, the best tour in the current population is four times as likely to be selected as the currently worst tour. Perturbation and local search are then applied to s_j, resulting in a new tour s''_j. If s''_j is better than the best tour found so far during the search, it replaces s_j, otherwise, s''_j replaces the worst tour s_k in the current population, unless s''_j has lower quality than s_k, in which case s''_j is discarded. Note that for $I = \infty$ this algorithm is equivalent to performing m independent runs of standard ILS, where m is the population size, *i.e.*, the number of candidate solutions in the population.

Similar population-based extensions were studied by Stützle, who examined three population-based ILS algorithm for the TSP with varying degrees of interaction between the members of the population [Stützle, 1998]:

```
**procedure** *ss-ILS*
    Generate initial population $p$
    $sp := init(\pi')$
    $\hat{s} := best(\pi', sp)$
    **while** (**not** $terminate(\pi', sp)$) **do**
        $s_0 := select(\pi', sp)$
        $s_0 := s$
        **for** $i = 1$ **to** $j$ **do**
            $s' := perturb(\pi', s)$
            $s'' := localSearch(\pi', s')$
            **if** $(f(s'') < f(\hat{s}))$
                $\hat{s} := s''$
            **end**
            $s := accept(\pi', s, s'')$
        **end**
        **if** $(f(s) < f(s_0)$ **then**
            $sp := replace(\pi', sp, s)$
        **end**
    **end**
    **if** $\hat{s} \in S'$ **then**
        **return** $\hat{s}$
    **else**
        **return** $\emptyset$
    **end**
**end** *ss-ILS*
```

Figure 8.6: Algorithm outline for ss-ILS. $best(\pi', sp)$ denotes the individual from population sp with the best objective function value; $select(\pi', sp)$ selects a candidate solution from sp based on its objective function value; and $replace(\pi', sp, s)$ returns the result of replacing one individual in sp with candidate solution s. (For details, see text.)

(i) A variant that does not use any interaction within the population and effectively performs a fixed number of independent ILS runs.

(ii) A variant called *Replace-Worst*, in which after every I iterations the best tour within the current population replaces the currently worst tour. This approach gradually focusses the search around the best found tours, where the parameter I controls the rate of this process.

(iii) A variant *ss-ILS*, in wich standard ILS is only applied to one tour s_0 selected from the current population sp; if after j iterations of ILS an improvement over s_0 has been achieved, this improved tour replaces one of the tours in sp (see Figure 8.6.) Different choices of the functions $select$ and $replace$ will lead to different search behaviour.

Computational experiments with the population-based ILS algorithms for the TSP by Hong *et al.* and Stützle suggest that they achieve substantial performance improvements over conventional ILS algorithms using the $better$ acceptance criterion, in particular ILS-Descent [Hong *et al.*, 1997; Stützle, 1998]. For fixed run-time, the population-based algorithms have been shown to obtained solution qualities whose average deviation from the optimum is roughly half of that obtained by ILS-Descent on a number of TSPLIB instances [Stützle, 1998]. Suprisingly, the variants without interaction within the population were often found to be as effective as variants with interaction; further experimentation is required to see whether interaction within the population is more advantageous for larger TSP instances. Furthermore, current empirical evidence suggests that when using standard ILS algorithms with more complex acceptance criteria, such as ILS-FDD (see Section 8.3, page **??**), all current population-based ILS extensions appear to be inferior in performance.

Evolutionary Algorithms for the TSP

The TSP has been the target of a large amount of research on Evolutionary Algorithms, and many EAs for the TSP have been proposed and studied in the literature. An important general issue in the design of EAs for the TSP is the representation of candidate solutions. Most commonly, tours are represented as permutation of the vertex indices. Several other representations have been studied [Homaifar *et al.*, 1993; Whitley *et al.*, 1989; Walters, 1998], but it is not clear whether any of these has particular advantages when compared to the permutation representation. Most research efforts on EAs have been focussed on the design of recombination operators [Potvin, 1996; Merz and Freisleben, 2002] and the development of hybrid EAs that include efficient subsidiary local search algorithms to improve candidate solutions – so-called Memetic Algorithms (see Chapter 2, page **??***ff.*).

