Math. Prog. Comp. (2009) 1:119–163
DOI 10.1007/s12532-009-0004-6

FULL LENGTH PAPER

General k-opt submoves for the Lin–Kernighan TSP heuristic

Keld Helsgaun

Received: 25 March 2009 / Accepted: 3 June 2009 / Published online: 1 July 2009

Abstract Local search with k-exchange neighborhoods, k-opt, is the most widely used heuristic method for the traveling salesman problem (TSP). This paper presents an effective implementation of k-opt in LKH-2, a variant of the Lin–Kernighan TSP heuristic. The effectiveness of the implementation is demonstrated with experiments on Euclidean instances ranging from 10,000 to 10,000,000 cities. The runtime of the method increases almost linearly with the problem size. LKH-2 is free of charge for academic and non-commercial use and can be downloaded in source code.

Keywords Traveling salesman problem · TSP · Lin–Kernighan · k-opt

Mathematics Subject Classification (2000) 90C27 · 90C35 · 90C59

1 Introduction

The traveling salesman problem (TSP) is one of the most widely studied problems in combinatorial optimization. Given a collection of cities and the cost of travel between each pair of them, the TSP is to find the cheapest way of visiting all of the cities and returning to the starting point. 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 ($i, j = 1, \ldots, n$), find a permutation $(i_1, i_2, i_3, \ldots, i_n)$ of the integers from 1 through n that minimizes the quantity
>
> $$c_{i_1 i_2} + c_{i_2 i_3} + \cdots + c_{i_n i_1}.$$

K. Helsgaun (✉)
Department of Communication, Business and Information Technologies, Roskilde University, 4000 Roskilde, Denmark
e-mail: keld@ruc.dk

The TSP may also be stated as the problem of finding a Hamiltonian cycle (tour) of minimum weight in an edge-weighted graph:

> Let $G = (N, E)$ be a weighted graph where $N = \{1, 2, \ldots, n\}$ 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 *cycle* is a set of edges $\{(i_1, i_2), (i_2, i_3), \ldots, (i_k, i_1)\}$ with $i_p \neq i_q$ for $p \neq q$. A *Hamiltonian cycle* (or *tour*) is a cycle where $k = n$. The *weight* (or *cost*) of a tour T is the sum $\sum_{(i,j)\in T} c(i, j)$. An *optimal tour* is a tour of minimum weight.

For surveys of the problem and its applications, the reader is referred to the excellent volumes edited by Lawler et al. [24] and Gutin and Punnen [11].

Local search with k-exchange neighborhoods, k-opt, is the most widely used heuristic method for the TSP. k-opt is a tour improvement algorithm, where in each step k links of the current tour are replaced by k links in such a way that a shorter tour is achieved.

It has been shown [7] that k-opt may take an exponential number of iterations and that the ratio of the length of an optimal tour to the length of a tour constructed by k-opt can be arbitrarily large when $k \leq n/2 - 5$. Such undesirable cases, however, are very rare when solving practical instances [31]. Usually, high-quality solutions are obtained in polynomial time. This is, for example, the case for the Lin–Kernighan heuristic, one of the most effective methods for generating optimal or near-optimal solutions for the symmetric TSP. High-quality solutions are often obtained, even though only a small part of the k-exchange neighborhood is searched.

In the original version of the Lin–Kernighan heuristic [25], the allowable k-exchanges (or k-opt moves) are restricted to those that can be decomposed into a 2- or 3-exchange followed by a (possibly empty) sequence of 2-exchanges. This restriction simplifies implementation, but it need not be the best design choice. This paper explores the effect of widening the search.

This paper describes LKH-2, an implementation of the Lin–Kernighan heuristic, which allows all those moves that can be decomposed into a sequence of k-exchanges for any k where $2 \leq k \leq n$. These k-exchanges may be sequential as well as non-sequential. LKH-2 is an extension and generalization of a previous version, LKH-1 [16], which uses a sequential 5-exchange as its basic move component (submove).

The rest of the paper is organized as follows. Section 2 gives an overview of the original Lin–Kernighan algorithm. Section 3 gives a short description of the first version of LKH, LKH-1. Section 4 presents the facilities of its successor LKH-2. Section 5 describes how k-opt submoves are implemented in LKH-2. The effectiveness of the implementation is reported in Sect. 6. Finally, the conclusions about the implementation are given in Sect. 7.

2 The original Lin–Kernighan algorithm (LK)

The Lin–Kernighan algorithm [25] belongs to the class of so-called *local search algorithms* [17,18,20]. A local search algorithm starts at some location in the search space and subsequently moves from the present location to a neighboring location.

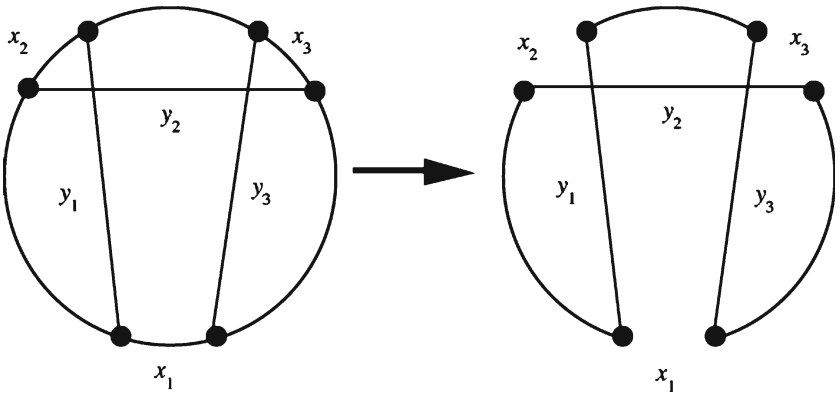

Fig. 1 A 3-opt move. x_1, x_2, x_3 are replaced by y_1, y_2, y_3

The algorithm is specified in *exchanges* (or *moves*) that can convert one candidate solution into another. Given a feasible TSP 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 Lin–Kernighan algorithm (LK) performs so-called k-opt moves on tours. A k-opt move changes a tour by replacing k edges from the tour by k edges in such a way that a shorter tour is achieved. The algorithm is described in more detail in the following.

Let T be the current tour. At each iteration step the algorithm attempts to find two sets of edges, $X = \{x_1, \ldots, x_k\}$ and $Y = \{y_1, \ldots, y_k\}$, such that, if the edges of X are deleted from T and replaced by the edges of Y, the result is a better tour. The edges of X are called *out-edges*. The edges of Y are called *in-edges*.

The two sets X and Y are constructed element by element. Initially X and Y are empty. In step i a pair of edges, x_i and y_i, are added to X and Y, respectively. Figure 1 illustrates a 3-opt move.

In order to achieve a sufficiently efficient algorithm, only edges that fulfill the following criteria may enter X or 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. 2).

As seen, the sequence $(x_1, y_1, x_2, y_2, x_3, \ldots, x_k, y_k)$ constitutes a chain of adjoining edges. A necessary (but not sufficient) condition that the exchange of edges X with edges Y results in a tour is that the chain is closed, i.e., $y_k = (t_{2k}, t_1)$. Such an exchange is called *sequential*. For such an exchange the chain of edges forms a cycle along which edges from X and Y appear alternately, a so-called *alternating cycle* (see Fig. 3). Generally, an improvement of a tour may be achieved as a sequential exchange by a suitable numbering of the affected edges. However, this is not always the case. Figure 4 shows an example where a sequential exchange is not possible.

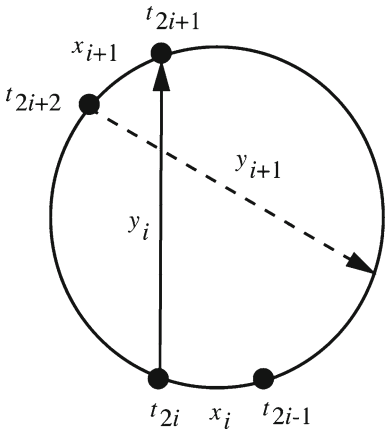

Fig. 2 Restricting the choice of x_i, y_i, x_{i+1}, and y_{i+1}

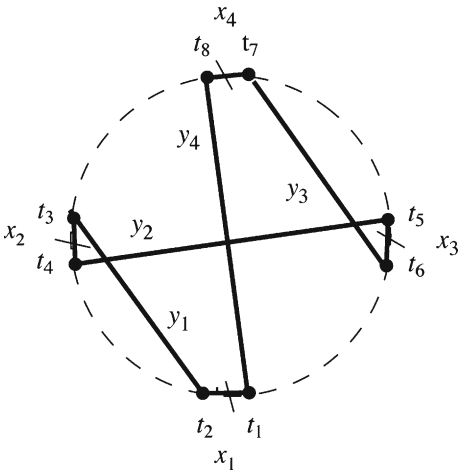

Fig. 3 Alternating cycle $(x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4)$

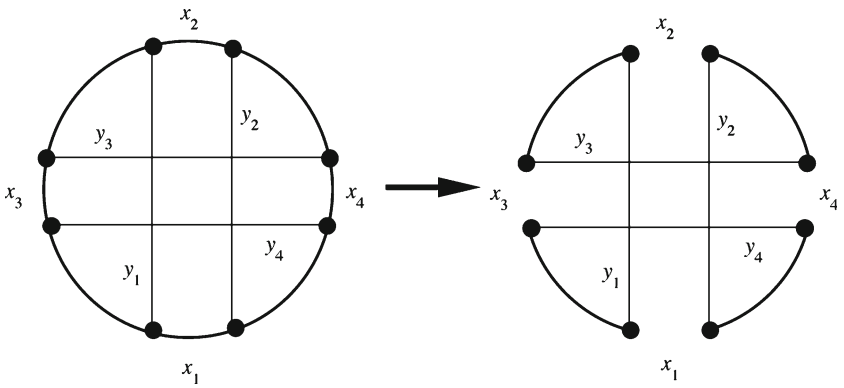

Fig. 4 Non-sequential exchange ($k = 4$)

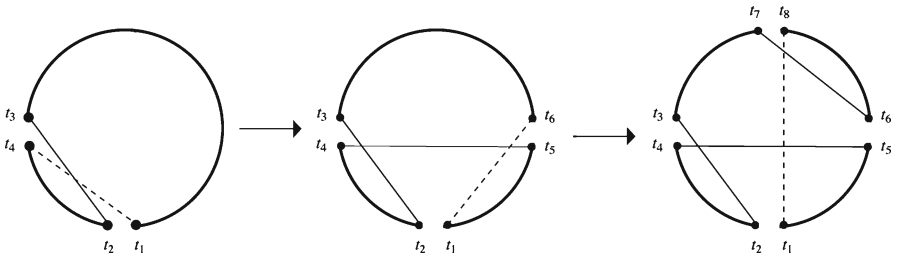

Fig. 5 Sequential 4-opt move performed by three 2-opt moves. Close-up edges are shown by *dashed lines*

Note that all 2- and 3-opt moves are sequential. The simplest non-sequential move is the 4-opt move shown in Fig. 4, the so-called *double-bridge move*.

(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 included in the algorithm both to reduce running time and to simplify the coding. It restricts the set of moves to be explored to those k-opt moves that can be performed by a 2- or 3-opt move followed by a sequence of 2-opt moves. In each of the subsequent 2-opt moves the first edge to be deleted is the last added edge in the previous move (the *close-up edge*). Figure 5 shows a sequential 4-opt move performed by a 2-opt move followed by two 2-opt moves.

(3) The positive gain criterion It is required that y_i is always chosen so that the *cumulative 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 $g_1 + g_2 + \cdots + g_i$. This stop criterion plays a major role in the efficiency of the algorithm.

(4) The disjunctivity criterion It is required that the sets X and Y are disjoint. This simplifies coding, reduces running time, and gives an effective stop criterion. To limit the search even more, Lin and Kernighan introduced some additional criteria of which the following one is the most important:

(5) The candidate set criterion The search for an edge to enter the tour, $y_i = (t_{2i}, t_{2i+1})$, is limited to the five nearest neighbors to t_{2i}.

3 The modified Lin–Kernighan algorithm (LKH-1)

Use of Lin and Kernighan's original criteria, as described in the previous section, results in a reasonably effective algorithm. Typical implementations are able to find solutions that are 1–2% above optimum. However, in [16] it was demonstrated that it was possible to obtain a much more effective implementation by revising these criteria. This implementation, in the following called LKH-1, made it possible to find optimum

solutions with an impressive high frequency. The revised criteria are described briefly below (for details, see [16]).

(1) The sequential exchange criterion This criterion has been relaxed a little. When a tour can no longer be improved by sequential moves, attempts are made to improve the tour by non-sequential 4- and 5-opt moves.

(2) The feasibility criterion A sequential 5-opt move is used as the basic submove. For $i \geq 1$ it is required that $x_{5i} = (t_{10i-1}, t_{10i})$, is chosen so that if t_{10i} is joined to t_1, the resulting configuration is a tour. 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 to a tour results in a tour improvement. Using a 5-opt move as the basic submove instead of 2- or 3-opt moves broadens the search and increases the algorithm's ability to find good tours, at the expense of an increase of running times.

(3) The positive gain criterion This criterion has not been changed.

(4) The disjunctivity criterion The sets X and Y need no longer be disjoint. In order to prevent an infinite chain of submoves the following rule applies: The last edge to be deleted in a 5-opt move must not previously have been added in the current chain of 5-opt moves. Note that this relaxation of the criterion makes it possible to generate certain non-sequential moves.

(5) The candidate set criterion The usual measure for nearness, the costs of the edges, is replaced by a new measure called the *α-measure*. Given the cost of a minimum 1-tree [14,15], the α-value of an edge is the increase of this cost when a minimum 1-tree is required to contain the edge. The α-values provide a good estimate of the edges' chances of belonging to an optimum tour. Using α-nearness it is often possible to restrict the search to relative few of the α-nearest neighbors of a node, and still obtain optimal tours.

4 LKH-2

Extensive computational experiments with LKH-1 have shown that the revised criteria provide an excellent basis for an effective implementation. In general, the solution quality is very impressive. However, these experiments have also shown that LKH-1 has its shortcomings. For example, solving instances with more than 100,000 nodes is computationally too expensive.

The new implementation, called LKH-2, eliminates many of the limitations and shortcomings of LKH-1. The new version extends the previous one with data structures and algorithms for solving very large instances, and facilities for obtaining solutions of even higher quality. A brief description of the main features of LKH-2 is given below.

4.1 General k-opt submoves

One of the most important means in LKH-2 for obtaining high-quality solutions is its use of general k-opt submoves. In the original version of the Lin–Kernighan algorithm moves are restricted to those that can be decomposed into a 2- or 3-opt move followed by a (possibly empty) sequence of 2-opt moves. This restriction simplifies implementation but is not necessarily the best design choice if high-quality solutions are sought. This has been demonstrated with LKH-1, which uses a 5-opt sequential move as the basic move component. LKH-2 takes this idea a step further. Now k-opt moves can be used as submoves, where k is any chosen integer greater than or equal to 2 and less than the number of cities. Each submove is sequential. However, during the search for such moves, non-sequential moves may also be examined. Thus, in contrast to the original version of the Lin–Kernighan algorithm, non-sequential moves are not just tried as a last resort but are integrated into the ordinary search.

