These techniques are important, because they are at the core of many more advanced SLS algorithms. They range from extremely fast constructive search algorithms, such as the Nearest Neighbour Heuristic, to complex variable depth search methods, in particular, variants of the Lin-Kernighan Algorithm, which make extensive use of a number of speedup techniques.

Nearest Neighbour and Insertion Construction Heuristics

There is a large number of constructive search algorithms for the TSP, ranging from extremely fast methods for metric TSP instances, whose run-time is only slightly larger than the time required for just reading the instance data from the hard-disk (see, for example, Platzman and Bartholdi III [1989]), to more sophisticated algorithms with non-trivial bounds 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 constructive search algorithm has been already discussed in Chapter 1, Section 1.4: The *Nearest Neighbour Heuristic (NNH)* starts tour construction from some randomly chosen vertex u_1 in the given graph and then iteratively extends the current partial tour $p = (u_1, \ldots, u_k)$ with an unvisited vertex u_{k+1} that is connected to u_k by a minimum weight edge (u_{k+1} is called a *nearest neighbour* of u_k); when all vertices have been visited, a complete tour is obtained by extending p with the initial vertex, u_1. The tours constructed by the NNH are called *nearest neighbour tours*.

For TSP instances that satisfy the triangle inequality, nearest neighbour tours are guaranteed to be at most by a factor of $1/2 \cdot (\lceil \log_2(n) \rceil + 1)$ worse than optimal tours in terms of solution quality [Rosenkrantz et al., 1977]. In the general case, however, there are TSP instances for which the Nearest Neighbour Heuristic returns tours that are by a factor of $1/3 \cdot (\log_2(n+1) + 4/3)$ worse than optimal, and hence, the approximation ratio of the NNH for the general TSP cannot be bounded by any constant [Rosenkrantz et al., 1977]. In practice, however, the NNH typically yields much better tours than these worst-case results may suggest; for metric and TSPLIB instances, nearest neighbour tours are typically only 20–35% worse than optimal. In most cases, nearest neighbour tours are locally similar to optimal solutions, but they include some very long edges that are added towards the end of the construction process in order to complete the tour (two examples are shown in Figure 8.4). This effect is avoided to some extent by a variant of the NNH that penalises insertions of such long edges [Reinelt, 1994]. Compared to the standard NNH, this variant requires only slightly more

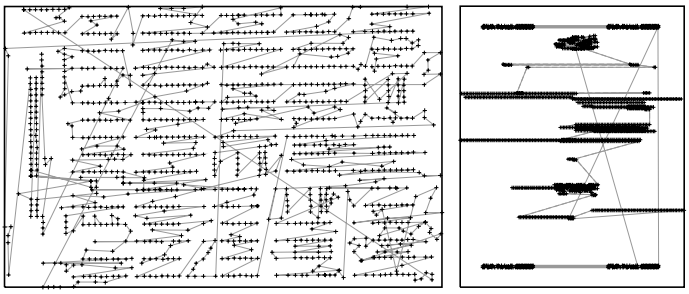

***Figure* 8.4** Two examples of nearest neighbour tours for TSPLIB instances. *Left:* `pcb1173` with 1 173 vertices, *right:* `fl1577` with 1 577 vertices. Note the long edges contained in both tours.

computation time, but when applied to TSPLIB instances, it finds tours that are about 5% closer to optimal.

Insertion heuristics construct tours in a way that is different from that underlying the NNH; in each step, they extend the current partial tour p by inserting a heuristically chosen vertex at a position that typically leads to a minimal length increase. Several variants of these heuristics exist, including

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

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

(iii) *farthest insertion*, where the next vertex to be inserted is a vertex u_i for which the minimum distance to a 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 long as an optimal tour [Rosenkrantz et al., 1977], while for random and farthest insertion, the solution quality is only guaranteed to be within a factor $O(\log n)$ of the optimum [Johnson and McGeoch, 2002]. In practice, however, the farthest and random insertion heuristics perform much better than nearest and cheapest insertion, yielding tours that, in the case of TSPLIB and RUE instances, are on average between 13% and 15% worse than optimal [Johnson and McGeoch, 2002; Reinelt, 1994].

