



ELSEVIER

European Journal of Operational Research 126 (2000) 106–130

EUROPEAN  
JOURNAL  
OF OPERATIONAL  
RESEARCH

[www.elsevier.com/locate/dsw](http://www.elsevier.com/locate/dsw)

## Theory and Methodology

# An effective implementation of the Lin–Kernighan traveling salesman heuristic

Keld Helsgaun

Department of Computer Science, Roskilde University, Building 42.1, Universitetsvej 1, Box 260, DK-4000 Roskilde, Denmark

Received 14 October 1998; accepted 13 April 1999

---

### Abstract

This paper describes an implementation of the Lin–Kernighan heuristic, one of the most successful methods for generating optimal or near-optimal solutions for the symmetric traveling salesman problem (TSP). Computational tests show that the implementation is highly effective. It has found optimal solutions for all solved problem instances we have been able to obtain, including a 13,509-city problem (the largest non-trivial problem instance solved to optimality today). © 2000 Elsevier Science B.V. All rights reserved.

**Keywords:** Traveling salesman problem; Heuristics; Lin–Kernighan

---

### 1. Introduction

The Lin–Kernighan heuristic [1] is generally considered to be one of the most effective methods for generating optimal or near-optimal solutions for the symmetric traveling salesman problem (TSP). However, the design and implementation of an algorithm based on this heuristic is not trivial. There are many design and implementation decisions to be made, and most decisions have a great influence on performance.

This paper describes the implementation of a new modified version of the Lin–Kernighan algorithm. Computational experiments have shown that the implementation is highly effective.

The new algorithm differs in many details from the original one. The most notable difference is found in the search strategy. The new algorithm uses larger (and more complex) search steps than the original one. Also new is the use of sensitivity analysis to direct and restrict the search.

Run times of both algorithms increase approximately as  $n^{2.2}$ . However, the new algorithm is much more effective. The new algorithm makes it possible to find *optimal* solutions to large-scale problems, in reasonable running times.

For a typical 100-city problem the optimal solution is found in less than a second, and for a typical 1000-city problem optimum is found in less than a minute (on a 300 MHz G3 Power Macintosh).

Even though the algorithm is approximate, optimal solutions are produced with an impres-

---

E-mail address: [keld@ruc.dk](mailto:keld@ruc.dk) (K. Helsgaun).

sively high frequency. It has produced optimal solutions for all solved problems we have been able to obtain, including a 13,509-city problem (at the time of writing, the largest non-trivial problem solved to optimality). Furthermore, the algorithm has improved the best known solutions for a series of large-scale problems with unknown optima, among these an 85,900-city problem.

The rest of this paper is organized as follows. Section 2 defines the TSP and gives an overview of solution algorithms. Section 3 describes the original algorithm of Lin and Kernighan (including their own refinements of the algorithm). Sections 4 and 5 present the new modified algorithm and its implementation. The effectiveness of the implementation is reported in Section 6.

## 2. The traveling salesman problem

### 2.1. Formulation

A salesman is required to visit each of  $n$  given cities once and only once, starting from any city and returning to the original place of departure. What tour should he choose in order to minimize his total travel distance?

The distances between any pair of cities are assumed to be known by the salesman. Distance can be replaced by another notion, such as time or money. In the following the term ‘cost’ is used to represent any such notion.

This problem, the TSP, is one of the most widely studied problems in combinatorial optimization [2]. Mathematically, the problem may be stated as follows:

Given a ‘cost matrix’  $C = (c_{ij})$ , where  $c_{ij}$  represents the cost of going from city  $i$  to city  $j$ , ( $i, j = 1, \dots, n$ ), find a permutation  $(i_1, i_2, i_3, \dots, i_n)$  of the integers from 1 through  $n$  that minimizes the quantity

$$c_{i_1 i_2} + c_{i_2 i_3} + \dots + c_{i_n i_1}$$

Properties of the cost matrix  $C$  are used to classify problems.

- If  $c_{ij} = c_{ji}$  for all  $i$  and  $j$ , the problem is said to be *symmetric*; otherwise, it is *asymmetric*.

- If the triangle inequality holds ( $c_{ik} \leq c_{ij} + c_{jk}$  for all  $i, j$  and  $k$ ), the problem is said to be *metric*.
- If  $c_{ij}$  are Euclidean distances between points in the plane, the problem is said to be *Euclidean*. A Euclidean problem is, of course, both symmetric and metric.

### 2.2. Solution algorithms

The problem is easy to state, but hard to solve. The difficulty becomes apparent when one considers the number of possible tours – an astronomical figure even for a relatively small number of cities. For a symmetric problem with  $n$  cities there are  $(n-1)!/2$  possible tours. If  $n$  is 20, there are more than  $10^{18}$  tours. The 13,509-city problem, which is successfully solved by the algorithm described in this paper, contains about  $10^{50,000}$  possible tours. In comparison it may be noted that the number of elementary particles in the universe has been estimated to be ‘only’  $10^{87}$ .

It has been proven that TSP is a member of the set of NP-complete problems. This is a class of difficult problems whose time complexity is probably exponential. The members of the class are related so that if a polynomial time were found for one problem, polynomial time algorithms would exist for all of them. However, it is commonly believed that no such polynomial algorithm exists. Therefore, any attempt to construct a general algorithm for finding optimal solutions for the TSP in polynomial time must (probably) fail.

Algorithms for solving the TSP may be divided into two classes:

- *Exact* algorithms.
- *Approximate* (or *heuristic*) algorithms.

#### 2.2.1. Exact algorithms

The *exact* algorithms are guaranteed to find the optimal solution in a bounded number of steps.

The most effective exact algorithms are cutting-plane or facet-finding algorithms [3–5]. These algorithms are quite complex, with codes on the order of 10,000 lines. In addition, the algorithms are very demanding of computer power. For example, it took roughly 3–4 years of CPU time on a large network of computers to determine the

exact solution of a 7397-city problem [5]. The previously mentioned 13,509-city problem was solved over a period of three months using a cluster of 3 servers, a total of 12 processors, and 32 PCs [6].

### 2.2.2. Approximate algorithms

In contrast, the *approximate* algorithms obtain good solutions but do not guarantee that optimal solutions will be found. These algorithms are usually very simple and have (relatively) short running times. Some of the algorithms give solutions that in average differs only by a few percent from the optimal solution. Therefore, if a small deviation from optimum can be accepted, it may be appropriate to use an approximate algorithm.

The class of approximate algorithms may be subdivided into the following three classes:

- *Tour construction algorithms.*
- *Tour improvement algorithms.*
- *Composite algorithms.*

The *tour construction algorithms* gradually build a tour by adding a new city at each step. The *tour improvement algorithms* improve upon a tour by performing various exchanges. The *composite algorithms* combine these two features.

A simple example of a tour construction algorithm is the so-called *nearest-neighbor algorithm*: Start in an arbitrary city. As long as there are cities, that have not yet been visited, visit the nearest city that still has not appeared in the tour. Finally, return to the first city.

A simple example of a tour improvement algorithm is the so-called *2-opt algorithm*: Start with a given tour. Replace 2 links of the tour with 2 other links in such a way that the new tour length is shorter. Continue in this way until no more improvements are possible.

Fig. 1 illustrates a 2-opt exchange of links, a so-called *2-opt move*. Note that a 2-opt move keeps the tour feasible and corresponds to a reversal of a subsequence of the cities.

A generalization of this simple principle forms the basis for one the most effective approximate algorithms for solving the symmetric TSP, the *Lin–Kernighan algorithm* [1].

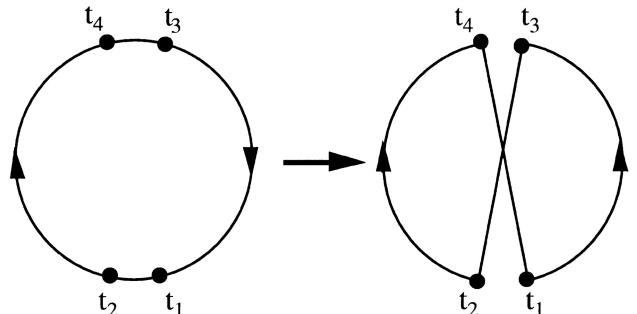


Fig. 1. A 2-opt move.

## 3. The Lin–Kernighan algorithm

### 3.1. The basic algorithm

The 2-opt algorithm is a special case of the  $\lambda$ -opt algorithm [7], where in each step  $\lambda$  links of the current tour are replaced by  $\lambda$  links in such a way that a shorter tour is achieved. In other words, in each step a shorter tour is obtained by deleting  $\lambda$  links and putting the resulting paths together in a new way, possibly reversing one or more of them.

The  $\lambda$ -opt algorithm is based on the concept  $\lambda$ -opt:

A tour is said to be  $\lambda$ -optimal (or simply  $\lambda$ -opt) if it is impossible to obtain a shorter tour by replacing any  $\lambda$  of its links by any other set of  $\lambda$  links.

From this definition it is obvious that any  $\lambda$ -opt tour is also  $\lambda'$ -opt for  $1 \leq \lambda' \leq \lambda$ . It is also easy to see that a tour containing  $n$  cities is optimal if and only if it is  $n$ -opt.

In general, the larger the value of  $\lambda$ , the more likely it is that the final tour is optimal. For fairly large  $\lambda$  it appears, at least intuitively, that a  $\lambda$ -opt tour should be optimal.

Unfortunately, the number of operations to test all  $\lambda$ -exchanges increases rapidly as the number of cities increases. In a naive implementation the testing of a  $\lambda$ -exchange has a time complexity of  $O(n^\lambda)$ . Furthermore, there is no non-trivial upper bound of the number of  $\lambda$ -exchanges. As a result, the values  $\lambda = 2$  and  $\lambda = 3$  are the most commonly

used. In one study the values  $\lambda=4$  and  $\lambda=5$  were used [8].

However, it is a drawback that  $\lambda$  must be specified in advance. It is difficult to know what  $\lambda$  to use to achieve the best compromise between running time and quality of solution.

Lin and Kernighan removed this drawback by introducing a powerful *variable*  $\lambda$ -opt algorithm. At each iteration step the algorithm examines, for ascending values of  $\lambda$ , whether an interchange of  $\lambda$  links may result in a shorter tour.

Given that the exchange of  $\lambda$  links is being considered, a series of tests is performed to determine whether  $\lambda+1$  link exchanges should be considered. This continues until some stopping conditions are satisfied.

At each step the algorithm considers a growing set of potential exchanges (starting with  $\lambda=2$ ). These exchanges are chosen in such a way that a feasible tour may be formed at any stage of the process. If the exploration succeeds in finding a new shorter tour, then the actual tour is replaced with the new tour.

The Lin–Kernighan algorithm belongs to the class of so-called *local optimization algorithms* [9, 10]. The algorithm is specified in terms of *exchanges* (or *moves*) that can convert one tour into another. Given a feasible tour, the algorithm repeatedly performs exchanges that reduce the length of the current tour, until a tour is reached for which no exchange yields an improvement. This process may be repeated many times from initial tours generated in some randomized way. The algorithm is described below in more detail.

Let  $T$  be the current tour. At each iteration step the algorithm attempts to find two sets of links,

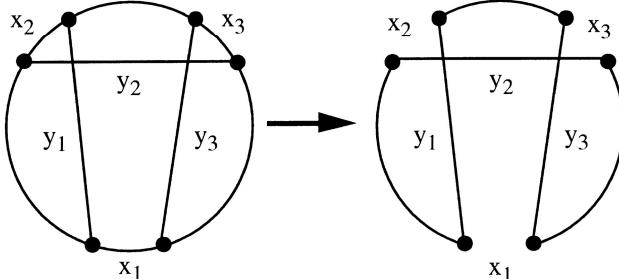


Fig. 2. A 3-opt move.

$X = \{x_1, \dots, x_r\}$  and  $Y = \{y_1, \dots, y_r\}$ , such that, if the links of  $X$  are deleted from  $T$  and replaced by the links of  $Y$ , the result is a better tour. This interchange of links is called a  $r$ -opt move. Fig. 2 illustrates a 3-opt move.