4.2 Partitioning

In order to reduce the complexity of solving large-scale problem instances, LKH-2 makes it possible to partition a problem into smaller subproblems. Each subproblem is solved separately, and its solution is used (if possible) to improve a given overall tour, T. The set of nodes is partitioned into subsets of a prescribed maximum size. Each subset, S, induces a subproblem consisting of all nodes of S, and with edges fixed between nodes that are connected by segments of T whose interior nodes do not belong S. Currently, LKH-2 implements the following six partitioning schemes: Tour segment, Karp, Delaunay, Rohe, K-means, and Space-filling Curve partitioning.

4.3 Tour merging

LKH-2 provides a tour merging procedure that attempts to produce the best possible tour from two or more given tours using local optimization on an instance that includes all tour edges, and where edges common to the tours are fixed. Tours that are close to optimum typically share many common edges. Thus, the input graph for this instance is usually very sparse, which makes it practicable to use k-opt submoves for rather large values of k.

4.4 Iterative partial transcription

Iterative partial transcription is a general procedure for improving the performance of a local search based heuristic algorithm. It attempts to improve two individual solutions by replacing certain parts of either solution by the related parts of the other solution. The procedure may be applied to the TSP by searching for subchains of two tours, which contain the same cities in a different order and have the same initial and final cities. LKH-2 uses the procedure on each locally optimum tour and the current best tour. The implemented algorithm is a simplified version of the algorithm described by Möbius et al. [29].

4.5 Backbone-guided search

The edges of the tours produced by a fixed number of initial trials may be used as candidate edges in the succeeding trials. The algorithm is a simplified version of the algorithm given by Zhang and Looks [34].

The rest of the paper describes and evaluates the implementation of general k-opt submoves in LKH-2. The other features will not be described further in this paper.

5 Implementation of general k-opt submoves

This section describes the implementation of general k-opt submoves in LKH-2. The description is divided into the following four parts:

(1) Search for sequential moves
(2) Search for non-sequential moves
(3) Determination of the feasibility of a move
(4) Execution of a feasible move.

The first two parts show how the search space of possible moves can be explored systematically. The third part describes how it is possible to decide whether a given move is feasible, that is, whether execution of the move on the current tour will result in a tour. Finally, it is shown how it is possible to execute a feasible move efficiently.

5.1 Search for sequential moves

A sequential k-opt move on a tour T may be specified by a sequence of nodes, $(t_1, t_2, \ldots, t_{2k-1}, t_{2k})$, where

(t_{2i-1}, t_{2i}) belongs to T $(1 \le i \le k)$, and
(t_{2i}, t_{2i+1}) does not belong to T $(1 \le i \le k$ and $t_{2k+1} = t_1)$.

The requirement that (t_{2i}, t_{2i+1}) does not belong to T is, in fact, not a part of the definition of a sequential k-opt move. Note, however, that if any of these edges belong to T, then the sequential k-opt move is also a sequential k'-opt move for some $k' < k$. Thus, when searching for k-opt moves, this requirement does not exclude any moves to be found. The requirement simplifies coding without doing any harm.

We may therefore generate all possible sequential k-opt moves by generating all t-sequences of length $2k$ that fulfill the two requirements. Generation of such t-sequences may, for example, be performed iteratively in $2k$ nested loops, where the loop at level i goes through all possible values for t_i. Generation of 5-opt moves in LKH-1 was implemented in this way. However, if we want to generate k-opt moves, where k may be chosen freely, this approach is not appropriate. In this case, we would like to use a variable number of nested loops. This is normally not possible in imperative languages like C, but it is well known that it may be simulated by use of recursion. In LKH-2 the search algorithm is implemented as a recursive function, which, in contrast to LK, terminates as soon as a gainful, feasible move has been found.

Table 1 Complexities for operations involved in the search for sequential submoves

Operation	Complexity
PRED	$O(1)$
SUC	$O(1)$
Added	$O(1)$
Deleted	$O(1)$
Excludable	$O(1)$
FeasibleKOptMove	$O(k \log k)$

The time complexity for the algorithm may be evaluated from the time complexities for the sub-operations involved. These sub-operations and their time complexities are given in Table 1.

PRED and *SUC* return for a given node respectively its predecessor and successor on the tour. *Added* and *Deleted* are used to ensure that no edge is added or deleted more than once in the submove under construction. *Excludable* is used to examine whether the last edge to be deleted in a k-opt move has previously been added in the current chain of k-opt submoves (Criterion 4 in Sect. 3). These five operations may be executed in constant time by maintaining a few pointers for each node.

Finally, *FeasibleKOptMove(k)* determines whether a given t-sequence, $(t_1, t_2, \ldots, t_{2k-1}, t_{2k})$, represents a feasible k-opt move. In Sect. 5.3 it is shown how this operation may be implemented with a time complexity of $O(k \log k)$.

Let d denote the maximum node degree in the candidate graph. Then, when searching for a move, there are at most d possible choices for each of its in-edges and at most 2 possible choices for each of its out-edges. Thus, the worst-case time complexity for the search algorithm grows exponentially with k if $d \geq 2$. This stresses the importance of choosing a sparse candidate graph if high values of k are wanted.

5.2 Search for non-sequential moves

In the original version of the Lin–Kernighan algorithm (LK) non-sequential moves are only used in one special case, namely when the algorithm can no longer find any sequential moves that improve the tour. In this case it tries to improve the tour by a non-sequential 4-opt move, a so-called *double bridge move* (see Fig. 4).

In LKH-1 this kind of post optimization moves is extended to include non-sequential 5-opt moves. However, unlike LK, 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.

LKH-2 takes this idea a step further. Now the search for non-sequential moves is integrated with the search for sequential moves. Furthermore, it is possible to search for non-sequential k-opt moves for any value of $k \geq 4$.

The basic idea is the following. If, during the search for a sequential move, a non-feasible move is found, this non-feasible move may be used as a starting point for

construction of a feasible non-sequential move. Observe that the non-feasible move would, if executed on the current tour, result in two or more disjoint cycles. Therefore, we can obtain a feasible move if these cycles somehow can be patched together to form one and only one cycle.

The solution to this cycle patching problem is straightforward. Given a set of disjoint cycles, we can always patch these cycles by one or more alternating cycles. The method is best explained through an example. Suppose that execution of a non-feasible k-opt move, $k \geq 4$, would result in four disjoint cycles. As shown in Fig. 6 the four cycles may be transformed into a tour by use of one alternating cycle, which is represented by the node sequence $(s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_1)$. Note that the alternating cycle alternately deletes an edge from one of the four cycles and adds an edge that connects two of the four cycles.

Figure 7 shows how four disjoint cycles can be patched by two alternating cycles: $(s_1, s_2, s_3, s_4, s_1)$ and $(t_1, t_2, t_3, t_4, t_5, t_6, t_1)$. Note that both alternating cycles are necessary in order to achieve a tour.

Figure 8 illustrates that it is also possible to use three alternating cycles: $(s_1, s_2, s_3, s_4, s_1)$, $(t_1, t_2, t_3, t_4, t_1)$, and $(u_1, u_2, u_3, u_4, u_1)$. In general, k disjoint cycles may be transformed into a tour using up to $k - 1$ alternating cycles.

With the addition of non-sequential moves, the number of different types of k-opt moves that the algorithm must be able to handle has increased considerably. In the following this statement is quantified.

Let MT(k) denote the number of k-opt move types. Then MT(k) can be computed as the product of the number of inversions of $k - 1$ segments and the number of permutations of $k - 1$ segments [10]:

$$\mathrm{MT}\,(k) = 2^{k-1}\,(k-1)!.$$

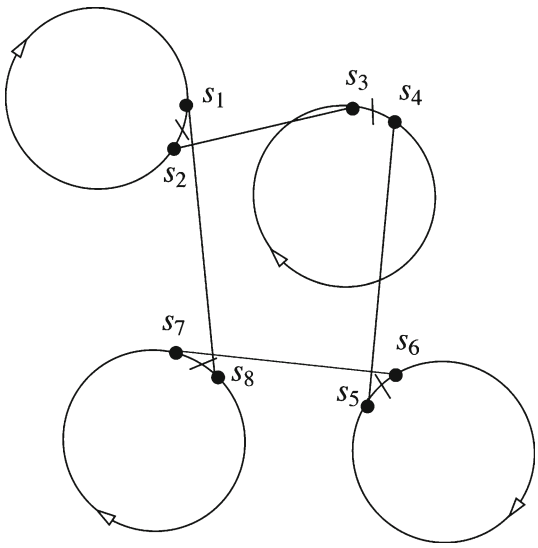

Fig. 6 Four disjoint cycles patched by one alternating cycle

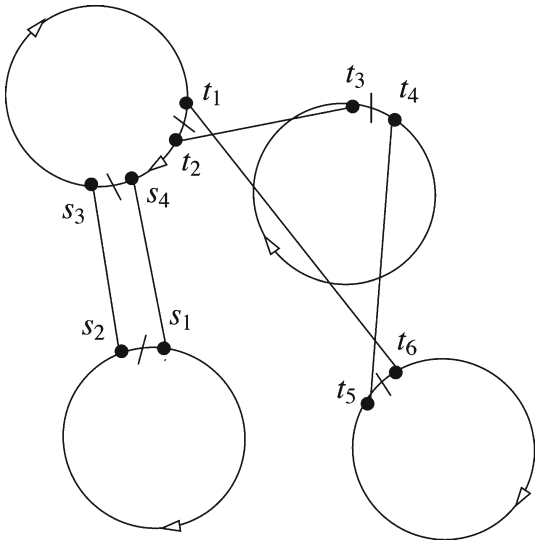

Fig. 7 Four disjoint cycles patched by two alternating cycles

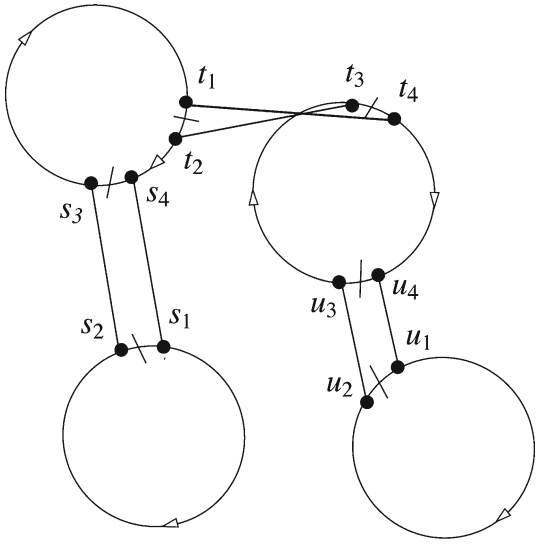

Fig. 8 Four disjoint cycles patched by three alternating cycles

However, MT(k) includes the number of moves, which reinserts one or more of the deleted edges. Since such moves may be generated by k'-opt moves where $k' < k$, we are more interested in computing PMT(k), the number of *pure* k-opt moves, that is, moves for which the set of removed edges and the set of added edges are disjoint. An explicit formula for PMT(k) may be derived from the formula for series A061714 in

The On-Line Encyclopedia of Integer Sequences [33]:

$$\text{PMT}\,(k) = (-1)^k + \sum_{1=0}^{k-1} (-1)^{k-1+i} \binom{k}{i+1} i!2^i \quad \text{for } k \geq 2.$$

The set of pure moves accounts for both sequential and non-sequential moves. To examine how much the search space has been enlarged by the inclusion of non-sequential moves, we will compute SPMT(k), the number of sequential, pure k-opt moves, and compare this number with the PMT(k). The explicit formula for SPMT(k) shown below has been derived from a formula given by Hanlon, Stanley and Stembridge [12, pp. 167–168]:

$$\text{SPMT}\,(k) = \frac{2^{3k-2} k!\,(k-1)!^2}{(2k!)} + \sum_{a=1}^{k-1} \sum_{b=1}^{\min(a,k-a)} c_{a,b}\,(2) \left[\frac{2^{a-b-1}\,(2b)!\,(a-1)!\,(k-a+b+1)}{(2b-1)\,b!} \right]^2,$$

where

$$c_{a,b}\,(2) = (-1)^k \frac{(2)^{a-b+1}\,k\,(2a-2b+1)\,(a-1)!}{(k+a-b+1)\,(k+a-b)\,(k-a+b-1)\,(k-a-b)!\,(2a-1)!\,(b-1)!}.$$

Table 2 depicts MT(k), PMT(k), SPMT(k), and the ratio SPMT(k)/MPT(k) for selected values of k. As seen, the ratio SMPT(k)/PMT(k) decreases as k increases. For $k \geq 10$, there are fewer types of sequential moves than types of non-sequential moves.

From the table it also appears that the number of types of non-sequential, pure moves constitutes 20% or more of all types of pure moves for $k \geq 4$. It is therefore very important that an implementation of non-sequential move generation is runtime efficient. Otherwise, its practical value will be limited.

Let there be a given set of disjoint cycles, C, corresponding to some non-feasible, sequential move. Then LKH-2 searches for a set of alternating cycles, AC, which when applied to C results in an improved tour. The set AC is constructed element by element. The search process is restricted by the following rules:

Table 2 Growth of move types for k-opt moves

k	2	3	4	5	6	7	8	9	10	50	100
MT(k)	2	8	48	384	3,840	46,080	645,120	1.0E7	1.9E8	3.4E77	5.9E185
PMT(k)	1	4	25	208	2,121	25,828	365,457	5.9E6	1.1E8	2.1E77	3.6E185
SPMT(k)	1	4	20	148	1,348	15,104	198,144	3.0E6	5.1E7	4.3E76	5.9E184
$\frac{\text{SPMT}(k)}{\text{PMT}(k)}$	1	1	0.80	0.71	0.63	0.58	0.54	0.51	0.48	0.21	0.17

The same values of MT(k) and PMT(k) have been reported in [10]

(1) No two alternating cycles have a common edge.
(2) All out-edges of an alternating cycle belong to the intersection of the current tour and C.
(3) All in-edges of an alternating cycle must connect two cycles in C.
(4) The starting node for an alternating cycle must belong to the shortest cycle (the cycle in C with the lowest number of edges).
(5) Construction of an alternating cycle is only started if the current gain is positive.