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 alternative approach is to iteratively build several tour fragments 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 minimum weight edge, in linear order. An edge e is added to the current partial candidate solution p if inserting it into G', the graph that contains all vertices of G and the edges in p, does not result in 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* [Applegate et al., 1999], which is inspired by the minimum spanning tree algorithm by Borůvka [1926]. First, the vertices in G are sorted arbitrarily (e.g., for metric TSP instances, the vertices can be sorted according to their first coordinate values). Then, the vertices are processed in the given order. For each vertex u_i of degree less than two in G', all edges incident to u_i that appear in G but not in G' are considered. Of these, the minimum weight edge that results neither in a cycle of length less than n nor in a vertex of degree larger than two, is added to G'. Note that at most two scans of the vertices have to be performed to generate a tour.

Another construction heuristic that is based on building multiple partial tours, is the *Savings Heuristic*, which was initially proposed for a vehicle routing problem [Clarke and Wright, 1964]. It works by first choosing a base vertex u_b and $n-1$ cyclic paths (u_b, u_i, u_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 u_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 the 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 longer than an optimal tour, while the length of a savings tour is at most a factor of $(1+\log n)$ 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 length 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, ranging for 1 million vertex RUE instances from 22 (for Quick-Borůvka) to around 100 seconds (for Greedy and Savings) on a 500 MHz Alpha CPU.

Construction Heuristics Based on Minimum Spanning Trees

Yet another class of construction heuristics builds tours based on minimum-weight spanning trees (MSTs). In the simplest case, such an algorithm consists of the following four steps: First, an MST t for the given graph G is computed; then, by doubling each edge in t, a new graph G' is obtained. In the third step, a Eulerian tour p of G', that is, a cyclic path that uses each edge in G' exactly once, is generated; a Eulerian tour can be found in $O(e)$, where e is the number of edges 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 the given TSP instance satisfies the triangle inequality; hence, in this case, the final tour is at most twice as long as an optimal tour. However, empirically this construction heuristic performs rather poorly, with solution qualities that are on average around 40% above the optimal tour lengths for TSPLIB and RUE instances [Reinelt, 1994; Johnson and McGeoch, 2002].

Much better performance is obtained by the *Christofides Heuristic* [Christofides, 1976]. 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 $O(k^3)$, 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, that is, a graph containing an Eulerian tour. As described previously, in a final step, this Eulerian tour is converted into a Hamiltonian cycle. 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, the Savings and Greedy Heuristics [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. The resulting variant of the Christofides Heuristic appears to be the best-performing construction heuristic for the TSP in terms of the solution quality achieved; however, its run-time is higher than that of the Savings Heuristic by a factor that increases with instance size from about 3.2 for RUE instances with

1 000 vertices to about 8 for RUE instances with 3.16 million vertices [Johnson et al., 2003a].

k-Exchange Iterative Improvement Methods

Most iterative improvement algorithms for the TSP are based on the k-exchange neighbourhood relation, in which candidate solutions s and s' are direct neighbours if, and only 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 relation, $k = 2$ and $k = 3$ are the most common choices. 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 ways in which the resulting fragments can be reconnected into a candidate solution different from s depends on k; for $k = 2$, after deleting two edges (u_i, u_j) and (u_k, u_l), the only way to rewire the two partial tours into a different complete tour is by introducing the edges (u_i, u_k) and (u_l, u_j). Note that after a 2-exchange move, one of the two partial tours is reversed. (For an illustration, see Figure 1.6, page 44.)

For $k = 3$, there are several ways of reconnecting the three tour fragments 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 8.5 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 sets of edges deleted from and subsequently added to the given candidate tour have one element in common. Allowing an 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 $k' < k$.