If one general conclusion can be drawn from all these research efforts, then it is that Memetic Algorithms (MAs), *i.e.*, combinations of EAs with efficient subsidiary local search algorithms, are generally superior to EAs that do not use a subsidiary local search. A second general conclusion is that different types of recombination operators can result in significant performance differences; yet, in Memetic Algorithms for the TSP, the influence of the recombination operator diminishes as higher performance subsidiary local search procedures are used. One very important property of recombination operators for the TSP is its *respectfulness* [Radcliffe and Surry, 1994; Merz and Freisleben, 2002]: respectful recombination operators ensure that solution components (here: edges) that are common to all parents are present in any offspring. Intuitively, the importance of respectful recombination is tightly connected to the typical search space structure for TSP instances, which is characterised by high fitness-distance correlation values (see also Section **??** in Chapter 5).

The Memetic Algorithm by Merz and Freisleben

The memetic algorithm by Merz and Freisleben (MA-MF) is probably one of the best studied and most effective MAs for the TSP. Initially proposed in 1996 [Freisleben and Merz, 1996], MA-MF has been continuously improved through the incorporation of more efficient subsidiary local search procedures and better recombination operators, as well as through the addition of restart mechanisms [Merz and Freisleben, 1997; Merz, 2000; Merz and Freisleben, 2002]. In the following, we describe the main features of the latest and best performing MA-MF variant [Merz and Freisleben, 2002].

The initial population in MA-MF is generated by a randomised variant of the Greedy construction heuristic (see Section 8.2). This constructive search procedure first iteratively inserts $n/4$ edges which are selected as follows: after choosing uniformly at random a vertex v of the given graph G that is not contained in the current partial tour, the cheapest feasible edge incident to v is selected with probability $2/3$ and the second-cheapest feasible edge otherwise; edges are feasible, if one of their endpoints is not contained in the current partial tour. The partial tour obtained after this initial random edge selection is then completed using the standard construction mechanism of the Greedy Heuristic. As usual in MAs, the search initialisation is completed by applying the subsidiary local search procedure (here: an LK variant) to all tours in the population.

Various recombination operators are used in different variants of MA-MF; of these, a greedy recombination operator GX, which is based on ideas from the Greedy construction heuristic, achieves the best performance. The GX procedure generates one offspring from two parent tours and consists of four phases:

1. Copy some or all edges common to the two parents to the offspring. (A parameter p_e gives the fraction of common edges to be copied.)

2. Add new short edges that are not contained in any of the parents. For a node v_i one of the five nearest neighbours is chosen such that edge (v_i, v_j) is not contained in any of the parents and edge (v_i, v_j) is feasible; the number of edges to be chosen in this way is determined by a parameter p_n.

3. Copy edges from the parents, where edges are ordered according to increasing length. Only edges that do not lead to a violation of the TSP constraints are considered and edges not common in the parents may be included; the number of edges included in this way is determined by a third parameter p_c.

4. If necessary, the candidate tour is completed using a randomised greedy construction heuristic.

Experimental results show that the best performance is obtained when GX is maximally respectful, *i.e.*, for $p_e = 1$ [Merz and Freisleben, 2002]. Good settings for the parameters p_n and p_c very between TSP instances. Recombination is applied to $n/2$ pairs of tours which are chosen uniformly at random from the current population.

The mutation operator used in MA-MF is the standard double-bridge move that is also used in most ILS algorithms. The candidate solutions to which mutation is applied are chosen uniformly at random. MA-MF uses a $(\mu+\lambda)$ selection strategy for determining the new population after each generation. The new population consists of the μ best tours among the μ from the current population plus the λ new tours obtained from the application of recombination, mutation, and local search but avoiding duplicate tours. Finally, an additional restart operator is applied conditionally, to maintain a certain diversity of the population: If the average distance between the tours in the population (measured as the number of different edges) falls below ten or if the average solution quality of the population did not change for 30 iterations, a random k-exchange move, with $k = 0.1n$, and subsequent local search is applied to all tours in the population except for the one with the best solution quality.