The two sets  $X$  and  $Y$  are constructed element by element. Initially  $X$  and  $Y$  are empty. In step  $i$  a pair of links,  $x_i$  and  $y_i$ , are added to  $X$  and  $Y$ , respectively.

In order to achieve a sufficient efficient algorithm, only links that fulfill the following criteria may enter  $X$  and  $Y$ .

(1) *The sequential exchange criterion:*  $x_i$  and  $y_i$  must share an endpoint, and so must  $y_i$  and  $x_{i+1}$ . If  $t_1$  denotes one of the two endpoints of  $x_1$ , we have in general:  $x_i = (t_{2i-1}, t_{2i})$ ,  $y_i = (t_{2i}, t_{2i+1})$  and  $x_{i+1} = (t_{2i+1}, t_{2i+2})$  for  $i \geq 1$ . See Fig. 3.

As seen, the sequence  $(x_1, y_1, x_2, y_2, x_3, \dots, x_r, y_r)$  constitutes a chain of adjoining links.

A necessary (but not sufficient) condition that the exchange of links  $X$  with links  $Y$  results in a tour is that the chain is closed, i.e.,  $y_r = (t_{2r}, t_1)$ . Such an exchange is called *sequential*.

Generally, an improvement of a tour may be achieved as a sequential exchange by a suitable numbering of the affected links. However, this is not always the case. Fig. 4 shows an example where a sequential exchange is not possible.

(2) *The feasibility criterion:* It is required that  $x_i = (t_{2i-1}, t_{2i})$  is chosen so that, if  $t_{2i}$  is joined to  $t_1$ , the resulting configuration is a tour. This feasibility criterion is used for  $i \geq 3$  and guarantees that it is possible to *close up* to a tour. This criterion was

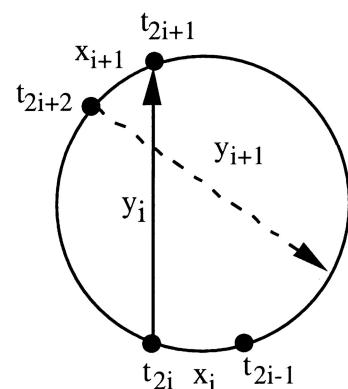


Fig. 3. Restricting the choice of  $x_i$ ,  $y_i$ ,  $x_{i+1}$ , and  $y_{i+1}$ .

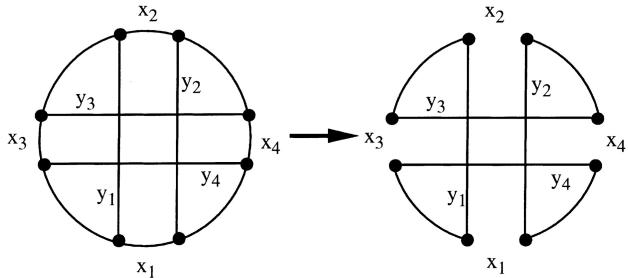


Fig. 4. Non-sequential exchange ( $r = 4$ ).

included in the algorithm both to reduce running time and to simplify the coding.

(3) *The positive gain criterion:* It is required that  $y_i$  is always chosen so that the gain,  $G_i$ , from the proposed set of exchanges is positive. Suppose  $g_i = c(x_i) - c(y_i)$  is the gain from exchanging  $x_i$  with  $y_i$ . Then  $G_i$  is the sum  $g_1 + g_2 + \dots + g_i$ .

This stop criterion plays a great role in the efficiency of the algorithm. The demand that every partial sum,  $G_i$ , must be positive seems immediately to be too restrictive. That this, however, is not the case, follows from the following simple fact: If a sequence of numbers has a positive sum, there is a cyclic permutation of these numbers such that every partial sum is positive. The proof is simple and can be found in [1].

(4) *The disjunctivity criterion:* Finally, it is required that the sets  $X$  and  $Y$  are disjoint. This simplifies coding, reduces running time and gives an effective stop criterion.

### 3.2. Lin and Kernighan's refinements

A bottleneck of the algorithm is the search for links to enter the sets  $X$  and  $Y$ . In order to increase efficiency, special care therefore should be taken to limit this search. Only exchanges that have a reasonable chance of leading to a reduction of tour length should be considered.

The basic algorithm as presented in the preceding section limits its search by using the following four rules:

- (1) Only sequential exchanges are allowed.
- (2) The provisional gain must be positive.
- (3) The tour can be 'closed' (with one exception,  $i = 2$ ).

(4) A previously broken link must not be added, and a previously added link must not be broken.

To limit the search even more Lin and Kernighan refined the algorithm by introducing the following rules:

(5) The search for a link to enter the tour,  $y_i = (t_{2i}, t_{2i+1})$ , is limited to the five nearest neighbors to  $t_{2i}$ .

(6) For  $i \geq 4$ , no link,  $x_i$ , on the tour must be broken if it is a common link of a small number (2–5) of solution tours.

(7) The search for improvements is stopped if the current tour is the same as a previous solution tour.

Rules 5 and 6 are heuristic rules. They are based on expectations of which links are likely to belong to an optimal tour. They save running time, but sometimes at the expense of not achieving the best possible solutions.

Rule 7 also saves running time, but has no influence on the quality of solutions being found. If a tour is the same as a previous solution tour, there is no point in attempting to improve it further. The time needed to check that no more improvements are possible (the checkout time) may therefore be saved.

In addition to these refinements, whose purpose is primarily to *limit* the search, Lin and Kernighan added some refinements whose purpose is primarily to *direct* the search. Where the algorithm has a choice of alternatives, heuristic rules are used to give priorities to these alternatives. In cases where only one of the alternatives must be chosen, the one with the highest priority is chosen. In cases where several alternatives must be tried, the alternatives are tried in descending priority order (using backtracking). To be more specific, the following rules are used:

(8) When link  $y_i$  ( $i \geq 2$ ) is to be chosen, each possible choice is given the priority  $c(x_{i+1}) - c(y_i)$ .

(9) If there are two alternatives for  $x_4$ , the one where  $c(x_4)$  is highest is chosen.

Rule 8 is a heuristic rule for ranking the links to be added to  $Y$ . The priority for  $y_i$  is the length of the next (unique) link to be broken,  $x_{i+1}$ , if  $y_i$  is included in the tour, minus the length of  $y_i$ . In this

way, the algorithm is provided with some lookahead. By maximizing the quantity  $c(x_{i+1}) - c(y_i)$ , the algorithm aims at breaking a long link and including a short link.

Rule 9 deals with the special situation where there are two choices for  $x_4$ . The rule gives preference to the longest link in this case. In three other cases, namely for  $x_1$ ,  $x_2$ , and sometimes  $x_3$ , there are two alternatives available. In these situations the algorithm examines both choices using backtracking (unless an improved tour was found). In their paper Lin and Kernighan do not specify the sequence in which the alternatives are examined.

As a last refinement, Lin and Kernighan included a limited defense against the situations where only non-sequential exchanges may lead to a better solution. After a local optimum has been found, the algorithm tests, among the links allowed to be broken, whether it is possible to make a further improvement by a non-sequential 4-opt change (as shown in Fig. 4).

#### 4. The modified Lin–Kernighan algorithm

Lin and Kernighan's original algorithm was reasonably effective. For problems with up to 50 cities, the probability of obtaining optimal solutions in a single trial was close to 100%. For problems with 100 cities the probability dropped to between 20% and 30%. However, by running a few trials, each time starting with a new random tour, the optimum for these problems could be found with nearly 100% assurance.

The algorithm was evaluated on a spectrum of problems, among these a drilling problem with 318 points. Due to computer-storage limitations, the problem was split into three smaller problems. A solution tour was obtained by solving the subproblems separately, and finally joining the three tours. At the time when Lin and Kernighan wrote their paper (1971), the optimum for this problem was unknown. Now that the optimum is known, it may be noted that their solution was 1.3% above optimum.

In the following, a modified and extended version of their algorithm is presented. The new al-

gorithm is a considerable improvement of the original algorithm. For example, for the mentioned 318-city problem the optimal solution is now found in a few trials (approximately 2), and in a very short time (about one second on a 300 MHz G3 Power Macintosh). In general, the quality of solutions achieved by the algorithm is very impressive.

The increase in efficiency is primarily achieved by a revision of Lin and Kernighan's heuristic rules for restricting and directing the search. Even if their heuristic rules seem natural, a critical analysis shows that they suffer from considerable defects.

##### 4.1. Candidate sets

A central rule in the original algorithm is the heuristic rule that restricts the inclusion of links in the tour to the five nearest neighbors to a given city (Rule 5 in Section 3.2). This rule directs the search against short tours and reduces the search effort substantially. However, there is a certain risk that the application of this rule may prevent the optimal solution from being found. If an optimal solution contains one link, which is not connected to the five nearest neighbors of its two end cities, then the algorithm will have difficulties in obtaining the optimum.

The inadequacy of this rule manifests itself particularly clearly in large problems. For example, for a 532-city problem [11] one of the links in the optimal solution is the 22nd nearest neighbor city for one of its end points. So in order to find the optimal solution to this problem, the number of nearest neighbors to be considered ought to be at least 22. Unfortunately, this enlargement of the set of candidates results in a substantial increase in running time.

The rule builds on the assumption that the shorter a link is, the greater is the probability that it belongs to an optimal tour. This seems reasonable, but used too restrictively it may result in poor tours.

In the following, a measure of *nearness* is described that better reflects the chances of a given link being a member of an optimal tour. This

measure, called  $\alpha$ -nearness, is based on sensitivity analysis using minimum spanning 1-trees.

First, some well-known graph theoretical terminology is reviewed.

Let  $G = (N, E)$  be a undirected weighted graph where  $N = \{1, 2, \dots, n\}$  is the set of nodes and  $E = \{(i, j) \mid i \in N, j \in N\}$  is the set of edges. Each edge  $(i, j)$  has associated a weight  $c(i, j)$ .

A path is a set of edges  $\{(i_1, i_2), (i_2, i_3), \dots, (i_{k-1}, i_k)\}$  with  $i_p \neq i_q$  for all  $p \neq q$ .

A cycle is a set of edges  $\{(i_1, i_2), (i_2, i_3), \dots, (i_{k-1}, i_1)\}$  with  $i_p \neq i_q$  for all  $p \neq q$  i.e., a path where the endpoints coincide.

A tour is a cycle where  $k = n$ .

For any subset  $S \subseteq E$  the length of  $S$ ,  $L(S)$ , is given by  $L(S) = \sum_{(i,j) \in S} c(i, j)$ .

An optimal tour is a tour of minimum length. Thus, the symmetric TSP can simply be formulated as: "Given a weighted graph  $G$ , determine an optimal tour of  $G$ ".

A graph  $G$  is said to be connected if it contains for any pair of nodes a path connecting them.

A tree is a connected graph without cycles. A spanning tree of a graph  $G$  with  $n$  nodes is a tree with  $n - 1$  edges from  $G$ . A minimum spanning tree is a spanning tree of minimum length.

Now the important concept of a 1-tree may be defined.

A 1-tree for a graph  $G = (N, E)$  is a spanning tree on the node set  $N \setminus \{1\}$  combined with two edges from  $E$  incident to node 1.

The choice of node 1 as a special node is arbitrary. Note that a 1-tree is not a tree since it contains a cycle (containing node 1; see Fig. 5).

A minimum 1-tree is a 1-tree of minimum length.

The degree of a node is the number of edges incident to the node.

It is easy to see [12,13] that:

1. an optimal tour is a minimum 1-tree where every node has degree 2;
2. if a minimum 1-tree is a tour, then the tour is optimal.

Thus, an alternative formulation of the symmetric TSP is: "Find a minimum 1-tree all whose nodes have degree 2".