Based on the 2-exchange and 3-exchange neighbourhood relations, 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, because 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 mechanism used for selecting an improving neighbouring candidate solution; see also Chapter 2, Section 2.1). In general, first-improvement

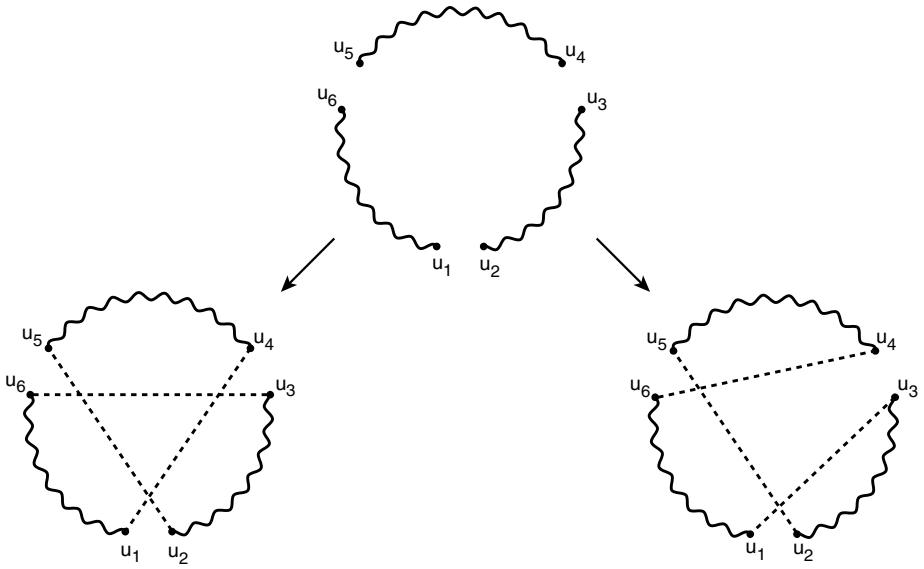

Figure **8.5** Two possible ways of reconnecting partial tours in a 3-exchange move after edges (u_1, u_2), (u_3, u_4) and (u_5, u_6) have been removed from a complete tour. Note that in the left result, the relative direction of all three tour fragments is preserved.

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 $O(n^2)$ and $O(n^3)$ 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 exists a number of speedup techniques that result in significant improvements in the time complexity of local search steps [Bentley, 1992; Johnson and McGeoch, 1997; Martin et al., 1991; Reinelt, 1994].

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 u_i, two searches are performed that consider each of

the two tour neighbours of u_i as a vertex u_j, respectively. For a given u_j, a search around u_i is performed for vertices u_k that are closer to u_i than $w((u_i, u_j))$, the radius of the search. For each vertex u_k found in this *fixed radius near neighbour* search, removing one of its two incident edges in s leads to a feasible 2-exchange move. The first such 2-exchange move that results in 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-optimal.

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

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 u_i by edges with low weight, for example, in the form of a list of neighbouring vertices u_k that is sorted according to edge weight $w((u_i, u_k))$ in ascending order. By using such *candidate lists* for all vertices in G, fixed radius near neighbour searches can be performed very efficiently; this is illustrated by the empirical results reported in Example 8.1 on page 376*f.*. 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 search steps that are evaluated first when using candidate lists are more effective than other k-exchange steps.

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

Typically, candidate lists of length 10 to 40 are used, although shorter lengths are sometimes chosen. Simply using short candidate lists that consist of 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 (page 360). For metric TSP instances, alternative approaches to constructing bounded-length candidate lists include so-called quadrant-nearest neighbour lists [Pekny and Miller, 1994; 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 to the Held-Karp lower bounds (see Section 8.1) [Helsgaun, 2000]. This mechanism works as follows: Based on the modified edge weights $w'((u_i, u_j))$ obtained from an approximation to the Held-Karp lower bounds, so-called α-values are computed for each edge (u_i, u_j) as $\alpha((u_i, u_j)) := w'(t^+(u_i, u_j)) - w'(t)$, where $w'(t)$ is the weight of a minimum weight one-tree t and $w'(t^+(u_i, u_j))$ is the weight of a minimum weight one-tree $t^+(u_i, u_j)$ that is forced to contain edge (u_i, u_j). For each edge, $\alpha(u_i, u_j) \geq 0$, and $\alpha(u_i, u_j) = 0$ if the edge (u_i, u_j) is contained in some minimum weight one-tree. A candidate list for a vertex u_i can now be obtained by sorting the edges incident to u_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 threshold. The vertices contained in these candidate lists are called *α-nearest neighbours*.