Computational results confirm that MA-MF is amongst the best performing MAs for the TSP [Merz and Freisleben, 2002]. For TSPLIB instances with up to 1,002 vertices, the known optimal solutions could be consistently identified within an average time of about two CPU minutes on a Pentium III 500 MHz processor. When applied to instances with up to 3,795 vertices, the probability for finding optimal solutions dropped significantly, although still very good average solution qualities with a deviation of less than 0.08% from the optimum could be achieved. Additional experiments showed that high quality solutions could also be identified for the largest TSPLIB instances; except for the largest instance with 85,900 vertices, solution qualities within 1% of the optimum were reached within an average computation time of one CPU hour on a Pentium III 500 MHz processor. However, for some instances, MA-MF does not reach the solution qualities obtained by Helsgaun's LK variant (HLK); generally, it is not clear whether using a similarly efficient LK variant for the subsidiary local search in MA-MF (or any other MA) could lead to competitive performance with state-of-the-art SLS algorithms for the TSP, such as IHLK.

The Repair-based MA by Walters

The memetic algorithm by Walters (MA-W) differs in several key aspects from most recent MA approaches to the TSP. Firstly, MA-W uses a solution representation that is based on a nearest neighbour indexing. Let nn_i^k be the kth nearest neighbour of vertex v_i, *i.e.*, nn_i^1 is the nearest neighbour to vertex v_i, nn_i^2 is the second nearest neighbour, *etc.*. Then, for a given tour orientation, the successor of v_i is encoded by its index in v_i's nearest neighbour list. A tour p is represented by a vector $s = (s_1, \dots, s_n)$ such that $s_i = k$ if and only if the successor of v_i in p is the kth nearest neighbour of v_i. As a side-effect, this representation leads to some redundancy because of the directionality imposed by the encoding of successors: For symmetric TSP the direction in which a tour is traversed does not matter, but if we would encode the predecessor relationship in the tour, a different representation of the same tour is obtained.

Secondly, MA-W uses an ingenious repair mechanism to transform infeasible paths, which may be generated by applying standard recombination operators, into valid tours. This repair mechanism preserves as many edges of the infeasible tour as possible; if for some vertex, an outgoing edge e needs to be replaced in order to obtain a valid tour, it is replaced by an edge e' that is as close as possible in weight to e.

In more detail, the repair process works as follows: First, a working list of edges is created that comprises all edges contained in the infeasible path p. Next, this list is sorted according to edge weight plus a small amount of random noise (about 20% of the edge weight); the use of the noise randomises the order and thus helps to prevent domination of the end result, *i.e.*, the feasible tour returned by the repair process, by specific short edges. Then, the working list is traversed in ascending order, and edges are included into the new path p' to be constructed if they do not lead to any cycle with less than n edges. If an edge e cannot be included into p', it is replaced in the working list by another edge e' that is incident to the same vertex and whose weight is as close as possible to that of e, while e is moved to a list of failed edges. If after a full traversal of the working list, p' is not a valid tour, the repair process is repeated starting with path p'. This repair mechanism allows MA-W to use generic recombination and mutation operators that are not guaranteed to produce valid tours. In particular, MA-W uses a modified two-point recombination operator [Walters, 1998]. The mutation operator modifies the nearest neighbour indices of randomly selected vertices. The new index is selected according to a probability distribution which is also used to initialise the population: The indices corresponding to the three nearest neighbours are selected with a probability of 0.45, 0.25, and 0.15 respectively; in the remaining cases, an index between four and ten is chosen uniformly at random. (For further details on MA-W, including parameter settings, we refer to Walter's original paper [Walters, 1998].)

MA-W uses an efficient implementation of the `3-opt` local search as its subsidiary local search procedure. Compared to state-of-the-art TSP algorithms that are based on variants of the `LK` algorithm, the empirical performance results for MA-W are promising [Walters, 1998; Stützle *et al.*, 2000]. For example, the MA-W algorithm requires an average run-time of 572.7 CPU seconds on a Pentium II 450 MHz processor for finding an optimal solution to TSPLIB instance `pr2392`, a result that is only surpassed by the best performing SLS algorithms for the TSP known todate. However, on other instances, MA-W performs is significantly weaker than current state-of-the-art algorithms and it is not clear how its performance scales to large instances.