Rules 1–3 are visualized in Figs. 6, 7, and 8. An alternating cycle moves from cycle to cycle, finally connecting the last visited node with the starting node. It is easy to see that an alternating cycle with $2m$ edges ($m \geq 2$) reduces the number of cycle components by $m-1$. Suppose that an infeasible k-opt move results in $m \geq 2$ disjoint cycles. Then these cycles can be patched using at least one and at most $m-1$ alternating cycles. If only one alternating cycle is used, it must contain precisely $2m$ edges. If $m-1$ alternating cycles are used, each of them must contain exactly 4 edges. Hence, the constructed feasible move is a L-opt move, where $k+2m/2 \leq L \leq k+4(m-1)/2$, that is, $k+m \leq L \leq k+2m-2$. Since m at most can be k, we can conclude that Rules 1–3 permit feasible, non-sequential L-opt moves, where $k+2 \leq L \leq 3k-2$. For example, if a non-feasible 5-opt submove results in 5 cycles, it may be the starting point for finding a feasible, non-sequential 7- to 13-opt submove.

Rule 4 minimizes the number of possible starting nodes. In this way the algorithm attempts to minimize the number of possible alternating cycles to be explored.

Rule 5 is analogous with the positive gain criterion for sequential moves (see Sect. 2). During the construction of a move, no alternating cycle will be closed unless the cumulated gain plus the cost of the close-up edge is positive. In order to reduce the search even more the following greedy rule is employed:

(6) The last three edges of an alternating cycle must be those that contribute most to the total gain. In other words, given an alternating cycle $(s_1, s_2, \ldots, s_{2m-2}, s_{2m-1}, s_{2m}, s_1)$, the quantity

$$-c(s_{2m-2}, s_{2m-1}) + c(s_{2m-1}, s_{2m}) - c(s_{2m}, s_1)$$

should be maximum.

Furthermore, the user of LKH-2 may restrict the search for non-sequential moves by specifying an upper limit for the number of cycles that can be patched, and an upper limit for the number of alternating cycles to be used for patching.

In the following it is described how cycle patching is implemented in LKH-2. To make the description more comprehensible we will first show the implementation of an algorithm that performs cycle patching by use of only one alternating cycle.

We need to be able to traverse those nodes that belong to the smallest cycle component (Rule 4), and for a given node to determine quickly to which of the current cycles it belongs to (Rule 3). For that purpose we first determine the permutation p corresponding to the order in which the nodes $t_1, \ldots t_{2k}$ occur on the tour in a clockwise direction, starting with t_1. For example, $p = (1\ 2\ 3\ 4\ 9\ 10\ 7\ 8\ 5\ 6)$ for the non-feasible 5-opt move shown in Fig. 9.

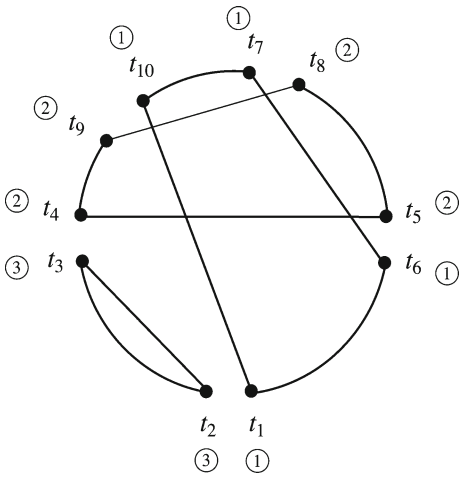

Fig. 9 Non-feasible 5-opt move (3 disjoint cycles)

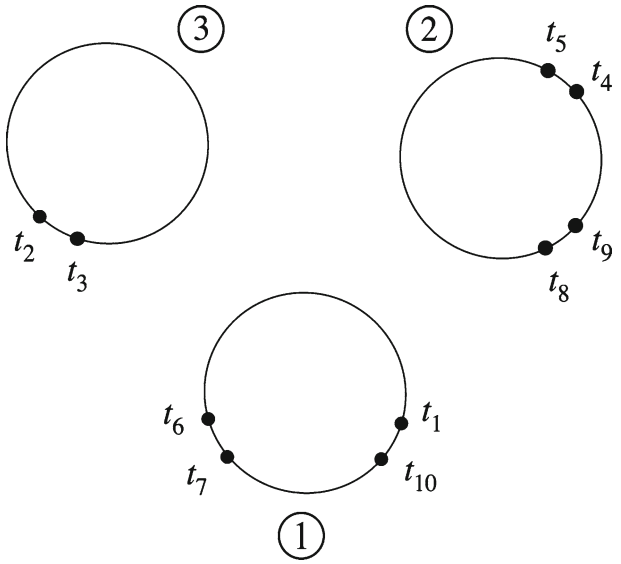

Fig. 10 The three cycles produced by the 5-opt move

Next, the number of cycles is determined and each node $t_1, \ldots t_{2k}$ is labeled with the number of the cycle it belongs to. Execution of the 5-opt move shown in Fig. 9 produces three disjoint cycles, one represented by the node sequence $(t_1, t_{10}, t_7, t_6, t_1)$, one represented by the node sequence (t_4, t_9, t_8, t_5), and one represented by the node sequence (t_2, t_3, t_2). The nodes of the first sequence are labeled with 1, the nodes of the second sequence with 2, and nodes of the third sequence with 3.

Figure 10 visualizes the three disjoint cycles that would arise if the 5-opt move shown in Fig. 9 were executed.

Next, the size of each cycle is calculated and the one that contains the lowest number of nodes is selected.

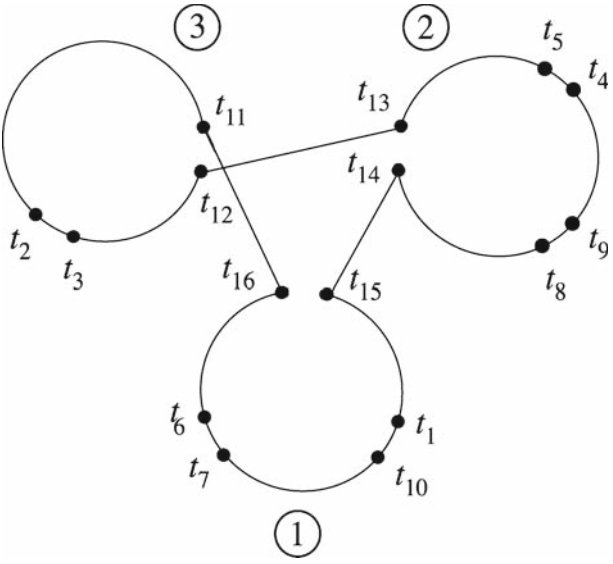

Fig. 11 Patching of the three cycles

The tour segments of the shortest cycle may now be traversed by exploiting the fact that $t_{p[2i]}$ and $t_{p[2i+1]}$ are the two end points for a tour segment of a cycle, for $1 \leq i \leq k$ and $p[2k+1] = p$ [1]. Which cycle a tour segment is part of may be determined simply by retrieving the cycle number associated with one of its two end points.

Assume the shortest cycle in Fig. 10 is the cycle labeled 3. For each edge (t_{11}, t_{12}) on this cycle, except (t_3, t_2), an attempt is made to find an alternating cycle $(t_{11}, t_{12}, t_{13}, t_{14}, t_{15}, t_{16}, t_{11})$ that patches the three cycles and gives a positive total gain (see Fig. 11). The search for node t_{13} is restricted to those candidate neighbors to t_{12} that does not belong to cycle 3. There are two choices for t_{14}, the predecessor and the successor to t_{13} on its cycle. However, the edge (t_{13}, t_{14}) must not be a previously added edge: (t_3, t_2), (t_5, t_4), (t_9, t_8), (t_1, t_{10}), (t_7, t_6). Finally, the nodes t_{15} and t_{16} are chosen in the same way as t_{13} and t_{14}. Node t_{15} must not belong to the same cycle as t_{11} and t_{13}.

A move is represented by the nodes in an array t, where the first $2k$ elements are the nodes of the given non-feasible sequential k-opt move, and the subsequent elements are the nodes of the alternating cycle(s). In order to be able to determine quickly whether an edge is an in- or out-edge of the current move we maintain an array, *incl*, such that $incl[i] = j$ and $incl[j] = i$ is true if and only if the edge $(t[i], t[j])$ is an in-edge. For example, in Fig. 9 $incl = [10, 3, 2, 5, 4, 7, 6, 9, 8, 1]$. It is easy to see that there is no reason to maintain similar information about out-edges as they always are those edges $(t[i], t[i+1])$ for which i is odd.

The time complexity for cycle patching may be evaluated from the time complexities for the sub-operations involved. The sub-operations may be implemented with the time complexities given in Table 3.

Table 3 Complexities for the sub-operations of cycle patching

Operation	Complexity
PRED	$O(1)$
SUC	$O(1)$
BETWEEN	$O(1)$
Deleted	$O(1)$
FindPermutation	$O(k \log k)$
Cycles	$O(k)$
ShortestCycle	$O(k)$
Cycle	$O(\log k)$

FindPermutation computes the permutation p by sorting the nodes of the first $2k$ elements of the t-sequence. The node comparisons are made by the operation *BETWEEN*(a, b, c), which in constant time can determine whether a node b is placed between two other nodes, a and c, on the tour. The constant time complexity of BETWEEN is achieved by using the two-level tree data structure for tour representation [9]. In addition, the operation determines in $O(k)$ time $q[1..2k]$ as the inverse permutation to p, that is, the permutation for which $q[p[i]] = i$ for $1 \leq i \leq 2k$.

The permutation p and its inverse is used by the operation *Cycles* to calculate the number of cycles and to associate with each node in $t[1..2k]$ the number of the cycle it belongs to. As will be shown in Sect. 5.3, all t-nodes of a cycle can be traversed in $O(k)$ time.

The operation *ShortestCycle* determines the cycle that contains the lowest number of nodes. The size of each cycle is found by adding the size of each tour segment belonging to the cycle. By using an appropriate data structure for representing a tour, the size of a tour segment may be computed (or estimated) in constant time. As there are $2k$ such segments, the time complexity of *ShortestCycle* is $O(k)$.

Finally, the operation *Cycle* determines the number of the cycle containing a given node. Only nodes in the current t-sequence are labeled with cycle numbers, but we can use binary search to find a t-node on the same cycle as the given node.

The algorithm as described above only allows cycle patching by means of one alternating cycle. However, it is relatively simple to extend the algorithm such that more than one alternating cycle can be used. Only a few lines of code need to be added.

The algorithm described in this section is somewhat simplified in relation to the algorithm implemented in LKH-2. For example, when two cycles arise, LKH-2 will attempt to patch them, not only by means of an alternating cycle consisting of 4 edges (a 2-opt move), but also by means of an alternating cycle consisting of 6 edges (a 3-opt move).

5.3 Determination of the feasibility of a move

Given a tour T and a k-opt move, how can it quickly be determined if the move is feasible, that is, whether the result will be a tour if the move is applied to T? Consider Fig. 12.

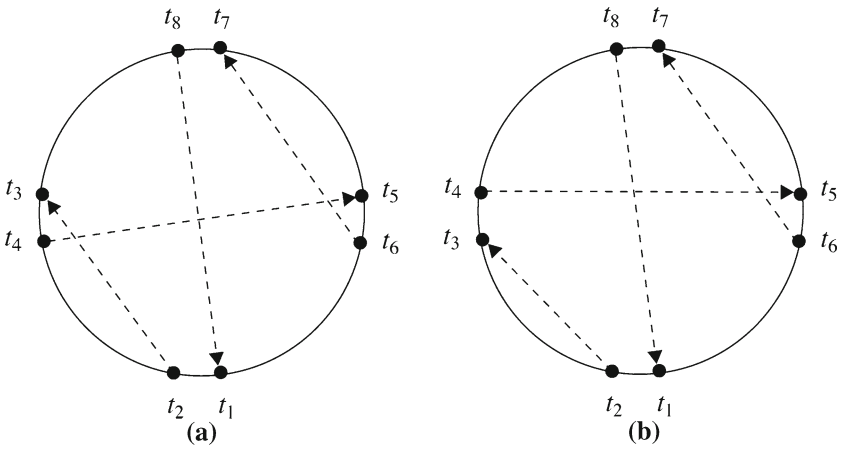

Fig. 12 **a** Feasible 4-opt move. **b** Non-feasible 4-opt move

Figure 12a depicts a feasible 4-opt move. Execution of the move will result in precisely one cycle (a tour), namely $(t_2t_3–t_8t_1–t_6t_7–t_5t_4–t_2)$. On the other hand, the 4-opt move in Fig. 12b is not feasible, since the result will be two cycles, $(t_2t_3–t_2)$ and $(t_4t_5–t_7t_6–t_1t_8–t_4)$.

Deciding whether a k-opt move is feasible is a frequent problem in the algorithm. Each time a gainful move is found, the move is checked for feasibility. Non-gainful moves are also checked to ensure that they can enter the current chain of sequential moves. Hence, it is very important that such checks are fast.

A simple algorithm for checking feasibility is to start in an arbitrary node and then walk from node to node in the graph that would arise if the move were executed, until the starting node is reached again. The move is feasible if and only if the number of visited nodes is equal to the number of nodes, n, in the original tour. However, the complexity of this algorithm is $O(n)$, which makes it unsuited for the job.

Can we construct a faster algorithm? Yes, because we do not need to visit every node on a tour. We can restrict ourselves to only visiting the t-nodes that represent the move. In other words, we can jump from t-node to t-node. A move is feasible if and only if all t-nodes of the move are visited in this way. For example, if we start in node t_6 in Fig. 12a, and jump from t-node to t-node alternately following an in-edge and leaping to the other end of an unbroken tour segment, then all t-nodes are visited in the following sequence: $t_6, t_7, t_5, t_4, t_2, t_3, t_8, t_1$.

It is easy to jump from one t-node, t_a, to another, t_b, if the edge (t_a, t_b) is an in-edge. We only need to maintain an array, *incl*, which represents the current in-edges. If (t_a, t_b) is an in-edge for a k-opt move $(1 \leq a, b \leq 2k)$, this fact is registered in the array by setting $incl[a] = b$ and $incl[b] = a$. In Fig. 12a, b $incl$ = [8, 3, 2, 5, 4, 7, 6, 1]. By this means each such jump can be made in constant time (by a table lookup).

On the other hand, it is not obvious how we can skip those nodes that are not t-nodes. If we start in node t_6 in Fig. 12a, then we must skip the nodes between t_7 and t_5, between t_4 and t_2, between t_3 and t_8, and between t_1 and t_6 before we return

to t_6. It turns out that a little preprocessing solves the problem. If we know the cyclic order in which the t-nodes occur on the original tour, then it becomes easy to skip all nodes that lie between two t-nodes on the tour. For a given t-node we just have to select either its predecessor or its successor in this cyclic ordering. For example, for t_7 in Fig. 12a we must select t_5, not t_8. Which of the two cases we should choose can be determined in constant time. This kind of jump may therefore be made in constant time if the t-nodes have been sorted. In the following it is shown that this sorting can be done in $O(k \log k)$ time.