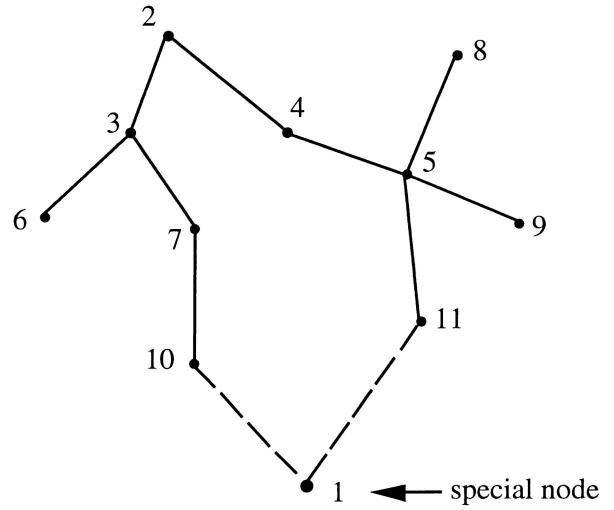


Fig. 5. A 1-tree.

Usually a minimum spanning tree contains many edges in common with an optimal tour. An optimal tour normally contains between 70% and 80% of the edges of a minimum 1-tree. Therefore, minimum 1-trees seem to be well suited as a heuristic measure of 'nearness'. Edges that belong, or 'nearly belong', to a minimum 1-tree, stand a good chance of also belonging to an optimal tour. Conversely, edges that are 'far from' belonging to a minimum 1-tree have a low probability of also belonging to an optimal tour. In the Lin-Kernighan algorithm these 'far' edges may be excluded as candidates to enter a tour. It is expected that this exclusion does not cause the optimal tour to be missed.

More formally, this measure of nearness is defined as follows:

Let  $T$  be a minimum 1-tree of length  $L(T)$  and let  $T^+(i, j)$  denote a minimum 1-tree required to contain the edge  $(i, j)$ . Then the  $\alpha$ -nearness of an edge  $(i, j)$  is defined as the quantity

$$\alpha(i, j) = L(T^+(i, j)) - L(T).$$

That is, given the length of (any) minimum 1-tree, the  $\alpha$ -nearness of an edge is the increase of length when a minimum 1-tree is required to contain this edge.

It is easy to verify the following two simple properties of  $\alpha$ :

- (1)  $\alpha(i, j) \geq 0$ .
- (2) If  $(i, j)$  belongs to some minimum 1-tree, then  $\alpha(i, j) = 0$ .

The  $\alpha$ -measure can be used to systematically identify those edges that could conceivably be included in an optimal tour, and disregard the remainder. These ‘promising edges’, called the *candidate set*, may, for example, consist of the  $k$   $\alpha$ -nearest edges incident to each node, and/or those edges having an  $\alpha$ -nearness below a specified upper bound.

In general, using the  $\alpha$ -measure for specifying the candidate set is much better than using nearest neighbors. Usually, the candidate set may be smaller, without degradation of the solution quality.

The use of  $\alpha$ -nearness in the construction of the candidate set implies computations of  $\alpha$ -values. The efficiency, both in time or space, of these computations is therefore important. The method is not of much practical value, if the computations are too expensive. In the following an algorithm is presented that computes all  $\alpha$ -values. The algorithm has time complexity  $O(n^2)$  and uses space  $O(n)$ .

Let  $G = (N, E)$  be a *complete* graph, that is, a graph where for all nodes  $i$  and  $j$  in  $N$  there is an edge  $(i, j)$  in  $E$ . The algorithm first finds a minimum 1-tree for  $G$ . This can be done by determination of a minimum spanning tree that contains the nodes  $\{2, 3, \dots, n\}$ , followed by the addition of the two shortest edges incident to node 1. The minimum spanning tree may, for example, be determined using Prim’s algorithm [14], which has a run time complexity of  $O(n^2)$ . The additional two edges may be determined in time  $O(n)$ . Thus, the complexity of this first part is  $O(n^2)$ .

Next, the nearness  $\alpha(i, j)$  is determined for all edges  $(i, j)$ . Let  $T$  be a minimum 1-tree. From the definition of a minimum spanning tree, it is easy to see that a minimum spanning tree  $T^+(i, j)$  containing the edge  $(i, j)$  may be determined from  $T$  using the following action rules:

- (a) If  $(i, j)$  belongs to  $T$ , then  $T^+(i, j)$  is equal to  $T$ .

(b) Otherwise, if  $(i, j)$  has 1 as end node ( $i = 1 \vee j = 1$ ), then  $T^+(i, j)$  is obtained from  $T$  by replacing the longest of the two edges of  $T$  incident to node 1 with  $(i, j)$ .

(c) Otherwise, insert  $(i, j)$  in  $T$ . This creates a cycle containing  $(i, j)$  in the spanning tree part of  $T$ . Then  $T^+(i, j)$  is obtained by removing the longest of the other edges on this cycle.

Cases a and b are simple. With a suitable representation of 1-trees they can both be treated in constant time.

Case c is more difficult to treat efficiently. The number of edges in the produced cycles is  $O(n)$ . Therefore, with a suitable representation it is possible to treat each edge with time complexity  $O(n)$ . Since  $O(n^2)$  edges must be treated this way, the total time complexity becomes  $O(n^3)$ , which is unsatisfactory.

However, it is possible to obtain a total complexity of  $O(n^2)$  by exploiting a simple relation between the  $\alpha$ -values [15,16].

Let  $\beta(i, j)$  denote the length of the edge to be removed from the spanning tree when edge  $(i, j)$  is added. Thus  $\alpha(i, j) = c(i, j) - \beta(i, j)$ . Then the following fact may be exploited (see Fig. 6). If  $(j_1, j_2)$  is an edge of the minimum spanning tree,  $i$  is one of the remaining nodes and  $j_1$  is on that cycle that arises by adding the edge  $(i, j_2)$  to the tree, then  $\beta(i, j_2)$  may be computed as the maximum of  $\beta(i, j_1)$  and  $c(j_1, j_2)$ .

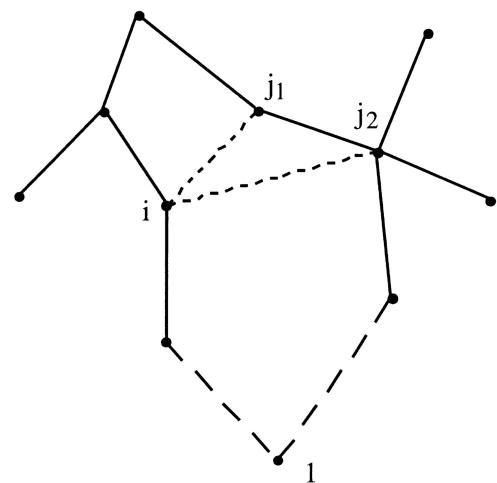


Fig. 6.  $\beta(i, j_2)$  may be computed from  $\beta(i, j_1)$ .

Thus, for a given node  $i$  all the values  $\beta(i,j), j = 1, 2, \dots, n$ , can be computed with a time complexity of  $O(n)$ , if only the remaining nodes are traversed in a suitable sequence. It can be seen that such a sequence is produced as a byproduct of Prim's algorithm for constructing minimum spanning trees, namely a topological order, in which every node's descendants in the tree are placed after the node. The total complexity now becomes  $O(n^2)$ .

Fig. 7 sketches in C-style notation an algorithm for computing  $\beta(i,j)$  for  $i \neq 1, j \neq 1, i \neq j$ . The algorithm assumes that the father of each node  $j$  in the tree,  $\text{dad}[j]$ , precedes the node (i.e.,  $\text{dad}[j] = i \Rightarrow i < j$ ).

Unfortunately this algorithm needs space  $O(n^2)$  for storing  $\beta$ -values. Some space may be saved by storing the  $c$ - and  $\beta$ -values in one quadratic matrix, so that, for example, the  $c$ -values are stored in the lower triangular matrix, while the  $\beta$ -values are stored in the upper triangular matrix. For large values of  $n$ , however, storage limitations may make this approach impractical.

Half of the space may be saved if the  $c$ -values are not stored, but computed when needed (for

```
for (i = 2; i < n; i++) {
    β[i][i] = - ;
    for (j = i+1; j <= n; j++)
        β[i][j] = β[j][i] = max(β[i][dad[j]], c(j,dad[j]));
}
```

Fig. 7. Computation of  $\beta(i,j)$  for  $i \neq 1, j \neq 1, i \neq j$ .

```
for (i = 2; i <= n; i++)
    mark[i] = 0;
for (i = 2; i <= n; i++) {
    b[i] = - ;
    for (k = i; k != 2; k = j) {
        j = dad[k];
        b[j] = max(b[k], c(k,j));
        mark[j] = i;
    }
    for (j = 2; j <= n; j++) {
        if (j != i) {
            if (mark[j] != i)
                b[j] = max(b[dad[j]], c(j,dad[j]));
            /* α(i,j) is now available as c(i,j) - b[j] */
        }
    }
}
```

Fig. 8. Space efficient computation of  $\alpha$ .

example as Euclidean distances). The question is whether it is also possible to save the space needed for the  $\beta$ -values. At first sight it would seem that the  $\beta$ -values must be stored in order to achieve  $O(n^2)$  time complexity for their computation. That this is not the case will now be demonstrated.

The algorithm, given in Fig. 8, uses two one-dimensional auxiliary arrays,  $b$  and  $\text{mark}$ . Array  $b$  corresponds to the  $\beta$ -matrix but only contains  $\beta$ -values for a given node  $i$ , i.e.,  $b[j] = \beta(i,j)$ . Array  $\text{mark}$  is used to indicate that  $b[j]$  has been computed for node  $i$ .

The determination of  $b[j]$  is done in two phases. First,  $b[j]$  is computed for all nodes  $j$  on the path from node  $i$  to the root of the tree (node 2). These nodes are marked with  $i$ . Next, a forward pass is used to compute the remaining  $b$ -values. The  $\alpha$ -values are available in the inner loop.

It is easy to see that this algorithm has time complexity  $O(n^2)$  and uses space  $O(n)$ .

The  $\alpha$ -values provide a good estimate of the edges' chances of belonging to an optimal tour. The smaller  $\alpha$  is for an edge, the more promising is this edge. Using  $\alpha$ -nearness it is often possible to limit the search to relative few of the  $\alpha$ -nearest neighbors of a node, and still obtain an optimal tour. Computational tests have shown that the  $\alpha$ -measure provides a better estimate of the likelihood of an edge being optimal than the usual  $c$ -measure. For example, for the 532-city problem the worst case is an optimal edge being the 22nd  $c$ -nearest edge for a node, whereas the worst case when using the  $\alpha$ -measure is an optimal edge being the 14th  $\alpha$ -nearest. The average rank of the optimal edges among the candidate edges is reduced from 2.4 to 2.1.

This seems to be quite satisfactory. However, the  $\alpha$ -measure can be improved substantially by making a simple transformation of the original cost matrix. The transformation is based on the following observations [13]:

1. Every tour is a 1-tree. Therefore the length of a minimum 1-tree is a lower bound on the length of an optimal tour.
2. If the length of all edges incident to a node are changed with the same amount,  $\pi$ , any optimal

tour remains optimal. Thus, if the cost matrix  $C = (c_{ij})$  is transformed to  $D = (d_{ij})$ , where

$$d_{ij} = c_{ij} + \pi_i + \pi_j,$$

then an optimal tour for the  $D$  is also an optimal tour for  $C$ . The length of every tour is increased by  $2\sum\pi_i$ . The transformation leaves the TSP invariant, but usually changes the minimum 1-tree.

3. If  $T_\pi$  is a minimum 1-tree with respect to  $D$ , then its length,  $L(T_\pi)$ , is a lower bound on the length of an optimal tour for  $D$ . Therefore  $w(\pi) = L(T_\pi) - 2\sum\pi_i$  is lower bound on the length of an optimal tour for  $C$ .

The aim is now to find a transformation,  $C \rightarrow D$ , given by the vector  $\pi = (\pi_1, \pi_2, \dots, \pi_n)$ , that maximizes the lower bound  $w(\pi) = L(T_\pi) - 2\sum\pi_i$ .

If  $T_\pi$  becomes a tour, then the exact optimum has been found. Otherwise, it appears, at least intuitively, that if  $w(\pi) > w(0)$ , then  $\alpha$ -values computed from  $D$  are better estimates of edges being optimal than  $\alpha$ -values computed from  $C$ .