ACO algorithms for the TSP

The TSP has played a central role in the development of ACO algorithms, because the first ACO algorithm, Ant System [Dorigo *et al.*, 1991; Dorigo, 1992; Dorigo *et al.*, 1996] and most of its successors, including Ant Colony System [Dorigo and Gambardella, 1997], $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System [Stützle and Hoos, 1997; Stützle and Hoos, 2000], Rank-based Ant System [Bullnheimer *et al.*, 1999], and Best-Worst Ant System [Cordón *et al.*, 2000] were first applied to the TSP. All these algorithms follow the same basic outline shown in Figure 2.10 on page 89, but they differ in several important algorithmic details.

The currently most successful ACO algorithms for the TSP share two important features: (i) they include effective mechanisms for achieving a good balance between intensification and diversification of the search [Dorigo and Gambardella, 1997; Stützle and Hoos, 2000] and (ii) before updating the pheromone trails, a subsidiary local search procedure is applied to the tours constructed by the ants [Dorigo and Gambardella, 1997; Stützle and Hoos, 2000; Stützle, 1998; Stützle and Hoos, 1997]. Hence, these ACO algorithms are hybrid SLS techniques that combine probabilistic solution construction with standard local search techniques.

Max-Min Ant System

One of the most effective ACO algorithms for the TSP is $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System ($\mathcal{MM}$AS). $\mathcal{MM}$AS is closely related to Ant System (AS) [Dorigo *et al.*, 1991; Dorigo, 1992; Dorigo *et al.*, 1996] (see Chapter 2, Section 2.3); in particular, the same mechanism for constructing candidate solutions is used: Starting from a randomly chosen vertex v_0, in each construction step, the current partial tour p is extended with avertex v_j that is adjacent to its endpoint v_i; v_j is chosen probabilistically from the set of all vertices not contained in p; the probability of choosing vertex v_j is computed from the pheromone trail strength $\tau_{ij}(t)$ for edge (v_i, v_j) and a heuristic value $\eta_{ij} = 1/w((v_i, v_j))$ according to Equation 2.2 on page 86.

$\mathcal{MM}$AS introduces four major modifications to AS. Firstly, it strongly exploits the best solutions found during the search — a feature it has in common with a variety of other ACO algorithms [Dorigo and Gambardella, 1997; Bullnheimer *et al.*, 1999; Cordón *et al.*, 2000]. $\mathcal{MM}$AS uses the modified pheromone update rule

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^{\mathit{best}},$$

where $\Delta\tau_{ij}^{\mathit{best}} = 1/w(p^{\mathit{best}})$ and only one ant is allowed to update the pheromone trail levels according to its tour, p^{best}; this may either be the "globally best" ant, *i.e.*, the ant corresponding to the best candidate solution found since the start of the algorithm, or the "iteration-best" ant, *i.e.*, the ant that represents the best tour obtained in the current iteration. The choice between the use of the iteration-best and the globally best ant influences the greediness of the search: When the globally best ant always deposits pheromone, the search focuses quickly around the best tour found so far, while when allowing the iteration-best ant to deposit pheromone, over time, a larger number of edges are reinforced and resulting in a less directed search. Empirical results indicate that while for rather small TSP instances it may be best to use iteration-best pheromone update only, for large TSPs with several hundreds of vertices the best performance is obtained by giving a stronger emphasis to the globally best ant. This can be achieved, for example, by choosing the globally best ant for trail update with a frequency that gradually increasing over time [Stützle, 1998].

One disadvantage of the greedier trail update procedure is an increased danger of search stagnation. Therefore, a second important feature of $\mathcal{MM}$AS is its use of strict lower and upper limits τ_{min} and τ_{max} on the pheromone trail level for each individual edge, which effectively avoids search stagnation. In particular, these limits have the effect of bounding the probability

p_{ij} of selecting a vertex j when an ant is in vertex i according to Equation 2.2 to an interval $[p_{\min}, p_{\max}]$, with $0 < p_{\min} \leq p_{ij} \leq p_{\max} \leq 1$. Only if an an ant has visited all but one vertices in the given graph, it will deterministically choose that vertex for its next and penultimate step, such that $p_{\min} = p_{\max} = 1$.