Empirically it was 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 the 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 u_i (e.g., in a fixed radius near neighbour search), it is unlikely that an improving move involving u_i will be found in future search steps, unless at least one of the edges incident to u_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 started only at 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 speedup that can be achieved by using DLBs is illustrated by the empirical results for various variants of 2-opt shown in Example 8.1.

The DLB mechanism can be easily integrated into more complex SLS methods, such as Iterated Local Search or Memetic Algorithms. One possibility is to set only the DLBs of those vertices to zero that are incident to edges that were 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 when compared to resetting 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 Speedup Techniques for 2-opt**

To illustrate the effectiveness of the previously discussed speedup techniques, we empirically evaluated three variants of 2-opt: a straight-forward implementation that in each search step evaluates every possible 2-exchange move (2-opt-std); a fixed radius near neighbour search that uses candidate lists of unbounded length (2-opt-fr+cl); and a fixed radius near neighbour search that uses candidate lists of unbounded length as well as DLBs (2-opt-fr+cl+dlb). For all variants, the search process was initialised at a random permutation of the vertices, and it was terminated as soon as a local minimum was encountered. These algorithms were run 1 000 times on several benchmark instances from TSPLIB using an Athlon 1.2 GHz MP CPU with 1 GB RAM running Suse Linux 7.3. (The 2-opt implementation used for these experiments is available from `www.sls-book.net`.)

The results reported in Table 8.1 show that the speedup techniques achieve substantial decreases in run-time over a standard implementation, an effect that increases strongly with instance size. The most significant speedup seems to be due to the combination of fixed-radius nearest neighbour search with candidate lists, while the additional use of DLBs can sometimes reduce the computation times by another factor of two. In addition, the bias in the local search towards first examining the most promising moves that is introduced by the use of candidate lists results in a significant improvement in the solution quality obtained by 2-opt; the use of DLBs diminishes this effect only slightly. When bounded length candidate lists are used, very

	2-opt-**std**		2-opt-**fr** + **cl**		2-opt-**fr** + **cl** + **dlb**		3-opt-**fr** + **cl**	
Instance	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}	Δ_{avg}	t_{avg}
`rat783`	13.0	93.2	3.9	3.9	8.0	3.3	3.7	34.6
`pcb1173`	14.5	250.2	8.5	10.8	9.3	7.1	4.6	66.5
`d1291`	16.8	315.6	10.1	13.0	11.1	7.4	4.9	76.4
`fl1577`	13.6	528.2	7.9	21.1	9.0	11.1	22.4	93.4
`pr2392`	15.0	1 421.2	8.8	47.9	10.1	24.9	4.5	188.7
`pcb3038`	14.7	3 862.4	8.2	73.0	9.4	40.2	4.4	277.7
`fnl4461`	12.9	19 175.0	6.9	162.2	8.0	87.4	3.7	811.6
`pla7397`	13.6	80 682.0	7.1	406.7	8.6	194.8	6.0	2 260.6
`rl11849`	16.2	360 386.0	8.0	1 544.1	9.9	606.6	4.6	8 628.6
`usa13509`	—		7.4	1 560.1	9.0	787.6	4.4	7 807.5

***Table* 8.1** Computational results for different variants of 2-opt and 3-opt. Δ_{avg} denotes the average percentage deviation from the optimal solution quality over 1 000 runs per instance, and t_{avg} is the average run-time for 1 000 runs of the respective algorithm, measured in CPU milliseconds on an Athlon 1.2 GHz CPU with 1GB of RAM. (For further details, see text.)

similar results were obtained for most instances (not shown here); only on the pathologically clustered instance `fl1577`, the solution quality decreases to an average of almost 60% above the optimum, while the computation time is reduced by about 10% (these observations were made for a length bound of 40).

3-opt achieves better-quality solutions than the previously mentioned 2-opt variants at the cost of substantially higher computation times; this is illustrated by the results for 3-opt with a fixed-radius search using candidate lists of a length limited to 40 shown in the last column of Table 8.1. Interestingly, using unbounded candidate lists for 3-opt leads to computation times that can be substantially higher than the ones reported in Table 8.1. This illustrates that bounding the length of candidate lists becomes increasingly important in the context of local search algorithms based on larger neighbourhoods.

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