First, we realize that is not necessary to sort all t-nodes. We can restrict ourselves to sorting half of the nodes, namely for each out-edge the first end point met when the tour is traversed in a given direction. If in Fig. 12a the chosen direction is "clockwise", we may restrict the sorting to the four nodes t_1, t_4, t_5, and t_8. If the result of a sorting is represented by a permutation, p_{half}, then p_{half} will be equal to (1 4 8 5). This permutation may easily be extended to a full permutation containing all node indices, $p = (1\ 2\ 4\ 3\ 8\ 7\ 5\ 6)$. The missing node indices are inserted by using the following rule: if $p_{\text{half}}[i]$ is odd, then insert $p_{\text{half}}[i]+1$ after $p_{\text{half}}[i]$, otherwise insert $p_{\text{half}}[i]-1$ after $p_{\text{half}}[i]$.

Let a move be represented by the nodes $t[1..2k]$. First, the operation *FindPermutation* is used to find the permutation $p[1..2k]$ that corresponds to their visiting order when the tour is traversed in the *SUC*-direction. In addition, the operation determines $q[1..2k]$ as the inverse permutation to p. After having determined p and q, we can determine in $O(k)$ time whether a k-opt move represented by the contents of the arrays t and *incl* is feasible. In each iteration of a loop the two end nodes of an in-edge are visited, namely $t[p[i]]$ and $t[incl[p[i]]]$. The inverse permutation, q, makes it possible to skip all nodes between $t[incl[p[i]]]$ and the next t-node on the tour. If the position of $incl[p[i]]$ in p, that is $q[incl[p[i]]]$, is even, then in the next iteration i should be equal to this position plus one. Otherwise, it should be equal to this position minus one. The loop may be expressed very compactly in C as

```
for (i = 2 * k, count = 1; (i = q[incl[p[i]]] ^ 1) != 0); count++);
```

The move is feasible if and only if count becomes equal to k after execution of the loop. The loop terminates when i becomes zero, which happens when node $t[p[1]]$ has been visited (since $1 \,\hat{}\, 1 = 0$, where $\hat{}$ is the exclusive OR operator). The loop always terminates since $t[p[1]]$ belong to the same cycle as the starting node, $t[p[2k]]$, and no node is visited more than once.

Since the sorting made by *FindPermutation* on average takes $O(k \log k)$ time, we can conclude that the average-time complexity for the *FeasibleKOptMove* operation is $O(k \log k)$. Normally, k is very small compared to the total number of nodes, n. Thus, we have obtained an algorithm that is efficient in practice.

5.4 Execution of a feasible move

In order to simplify execution of a feasible k-opt move, the following fact may be used: Any k-opt move ($k \geq 2$) is equivalent to a finite sequence of 2-opt moves [8,26]. In the case of 5-opt moves it can be shown that any 5-opt move is equivalent to a

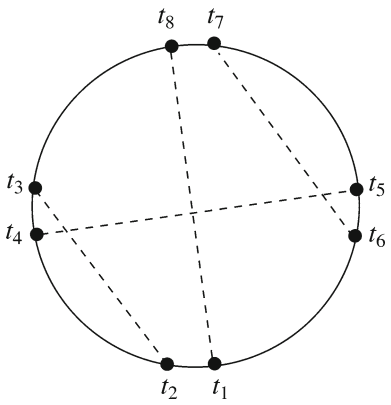

Fig. 13 Feasible 4-opt move

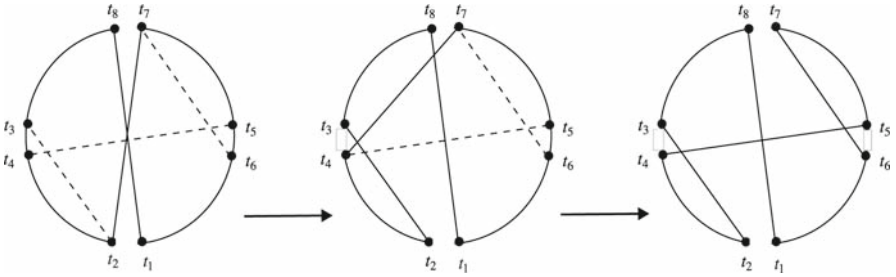

Fig. 14 Execution of the 4-opt move by means of 3 flips

sequence of at most five 2-opt moves. Any 3-opt move as well as any 4-opt move is equivalent to a sequence of at most three 2-opt moves. In general, any feasible k-opt move may be executed by at most k 2-opt moves. For a proof, see [28].

Let FLIP(a, b, c, d) denote the operation of replacing the two edges (a, b) and (c, d) of the tour by the two edges (b, c) and (d, a). Then the 4-opt move depicted in Fig. 13 may be executed by the following sequence of FLIP-operations:

FLIP(t_2, t_1, t_8, t_7)
FLIP(t_4, t_3, t_2, t_7)
FLIP(t_7, t_4, t_5, t_6)

The execution of the flips is illustrated in Fig. 14. The 4-opt move of Fig. 13 may be executed by many other flip sequences, for example (see Fig. 15):

FLIP(t_2, t_1, t_5, t_6)
FLIP(t_5, t_1, t_3, t_4)
FLIP(t_3, t_1, t_8, t_7)
FLIP(t_3, t_7, t_6, t_2)

However, this sequence contains one more FLIP-operation than the previous sequence. Therefore, the first one of these two is preferred.

A central question is, for any feasible k-opt move, how to find a FLIP-sequence that corresponds to the move. In addition, we want the sequence to be as short as possible.

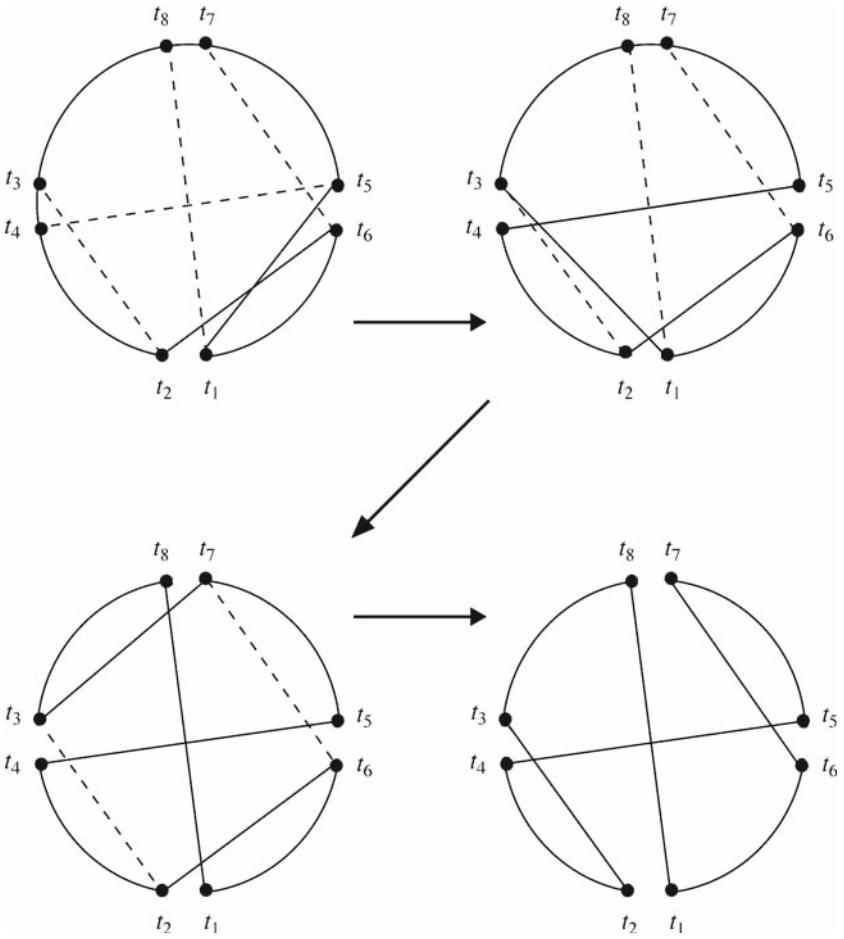

Fig. 15 Execution of the 4-opt move by means of 4 flips

In the following it is shown that an answer to this question can be found by transforming the problem into an equivalent problem, which has a known solution. Consider Fig. 16, which shows the resulting tour after a 4-opt move has been applied. Note that any 4-opt move may effect the reversal of up to 4 segments of the tour. In the figure each of these segments has been labeled with an integer whose numerical value is the order in which the segment occurs in the resulting tour. The sign of the integer specifies whether the segment in the resulting tour has the same (+) or the opposite orientation (−) as in the original tour. Starting in the node t_2, the segments in the new tour occur in the order 1–4. A negative sign associated with the segments 2 and 4 specifies that they have been reversed in relation to their direction in the original tour (clockwise).

If we write the segment labels in the order the segments occur in the original tour, we get the following sequence, a so-called *signed permutation*.

$$(+1 - 4 - 2 + 3)$$

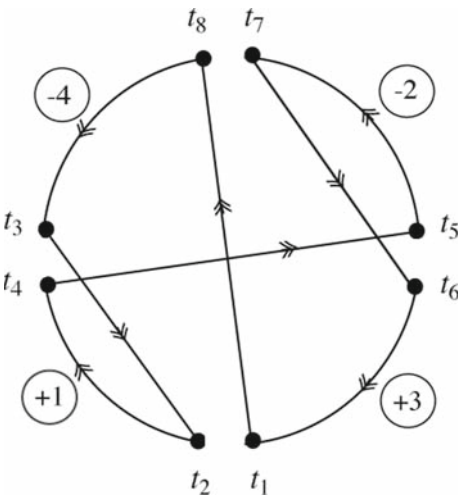

Fig. 16 Segments labeled with orientation and rank

We want this sequence to be transformed into the sequence (the *identity permutation*):

$(+1 + 2 + 3 + 4)$

Notice now that a FLIP-operation corresponds to a reversal of a segment of the signed permutation, where both the order and the sign of the elements of this segment are changed. In Fig. 15 an execution of the flip sequence

FLIP(t_2, t_1, t_8, t_7)
FLIP(t_4, t_3, t_2, t_7)
FLIP(t_7, t_4, t_5, t_6)

corresponds to the following sequence of signed reversals:

$(+1\ -4\ \underline{-2\ +3})$
$(+1\ \underline{-4\ -3\ +2})$
$(+1\ \underline{-2}\ +3\ +4)$
$(+1 + 2 + 3 + 4)$

Reversed segments are underlined.

Suppose now that, given a signed permutation of $\{1, 2, \ldots, k\}$, we are able to determine the shortest possible sequence of signed reversals that transforms the permutation into the identity permutation $(+1, +2, \ldots, +k)$. Then, given a feasible k-opt move, we will also be able to find a shortest possible sequence of FLIP-operations that can be used to execute the move.

However, this problem, called *Sorting signed permutations by reversals*, is a well-studied problem in computational molecular biology. The problem arises, for example, when one wants to determine the genetic distance between two species, that is, the minimum number of mutations needed to transform the genome of one species into the genome of the other. The most important mutations are those that rearrange the

genomes by reversals, and since the order of genes in a genome may be described by a permutation, the problem is to find the shortest number of reversals that transform one permutation into another.

The problem can more formally be defined as follows. Let $\pi = (\pi_1 \dots \pi_n)$ be a permutation of $\{1, \dots, n\}$. A *reversal* $\rho(i, j)$ of π is an inversion of a segment $(\pi_i \dots \pi_j)$ of π, that is, it transforms the permutation $(\pi_1 \dots \underline{\pi_i \dots \pi_j} \dots \pi_n)$ into $(\pi_1 \dots \pi_j \dots \pi_i \dots \pi_n)$. The problem of *Sorting by reversals* (SBR) is the problem of finding the shortest possible sequence of reversals $(\rho_1 \dots \rho_{d(n)})$ such that $\pi\rho_1 \dots \rho_{d(n)} = (12 \dots n-1\ n)$, where $d(n)$ is called the *reversal distance* for π.

A special version of the problem is defined for *signed permutations*. A signed permutation $\sigma = (\sigma_1 \dots \sigma_m)$ is obtained from an ordinary permutation $\pi = (\pi_1 \dots \pi_m)$ by replacing each of its elements π_i by either $+\pi_i$ or $-\pi_i$. A reversal $\rho(i, j)$ of a signed permutation σ reverses both the order and the signs of the elements $(\sigma_i \dots \sigma_j)$. The problem of *Sorting signed permutations by reversals* (SSBR) is the problem of finding the shortest possible sequence of reversals $(\rho_1 \dots \rho_{d(n)})$ such that $\sigma\rho_1 \dots \rho_{d(n)} = (+1 + 2 \cdots + (m-1)m)$.

It is easy to see that determination of a shortest possible FLIP-sequence for a k-opt move is a SSBR problem. We are therefore interested in finding an efficient algorithm for solving SSBR. It is known that the unsigned version, SBR, is NP-hard [6], but, fortunately, the signed version, SSBR, has been shown to be polynomial by Hannenhalli and Pevzner in 1995, and they gave an algorithm for solving the problem in $O(n^4)$ time [13]. Since then faster algorithms have been discovered, among others an $O(n^2)$ algorithm by Kaplan et al. [23]. The fastest algorithm for SSBR today has complexity $O\left(n\sqrt{n \log n}\right)$ [32].

Several of these fast algorithms are difficult to implement. We have chosen to implement a very simple algorithm described by Bergeron [5]. The algorithm has a worst-time complexity of $O(n^3)$. However, as it on average runs in $O(n^2)$ time, and hence in $O(k^2)$ for a k-opt move, the algorithm is sufficiently efficient for our purpose. If we assume $k << n$, where n is the number of cities, the time for determining the minimal flip sequence is dominated by the time to make the flips, which is $O\left(\sqrt{n}\right)$, since the tour is represented by the two-level tree data structure [9].

The operation *MakeKOptMove* exploits Bergeron's algorithm for executing a k-opt move using a minimum number of flips. Its worst-time complexity is $O\left(k^3 + k\sqrt{n}\right)$, since the worst time complexity of Bergeron's algorithm is $O(k^3)$, and there are at most k FLIP-operations, each of which has complexity $O\left(\sqrt{n}\right)$.

The operation minimizes the number of flips. However, this need not be the best way to minimize running time. The lengths of the tour segments to be flipped should not be ignored. Currently, however, no algorithm is known that solves this sorting problem optimally. It is an interesting area of future research.

6 Experimental evaluation

This section presents the results of a series of computational experiments, the purpose of which is to examine LKH-2's performance when general k-opt moves are used as submoves. The results include its qualitative performance and its runtime efficiency.

Runtimes are measured in seconds on an Intel Xeon 2.66 GHz processor. The runtimes do not include the time used for preprocessing.

The performance has been evaluated on the following symmetric problems taken from the 8th DIMACS Implementation Challenge [21]:

E-instances: Instances consisting of uniformly distributed points in a square.
C-instances: Instances consisting of clustered points in a square.