It is easy to show that in the limit, the pheromone trail level on any edge is bounded from above by $w(p^*)/\rho$, where $w(p^*)$ is the weight of an optimal tour for the given TSP instance. Based on this result, $\mathcal{MM}$AS uses an estimate of this value, $w(p^{gb})/\rho$, to define τ_{max}. In fact, each time a new globally best tour is found, the value of τ_{max} is adapted. The lower pheromone trail limit is set to $\tau_{\mathit{min}} = \tau_{\mathit{max}}/a$, where a is a parameter of the algorithm [Stützle, 1998; Stützle and Hoos, 2000]. Experimental results [Stützle, 1998] suggest that for effectively avoiding search stagnation, the τ_{min} plays a more important role than τ_{max}.

Thirdly, at the start of the algorithm, $\mathcal{MM}$AS sets the initial pheromone trails to an estimate of $\tau_{\mathit{max}} = w(p_i)/\rho$, where tour p_i is obtained by applying local search to a nearest neighbour tour. Together with small values for the pheromone evaporation parameter ρ, this ensures that during its initial search phase, $\mathcal{MM}$AS is very explorative. As a further means for increasing the exploration of paths that have only a small probability of being chosen, $\mathcal{MM}$AS occasionally re-initialises the pheromone trails; trail re-initialisation is triggered when stagnation behaviour is detected (as measured by some statistics on the pheromone trails or the number of iterations during which no improvement of the incumbent tour occurred [Stützle and Hoos, 2000]).

While early versions of $\mathcal{MM}$AS used `2-opt` as a subsidiary local search procedure [Stützle and Hoos, 1996; 1997], more recent variants use an efficient `3-opt` procedure [Stützle, 1998; Stützle and Hoos, 2000]. Results for these latter variants suggest that $\mathcal{MM}$AS can find optimal solutions for TSP instances with a few hundred vertices up slightly more than 1,000 vertices rather efficiently, *i.e.*, in a few CPU minutes to around one CPU hour on a UltraSparc I 167 MHz processor). Limited experiments indicate that by using an `LK` algorithm as an subsidiary local search procedure, the performance of $\mathcal{MM}$AS can be significantly improved. However, probably because the `LK` implementation used for these experiments was not efficient enough, of this lates $\mathcal{MM}$AS algorithm did not reach the performance of the currently best SLS algorithms for the TSP.

Population-based Algorithms for the ATSP

All previously described population-based algorithms for the symmetric TSP can be extended to the ATSP in a rather straightforward way. In most cases, the only major difference lies in the use of a different subsidiary local search procedure that is suitable for the ATSP, such as `reduced 3-opt`. Additionally, some of the operators used in the context of Evolutionary Algorithms require adaptations, and for ACO algorithms, asymmetric pheromone trail levels need to be supported.

Empirical results suggest that the ATSP algorithms thus obtained can typically find optimal solutions of asymmetric TSPLIB instances with up to 170 vertices within reasonable run-times [Merz and Freisleben, 1997; Stützle and Hoos, 2000; Stützle, 1998; Walters, 1998]. When taking into account the differences between the machines that were used for these experiments, the best performance is probably achieved by the ATSP version of Walter's repair-based MA, followed by the population-based ILS variants and $\mathcal{MAX}$–$\mathcal{MIN}$ Ant System. Recently, very good results were also reported for the memetic algorithm of Buriol, França, and Moscato [Buriol *et al.*, 2002], which uses a specialised local search algorithm for the ATSP as its subsidiary local search procedure. However, there is some indication that none of these population-based algorithms reaches the performance of the best known SLS algorithms for the ATSP [Johnson *et al.*, 2002b], but further empirical analysis is required for a conclusive performance comparison.