A suitable method for maximizing  $w(\pi)$  is *subgradient optimization* [13]. It is an iterative method in which the maximum is approximated by step-wise changes of  $\pi$ .

At each step  $\pi$  is changed in the direction of the subgradient, i.e.,  $\pi^{k+1} = \pi^k + t^k v^k$ , where  $v^k$  is a subgradient vector, and  $t^k$  is a positive scalar, called the *step size*.

For the actual maximization problem it can be shown that  $v^k = d^k - 2$  is a subgradient vector, where  $d^k$  is a vector having as its elements the degrees of the nodes in the current minimum 1-tree. This subgradient makes the algorithm strive towards obtaining minimum 1-trees with node degrees equal to 2, i.e., minimum 1-trees that are tours. Edges incident to a node with degree 1 are made shorter. Edges incident to a node with degree greater than 2 are made longer. Edges incident to a node with degree 2 are not changed.

The  $\pi$ -values are often called *penalties*. The determination of a (good) set of penalties is called an *ascent*.

The choice of step size is a very crucial decision from the viewpoint of algorithmic efficiency or

even adequacy. Whether convergence is guaranteed is often not important, as long as good approximations can be obtained in a short time.

No general methods to determine an optimum strategy for the choice of step size are known. However, many strategies have been suggested that are quite effective in practice [17–22]. These strategies are heuristics, and different variations have different effects on different problems. In the present implementation of the modified Lin–Kernighan algorithm the following strategy was chosen (inspired by Crowder [18] and Helbig-Hansen and Krarup [22]):

- The step size is constant for a fixed number of iterations, called a period.
- When a period is finished, both the length of the period and the step size are halved.
- The length of the first period is set to  $n/2$ , where  $n$  is the number of cities.
- The initial step size,  $t^0$ , is set to 1, but is doubled in the beginning of the first period until  $W$  does not increase, i.e.,  $w(\pi^k) \leq w(\pi^{k-1})$ . When this happens, the step size remains constant for the rest of the period.
- If the last iteration of a period leads to an increment of  $W$ , then the period is doubled.
- The algorithm terminates when either the step size, the length of the period or  $v^k$  becomes zero.

Furthermore, the basic subgradient algorithm has been changed on two points (inspired by Held and Karp [13]):

- The updating of  $\pi$ , i.e.,  $\pi^{k+1} = \pi^k + t^k v^k$ , is replaced by

$$\pi_{k+1} = \pi^k + t^k(0.7v^k + 0.3v^{k-1}),$$

$$\text{where } v^{-1} = v^0.$$

- The special node for the 1-tree computations is not fixed. A minimum 1-tree is determined by computing a minimum spanning tree and then adding an edge corresponding to the second nearest neighbor of one of the leaves of the tree. The leaf chosen is the one that has the longest second nearest neighbor distance.

Practical experiments have shown that these changes lead to better bounds.

Having found a penalty vector  $\pi$ , that maximizes  $w(\pi)$ , the transformation given by  $\pi$  of the

original cost matrix  $C$  will often improve the  $\alpha$ -measure substantially. For example, for the 532-city problem every edge of the optimal tour is among the 5  $\alpha$ -nearest neighbors for at least one of its endpoints. The improvement of the  $\alpha$ -measure reduces the average rank of the optimal edges among the candidate edges from 2.1 to 1.7. This is close to the ideal value of 1.5 (when every optimal edge has rank 1 or 2).

The transformation of the cost matrix has the effect that optimal edges come ‘nearer’, when measured by their  $\alpha$ -values. The transformation ‘conditions’ the problem, so to speak. Therefore, the transformed matrix is also used during the Lin–Kernighan search process. Most often the quality of the solutions is improved by this means.

The greatest advantage of the  $\alpha$ -measure, however, is its usefulness for the construction of the candidate set. By using the  $\alpha$ -measure the cardinality of the candidate set may generally be small without reducing the algorithm’s ability to find short tours. Thus, in all test problems the algorithm was able to find optimal tours using as candidate edges the 5  $\alpha$ -nearest edges incident to each node. Most of the problems could even be solved when search was restricted to only the 4  $\alpha$ -nearest edges.

The candidate edges of each node are sorted in ascending order of their  $\alpha$ -values. If two edges have the same  $\alpha$ -value, the one with the smallest cost,  $c_{ij}$ , comes first. This ordering has the effect that candidate edges are considered for inclusion in a tour according to their ‘promise’ of belonging to an optimal tour. Thus, the  $\alpha$ -measure is not only used to *limit* the search, but also to *focus* the search on the most promising areas.

To speed up the search even more, the algorithm uses a *dynamic* ordering of the candidates. Each time a shorter tour is found, all edges shared by this new tour and the previous shortest tour become the first two candidate edges for their end nodes.

This method of selecting candidates was inspired by Stewart [23], who demonstrated how minimum spanning trees could be used to accelerate 3-opt heuristics. Even when subgradient optimization is not used, candidate sets based on minimum spanning trees usually produce better results than nearest neighbor candidate sets of the same size.

Johnson [9] in an alternative implementation of the Lin–Kernighan algorithm used precomputed candidate sets that usually contained more than 20 (ordinary) nearest neighbors of each node. The problem with this type of candidate set is that the candidate subgraph need not be connected even when a large fraction of all edges is included. This is, for example, the case for geometrical problems in which the point sets exhibit clusters. In contrast, a minimum spanning tree is (by definition) always connected.

Other candidate sets may be considered. An interesting candidate set can be obtained by exploiting the Delaunay graph [24,25]. The Delaunay graph is connected and may be computed in linear time, on the average. A disadvantage of this approach, however, is that candidate sets can only be computed for geometric problem instances. In contrast, the  $\alpha$ -measure is applicable in general.

#### 4.2. Breaking of tour edges

A candidate set is used to prune the search for edges,  $Y$ , to be included in a tour. Correspondingly, the search of edges,  $X$ , to be excluded from a tour may be restricted. In the actual implementation the following simple, yet very effective, pruning rules are used:

1. The first edge to be broken,  $x_1$ , must not belong to the currently best solution tour. When no solution tour is known, that is, during the determination of the very first solution tour,  $x_1$  must not belong to the minimum 1-tree.
2. The last edge to be excluded in a basic move must not previously have been included in the current chain of basic moves.

The first rule prunes the search already at level 1 of the algorithm, whereas the original algorithm of Lin and Kernighan prunes at level 4 and higher, and only if an edge to be broken is a common edge of a number (2–5) of solution tours. Experiments have shown that the new pruning rule is more effective. In addition, it is easier to implement.

The second rule prevents an infinite chain of moves. The rule is a relaxation of Rule 4 in Section 3.2.

#### 4.3. Basic moves

Central in the Lin–Kernighan algorithm is the specification of allowable moves, that is, which subset of  $r$ -opt moves to consider in the attempt to transform a tour into a shorter tour.

The original algorithm considers only  $r$ -opt moves that can be decomposed into a 2- or 3-opt move followed by a (possibly empty) sequence of 2-opt moves. Furthermore, the  $r$ -opt move must be *sequential* and *feasible*, that is, it must be a connected chain of edges where edges removed alternate with edges added, and the move must result in a feasible tour. Two minor deviations from this general scheme are allowed. Both have to do with 4-opt moves. First, in one special case the first move of a sequence may be a sequential 4-opt move; the following moves must still be 2-opt moves. Second, non-sequential 4-opt moves are tried when the tour can no longer be improved by sequential moves (see Fig. 4).

The new modified Lin–Kernighan algorithm revises this basic search structure on several points.

First and foremost, the basic move is now a sequential 5-opt move. Thus, the moves considered by the algorithm are sequences of one or more 5-opt moves. However, the construction of a move is stopped immediately if it is discovered that a close up of the tour results in a tour improvement. In this way the algorithm attempts to ensure 2-, 3-, 4- as well as 5-opt.

Using a 5-opt move as the basic move broadens the search and increases the algorithm's ability to find good tours, at the expense of an increase of running times. However, due to the use of small candidate sets, run times are only increased by a small factor. Furthermore, computational experiments have shown that backtracking is no longer necessary in the algorithm (except, of course, for the first edge to be excluded,  $x_1$ ). The removal of backtracking reduces runtime and does not degrade the algorithm's performance significantly. In addition, the implementation of the algorithm is greatly simplified.

The new algorithm's improved performance compared with the original algorithm is in accordance with observations made by Christofides and

Eilon [8]. They observed that 5-opt should be expected to yield a relatively superior improvement over 4-opt compared with the improvement of 4-opt over 3-opt.

Another deviation from the original algorithm is found in the examination of non-sequential exchanges. In order to provide a better defense against possible improvements consisting of non-sequential exchanges, the simple non-sequential 4-opt move of the original algorithm has been replaced by a more powerful set of non-sequential moves.

This set consists of:

- any non-feasible 2-opt move (producing two cycles) followed by any 2- or 3-opt move, which produces a feasible tour (by joining the two cycles);
- any non-feasible 3-opt move (producing two cycles) followed by any 2-opt move, which produces a feasible tour (by joining the two cycles).

As seen, the simple non-sequential 4-opt move of the original algorithm belongs to this extended set of non-sequential moves. However, by using this set of moves, the chances of finding optimal tours are improved. By using candidate sets and the “positive gain criterion” the time for the search for such non-sequential improvements of the tour is small relative to the total running time.

Unlike the original algorithm the search for non-sequential improvements is not only seen as a post optimization maneuver. That is, if an improvement is found, further attempts are made to improve the tour by ordinary sequential as well as non-sequential exchanges.

#### 4.4. Initial tours

The Lin–Kernighan algorithm applies edge exchanges several times to the same problem using different initial tours.

In the original algorithm the initial tours are chosen at random. Lin and Kernighan concluded that the use of a construction heuristic only wastes time. Besides, construction heuristics are usually deterministic, so it may not be possible to get more than one solution.

However, the question of whether or not to use a construction heuristic is not that simple to answer. Adrabsinsky and Syslo [26], for instance, found that the farthest insertion construction heuristic was capable of producing good initial tours for the Lin-Kernighan algorithm. Perttunen [27] found that the Clarke and Wright savings heuristic [28] in general improved the performance of the algorithm. Reinelt [25] also found that it is better not to start with a random tour. He proposed using locally good tours containing some major errors, for example the heuristics of Christofides [29]. However, he also observed that the difference in performance decreases with more elaborate versions of the Lin-Kernighan algorithm.

Experiments with various implementations of the new modified Lin-Kernighan algorithm have shown that the quality of the final solutions does not depend strongly on the initial tours. However, significant reduction in run time may be achieved by choosing initial tours that are close to being optimal.

In the present implementation the following simple construction heuristic is used:

1. Choose a random node  $i$ .
2. Choose a node  $j$ , not chosen before, as follows:

If possible, choose  $j$  so that

- (a)  $(i, j)$  is a candidate edge,
- (b)  $\alpha(i, j) = 0$ , and
- (c)  $(i, j)$  belongs to the current best tour.

Otherwise, if possible, choose  $j$  such that  $(i, j)$  is a candidate edge.

Otherwise, choose  $j$  among those nodes not already chosen.

3. Let  $i = j$ . If not all nodes have been chosen, then go to Step 2.

When more than one node may be chosen at Step 2, the node is chosen at random among the alternatives. The sequence of chosen nodes constitutes the initial tour.

This construction procedure is fast, and the diversity of initial solutions is large enough for the edge exchange heuristics to find good final solutions.

## 5. Implementation

The modified Lin-Kernighan algorithm has been implemented in the programming language

C. The software, approximately 4000 lines of code, is entirely written in ANSI C and portable across a number of computer platforms and C compilers. The following sections describe the most central techniques employed in the implementation.

### 5.1. Representation of tours and moves

The representation of tours is a central implementation issue. The data structure chosen may have great impact on the run time efficiency. It is obvious that the major bottleneck of the algorithm is the search for possible moves (edge exchanges) and the execution of such moves on a tour. Therefore, special care should be taken to choose a data structure that allows fast execution of these operations.