6.1 Performance for E-instances

The E-instances consist of cities uniformly distributed in the 1,000,000 by 1,000,000 square under the Euclidean metric. For testing purposes, we have selected those instances that have 10,000 or more cities. Optima for these instances are currently unknown. We follow Johnson and McGeoch [22] in measuring tour quality in terms of percentage over the Held–Karp lower bound [14,15] on optimal tours. The Held–Karp bound appears to provide a consistently good approximation to the optimal tour length [20].

Table 4 covers the lengths of the current best tours for the E-instances. These tours have all been found by LKH. The first two columns give the names of the instances and their number of nodes. The column labeled *CBT* contains the lengths of the current best tours. The column labeled *HK bound* contains the Held–Karp lower bounds. The table entries for the two largest instances (*E3M.0* and *E10M.0*), however, contain approximations to the Held–Karp lower bounds (lower bounds on the lower bounds).

The column labeled *HK gap* (%) gives for each instance the percentage excess of the current best tour over the Held–Karp bound:

$$\text{HK gap}\,(\%) = \frac{\text{CBT} - \text{HK bound}}{\text{HK bound}} \times 100\%$$

Table 4 Tour quality for E-instances

Instance	n	CBT	HK bound	HK gap (%)	$\frac{\text{CBT}}{\sqrt{n}}10^{-6}$
E10k.0	10,000	71,865,826	71,362,276	0.706	0.7187
E10k.1	10,000	72,031,630	71,565,485	0.651	0.7203
E10k.2	10,000	71,822,483	71,351,795	0.660	0.7182
E31k.0	31,623	127,282,138	126,474,847	0.638	0.7158
E31k.1	31,623	127,452,384	126,647,285	0.636	0.7167
E100k.0	100,000	225,787,421	224,330,692	0.649	0.7140
E100k.1	100,000	225,659,006	224,241,789	0.632	0.7136
E316k.0	316,228	401,307,462	398,760,105	0.639	0.7136
E1M.0	1,000,000	713,189,988	708,703,513	0.633	0.7132
E3M.0	3,162,278	1,267,369,147	1,260,000,000	0.585	0.7127
E10M.0	10,000,000	2,253,175,807	2,240,000,000	0.588	0.7125

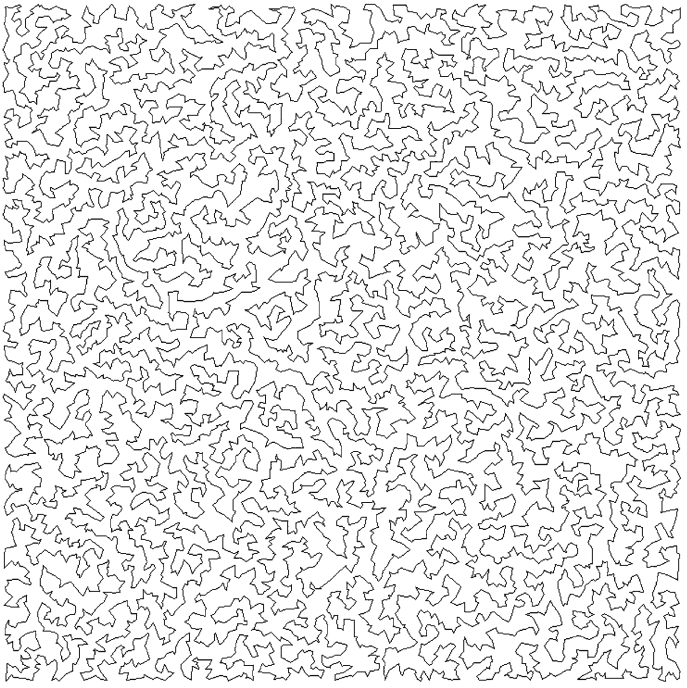

Fig. 17 Best tour for *E10k.0*

It is well known that for random Euclidean instances with n cities distributed uniformly randomly over a rectangular area of A units, the ratio of the optimal tour length to $\sqrt{n}\sqrt{A}$ approaches a limiting constant C_{OPT} as $n \to \infty$. Johnson, McGeoch, and Rothenberg [19] have estimated C_{OPT} to 0.7124 ± 0.0002. The last column of Table 4 contains these ratios. The results are consistent with this estimate for large n.

Using cutting-plane methods Concorde [1,3] has found a lower bound of 713,056, 616 for the 1,000,000-city instance *E1M.0* (William Cook, private communication, 2008). The bound shows that LKH's current best tour for this instance has a length at most 0.019% greater than the length of an optimal tour.

6.1.1 Results for E10k.0

The first test instance is *E10k.0*. Figure 17 depicts the current best tour for this instance.

First we will examine, for increasing values of k, how tour quality and CPU times are affected if we use sequential k-opt moves as submoves. The 5 α-nearest edges incident to each node are used as candidate set (see Fig. 18). As many as 99.5% of the edges of the best tour belong to this set.

For each value k between 2 and 8 ten local optima were found, each time starting from a new initial tour. Initial tours are constructed using self-avoiding random walks on the candidate edges [16].

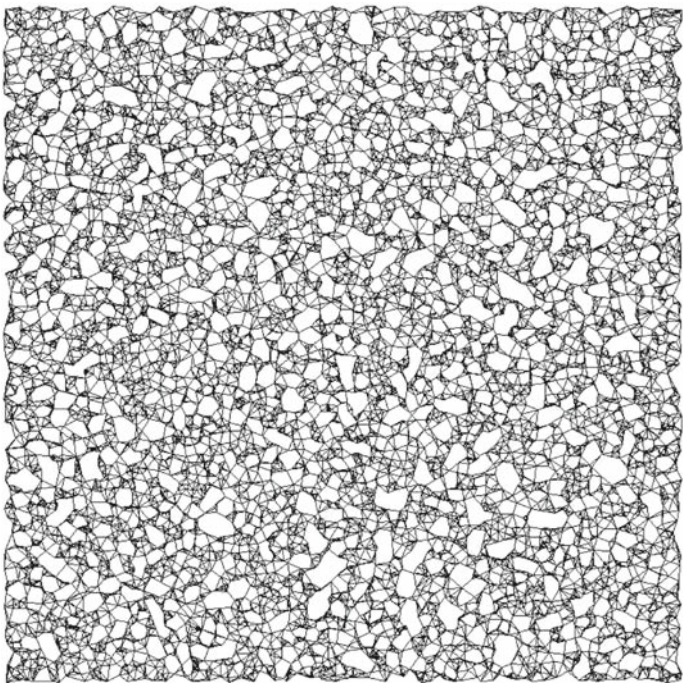

Fig. 18 The 5 α-nearest candidate set for *E10k.0*

The results from these experiments are shown in Table 5. The table covers the Held–Karp gap in percent and the CPU time in seconds used for each run. The program parameter *PATCHING_C* specifies the maximum number of cycles that may be patched during the search for moves. In this experiment, the value is zero, indicating that only non-sequential moves are to be considered. Note, however, that the post optimization procedure of LKH for finding improving non-sequential 4- or 5-opt moves is used in this as well as all the following experiments.

The results show, not surprisingly, that tour quality increases as k grows, at the cost of increasing CPU time. These facts are best illustrated by curves (Figs. 19, 20).

Table 5 Results for *E10k.0* (no patching, 1 trial, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
2	1.745	1.838	1.986	0	0	0
3	1.071	1.151	1.258	0	0	0
4	0.885	0.937	1.024	0	1	1
5	0.799	0.851	0.920	1	1	2
6	0.788	0.824	0.864	7	11	16
7	0.772	0.787	0.799	19	48	87
8	0.764	0.779	0.811	82	123	187

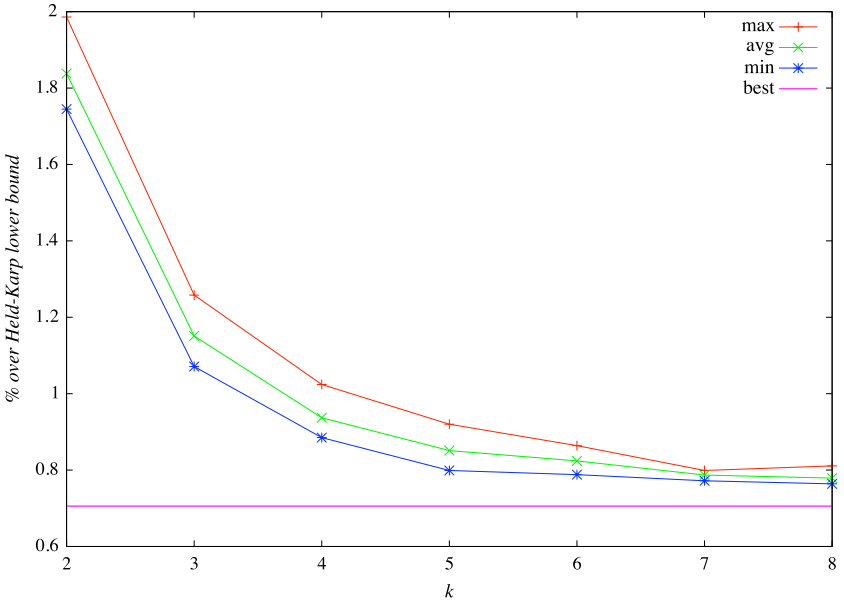

Fig. 19 *E10k.0*: percentage excess over HK bound (no patching, 1 trial, 10 runs)

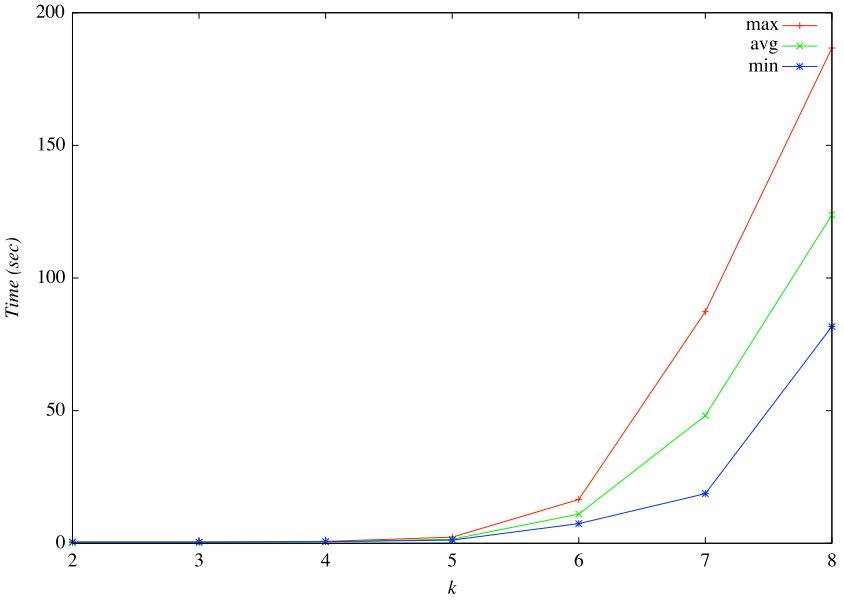

Fig. 20 *E10k.0*: CPU time (no patching, 1 trial, 10 runs)

As expected, CPU time grows exponentially with k. The average time per run grows as 0.03×2.87^k. What is the best choice of k for this instance? Unfortunately, there is no simple answer to this question. It is a tradeoff between quality and time. A possible

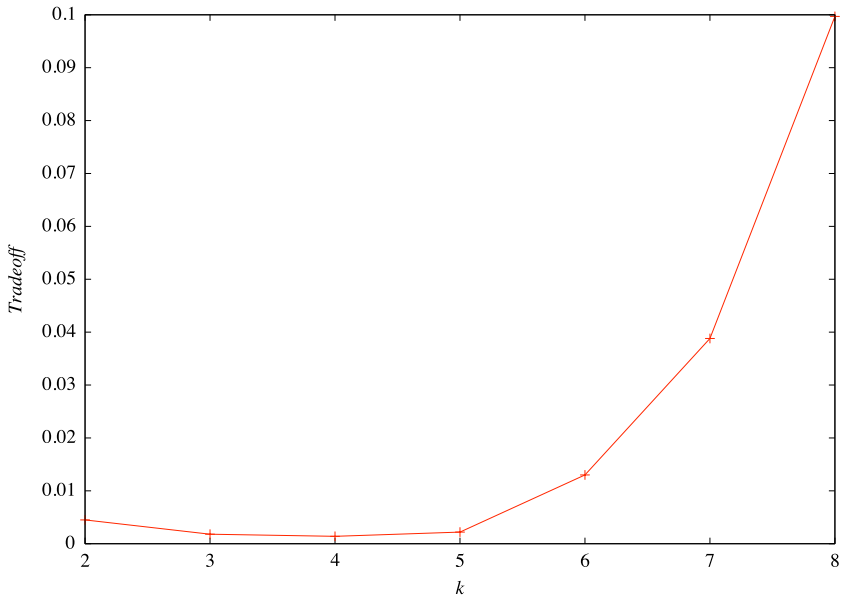

Fig. 21 *E10k.0*: quality-time tradeoff (no patching, 1 trial, 10 runs)

quality-time assessment of k could be defined as the product of time and excess over optimum (*OPT*):

$$\text{Time}(k) \times \frac{\text{Length}(k) - \text{OPT}}{\text{OPT}}$$

The smaller this value is for k the better. The measure gives an equal weight to time and quality. Figure 21 depicts the measure for this experiment, where *OPT* has been replaced by the length of the current best tour. As can be seen, 3, 4 and 5 are the best choices for k if this measure is used. Note, however, that if one wants the shortest possible tour, k should be as large as possible while respecting given time constraints.

We will now examine the effect of integrating non-sequential and sequential moves. To control the magnitude of the search for non-sequential moves LKH-2 provides the following two program parameters:

PATCHING_C: Maximum number of cycles that can be patched to form a tour
PATCHING_A: Maximum number of alternating cycles that can be used for patching

Suppose k-opt moves are used as basis for constructing non-sequential moves. Then *PATCHING_C* can be at most k. *PATCHING_A* must be less than or equal to *PATCHING_C* - 1. A search for non-sequential moves will be made only if PATCHING_C $\geq$ 2 and *PATCHING_A* $\geq$ 1.

Thus, for a given value of k a full exploration of all possible non-sequential move types would require

$$\sum_{i=2}^{k}(i-1) = \frac{k(k-1)}{2} \text{experiments.}$$

In order to limit the number of experiments, we have chosen for each k only to evaluate the effect of adding non-sequential moves to the search for the following two parameter combinations:

$$PATCHING\_C = k, PATCHING\_A = 1$$
$$PATCHING\_C = k, PATCHING\_A = k - 1$$

With the first combination, called *simple patching*, as many cycles as possible are patched using only one alternating cycle. With the second combination, called *full patching*, as many cycles as possible are patched with as many alternating cycles as possible.