8.5 Further Readings and Related Work

The literature on the TSP is vast and it is practically impossible to adequately cover in a single book chapter all relevant work on SLS algorithms for the TSP, not even to mention other types of TSP algorithms. ¿From this perspective, it is not surprising that there are a number of books that are entirely devoted to the TSP. A classic in the TSP literature the book by Lawler, Lenstra, Rinnooy Kan, and Shmoys (editors) [Lawler *et al.*, 1985] which covers many aspects of TSP research up to the year 1985. The book by Reinelt [Reinelt, 1994], published in 1994, provides an extensive and in-depth computational study of construction heuristics and local search algorithms. The recently published book by Gutin and Punnen (editors) [Gutin and Punnen, 2002] covers many aspects of the TSP and TSP solving, including exact algorithms, SLS methods, empirical analysis of heuristic TSP algorithms, and problems related to the TSP.

In the following, we provide a few selected references to important work related to the SLS methods covered in this chapter, starting with iterative improvement algorithms. The `2.5-opt` algorithm enhances `2-opt` by additionally checking whether a tour can be improved by inserting a vertex between the two vertices incident to the first edge that is removed in a 2-exchange move. `2.5-opt` was shown to obtain substantially better tours than `2-opt` at only a slight increase of computation time [Bentley, 1992]. The `Or-opt` algorithms generalises `2.5-opt` by allowing the tour segment that is inserted between two tour neighbours to be of maximal length three (the length of the segment is one in the case of `2.5-opt`) [Or, 1976]; however, no results for `Or-opt` using current speed-up techniques are available and therefore its performance potential is unclear.

The Generalised Insertion Procedure (GENI) combines a construction heuristic with a local search algorithm that is applied each time after adding a vertex to the current partial tour [Gendreau *et al.*, 1992]. This procedure is extended by a post-optimisation phase, during which iteratively the following steps are applied: remove a vertex, apply a local search and then re-insert the vertex into the tour [Gendreau *et al.*, 1992]. The Hyperopt algorithm combines enumerative algorithms with local search. It is based on the deletion of a set of edges and then enumerating all possible ways of reconnecting the resulting tour segments and individual vertices [Burke *et al.*, 2001]. However, the computational results obtained so far are not promising, neither for symmetric TSP [Johnson *et al.*, 2002a] nor for ATSPs [Johnson *et al.*, 2002b].

There are several other SLS algorithms for the TSP that are based on (exponentially) large neighbourhoods. Computational results for Dynasearch extensions (see Section 2.1 of Chapter 2) of the 2-exchange, 2.5-exchange and 3-exchange neighbourhoods and iterated versions of Dynasearch show promising performance of this approach, although they do not appear to be competitive with `LK` or iterated `LK` [Congram, 2000]. Ejection chains are another type of variable-depth search algorithms, which is closely related to `LK` [Glover, 1996a; Rego and Glover, 2002]. Compared to `LK` they allow more flexibility in building complex search steps (for more details we refer to [Rego and Glover, 2002]).

Apart from tour-merging (see Section 8.3), two further approaches make use of ILS algorithms as a sub-routine. These are the dynamic programming algorithm of Balas an Simonetti [Balas and Simonetti, 2001], which tries to find a best way of locally reordering a tour, and the multi-level approach by Walshaw [Walshaw, 2002].

Basically all of the 'simple' SLS techniques covered in Chapter 2, Section 2.2, have been applied to the TSP. Although in some cases, reasonably good performance was reported (mainly for TSPLIB instances), none of these SLS algorithms appears to be competitive with the best performing ILS or population-based algorithms [Johnson and McGeoch, 1997; Johnson *et al.*, 2002a]. However, the interested reader may learn more about these TSP algorithms from the following references.

Simulated Annealing is mainly covered in the book chapter by Johnson and McGeoch [Johnson and McGeoch, 1997], although also several earlier approaches exist. Tabu Search algorithms based on short term tabu memory are described by Knox [Knox, 1994] and Malek et al. [Malek *et al.*, 1989], while more advanced Tabu Search strategies are applied by Fiechter [Fiechter, 1994] and Dam and Zachariason [Zachariasen and Dam, 1996]. Among the Dynamic Local Search methods, two approaches were applied to the TSP: Guided Local Search by Voudouris and Tsang [Voudouris and Tsang, 1999b] and the Noising Method by Charon and Hudry [Charon and Hudry, 2000].