The data structure should support the following primitive operations:

1. find the predecessor of a node in the tour with respect to a chosen orientation (PRED);
2. find the successor of a node in the tour with respect to a chosen orientation (SUC);
3. determine whether a given node is between two other nodes in the tour with respect to a chosen orientation (BETWEEN);
4. make a move;
5. undo a sequence of tentative moves.

The necessity of the first three operations stems from the need to determine whether it is possible to ‘close’ the tour. The last two operations are necessary for keeping the tour up to date.

In the modified Lin-Kernighan algorithm a move consists of a sequence of basic moves, where each basic move is a 5-opt move ( $k$ -opt moves with  $k \leq 4$  are made in case an improvement of the tour is possible).

In order to simplify tour updating, the following fact may be used: Any  $r$ -opt move ( $r \geq 2$ ) is equivalent to a finite sequence of 2-opt moves [8,30]. In the case of 5-opt moves it can be shown that any 5-opt move is equivalent to a sequence of at most five 2-opt moves. Any 4-opt move as well as any 3-opt move is equivalent to a sequence of at most three 2-opt moves.

This is exploited as follows. Any move is executed as a sequence of one or more 2-opt moves.

During move execution, all 2-opt moves are recorded in a stack. A bad move is undone by unstacking the 2-opt moves and making the inverse 2-opt moves in this reversed sequence.

Thus, efficient execution of 2-opt moves is needed. A 2-opt move, also called a *swap*, consists of moving two edges from the current tour and reconnecting the resulting two paths in the best possible way (see Fig. 1). This operation is seen to reverse one of the two paths. If the tour is represented as an array of nodes, or as a doubly linked list of nodes, the reversal of the path takes time  $O(n)$ .

It turns out that data structures exist that allow logarithmic time complexity to be achieved [25,31–34]. These data structures, however, should not be selected without further notice. The time overhead of the corresponding update algorithms is usually large, and, unless the problem is large, typically more than 1000 nodes, update algorithms based on these data structures are outperformed by update algorithms based on the array and list structures. In addition, they are not simple to implement.

In the current implementation of the modified Lin–Kernighan algorithm a tour may be represented in two ways, either by a *doubly linked list*, or by a *two-level tree* [34]. The user can select one of these two representations. The doubly linked list is recommended for problems with fewer than 1000 nodes. For larger problems the two-level tree should be chosen.

A worst-case cost of  $O(\sqrt{n})$  per 2-opt move may be achieved using the two-level tree representation. This is currently the fastest and most robust representation on large instances that might arise in practice. The idea is to divide the tour into roughly  $\sqrt{n}$  segments. Each segment is maintained as a doubly linked list of nodes.

The segments are connected in a doubly linked list, and each segment contains a sequence number that represents its position in the list.

All the query operations (PRED, SUC and BETWEEN) are performed in constant time (as in the list representation, albeit with slightly larger constants), whereas the move operations have a worst-case cost of  $O(\sqrt{n})$  per move.

The implementation of the operations closely follows the suggestions given in [34]. See [34, pp. 444–446] for details.

## 5.2. Distance computations

A bottleneck in many applications is the computing of distances. For example, if Euclidean distances are used, a substantial part of run time may be spent in computing square roots.

If sufficient space is available, all distances may be computed once and stored in a matrix. However, for large problems, say more than 5000 nodes, this approach is usually not possible.

In the present implementation, distances are computed once and stored in a matrix, only if the problem is smaller than a specified maximum dimension. For larger problems, the following techniques are used to reduce run time.

(1) Each candidate edge including its length is associated to the node from which it emanates. A large fraction of edges considered during the solution process are candidate edges. Therefore, in many cases the length of an edge may be found by a simple search among the candidate edges associated with its end nodes.

(2) Computational cheap functions are used in calculating lower bounds for distances. For example, a lower bound for an Euclidean distance  $\sqrt{dx^2 + dy^2}$  may be quickly computed as the maximum of  $|dx|$  and  $|dy|$ . Often a reasonable lower bound for a distance is sufficient for deciding that there is no point in computing the true distance. This may, for example, be used for quickly deciding that a tentative move cannot possibly lead to a tour improvement. If the current gain, plus a lower bound for the distance of a closing edge, is not positive, then the tour will not be improved by this move.

(3) The number of distance computations is reduced by the caching technique described in [35]. When a distance between two nodes has been computed the distance is stored in a hash table. The hash index is computed from the identification numbers of the two nodes. Next time the distance between the same two nodes is to be computed, the table is consulted to see whether the distance is still available. See [35] for details. The effect of using the caching technique was measured in the solution of a 2392-node problem. Here optimum was found with about 70 times fewer ordinary distance calculations than without

the technique, and the running time was more than halved.

### 5.3. Reduction of checkout time

When the algorithm has found a local optimum, time is spent to check that no further progress is possible. This time, called the *checkout time*, can be avoided if the same local optimum has been found before. There is no point in attempting to find further improvements the situation has been previously been ‘checked out’. The checkout time often constitutes a substantial part of the running time. Lin and Kernighan report checkout times that are typically 30–50% of running time.

The modified algorithm reduces checkout time by using the following techniques:

(1) Moves in which the first edge  $(t_1, t_2)$  to be broken belongs to the currently best solution tour are not investigated.

(2) A hashing technique is used. A hash function maps tours to locations in a hash table. Each time a tour improvement has been found, the hash table is consulted to see whether the new tour happens to be local optimum found earlier. If this is the case, fruitless checkout time is avoided. This technique is described in detail in [36].

(3) The concept of the *don't look bit*, introduced by Bentley [37], is used. If for a given choice of  $t_1$  the algorithm previously failed to find an improvement, and if  $t_1$ 's tour neighbors have not changed since that time, then it is unlikely that an improving move can be made if the algorithm again looks at  $t_1$ . This is exploited as follows. Each node has a *don't look bit*, which initially is 0. The bit for node  $t_1$  is set to 1 whenever a search for an improving move with  $t_1$  fails, and it is set to 0 whenever an improving move is made in which it is an end node of one of its edges. In considering candidates for  $t_1$  all nodes whose don't look bit is 1 are ignored. This is done in maintaining a queue of nodes whose bits are zero.

### 5.4. Speeding up the ascent

Subgradient optimization is used to determine a lower bound for the optimum. At each step a

minimum 1-tree is computed. Since the number of steps may be large, it is important to speed up the computation of minimum 1-trees. For this purpose, the trees are computed in sparse graphs.

The first tree is computed in a complete graph. All remaining trees but the last are computed in a sparse subgraph determined by the  $\alpha$ -measure. The subgraph consists of a specified number of  $\alpha$ -nearest neighbor edges incident to each node.

Prim's algorithm [14] is used for computing minimum spanning trees. Therefore, to achieve a speed-up it is necessary to quickly find the shortest edge from a number of edges. In the current implementation a binary heap is used for this purpose.

The combination of these methods results in fast computation of minimum spanning trees, at least when the number of candidate edges allowed for each node is not too large. On the other hand, this number should not be so small that the lower bound computed by the ascent is not valid. In the present implementation, the number is 50 by default.

## 6. Computational results

The performance of an approximate algorithm such as the Lin–Kernighan algorithm can be evaluated in three ways:

1. by *worst-case* analysis;
2. by *probabilistic* (or *average-case*) analysis;
3. by *empirical* analysis.

The goal of *worst-case* analysis is to derive upper bounds for possible deviations from optimum; that is, to provide quality guarantees for results produced by the algorithm.

All known approximate algorithms for the TSP have rather poor worst-case behavior. Assume, for example, that the problems to be solved are metric (the triangle inequality holds). Then the approximate algorithm known to have the best worst-case behavior is the algorithm of Christofides [29]. This algorithm guarantees a tour length no more than 50% longer than optimum. For any  $r$ -opt algorithm, where  $r \leq n/4$  ( $n$  being the number of cities), problems may be

constructed such that the error is almost 100% [38]. For non-metric problems it can be proven that it is impossible to construct an algorithm of polynomial complexity which finds tours whose length is bounded by a constant multiple of the optimal tour length [39].

The purpose of the second method, *probabilistic* analysis, is to evaluate average behavior of the algorithms. For example, for an approximate TSP algorithm probability analysis can be used to estimate the expected error for large problem sizes.

The worst-case as well as the probability approach, however, have their drawbacks. The mathematics involved may be very complex and results achieved by these methods may often be of little use when solving *practical* instances of TSP. Statements concerning problems that almost certainly do not occur in practice ('pathological' problems, or problems with an 'infinite' number of cities) will often be irrelevant in connection with practical problem solving.

In this respect the third method, *empirical* analysis, seems more appropriate. Here the algorithm is executed on a number of test problems, and the results are evaluated, often in relation to optimal solutions. The test problems may be generated at random, or they may be constructed in a special way. If the test problems are representative for those problems the algorithm is supposed to solve, the computations are useful for evaluating the appropriateness of the algorithm.

Section 6.1 documents computational results of the modified Lin–Kernighan algorithm. The results include the qualitative performance and the run time efficiency of the current implementation. Run times are measured in seconds on a 300 MHz G3 Power Macintosh.

The performance of the implementation has been evaluated on the following spectrum of problems:

1. Symmetric problems.
2. Asymmetric problems.
3. Hamiltonian cycle problems.
4. Pathological problems.

Each problem has been solved by a number of independent *runs*. Each run consists of a series of *trials*, where in each trial a tour is determined by

the modified Lin–Kernighan algorithm. The trials of a run are not independent, since edges belonging to the best tour of the current run are used to prune the search.

In the experiments the number of runs varies. In problems with less than 1000 cities the number of runs is 100. In larger problems the number of runs is 10.

The number of trials in each run is equal to the dimension of the problem (the number of cities). However, for problems where optimum is known, the current series of trials is stopped if the algorithm finds optimum.

### 6.1. Symmetric problems

TSPLIB [40] is a library, which is meant to provide researchers with a set of sample instances for the TSP (and related problems). TSPLIB is publicly available via FTP from softlib.rice.edu and contains problems from various sources and with various properties.

At present, instances of the following problem classes are available: symmetric TSPs, asymmetric TSPs, Hamiltonian cycle problems, sequential ordering problems, and capacitated vehicle routing problems. Information on the length of optimal tours, or lower and upper bounds for this length, is provided (if available).

The library contains 111 symmetric TSPs, ranging from a problem with 14 cities to a problem with 85,900 cities. The performance evaluation that follows is based on those problems for which the optimum is known. Today there are 104 problems of this type in the library, the largest being a problem with 13,509 cities.

The test results are reported in Table 1.

The table gives the problem names along with the number of cities, the problem type, the optimal tour length, the ratio of runs succeeding in finding the optimal solution, the minimum and average number of trials in a run, the minimum and average gap between the length of the best solution obtained and optimum as a percentage of optimum, the preprocessing time, and the minimum and average time used for search in seconds per run.