Tables 6 and 7 report the results from the experiments with simple patching and full patching. Not surprisingly, these experiments show that better tour quality is achieved if non-sequential moves are allowed. However, it is a nice surprise that this increase in quality is obtained with very little time penalty. In fact, for $k \geq 6$ the algorithm uses less CPU time when non-sequential moves are allowed.

Table 6 Results for *E10k.0* (simple patching, 1 trial, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
2	1.587	1.668	1.787	0	0	1
3	1.011	1.060	1.148	0	1	1
4	0.860	0.896	0.963	1	1	1
5	0.799	0.835	0.893	2	3	3
6	0.755	0.794	0.874	7	8	9
7	0.764	0.787	0.827	20	31	50
8	0.736	0.762	0.796	67	105	132

Table 7 Results for *E10k.0* (full patching, 1 trial, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
3	1.027	1.070	1.122	0	1	1
4	0.849	0.884	0.928	1	1	2
5	0.800	0.833	0.898	2	3	4
6	0.774	0.799	0.844	7	9	13
7	0.767	0.786	0.808	19	35	61
8	0.748	0.761	0.788	54	96	142

6.1.2 Results for E100k.0

Are these conclusions also valid for larger E-instances? In order to answer this question, the same experiments as described in the previous section were made with the 100,000-city instance *E100k.0*. The results from these experiments are reported in Tables 8, 9, and 10. As can be seen from these tables, the same conclusions may be drawn: It pays off to use non-sequential moves.

Table 8 Results for *E100k.0* (no patching, 1 trial, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
2	1.747	1.779	1.901	5	6	9
3	1.099	1.109	1.154	5	7	10
4	0.904	0.915	0.924	9	10	12
5	0.818	0.820	0.824	21	26	37
6	0.769	0.773	0.776	129	165	213
7	0.739	0.745	0.759	405	569	826
8	0.707	0.711	0.740	1,980	2,355	3,513

Table 9 Results for *E100k.0* (simple patching, 1 trial, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
2	1.581	1.640	1.788	5	6	9
3	1.008	1.012	1.017	6	8	12
4	0.844	0.849	0.874	12	15	21
5	0.777	0.790	0.807	36	41	53
6	0.743	0.746	0.750	121	145	174
7	0.713	0.719	0.736	347	518	898
8	0.710	0.715	0.723	1,593	2,195	3,150

Table 10 Results for *E100k.0* (full patching, 1 trial, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
3	1.005	1.012	1.023	7	8	11
4	0.842	0.853	0.873	13	15	20
5	0.784	0.786	0.800	37	40	44
6	0.747	0.751	0.762	117	134	171
7	0.723	0.727	0.735	395	572	765
8	0.704	0.708	0.717	1,353	2,397	4,439

Table 11 Results for *E*-instances (no patching, 1 trial, 1 run)

k	HK gap (%)						
	E10k.0	*E31k.0*	*E100k.0*	*E316k.0*	*E1M.0*	*E3M.0*	*E10M.0*
4	0.924	0.888	0.924	0.929	0.923	0.883	0.892
5	0.832	0.827	0.824	0.834	0.833	0.786	0.789
6	0.848	0.775	0.776	0.784	0.776	0.725	0.733
7	0.788	0.757	0.759	0.739	0.740	0.691	0.696
k	Time (s)						
	E10k.0	*E31k.0*	*E100k.0*	*E316k.0*	*E1M.0*	*E3M.0*	*E10M.0*
4	1	2	10	56	305	1,364	6,144
5	2	8	32	123	816	2,470	11,967
6	9	63	165	795	3,478	13,465	50,559
7	36	175	553	3,015	12,769	51,635	175,317

Table 12 Results for *E*-instances (simple patching, 1 trial, 1 run)

k	HK gap (%)						
	E10k.0	*E31k.0*	*E100k.0*	*E316k.0*	*E1M.0*	*E3M.0*	*E10M.0*
4	0.926	0.878	0.874	0.851	0.857	0.806	0.811
5	0.893	0.792	0.796	0.786	0.779	0.727	0.736
6	0.815	0.738	0.750	0.733	0.736	0.689	0.693
7	0.803	0.738	0.714	0.718	0.711	0.663	0.667
k	Time (s)						
	E10k.0	*E31k.0*	*E100k.0*	*E316k.0*	*E1M.0*	*E3M.0*	*E10M.0*
4	1	6	21	79	375	1,492	6,213
5	3	11	61	237	901	3,151	11,321
6	9	39	151	673	2,576	11,032	35,097
7	32	178	352	1,751	8,458	37,352	108,041

6.1.3 Comparative results for E-instances

In order to examine the performance of the implementation as n grows, we used the following *E*-instances: *E10k.0, E31k.0, E100k.0, E316k.0, E1M.0, E3M.0, E10M.0*. The instance sizes are increasing half-powers of 10: 10^4, $10^{4.5}$, 10^5, $10^{5.5}$, 10^6, $10^{6.5}$, and 10^7. For each of these instances a local optimum was found using values of k between 4 and 7, and using either no patching, simple patching or full patching. Due to long computation times for the largest instances only one run was made for each instance. The results of the experiments are reported in Tables 11, 12, and 13. Figures 22, 23, 24, 25, 26, and 27 provide a graphical visualization of the results. As can be seen, the algorithm is very robust for this problem type. The runtime increases almost linearly with the problem size.

Table 13 Results for E-instances (full patching, 1 trial, 1 run)

k	Average HK gap (%)						
	E10k.0	*E31k.0*	*E100k.0*	*E316k.0*	*E1M.0*	*E3M.0*	*E10M.0*
4	0.901	0.835	0.853	0.846	0.845	0.796	0.803
5	0.823	0.783	0.786	0.768	0.775	0.721	0.728
6	0.826	0.729	0.751	0.729	0.730	0.683	0.689
7	0.786	0.696	0.730	0.714	0.709	0.659	0.664
k	Time (s)						
	E10k.0	*E31k.0*	*E100k.0*	*E316k.0*	*E1M.0*	*E3M.0*	*E10M.0*
4	2	5	22	89	431	1,453	6,572
5	3	12	48	172	806	3,132	12,228
6	9	32	148	486	2,206	8,253	30,606
7	27	92	416	2,026	8,131	24,977	111,078

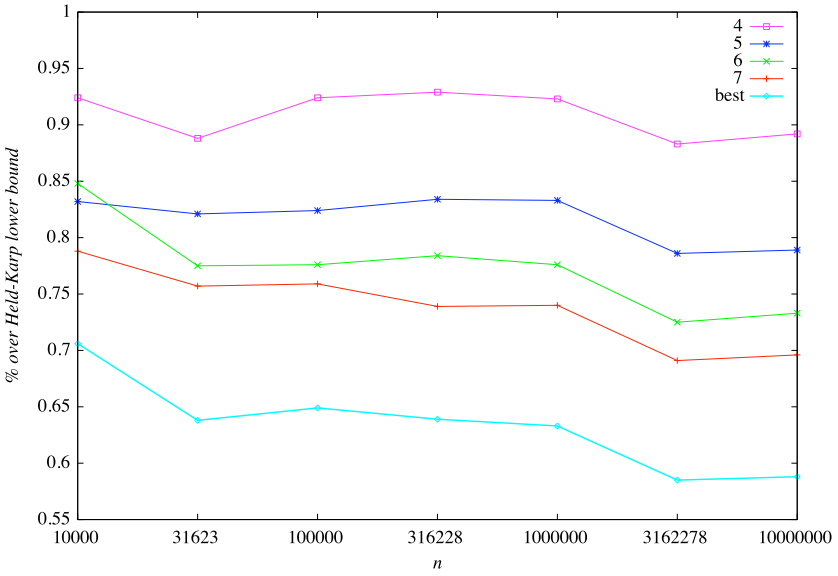

Fig. 22 E-instances: percentage excess over HK bound (no patching, 1 trial, 1 run)

6.1.4 Solving E10k.0 and E100k.0 by multi-trial LKH

In the experiments described until now only one trial per run was used. As each run takes a new initial tour as its starting point, the trials have been independent. Repeatedly starting from new tours, however, is an inefficient way to sample locally optimal tours. Valuable information is thrown away. A better strategy is to *kick* a locally optimal tour (that is, to perturb it slightly), and reapply the algorithm on this tour. If this effort

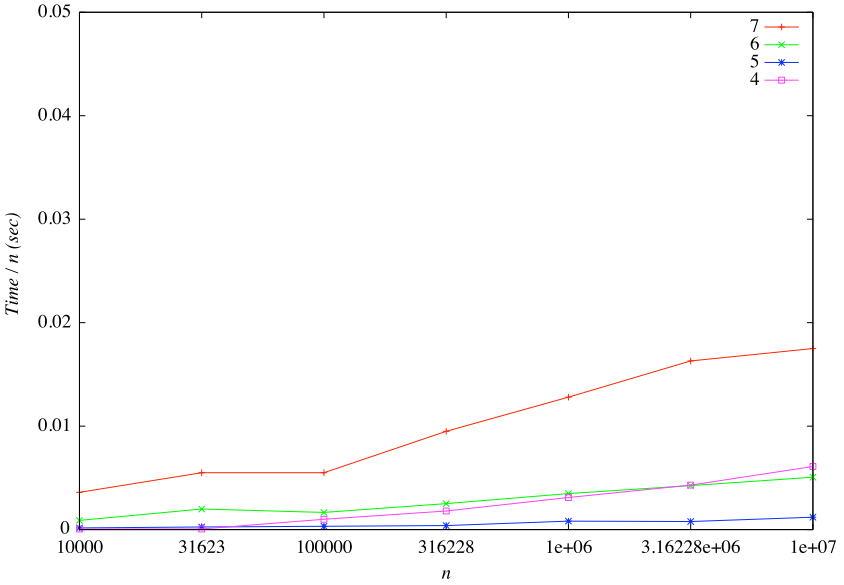

Fig. 23 *E*-instances: time per node (no patching, 1 trial, 1 run)

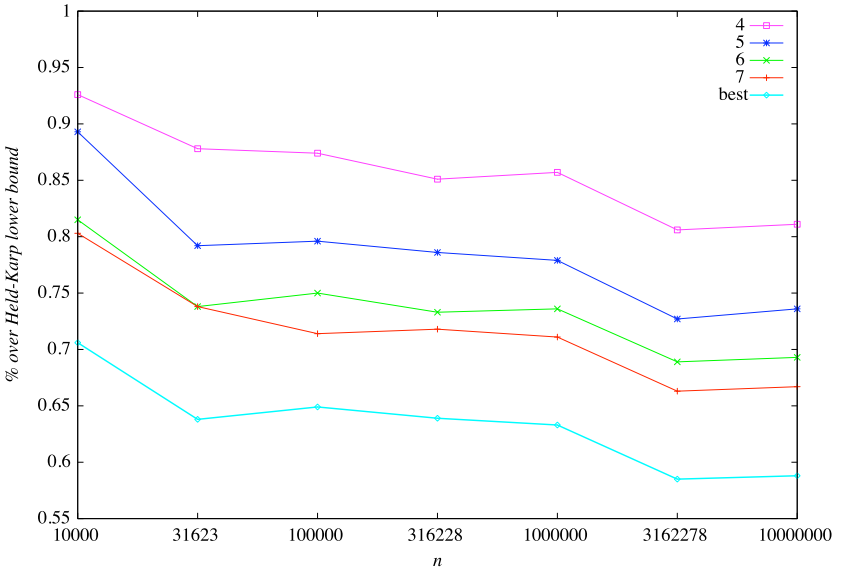

Fig. 24 *E*-instances: percentage excess over HK bound (simple patching, 1 trial, 1 run)

produces a better tour, we discard the old tour and work with the new one. Otherwise, we kick the old tour again. To kick the tour the double-bridge move (see Fig. 4) is often used [2,4,27].

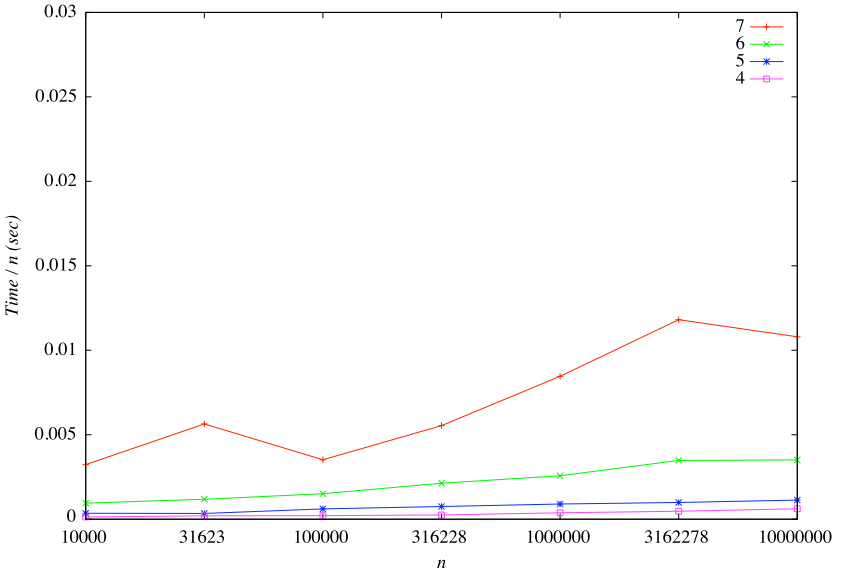

Fig. 25 E-instances: time per node (simple patching, 1 trial, 1 run)

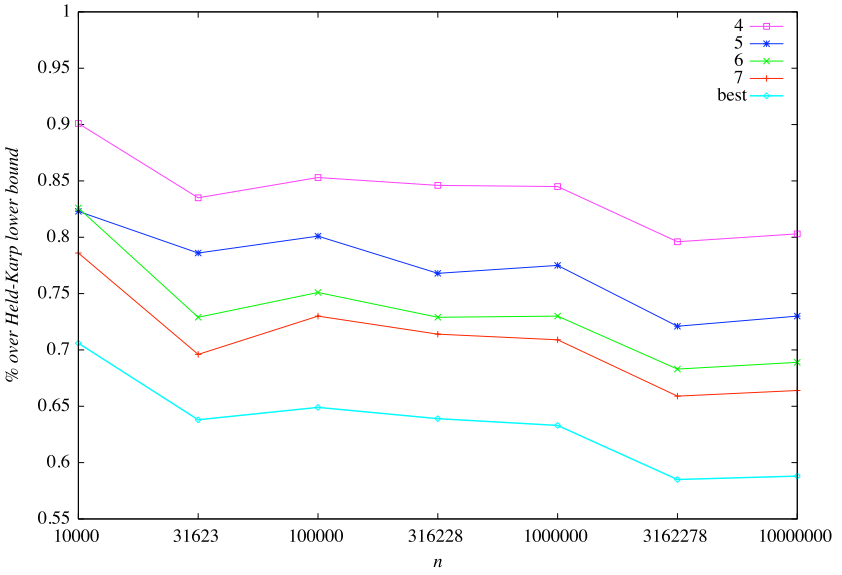

Fig. 26 E-instances: percentage excess over HK bound (full patching, 1 trial, 1 run)

An alternative strategy is used by LKH. The strategy differs from the standard approach in that it uses a random initial tour and restricts its search process by the following rule:

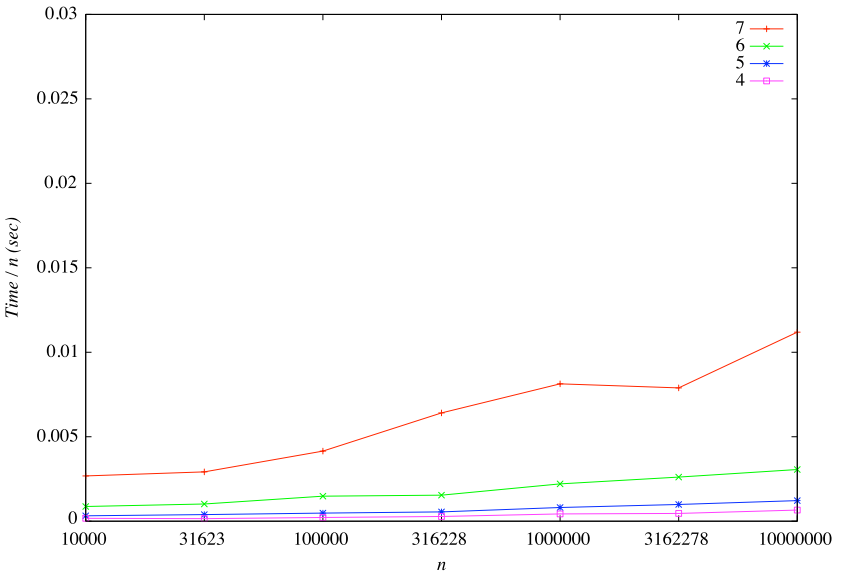

Fig. 27 *E*-instances: time per node (full patching, 1 trial, 1 run)

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

It has been observed that this dependence of the trials almost always results in significantly better tours than would be obtained by the same number of independent trials. In addition, the search restriction above makes it fast.