There is a large body of literature on population-based SLS algorithms for the TSP, in particular on Evolutionary Algorithms. In addition to the two Evolutionary Algorithms presented in Section 8.4, a few other recent algorithms are worth mentioning (for an overview of earlier approaches we refer to [Johnson and McGeoch, 1997; Merz and Freisleben, 2002; Potvin, 1996]). Nagata and Kobayashi [Nagata and Kobayashi, 1997] presented an EA with edge assembly crossover, which constructs offspring based on the union of the edge sets of the two parents and then applies a greedy construction algorithm to merge sub-tours. This EA does not use explicit local search, however, the recombination operator incorporates some local search features. Very good performance is reported for the GA of Seo and Moon [Seo and Moon, 2002], which uses a particular recombination operator called Voronoi Quantised Crossover.

There are a number of other population-based algorithms that share some connection to Evolutionary Algorithms, but introduce additional ideas.

One is the Iterative Partial Transcription (IPT) approach by Möbius *et al.* [Möbius *et al.*, 1999], which can be seen as a local search method that is based on information exchanges between pairs of solutions. Houdayer and Martin [Houdayer and Martin, 1999] propose a population-based algorithm that iteratively generates offspring by choosing k parents, freezing the common edges among the k parents, and solving a TSP in which only edges that are not frozen may be changed. For the various Ant Colony Optimisation applications to the TSP we refer to the overview article by Stützle and Dorigo [Stützle and Dorigo, 1999].

A number of extensive computational studies on heuristic TSP algorithms are now available. First and foremost, the 8th DIMACS Implementation Challenge on the TSP [Johnson *et al.*, 2002a] provides an online collection of empirical results for a large variety of implementations of construction heuristics, local search algorithms, and more complex SLS algorithms. A summary of the results as of July 1, 2001 is available in a book chapter by Johnson and McGeoch [Johnson and McGeoch, 2002]; the same book also includes a chapter on computational results with SLS algorithms for the ATSP [Johnson *et al.*, 2002b]. **[mention forthcoming DIMACS report if it appears before book deadline – TODO(ts).]** For the most recent results, we refer the interested to the web-pages, which provide pairwise performance comparisons of algorithms on sets of TSP instances as well as results for all algorithms on each single instance available for the implementation challenge. However, the way in which the challenge results are presented makes it difficult to draw final conclusions regarding the relative advantages of many advanced SLS algorithms, whose performance is often not dominated by any other algorithm on every single instance. This situation may be improved by using some of the empirical methods presented in Chapter 4.

Although the 8th DIMACS Implementation Challenge on the TSP provides extensive empirical results, there are a number of earlier, extensive studies that are quite relevant. These include the study of construction heuristics and local search algorithms by Bentley [Bentley, 1992], the previously mentioned book by Reinelt [Reinelt, 1994], and a book chapter on local search for the TSP again by Johnson and McGeoch [Johnson and McGeoch, 1997].

8.6 Summary

The TSP is a central problem in combinatorial optimisation with many theoretical and practical applications; it also was and still is at the core of attempts to push permanently further the boundary on the size of practically tractable optimisation problems. State-of-the-art complete TSP algorithms can solve instances up to several thousand vertices in reasonable computation times (CPU hours to several CPU days), while the best SLS algorithms can find solutions whose quality is within fractions of a percent of the optimum for much larger instances with up to millions of vertices.

Construction heuristics can find reasonably good solutions for TSP instances extremely fast. They also play an important role as initialisation procedures for various SLS algorithms. `2-opt` and `3-opt` are most prominent iterative improvement algorithms based on k-exchange neighbourhoods. Various speed-up techniques, including fixed-radius searches, candidate lists and don't look bits, play a crucial role in the design of efficient SLS algorithms for the TSP, particularly in the case of iterative improvement algorithms. Variable-depth search methods, such as the Lin-Kernighan algorithm, can find higher-quality solutions, but typically require longer runtimes. They also require considerable fine-tuning and are substantially harder to implement than simpler iterative improvement algorithms, such as `2-opt` and `3-opt`.