Table 1

Performance for symmetric problems

Name	Cities	Type	Optimum	Success	Trials		Gap (%)			Time (s)		
					Min	Average	Min	Average	Max	Pre	Min	Average
a280	280	EUC_2D	2579	100/100	1	1.0	0.000	0.000	0.000	0.7	0.0	0.1
ali535	535	GEO	202310	100/100	1	8.0	0.000	0.000	0.000	4.0	0.2	0.7
att48	48	ATT	10628	100/100	1	1.0	0.000	0.000	0.000	0.1	0.0	0.0
att532	532	ATT	27686	98/100	1	66.2	0.000	0.001	0.072	2.9	0.2	3.6
bayg29	29	GEO	1610	100/100	1	1.0	0.000	0.000	0.000	0.0	0.0	0.0
bays29	29	GEO	2020	100/100	1	1.0	0.000	0.000	0.000	0.0	0.0	0.0
berlin52	52	EUC_2D	7542	*1/0	1	1.0	0.000	0.000	0.000	0.1	0.0	0.0
bier127	127	EUC_2D	118282	100/100	1	1.1	0.000	0.000	0.000	0.3	0.0	0.0
brazil58	58	MATRIX	25395	100/100	1	1.0	0.000	0.000	0.000	0.1	0.0	0.0
brg180	180	MATRIX	1950	100/100	1	3.3	0.000	0.000	0.000	0.3	0.0	0.0
burma14	14	GEO	3323	*1/0	1	1.0	0.000	0.000	0.000	0.0	0.0	0.0
ch130	130	EUC_2D	6110	100/100	1	3.0	0.000	0.000	0.000	0.2	0.0	0.1
ch150	150	EUC_2D	6528	62/100	1	62.9	0.000	0.023	0.077	0.3	0.0	0.4
d198	198	EUC_2D	15780	100/100	1	13.6	0.000	0.000	0.000	0.4	0.2	1.0
d493	493	EUC_2D	35002	100/100	1	45.8	0.000	0.000	0.000	2.5	0.2	3.2
d657	657	EUC_2D	48912	100/100	1	85.8	0.000	0.000	0.000	5.1	0.3	3.3
d1291	1291	EUC_2D	50801	8/10	152	584.2	0.000	0.033	0.167	19.0	23.1	59.8
d1655	1655	EUC_2D	62128	10/10	98	494.6	0.000	0.000	0.000	47.0	10.5	41.1
dantzig42	42	MATRIX	699	100/100	1	1.0	0.000	0.000	0.000	0.0	0.0	0.0
dsj1000	1000	CEIL_2D	18659688	7/10	90	514.1	0.000	0.035	0.116	12.3	13.1	55.1
eil51	51	EUC_2D	426	100/100	1	1.0	0.000	0.000	0.000	0.1	0.0	0.0
eil76	76	EUC_2D	538	100/100	1	1.0	0.000	0.000	0.000	0.1	0.0	0.0
eil101	101	EUC_2D	629	100/100	1	1.0	0.000	0.000	0.000	0.2	0.0	0.0
fl417	417	EUC_2D	11861	88/100	1	64.2	0.000	0.052	0.430	2.0	1.2	12.4
fl1400	1400	EUC_2D	20127	1/10	13	1261.3	0.000	0.162	0.199	36.5	13.1	583.3
fl1577	1577	EUC_2D	22249	2/10	134	1350.1	0.000	0.046	0.063	45.2	120.2	1097.5
fl3795	3795	EUC_2D	28772	4/10	1576	3167.5	0.000	0.072	0.191	220.9	2448.9	5666.4
fnl4461	4461	EUC_2D	182566	6/10	665	2767.3	0.000	0.001	0.003	332.7	284.1	1097.3
fri26	26	MATRIX	937	100/100	1	1.0	0.000	0.000	0.000	0.0	0.0	0.0
gil262	262	EUC_2D	2378	100/100	1	13.3	0.000	0.000	0.000	0.6	0.1	0.4
gr17	17	MATRIX	2085	*1/0	1	1.0	0.000	0.000	0.000	0.0	0.0	0.0
gr21	21	MATRIX	2707	*1/0	1	1.0	0.000	0.000	0.000	0.0	0.0	0.0
gr24	24	MATRIX	1272	100/100	1	1.0	0.000	0.000	0.000	0.0	0.0	0.0
gr48	48	MATRIX	5046	100/100	1	1.0	0.000	0.000	0.000	0.1	0.0	0.0
gr96	96	GEO	55209	100/100	1	10.2	0.000	0.000	0.000	0.2	0.0	0.1
gr120	120	MATRIX	6942	100/100	1	1.0	0.000	0.000	0.000	0.2	0.0	0.0
gr137	137	GEO	69853	100/100	1	1.0	0.000	0.000	0.000	0.4	0.0	0.0
gr202	202	GEO	40160	100/100	1	1.0	0.000	0.000	0.000	0.4	0.0	0.1
gr229	229	GEO	134602	12/100	6	216.6	0.000	0.009	0.010	0.7	0.2	1.7
gr431	431	GEO	171414	17/100	81	401.4	0.000	0.053	0.077	2.0	4.4	13.1
gr666	666	GEO	294358	30/100	25	531.9	0.000	0.026	0.040	4.8	2.0	18.8
hk48	48	MATRIX	11461	100/100	1	1.0	0.000	0.000	0.000	0.0	0.0	0.0
kroA100	100	EUC_2D	21282	100/100	1	1.0	0.000	0.000	0.000	0.1	0.0	0.0
kroB100	100	EUC_2D	22141	100/100	1	1.4	0.000	0.000	0.000	0.1	0.0	0.1
kroC100	100	EUC_2D	20749	100/100	1	1.0	0.000	0.000	0.000	0.2	0.0	0.0
kroD100	100	EUC_2D	21294	100/100	1	1.3	0.000	0.000	0.000	0.2	0.0	0.0
kroE100	100	EUC_2D	22068	99/100	1	10.7	0.000	0.002	0.172	0.2	0.0	0.2
kroA150	150	EUC_2D	26524	100/100	1	1.2	0.000	0.000	0.000	0.3	0.0	0.1
kroB150	150	EUC_2D	26130	55/100	1	97.7	0.000	0.003	0.008	0.3	0.1	0.8
kroA200	200	EUC_2D	29368	100/100	1	2.1	0.000	0.000	0.000	0.7	0.1	0.2
kroB200	200	EUC_2D	29437	100/100	1	2.0	0.000	0.000	0.000	0.4	0.0	0.1

Table 1 (Continued)

Name	Cities	Type	Optimum	Success	Trials		Gap (%)			Time (s)		
					Min	Average	Min	Average	Max	Pre	Min	Average
lin105	105	EUC_2D	14379	100/100	1	1.0	0.000	0.000	0.000	0.2	0.0	0.0
lin318	318	EUC_2D	42029	71/100	1	154.1	0.000	0.076	0.271	1.5	0.1	2.0
linhp318	318	EUC_2D	41345	100/100	1	2.4	0.000	0.000	0.000	1.2	0.0	0.1
nrw1379	1379	EUC_2D	56638	3/10	414	1148.5	0.000	0.006	0.009	23.3	20.2	69.3
p654	654	EUC_2D	34643	100/100	1	46.6	0.000	0.000	0.000	4.5	1.1	9.7
pa561	561	MATRIX	2763	99/100	1	40.8	0.000	0.001	0.072	3.2	0.2	3.5
pcb442	442	EUC_2D	50778	93/100	1	102.7	0.000	0.001	0.014	2.0	0.1	4.0
pcb1173	1173	EUC_2D	56892	8/10	2	475.2	0.000	0.002	0.009	15.7	0.6	14.6
pcb3038	3038	EUC_2D	137694	9/10	121	1084.7	0.000	0.000	0.004	139.8	39.1	323.7
pla7397	7397	CEIL_2D	23260728	7/10	1739	4588.4	0.000	0.001	0.004	1065.6	4507.9	13022.0
pr76	76	EUC_2D	108159	100/100	1	1.0	0.000	0.000	0.000	0.1	0.0	0.1
pr107	107	EUC_2D	44303	100/100	1	1.0	0.000	0.000	0.000	0.2	0.0	0.0
pr124	124	EUC_2D	59030	100/100	1	1.4	0.000	0.000	0.000	0.2	0.0	0.1
pr136	136	EUC_2D	96772	100/100	1	1.0	0.000	0.000	0.000	0.3	0.1	0.2
pr144	144	EUC_2D	58537	100/100	1	1.0	0.000	0.000	0.000	0.3	0.1	0.1
pr152	152	EUC_2D	73682	100/100	1	3.4	0.000	0.000	0.000	0.5	0.1	0.1
pr226	226	EUC_2D	80369	100/100	1	1.0	0.000	0.000	0.000	0.6	0.1	0.1
pr264	264	EUC_2D	49135	100/100	1	7.5	0.000	0.000	0.000	0.6	0.2	0.5
pr299	299	EUC_2D	48191	100/100	1	4.5	0.000	0.000	0.000	0.9	0.3	0.8
pr439	439	EUC_2D	107217	98/100	1	72.8	0.000	0.001	0.041	2.0	0.1	1.6
pr1002	1002	EUC_2D	259045	10/10	38	215.1	0.000	0.000	0.000	15.6	1.0	3.4
pr2392	2392	EUC_2D	378032	10/10	37	396.6	0.000	0.000	0.000	84.6	8.7	54.5
rat99	99	EUC_2D	1211	100/100	1	1.0	0.000	0.000	0.000	0.2	0.0	0.0
rat195	195	EUC_2D	2323	100/100	1	3.5	0.000	0.000	0.000	0.3	0.1	0.4
rat575	575	EUC_2D	6773	77/100	2	290.6	0.000	0.004	0.030	3.2	0.4	8.2
rat783	783	EUC_2D	8806	100/100	1	6.6	0.000	0.000	0.000	6.5	0.1	0.3
rd100	100	EUC_2D	7910	100/100	1	1.1	0.000	0.000	0.000	0.1	0.0	0.0
rd400	400	EUC_2D	15281	99/100	1	51.0	0.000	0.000	0.020	1.7	0.1	1.1
rl1304	1304	EUC_2D	252948	8/10	14	840.0	0.000	0.019	0.161	19.9	1.1	35.8
rl1323	1323	EUC_2D	270199	1/10	244	1215.1	0.000	0.018	0.048	20.8	10.9	51.6
rl1889	1889	EUC_2D	316536	4/10	1	1418.1	0.000	0.002	0.004	49.6	1.2	113.8
rl5915	5915	EUC_2D	565530	0/10	5915	5915.0	0.009	0.028	0.041	620.4	1886.5	2734.3
rl5934	5934	EUC_2D	556045	0/10	5934	5934.0	0.006	0.089	0.179	629.3	2704.0	3548.9
si175	175	MATRIX	21407	100/100	1	6.8	0.000	0.000	0.000	0.3	0.1	0.2
si535	535	MATRIX	48450	33/100	70	460.8	0.000	0.006	0.017	2.8	5.2	30.0
si1032	1032	MATRIX	92650	2/10	49	844.6	0.000	0.057	0.071	12.6	3.4	20.1
st70	70	EUC_2D	675	100/100	1	1.0	0.000	0.000	0.000	0.1	0.0	0.0
swiss42	42	MATRIX	1273	100/100	1	1.0	0.000	0.000	0.000	0.0	0.0	0.0
ts225	225	EUC_2D	126643	100/100	1	1.0	0.000	0.000	0.000	0.5	0.1	0.1
tsp225	225	EUC_2D	3919	100/100	1	5.8	0.000	0.000	0.000	0.5	0.1	0.3
u159	159	EUC_2D	42080	100/100	1	1.0	0.000	0.000	0.000	0.3	0.0	0.0
u574	574	EUC_2D	36905	91/100	1	111.4	0.000	0.007	0.081	3.3	0.2	3.2
u724	724	EUC_2D	41910	98/100	4	162.5	0.000	0.000	0.005	5.2	0.6	6.8
u1060	1060	EUC_2D	224094	9/10	2	305.9	0.000	0.000	0.003	12.2	1.6	38.0
u1432	1432	EUC_2D	152970	10/10	3	59.9	0.000	0.000	0.000	25.7	0.8	8.0
u1817	1817	EUC_2D	57201	2/10	905	1707.8	0.000	0.078	0.124	42.9	199.6	252.9
u2152	2152	EUC_2D	64253	5/10	491	1706.5	0.000	0.029	0.089	65.2	85.0	274.2
u2319	2319	EUC_2D	234256	10/10	1	3.2	0.000	0.000	0.000	86.2	0.5	2.6
ulysses16	16	GEO	6859	*1/0	1	1.0	0.000	0.000	0.000	0.0	0.0	0.0
ulysses22	22	GEO	7013	*1/0	1	1.0	0.000	0.000	0.000	0.0	0.0	0.0
usa13509	13509	EUC-2D	19982859	0/10	13509	13509.0	0.004	0.008	0.013	3785.2	39808.0	43814.5
vm1084	1084	EUC_2D	239297	7/10	8	425.2	0.000	0.007	0.022	12.7	1.9	28.6
vm1748	1748	EUC_2D	336556	4/10	65	1269.6	0.000	0.023	0.054	40.6	7.3	1016.1

The problem type specifies how the distances are given. The entry MATRIX indicates that distances are given explicitly in matrix form (as full or triangular matrix). The other entry names refer to functions for computing the distances from city coordinates. The entries EUC\_2D and CEIL\_2D both indicate that distances are the 2-dimensional Euclidean distances (they differ in their rounding method). GEO indicates geographical distances on the Earth's surface, and ATT indicates a special ‘pseudo-Euclidean’ distance function. All distances are integer numbers. See [40] for details.