We made a series of experiments with the instances *E10k.0* and *E100k.0* to study how multi-trial LKH is affected when k is increased and cycle patching is added to the basic k-opt move. Tables 14, 15, and 16 report the results for 1,000 trials on *E10k.0*. As can be seen, tour quality increases as k increases, and as in the previous 1-trial experiments with this instance it is advantageous to use cycle patching.

Tables 17 report the experimental results for multi-trial LKH on the *E100k.0* instance. These results follow the same pattern as the results for *E10k.0*. Note that for this instance it is even more advantageous to use cycle patching.

Table 14 Results for *E10k.0* (no patching, 1,000 trials, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
4	0.713	0.725	0.743	227	263	310
5	0.711	0.721	0.731	310	354	395
6	0.709	0.717	0.732	1,261	1,679	2,327

Table 15 Results for *E10k.0* (simple patching, 1,000 trials, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
4	0.717	0.728	0.760	181	210	249
5	0.711	0.715	0.726	195	236	285
6	0.709	0.714	0.730	362	453	578

Table 16 Results for *E10k.0* (full patching, 1,000 trials, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
4	0.715	0.729	0.758	186	220	239
5	0.709	0.715	0.725	214	253	307
6	0.709	0.714	0.724	393	467	523

Table 17 Results for *E100k.0* (1,000 trials, 1 run)

k	HK gap (%)			Time (s)		
	No	Simple	Full	No	Simple	Full
4	0.783	0.681	0.686	7,023	4,352	4,763
5	0.716	0.671	0.673	13,761	6,144	6,652
6	0.699	0.661	0.660	102,322	11,909	13,702

6.2 Performance for C-instances

It is well known that geometric instances with clustered points are difficult for the Lin–Kernighan heuristic. When it tries to remove an edge bridging two clusters, it is tricked into long and often fruitless searches. Each time a long edge is removed, the cumulative gain rises enormously, and the heuristic is encouraged to perform very deep searches. The cumulative gain criterion is too optimistic and does not effectively prune the search space for this type of instances [30].

To examine LKH's performance for clustered problems we performed experiments on the eight largest C-instances of the 8th DIMACS TSP Challenge. Table 18 covers the lengths of the current best tours for these instances. These tours have all been found by LKH. The experiments with the C-instances are very similar to those performed with the E-instances.

6.2.1 Results for C10k.0

Figure 28 depicts the current best tour for the 10,000-city instance *C10k.0*. Its clustered nature is clear. The cities are grouped so that distances between cities in distinct groups are large in comparison to distances between cities within a group.

As in the experiments with E-instances a candidate set based on the α-measure could be used. Figure 29 depicts the 5 α-nearest candidate set for *C10k.0*. Since the

Table 18 Tour quality for *C*-instances

Instance	*n*	CBT	HK bound	HK gap (%)
C10k.0	10,000	33,001,034	32,782,155	0.668
C10k.1	10,000	33,186,248	32,958,946	0.690
C10k.2	10,000	33,155,424	32,926,889	0.694
C31k.0	31,623	59,545,390	59,169,193	0.636
C31k.1	31,623	59,293,266	58,840,096	0.770
C100k.0	100,000	104,633,819	103,916,254	0.691
C100k.1	100,000	105,390,777	104,663,040	0.695
C316k.0	316,228	186,909,997	185,576,667	0.733

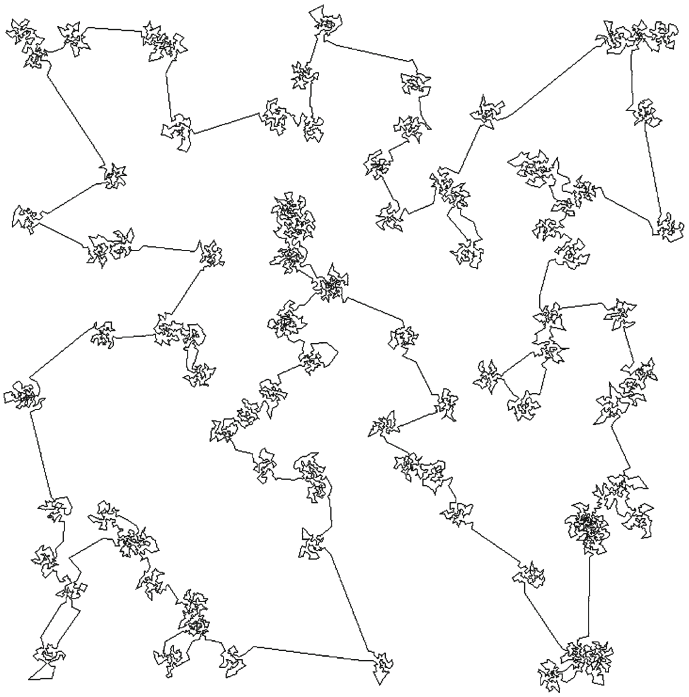

Fig. 28 Best tour for *C10k.0*

set contains as many as 99.3% of the edges of the current best tour, it seems to be well qualified as a candidate set. But, unfortunately, some of the long edges of the best tour are missing, which means that we cannot expect high-quality tours to be found.

For geometric instances, Johnson [18] has suggested using *quadrant-based neighbors*, that is, the least costly edges in each of the four geometric quadrants (for 2-dimensional instances) around the city. For example, for each city its neighbors could be chosen so as to include, if possible, the closest city in each of its four surrounding quadrants.

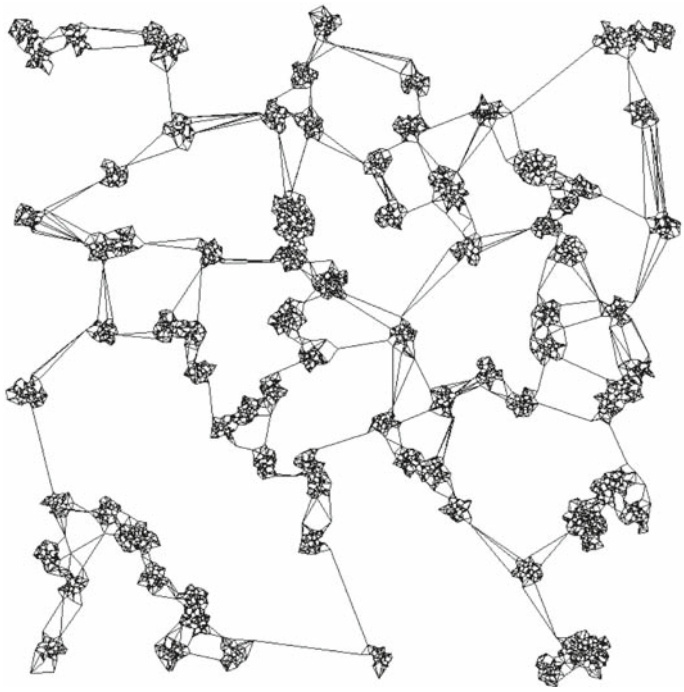

Fig. 29 The 5 α-nearest candidate set for *C10k.0*

For clustered instances we have chosen to use candidate sets defined by the union of the 4 α-nearest neighbors and the 4 quadrant-nearest neighbors (the closest city in each of the four quadrants). Figure 30 depicts this candidate set for *C10k.0*. Even though this candidate subgraph is very sparse (average number of neighbors is 5.1), it has proven to be sufficiently rich to produce excellent tours. It contains 98.3% of the edges of the current best tour, which is less than for the candidate subgraph defined by 5 α-nearest neighbors. In spite of this, it leads to better tours.

However, even if this sparse candidate set is used, our experiments with cycle patching on clustered instances have revealed that the search space is large. To prune the search space we decided to restrict cycle patching for this type of instances by adding the following rule:

All in-edges of an alternating cycle must belong to the candidate set.

Note that the algorithm for generating alternating cycles described in Sects. 5.1 and 5.2 already guarantees that all in-edges, except the last one, are candidate edges. The rule is put into force by a program parameter.

The results from experiments with 1-trial solution of *C10k.0* are shown in Tables 19, 20, and 21. As can be seen, tour quality and CPU time are acceptable. But note that increasing k from 5 to 6 incurs a considerable runtime penalty.

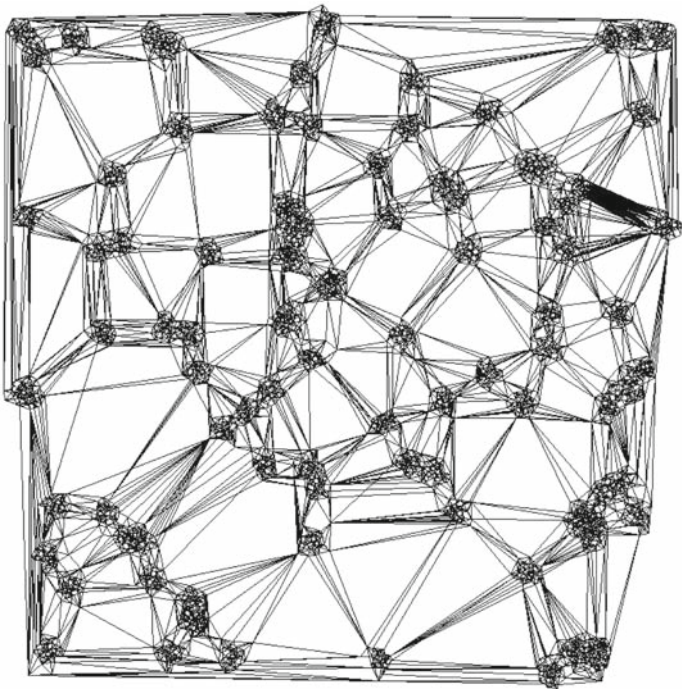

Fig. 30 The 4 α-nearest +4 quadrant-nearest candidate set for *C10k.0*

Table 19 Results for *C10k.0* (no patching, 1 trial, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
3	1.771	1.991	2.226	0	1	1
4	0.811	1.024	1.387	1	1	1
5	0.841	1.184	2.026	5	7	9
6	0.836	1.061	1.710	64	93	150

Table 20 Results for *C10k.0* (simple patching, 1 trial, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
3	1.157	2.185	3.159	1	1	1
4	1.883	2.091	2.764	2	3	3
5	1.308	1.607	2.273	9	13	19
6	0.910	1.233	2.629	62	79	104

Table 21 Results for *C10k.0* (full patching, 1 trial, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
3	1.707	1.873	2.483	1	1	1
4	0.756	1.108	2.470	2	2	3
5	0.801	1.013	1.995	11	14	20
6	0.810	1.083	1.863	61	80	97

Table 22 Results for *C100k.0* (no patching, 1 trial, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
3	2.838	2.991	3.275	6	8	10
4	1.969	2.102	2.491	15	17	22
5	1.515	1.635	1.891	64	73	85
6	1.248	1.293	1.413	628	808	957

Table 23 Results for *C100k.0* (simple patching, 1 trial, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
3	2.910	3.061	3.405	10	12	14
4	2.011	2.175	2.698	34	38	45
5	1.495	1.613	1.815	151	184	269
6	1.143	1.283	1.437	853	1,009	1,134

Table 24 Results for *C100k.0* (full patching, 1 trial, 10 runs)

k	HK gap (%)			Time (s)		
	Min	Avg	Max	Min	Avg	Max
3	2.705	3.050	4.529	10	12	15
4	1.934	2.034	2.269	35	38	45
5	1.392	1.480	1.840	156	193	237
6	1.051	1.174	1.400	780	964	1,263

6.2.2 Results for C100k.0

Looking at the results for 1-trial solution of *C10k.0* (Tables 19, 20, 21), there seems to be only little advantage in using cycle patching. To see whether this also applies to larger instances we made the same experiments with the 100,000-city instance *C100k.0*. The results of these experiments are covered in Tables 22, 23, and 24. As can be seen, it is also questionable whether patching is useful for this instance. In

Table 25 Results for *C*-instances (no patching, 1 trial, 1 run)

k	HK gap (%)				Time (s)			
	C10k.0	*C31k.0*	*C100k.0*	*C316k.0*	*C10k.0*	*C31k.0*	*C100k.0*	*C316k.0*
3	2.226	3.935	3.275	4.691	1	2	7	40
4	1.387	2.709	2.491	2.466	1	4	22	86
5	2.011	2.375	1.891	1.874	6	27	83	359

Table 26 Results for *C*-instances (simple patching, 1 trial, 1 run)

k	HK gap (%)				Time (s)			
	C10k.0	*C31k.0*	*C100k.0*	*C316k.0*	*C10k.0*	*C31k.0*	*C100k.0*	*C316k.0*
3	3.159	4.348	3.405	4.233	1	3	14	68
4	2.764	3.063	2.698	2.849	3	11	45	172
5	2.273	2.908	1.815	2.060	19	72	268	1,021

Table 27 Results for *C*-instances (full patching, 1 trial, 1 run)

k	HK gap (%)				Time (s)			
	C10k.0	*C31k.0*	*C100k.0*	*C316k.0*	*C10k.0*	*C31k.0*	*C100k.0*	*C316k.0*
3	2.483	3.743	4.529	5.128	1	21	13	64
4	1.424	2.950	2.269	2.726	2	21	44	192
5	1.995	2.160	1.840	1.739	16	90	236	835

addition, the runtime penalty for increasing k from 5 to 6 is even more conspicuous for this instance.