Somewhat surprisingly, current empirical evidence suggests that one of the conceptually simplest hybrid SLS methods, Iterated Local Search, gives rise to some of the best performing TSP algorithms currently known. Only if extremely high solution qualities are required, other approaches may be preferable, such as the tour merging approach, which uses an ILS algorithm as a subroutine in a more complex, hybrid search algorithm. Population-based extensions of ILS or slightly more complex SLS algorithms, such as Memetic Algorithms or Ant Colony Optimisation also achieve very high performance; however, there is currently no strong evidence that these population-based algorithms achieve any performance advantage over conceptually more simple state-of-the-art ILS algorithms for the TSP.

The TSP is an ideal playground for the design and analysis of SLS algorithms. Firstly, there is substantial evidence that the SLS techniques underlying the best-performing TSP algorithms often achieve excellent or even state-of-the-art performance for many other combinatorial optimisation problems. This is the case for many ACO algorithms as well as Evolutionary Algorithms (in particular, for Memetic Algorithms). Secondly, because the TSP is easy to understand and does not involve side constraints that complicate algorithm design, it is particularly well suited for gaining practical experience in the design and implementation of high performance SLS algorithms for combinatorial optimisation problems.

8.7 Exercises

Exercise 8.1 (Medium) In Section 8.1 we stated that adding penalties p_i to each vertex v_i and defining modified edge weights $w'((v_i, v_j)) = w((v_i, v_j)) + p_i + p_j$ preserves optimality of tours, but it may result in different one trees. Show that the optimality of tours is preserved. Give an example of a weighted graph, where the optimum one-tree changes after the addition of appropriate vertex penalites.

Exercise 8.2 (Easy) Specify all the possible ways of re-wiring the tour segments in a 3-exchange algorithm after 3 edges were deleted. How many possibilities to exist when applying a 4-exchange algorithm?

Exercise 8.3 (Implementation, easy) The power of the pruning achieved by fixed-radius search around a vertex v_i depends strongly the nearest neighbour indices of v_i's tour neighbours. For example, when edge (v_i, v_j) is broken and v_j is the closest vertex to v_i no search at all is done. Analyse the distribution of the nearest neighbour indices for RUE, RCE, TSPLIB, and RDM instances.

Exercise 8.4 (Easy) Prove the following statement. If fixed radius near neighbour searches for all vertices do not result in any improving 2-exchange move, the current tour is `2-opt`.

Exercise 8.5 (Medium) Explain how the complex steps in the Lin-Kernighan algorithm, which were introduced as sequences of 1-exchange moves, can also be interpreted as sequences of 2-exchange moves.

Exercise 8.6 (Medium) In the original Lin-Kernighan algorithm the set of deleted edges X and the set of added edges Y is required to be disjoint. Why may the variable depth search be unbounded, if this criterion is dropped?

What if we require instead that "no deleted edge can be added subsequently" or "no added edge can be deleted subsequently"? Can either of these criteria lead to unbounded searches?

Exercise 8.7 (Implementation; easy) Study the influence of the don't look bit resetting strategy after a perturbation in ILS. Consider the following three re-setting strategies: (i) reset all don't look bits to zero, (ii) reset only don't look bits of end-points of broken edges, and (iii) reset don't look bits of the end-points of broken edges plus the 25 tour neighbours.

Exercise 8.8 (Implementation; medium) One possibility to obtain a stochastic local search procedure from an, at least in principle, deterministic k-opt algorithm is to generate a random permutation of the vertex indices and to choose starting vertices for the searches of improving moves according to this random order. (This is a possibility also considered in the accompanying code). Study the cost distribution of the local optima returned by such a randomised local search procedure by following these steps:

1. Generate one nearest neighbour tour, one greedy tour and one tour by random insertion.
2. Apply to each of these tours 10,000 times a randomised `2-opt` algorithm, which includes all the available speed-up techniques on instances from TSPLIB that are larger than 500 vertices.
3. Generate the resulting empirical distributions and try to approximate these with known distributions from statistics.
4. Compare the cost distributions regarding location and shape to a random restart heuristics that starts a *deterministic* local search from 10,000 random initial solutions.

[The implementation exercises above make use of TSP solvers that will be provided on the companion webpage for SLS book.]