The preprocessing time is the time used for reading the problem and determining the candidate set (using subgradient optimization).

For example, for the problem att532 of Padberg and Rinaldi [11] the optimal solution was determined in 98 runs out of 100. The minimum number of trials made to find optimum was 1. The average number of trials in a run was 66.2 (a run is stopped if the optimal solution is found, or the number of trials made equals the number of cities). The average gap between the length of the best solution obtained and the optimum as a percentage of the optimum was 0.001%. The maximum gap was 0.072%. The preprocessing time was 2.9 seconds. Finally, the minimum and average CPU time used in a run was 0.2 and 3.6 seconds, respectively.

For some of the small problems (berlin52, burma14, fri26, gr17, gr21, gr24, ulysses16 and ulysses22) optima were determined by subgradient optimization (marked with a \* in the Success column).

The table shows that optima were found with the default parameter settings for 101 of the problems.

Optima for the remaining 3 problems (rl5915, rl5934 and usa13,509) were obtained in the following way. First, for each of the problems a short tour was found using the default settings. The search was then continued, but the following rule was added to the set of pruning rules:

- The last link to be removed in a non-gainful move ( $x_5$  in a 5-opt move) must not belong to the shortest available tour.

Optima for rl5915, 5934 and usa13,509 were found by this means using approximately 2 days, 1 day and 3 days of CPU time, respectively.

In addition to these problems, TSPLIB contains 7 symmetric problems for which no optimum solutions are known today. The dimension of these problems varies from 2103 to 85,900 cities.

Table 2 lists for each of these problems the currently best known lower and upper bound (published in TSPLIB, June 1998).

When the new algorithm was executed on these problems, it found tour lengths equal to the best known upper bounds for 1 of the problems (d2103). However, for the remaining 6 problems, the algorithm was able to find tours *shorter* than the best known upper bounds. These new upper bounds are listed in Table 3.

The new upper bound for the largest of these problems, pla85900, was found using two weeks of CPU time.

It is difficult to predict the running time needed to solve a given problem with the algorithm. As can be seen from Table 1, the size alone is not enough. One problem may require much more running time than a larger problem.

However, some guidelines may be given. Random problems may be used to provide estimates of running times. By solving randomly generated problems of different dimensions and measuring the time used to solve these problems, it is possible to get an idea of how running time grows as a function of problem dimension.

An algorithm for randomly generating TSP with known optimal tours, described by Arthur and Frenetewey [41], was used for this purpose. The test results [42] indicated that the average running time of the algorithm is approximately  $O(n^{2.2})$ .

Another method often used in studying the average performance of TSP heuristics is to solve problem instances consisting of random points within a rectangle under the Euclidean metric. The instances are solved for increasing values of  $n$  and compared to the theoretical value for the *expected* length of an optimal tour ( $L_{\text{opt}}$ ). A well-known formula is  $L_{\text{opt}}(n, A) = K\sqrt{n}\sqrt{A}$  when  $n$  cities are distributed uniformly randomly over a rectangular

Table 2  
Problems in TSPLIB with unknown optimal solutions

Name	Cities	Type	Lower bound	Upper bound
brd14051	14051	EUC_2D	469272	469445
d2103	2103	EUC_2D	80330	80450
d15112	15112	EUC_2D	1572863	1573152
d18512	18512	EUC_2D	645092	645300
pla33810	33810	CEIL_2D	65960739	66116530
pla85900	85900	CEIL_2D	142244225	142482068
rl11849	11849	EUC_2D	923132	923368

Table 3  
Improved upper bounds

Name	New upper bound
brd14051	469395
d15112	1573089
d18512	645250
pla33810	66060236
pla85900	142416327
rl11849	923307

area of  $A$  units [43]. That is, the ratio of the optimal tour length to  $\sqrt{n}\sqrt{A}$  approaches a constant  $K$  for  $N \rightarrow \infty$ . Experiments of Johnson, McGeoch and Rothenberg [44] suggest that  $K$  is approximated by 0.7124.

Fig. 9 shows the results obtained when the modified Lin–Kernighan algorithm was used to solve such problems on a  $10^4 \times 10^4$  square for  $n$  ranging from 500 to 5000 and  $n$  increasing by 500. The figure depicts the tour length divided by  $\sqrt{n} * 10^4$ . These results are consistent with the estimate  $K \approx 0.7124$  for large  $n$ .

## 6.2. Asymmetric problems

The implemented algorithm is primarily intended for solving symmetric TSPs. However, any asymmetric problem may be transformed into a symmetric problem and therefore be solved by the algorithm.

The transformation method of Jonker and Volgenant [45] transforms an asymmetric problem with  $n$  nodes into a problem  $2n$  nodes. Let  $C = (c_{ij})$  denote the  $n \times n$  cost matrix of the asymmetric problem. Then let  $C' = (c'_{ij})$  be the  $2n \times 2n$  symmetric matrix computed as follows:

$$c'_{n+i,j} = c'_{j,n+i} = c_{i,j} \quad \text{for } i = 1, 2, \dots, n, \\ j = 1, 2, \dots, n, \\ \text{and } i \neq j$$

$$c'_{n+i,i} = c'_{i,n+i} = -M \quad \text{for } i = 1, 2, \dots, n \\ c'_{i,j} = M \quad \text{otherwise,}$$

where  $M$  is a sufficiently large number, e.g.,  $M = \max(c_{i,j})$ .

It is easy to prove that any optimal solution of the new symmetric problem corresponds to an optimal solution of the original asymmetric problem.

An obvious disadvantage of the transformation is that it doubles the size of the problem. Therefore, in practice it is more advantageous to use algorithms dedicated for solving asymmetric problems. However, as can be seen from Table 4 the performance of the modified Lin–Kernighan algorithm for asymmetric problems is quite impressive. The optimum was obtained for all

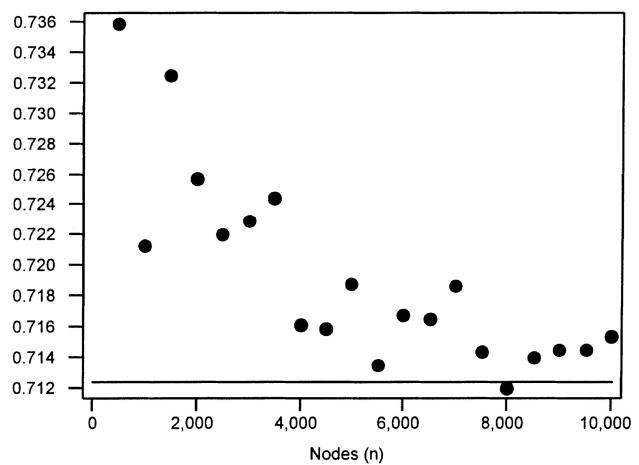


Fig. 9. Solutions of random Euclidean problems on a square.

Table 4  
Performance for asymmetric problems.

Name	Cities	Optimum	Success	Trials		Gap (%)			Time (s)		
				Min	Average	Min	Average	Max	Pre	Min	Average
br17	17	39	100/100	1	1.0	0.000	0.000	0.000	0.0	0.0	0.0
ft53	53	6905	100/100	1	8.0	0.000	0.000	0.000	0.1	0.0	0.0
ft70	70	38673	100/100	1	1.6	0.000	0.000	0.000	0.2	0.0	0.0
ftv33	34	1286	100/100	1	1.0	0.000	0.000	0.000	0.1	0.0	0.0
ftv35	36	1473	47/100	1	43.6	0.000	0.072	0.136	0.1	0.0	0.1
ftv38	39	1530	53/100	1	40.3	0.000	0.061	0.131	0.1	0.0	0.1
ftv44	45	1613	100/100	1	2.8	0.000	0.000	0.000	0.1	0.0	0.0
ftv47	48	1776	100/100	1	1.5	0.000	0.000	0.000	0.1	0.0	0.0
ftv55	56	1608	100/100	1	3.5	0.000	0.000	0.000	0.1	0.0	0.0
ftv64	65	1839	100/100	1	3.6	0.000	0.000	0.000	0.2	0.0	0.0
ftv70	71	1950	100/100	1	3.6	0.000	0.000	0.000	0.2	0.0	0.0
ftv170	171	2755	88/100	1	119.7	0.000	0.039	0.327	1.1	0.0	1.0
kro124p	100	36230	95/100	1	24.6	0.000	0.002	0.030	0.3	0.0	0.1
p43	43	5620	21/100	1	72.2	0.000	0.014	0.018	0.0	0.0	0.5
rbg323	323	1326	97/100	17	116.3	0.000	0.018	0.679	41.3	1.9	8.5
rbg358	358	1163	99/100	19	96.8	0.000	0.060	6.019	105.1	2.5	8.3
rbg403	403	2465	100/100	19	87.5	0.000	0.000	0.000	183.2	2.6	11.3
rbg443	443	2720	100/100	18	105.2	0.000	0.000	0.000	237.8	2.5	12.6
ry48p	48	1442	99/100	1	9.8	0.000	0.002	0.166	0.1	0.0	0.0

asymmetric problems of TSPLIB. The largest of these problems, rbg443, has 443 nodes (thus, the transformed problem has 886 nodes).

### 6.3. Hamiltonian cycle problems

The Hamiltonian cycle problem is the problem of deciding if a given undirected graph contains a cycle. The problem can be answered by solving a symmetric TSP in the complete graph where all edges of  $G$  have cost 0 and all other edges have a positive cost. Then  $G$  contains a Hamiltonian cycle if and only if an optimal tour has cost 0.

An interesting special Hamilton cycle problem is the so-called knight's-tour problem. A knight is to be moved around on a chessboard in such a way that all 64 squares are visited exactly once and the moves taken constitute a round trip on the board. Fig. 10 depicts one solution of the problem. The two first moves are shown with arrows.

The problem is an instance of the general “leaper” problem: “Can a  $\{r, s\}$ -leaper, starting at any square of a  $m \times n$  board, visit each square ex-

actly once and return to its starting square” [46]. The knight's-tour problem is the 1,2-leaper problem on a  $8 \times 8$  board.

Table 5 shows the performance statistics for a few leaper problems. A small C-program, included in TSPLIB, was used for generating the leaper graphs. Edges belonging to the graph cost 0, and the remaining edges cost 1.

The last of these problems has an optimal tour length of 18. All the other problems contain a Hamiltonian cycle, i.e., their optimum tour length is 0.

Table 6 shows the performance for the same problems, but now the edges of the leaper graph cost 100, and the remaining edges cost 101. Lin and Kernighan observed that knight's-tour problems with these edge costs were hard to solve for their algorithm [1].

As seen the new implementation is almost unaffected.

### 6.4. Pathological problems

The Lin–Kernighan algorithm is not always as effective as it seems to be with “random” or

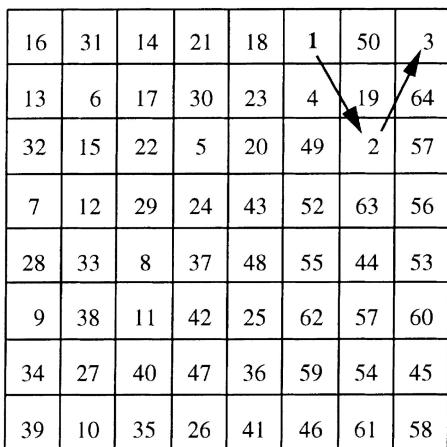


Fig. 10. One solution of the knight's-tour problem

"typical" problems. Papadimitriou and Steiglitz [47] have constructed a special class of instances of the TSP for which local search algorithms, such as the Lin–Kernighan algorithm, appears to be very ineffective. Papadimitriou and Steiglitz denote this class of problems as 'perverse'.

Each problem has  $n=8k$  nodes. There is exactly one optimal tour with cost  $n$ , and there are  $2^{k-1}(k-1)!$  tours that are next best, have arbitrary

large cost, and cannot be improved by changing fewer than  $3k$  edges. See [47] for a precise description of the constructed problems.