6.2.3 Comparative results for C-instances

In order to evaluate the scalability of the implementation we also performed experiments with the instances *C31k.0* and *C316k.0*. Tables 25, 26, and 27 contain comparative results for the four chosen *C*-instances. As can be seen from Figs. 31, 32, and 33, the runtime increases almost linearly with problem size.

6.2.4 Solving C10k.0 and C100k.0 by multi-trial LKH

The performance for 1-trial solution of the *C*-instances is not impressive. However, as shown by the following results from 1,000-trial experiments with *C10k.0* and *C100k.0*,

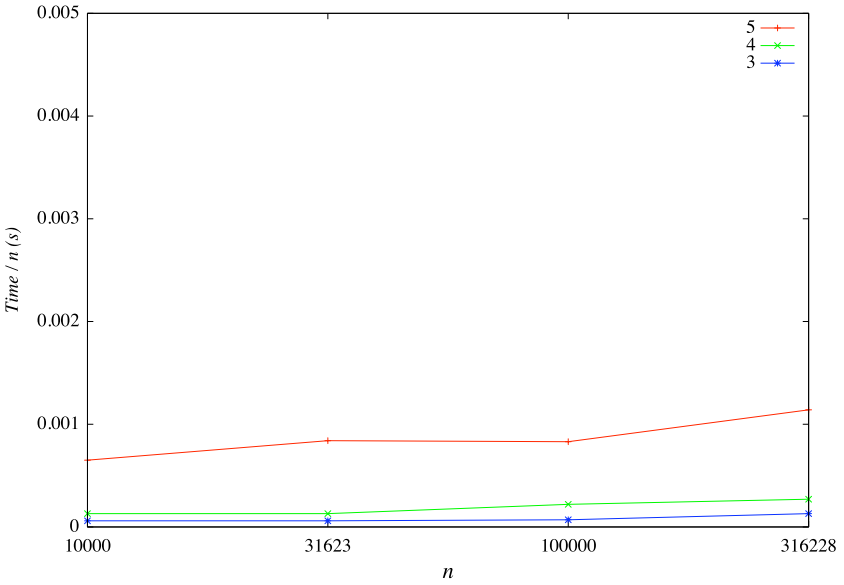

Fig. 31 *C*-instances: time per node (no patching, 1 trial, 1 run)

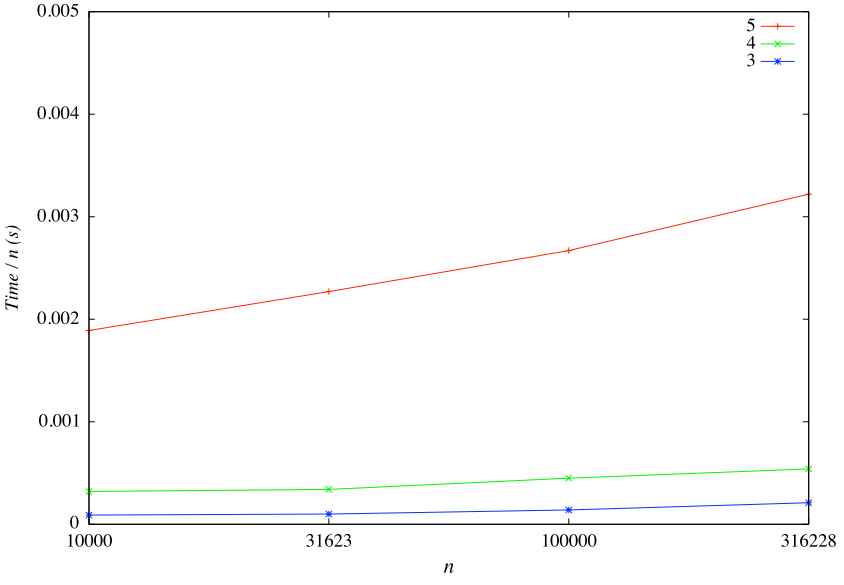

Fig. 32 *C*-instances: time per node (simple patching, 1 trial, 1 run)

high-quality solutions may be achieved using few trials. It is interesting that for this instance type it does not pay off to set k to other values than 4, and that little is gained by using non-sequential moves (Tables 28, 29).

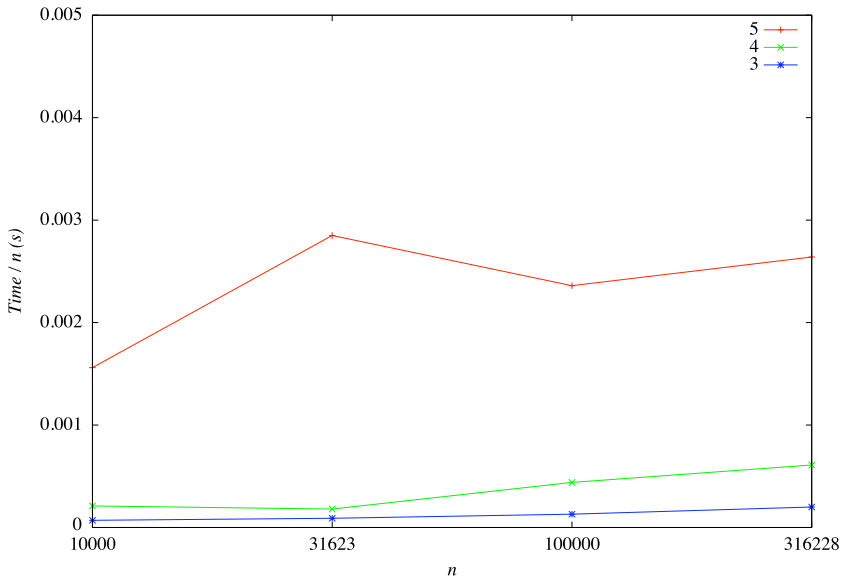

Fig. 33 *C*-instances: time per node (full patching, 1 trial, 1 run)

Table 28 Results for *C10k.0* (1,000 trials, best of 10 runs)

k	HK gap (%)			Time (s)		
	No	Simple	Full	No	Simple	Full
3	0.668	0.668	0.668	324	329	341
4	0.668	0.688	0.668	320	408	642
5	0.668	0.668	0.668	663	938	957

Table 29 Results for *C100k.0* (1,000 trials, 1 run)

k	HK gap (%)			Time (s)		
	No	Simple	Full	No	Simple	Full
3	0.989	1.083	0.876	4,117	4,981	4,765
4	0.814	0.778	0.832	3,793	5,947	6,151
5	0.861	0.743	0.833	7,130	15,276	14,877

7 Conclusions

This paper has described the implementation of a general k-opt submove for the Lin–Kernighan heuristic. The computational experiments have shown that the implementation is both effective and scalable. It should be noted, however, that the usefulness of general k-opt submoves depends on the candidate graph. Unless the candidate graph is sparse (for example defined by the five α-nearest neighbors), it will often be too time consuming to choose k larger than 4. Furthermore, the instance should not be heavily clustered.

The implementation allows the search for non-sequential moves to be integrated with the search for sequential moves. It is interesting to note that in many cases the use of non-sequential moves not only results in better tours but also, what is surprising, reduce running time.

In the current implementation the user may choose the value of k as well the extent of non-sequential moves. These choices are constant during program execution. A possible future path for research would be to explore strategies for varying k dynamically during a run.

LKH-2 is free of charge for academic and non-commercial use and can be downloaded in source code from http://www.ruc.dk/~keld/research/LKH.

Appendix: Parameter settings

This appendix provides representative examples of the parameter settings used in producing the reported results in this paper.

Parameters common to all *E*-instances:

```
CANDIDATE_SET_TYPE = DELAUNAY
MAX_CANDIDATES = 5
INITIAL_PERIOD = 100
```

Example of additional parameters for *E10k.0*:

```
PROBLEM_FILE = E10k.0.tsp
MAX_TRIALS = 1
RUNS = 10
MOVE_TYPE = 8
PATCHING_C = 8
PATCHING_A = 7
```

Parameters common to all *C*-instances:

```
CANDIDATE_SET_TYPE = DELAUNAY
MAX_CANDIDATES = 4
EXTRA_CANDIDATE_SET_TYPE = QUADRANT
EXTRA_CANDIDATES = 4
INITIAL_PERIOD = 100
```

Example of additional parameters for *C10k.0*:

```
PROBLEM_FILE = C10k.0.tsp
MAX_TRIALS = 1000
RUNS = 10
MOVE_TYPE = 3
PATCHING_C = 3 RESTRICTED
PATCHING_A = 2 RESTRICTED
```

References

1. Applegate, D., Bixby, R., Chvátal, V., Cook, W.: Concorde: a code for solving traveling salesman problems. http://www.tsp.gatech.edu/concorde.html (1999)

2. Applegate, D., Bixby R., Chvátal V., Cook W.: Finding tours in the TSP. Technical Report 99885, Forschungsinstitut für Diskrete Mathematik, Universität Bonn (1999)
3. Applegate, D., Bixby, R., Chvátal, V., Cook, W.: Implementing the Dantzig–Fulkerson–Johnson algorithm for large traveling salesman problems. Math. Prog. **97**, 91–153 (2003)
4. Applegate, D., Cook, W., Rohe, A.: Chained Lin–Kernighan for large traveling salesman problems. INFORMS J. Comput. **15**, 82–92 (2003)
5. Bergeron, A.: A very elementary presentation of the Hannenhalli–Pevzner theory. LNCS **2089**, 106–117 (2001)
6. Caprara, A.: Sorting by reversals is difficult. In: Proceedings of the First International Conference on Computational Molecular Biology, pp. 75–83 (1997)
7. Chandra, B., Karloff, H., Tovey, C.: New results on the old k-opt algorithm for the TSP. In: Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 150–159 (1994)
8. Christofides, N., Eilon, S.: Algorithms for large-scale traveling salesman problems. Oper. Res. Quart. **23**, 511–518 (1972)
9. Fredman, M.L., Johnson, D.S., McGeoch, L.A., Ostheimer, G.: Data structures for traveling salesmen. J. Algorithms **18**(3), 432–479 (1995)
10. Funke, B., Grünert, T., Irnich, S.: Local search for vehicle routing and scheduling problems: review and conceptual integration. J. Heuristics **11**, 267–306 (2005)
11. Gutin, G., Punnen, A.P.: Traveling Salesman Problem and Its Variations. Kluwer, Dordrecht (2002)
12. Hanlon, P.J., Stanley, R.P., Stembridge, J.R.: Some combinatorial aspects of the spectra of normally distributed random matrices. Contemp. Math. **138**, 151–174 (1992)
13. Hannenhalli, S., Pevzer, P.A.: Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. In: Proceedings of the 27th ACM-SIAM Symposium on Theory of Computing, pp. 178–189 (1995)
14. Held, M., Karp, R.M.: The traveling-salesman problem and minimum spanning trees. Oper. Res. **18**, 1138–1162 (1970)
15. Held, M., Karp, R.M.: The traveling-salesman problem and minimum spanning trees: Part II. Math. Prog. **1**, 6–25 (1971)
16. Helsgaun, K.: An effective implementation of the Lin–Kernighan traveling salesman heuristic. EJOR **12**, 106–130 (2000)
17. Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann, Menlo Park (2004)
18. Johnson, D.S.: Local optimization and the traveling salesman problem. LNCS **442**, 446–461 (1990)
19. Johnson, D.S., McGeoch, L.A., Rothberg, E.E.: Asymptotic experimental analysis for the Held–Karp traveling salesman bound. In: Proceedings of 7th Ann. ACM-SIAM Symp. on Discrete Algorithms, pp. 341–350 (1996)
20. Johnson, D.S., McGeoch, L.A.: The traveling salesman problem: a case study in local optimization. In: Aarts, E.H.L., Lenstra, J.K. (eds.) Local Search in Combinatorial Optimization, pp. 215–310. Wiley, New York (1997)
21. Johnson, D.S., McGeoch, L.A., Glover, F., Rego, C.: Proceedings of the 8th DIMACS Implementation Challenge: The Traveling Salesman Problem. http://www.research.att.com/~dsj/chtsp/ (2000)
22. Johnson, D.S., McGeoch, L.A.: Experimental analysis of heuristics for the STSP. In: Gutin, G, Punnen, A., (eds.) The Traveling Salesman Problem and Its Variations, pp. 369–443 (2002)
23. Kaplan, H., Shamir, R., Tarjan, R.E.: Faster and simpler algorithm for sorting signed permutations by reversals. In: Proceedings of 8th annual ACM-SIAM Symp. on Discrete Algorithms (SODA 97), pp. 344–351 (1997)
24. Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., Shmoys, D.B. (eds.): The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. Wiley, New York (1985)
25. Lin, S., Kernighan, B. W.: An effective heuristic algorithm for the traveling-salesman problem. Oper. Res. **21**, 498–516 (1973)
26. Mak, K.T., Morton, A.J.: Distances between traveling salesman tours. Discret. Appl. Math. **58**, 281–291 (1995)
27. Martin, O., Otto, S.W., Felten, E.W.: Large-step Markov chains for the TSP incorporating local search heuristics. Oper. Res. Lett. **11**, 219–224 (1992)
28. Mendivil, D., Shonkwiler, R., Spruill, M.C.: An analysis of random restart and iterated improvement for global optimization with an application to the traveling salesman problem. J. Optim. Theory Appl. **124**(4), 407–433 (2005)

29. Möbius, A., Freisleben, B., Merz, P., Schreiber, M.: Combinatorial optimization by iterative partial transcription. Phys. Rev. E **59**(4), 4667–4674 (1999)
30. Neto, D.: Efficient Cluster Compensation For Lin–Kernighan Heuristics. PhD thesis, University of Toronto (1999)
31. Okada, M.: Studies on Probabilistic Analysis of λ-opt for Traveling Salesperson Problems. Doctor's thesis, Nara Institute of Science and Technology (1999)
32. Tannier, E., Sagot, M.: Sorting by reversals in subquadratic time. Rapport de recherche No 5097, l'INRIA (2004)
33. The On-Line Encyclopedia of Integer Sequences. http://www.research.att.com/~njas/sequences/A061714
34. Zhang, W., Looks, M.: A Novel Local Search Algorithm for the Traveling Salesman Problem that Exploits Backbones. IJCAI 2005, pp. 343–350 (2005)