The difficulty of the problem class is illustrated in Table 7 showing the performance of the algorithm when subgradient optimization is left out. Note the results for the cases  $k=3$  and  $k=5$ . Here the optimum was frequently found, whereas the implementation of the Lin–Kernighan algorithm by Papadimitriou and Steiglitz was unable to discover the optimum even once.

However, all these problems can be solved without any search, if subgradient optimization is used (provided that the initial period is sufficiently large).

Table 8 shows the time (in seconds) to find optimal solutions by subgradient optimization.

## 7. Conclusions

This paper has described a modified Lin–Kernighan algorithm and its implementation. In the development of the algorithm great emphasis was put on achieving high quality solutions, preferably optimal solutions, in a reasonable short time.

Table 5  
Performance for leaper problems (Variant 1)

Problem	Success	Trials		Gap (%)		Time (s)	
		Min	Average	Max	Min	Min	Average
{1,2}, 8×8	100/100	1	1.0	0.000	0.000	0.0	0.0
{1,2}, 10×10	100/100	1	1.0	0.000	0.000	0.0	0.0
{1,2}, 20×20	100/100	1	2.1	0.000	0.000	0.0	0.0
{7,8}, 15×106	10/10	4	8.6	0.000	0.000	0.3	0.7
{6,7}, 13×76	100/100	1	1.1	0.000	0.000	0.3	0.5

Table 6  
Performance for leaper problems (Variant 2)

Problem	Success	Trials		Gap (%)		Time (s)	
		Min	Average	Max	Min	Min	Average
{1,2}, 8×8	100/100	1	1.0	0.000	0.000	0.0	0.0
{1,2}, 20×20	100/100	1	1.1	0.000	0.000	0.0	0.0
{7,8}, 15×106	10/10	3	10.0	0.000	0.000	0.4	1.0
{6,7}, 13×76	100/100	1	1.1	0.000	0.000	0.4	0.7

Table 7  
Performance for perverse problems (without subgradient optimization)

k	n	Success	Trials		Gap (%)		Time (s)	
			Min	Average	Max	Min	Min	Average
3	24	46/100	1	14.9	2479.5	8291.7	0.0	0.0
4	32	55/100	1	17.4	1732.8	6209.4	0.0	0.0
5	40	48/100	1	24.1	1430.4	4960.0	0.0	0.0
6	48	53/100	1	26.6	1045.1	4127.1	0.0	0.0
7	56	32/100	1	41.2	1368.2	3532.1	0.0	0.0
8	64	41/100	1	41.2	1040.2	3085.9	0.0	0.0
9	72	1/100	5	71.3	1574.1	2738.9	0.0	0.1
10	80	2/100	1	78.6	1700.5	4958.8	0.0	0.2

Table 8  
Time to find the optimal solutions for perverse problems (with subgradient optimization)

k	n	Time (s)
3	24	0.0
4	32	0.0
5	40	0.0
6	48	0.1
7	56	0.1
8	64	0.1
9	72	0.1
10	80	0.2

Achieving simplicity was of minor concern here. In comparison, the modified Lin–Kernighan algorithm of Mak and Morton [48] has a very simple algorithmic structure. However, this simplicity has been achieved with the expense of a reduced ability to find optimal solutions. Their algorithm does not even guarantee 2-opt optimality.

Computational experiments have shown that the new algorithm is highly effective. An optimal solution was obtained for all problems with a known optimum. This is remarkable, considering that the modified algorithm does not employ backtracking.

The running times were satisfactory for all test problems. However, since running time is approximately  $O(n^{2.2})$ , it may be impractical to solve very large problems. The current implementation seems to be feasible for problems with fewer than 100,000 cities (this depends, of course, of the available computer resources).

When distances are implicitly given, space requirements are  $O(n)$ . Thus, the space required for geometrical problems is linear. For such problems, not space, but run time, will usually be the limiting factor.

The effectiveness of the algorithm is primarily achieved through an efficient search strategy. The search is based on 5-opt moves restricted by carefully chosen candidate sets. The  $\alpha$ -measure, based on sensitivity analysis of minimum spanning trees, is used to define candidate sets that are small, but large enough to allow excellent solutions to be found (usually the optimal solutions).

## References

- [1] S. Lin, B.W. Kernighan, An effective heuristic algorithm for the traveling salesman problem, *Operations Research* 21 (1973) 498–516.
- [2] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys (Eds.), *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, Wiley, New York, 1985.
- [3] M.W. Padberg, G. Rinaldi, A branch-and-cut algorithm for the resolution of large-scalesymmetric traveling salesman problems, *SIAM Review* 33 (1991) 60–100.
- [4] M. Grötschel, O. Holland, Solution of large scale symmetric traveling salesman problems, *Mathematical Programming* 51 (1991) 141–202.
- [5] D. Applegate, R. Bixby, V. Chvátal, W. Cook, Finding cuts in the TSP (A preliminary report), DIMACS, Tech. Report 95-05, 1995.
- [6] D. Applegate, R. Bixby, V. Chvátal, W. Cook, Microelectronics Technology Alert, July 1998. [http://www.crcpc.ice.edu/CRPC/newsArchive/mta\\_7\\_24\\_98.html](http://www.crcpc.rice.edu/CRPC/newsArchive/mta_7_24_98.html).

- [7] S. Lin, Computer solutions of the traveling salesman problem, *Bell System Technical Journal* 44 (1965) 2245–2269.
- [8] N. Christofides, S. Eilon, Algorithms for large-scale traveling salesman problems, *Operations Research Quarterly* 23 (1972) 511–518.
- [9] D.S. Johnson, Local optimization and the traveling salesman problem, *Lecture Notes in Computer Science* 442 (1990) 446–461.
- [10] D.S. Johnson, L.A. McGeoch, The traveling salesman problem: A case study in local optimization, in: E.H.L. Aarts, J.K. Lenstra (Eds.), *Local Search in Combinatorial Optimization*, Wiley, New York, 1997.
- [11] M.W. Padberg, G. Rinaldi, Optimization of a 532-city symmetric traveling salesman problem by branch and cut, *Operations Research Letters* 6 (1987) 1–7.
- [12] M. Held, R.M. Karp, The traveling salesman problem and minimum spanning trees, *Operations Research* 18 (1970) 1138–1162.
- [13] M. Held, R.M. Karp, The traveling salesman problem and minimum spanning trees: Part II, *Mathematical Programming* 1 (1971) 16–25.
- [14] R.C. Prim, Shortest connection networks and some generalizations, *Bell System Technical Journal* 36 (1957) 1389–1401.
- [15] T. Volgenant, R. Jonker, The symmetric traveling salesman problem and edge exchanges  $i$  minimal 1-trees, *European Journal of Operational Research* 12 (1983) 394–403.
- [16] G. Carpaneto, M. Fischetti, P. Toth, New lower bounds for the symmetric traveling salesman problem, *Mathematical Programming* 45 (1989) 233–254.
- [17] M. Held, P. Wolfe, H.P. Crowder, Validation of subgradient optimization, *Mathematical Programming* 6 (1974) 62–88.
- [18] H.P. Crowder, Computational improvements of subgradient optimization, *IBM Research Report RC 4907* (21841) (1974).
- [19] P.M. Camerini, L. Fratta, F. Maffioli, On improving relaxation methods by modified gradient techniques, *Mathematical Programming Study* 3 (1976) 26–34.
- [20] M.S. Bazaraa, H.D. Sherali, On the choice of step size in subgradient optimization, *European Journal of Operational Research* 7 (1981) 380–388.
- [21] T. Volgenant, R. Jonker, A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation, *European Journal of Operational Research* 9 (1982) 83–89.
- [22] K.H. Helbig-Hansen, J. Krarup, Improvements of the Held-Karp algorithm for the symmetric traveling salesman problem, *Mathematical Programming* 7 (1974) 87–96.
- [23] W.R. Stewart Jr., Accelerated branch exchange heuristics for symmetric traveling salesman problems, *Networks* 17 (1987) 423–437.
- [24] G. Reinelt, Fast heuristics for large geometric traveling salesman problems, *ORSA Journal on Computing* 2 (1992) 206–217.
- [25] G. Reinelt, The traveling salesman: Computational solutions for TSP applications, *Lecture Notes in Computer Science* 840 (1994).
- [26] A. Adrabinski, M.M. Syslo, Computational experiments with some approximation algorithms for the traveling salesman problem, *Zastosowania Matematyki* 18 (1983) 91–95.
- [27] J. Perttunen, On the Significance of the initial solution in traveling salesman heuristics, *Journal of the Operational Research Society* 45 (1994) 1131–1140.
- [28] G. Clarke, J.W. Wright, Scheduling of vehicles from a central depot to a number of delivery points, *Operations Research* 12 (1964) 568–581.
- [29] N. Christofides, Worst Case Analysis of a New Heuristic for the Traveling Salesman Problem, Report 388. Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, 1976.
- [30] K.-T. Mak, A.J. Morton, Distances between traveling salesman tours, *Discrete Applied Mathematics* 58 (1995) 281–291.
- [31] D. Applegate, R.E. Bixby, V. Chvátal, W. Cook, Data Structures for the Lin-Kernighan Heuristic, Talk presented at the TSP-Workshop, CRCP, Rice University, 1990.
- [32] L. Hårs, Reversible-segment list, Report No 89596-OR, Forschungsinstitut für Discrete Mathematik, Bonn, 1989.
- [33] F. Margot, Quick updates for  $p$ -opt TSP heuristics, *Operations Research Letters* 11 (1992) 45–46.
- [34] M.L. Fredman, D.S. Johnson, L.A. McGeoch, Data structures for traveling salesmen, *Journal of Algorithms* 16 (1995) 432–479.
- [35] J.L. Bentley,  $K-d$  trees for semidynamic point sets, Sixth Annual ACM Symposium on Computational Geometry, Berkeley, CA, 1990, pp. 187–197.
- [36] O. Martin, S.W. Otto, E.W. Felten, Large-step markov chains for the traveling salesman problem, *Journal of Complex Systems* 5 (1991) 219–224.
- [37] J.L. Bentley, Fast algorithms for geometric traveling salesman problems, *ORSA Journal on Computing* 4 (1992) 347–411.
- [38] D.E. Rosenkrantz, R.E. Stearns, P.M. Lewis II, An analysis of several heuristics for the traveling salesman problem, *SIAM Journal on Computing* 6 (1977) 563–581.
- [39] S. Sahni, T. Gonzales,  $P$ -complete approximation algorithms, *Journal of the Association for Computing Machinery* 23 (1976) 555–565.
- [40] G. Reinelt, TSPLIB – A traveling salesman problem library, *ORSA Journal on Computing* 3 (4) (1991) 376–385.
- [41] J.L. Arthur, J.O. Frendewey, Generating traveling salesman problems with known optimal tours, *Journal of the Operational Research Society* 39 (1988) 153–159.
- [42] K. Helsgaun, An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic, *DATALOG-ISKE SKRIFTER*, Roskilde University, No. 81, 1998.
- [43] J. Beardswood, J.M. Hammersley, The shortest path through many points, in: *Proceedings of the Cambridge Philosophical Society*, 55, 1959, pp. 299–327.

- [44] D.S. Johnson, L.A. McGeoch, E.E. Rothenberg, Asymptotic experimental analysis for the Held–Karp traveling salesman problem, in: Proceedings of Seventh ACM SIAM Symposium on Discrete Algorithms Society of Industrial and Applied Mathematics, Philadelphia, York, 1996.
- [45] R. Jonker, T. Volgenant, Transforming asymmetric into symmetric traveling salesman problems, *Operations Research Letters* 2 (1983) 161–163.
- [46] D.E. Knuth, Leaper graphs, *The Mathematical Gazette* 78 (1994) 274–297.
- [47] C.H. Papadimitriou, K. Steiglitz, Some examples of difficult traveling salesman problems, *Operations Research* 26 (1978) 434–443.
- [48] K.-T. Mak, A.J. Morton, A modified Lin–Kernighan traveling salesman heuristic, *Operations Research Letters* 13 (1993) 127–132.