An Effective Implementation of *K*-opt Moves for the Lin-Kernighan TSP Heuristic

Keld Helsgaun
E-mail: keld@ruc.dk

Computer Science
Roskilde University
DK-4000 Roskilde, Denmark

Abstract

Local search with k-change neighborhoods, k-opt, is the most widely used heuristic method for the traveling salesman problem (TSP). This report presents an effective implementation of k-opt for the Lin-Kernighan TSP heuristic. The effectiveness of the implementation is demonstrated with extensive experiments on instances ranging from 10,000 to 10,000,000 cities.

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 traveling salesman problem is to find the cheapest way of visiting all of the cities and returning to the starting point. 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, $(i, j = 1, ..., n)$, find a permutation $(i_1, i_2, i_3, ..., i_n)$ of the integers from 1 through n that minimizes the quantity
>
> $$c_{i_1 i_2} + c_{i_2 i_3} + \cdots + c_{i_n i_1}.$$

Writings on Computer Science, No. 109, Roskilde University, 2006
Revised November 20, 2007

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 $N = \{1, 2, ..., n\}$ 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 $\{(i_1, i_2), (i_2, i_3), ..., (i_k, i_1)\}$ 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 $\Sigma_{(i, j) \in T}\, c(i, j)$. An *optimal tour* is a tour of minimum weight.

For surveys of the problem and its applications, I refer the reader to the excellent volumes edited by Lawler et al. [26] and Gutin and Punnen [13].

Local search with *k*-change neighborhoods, *k*-opt, is the most widely used heuristic method for the traveling salesman problem. *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 [8] 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 [33]. Usually high-quality solutions are obtained in polynomial time. This is for example the case for the Lin-Kernighan heuristic [26], one of the most effective methods for generating optimal or near-optimal solutions for the symmetric traveling salesman problem. High-quality solutions are often obtained, even though only a small part of the *k*-change neighborhood is searched.

In the original version of the heuristic, the allowable *k*-changes (or *k-opt moves*) are restricted to those that can be decomposed into a 2- or 3-change followed by a (possibly empty) sequence of 2-changes. This restriction simplifies implementation, but it need not be the best design choice. This report explores the effect of widening the search.

The report describes LKH-2, an implementation of the Lin-Kernighan heuristic, which allows all those moves that can be decomposed into a sequence of K-changes for any K where $2 \leq K \leq n$. These K-changes may be sequential as well as non-sequential. LKH-2 is an extension and generalization of a previous version, LKH-1 [18], which uses a 5-change as its basic move component.

The rest of this report is organized as follows. Section 2 gives an overview of the 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 general k-opt moves are implemented in LKH-2. The effectiveness of the implementation is reported in Section 6. The evaluation is based on extensive experiments for a wide variety of TSP instances ranging from 10,000-city to 10,000,000-city instances. Finally, the conclusions about the implementation are given in Section 7.

2. The Lin-Kernighan Algorithm

The Lin-Kernighan algorithm [27] belongs to the class of so-called *local search algorithms* [19, 20, 22]. A local search algorithm starts at some location in the search space and subsequently moves from the present location to a neighboring location. 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, $X = \{x_1, ..., x_k\}$ and $Y = \{y_1, ..., y_k\}$,, 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 2.1 illustrates a 3-opt move.

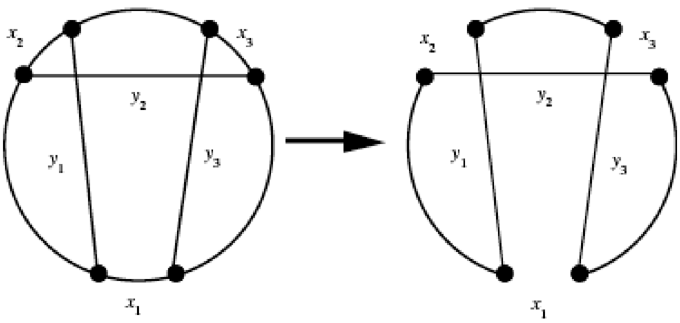

Figure 2.1. A 3-opt move.
x_1, x_2, x_3 are replaced by y_1, y_2, y_3.

In order to achieve a sufficiently efficient algorithm, only edges 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 Figure 2.2.

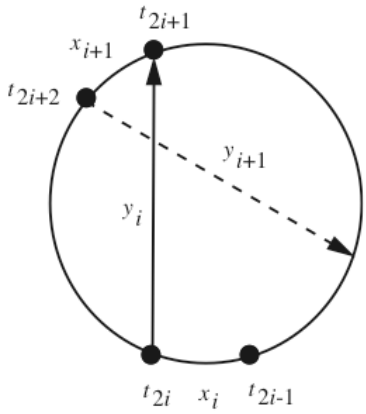

Figure 2.2. Restricting the choice of x_i, y_i, x_{i+1}, and y_{i+1}.

As seen, the sequence $(x_1, y_1, x_2, y_2, x_3, ..., x_k, y_k)$ 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 Figure 2.3.

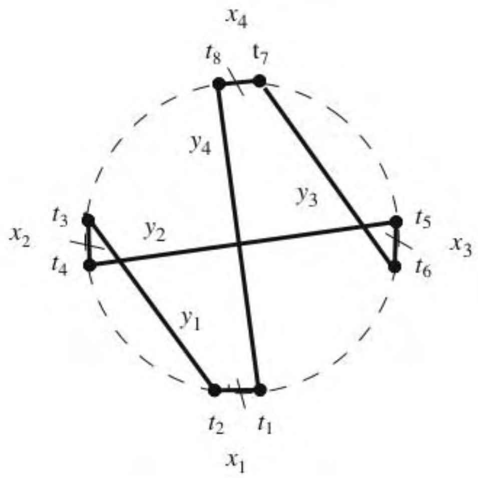

Figure 2.3. Alternating cycle (x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4).

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 2.4 shows an example where a sequential exchange is not possible.

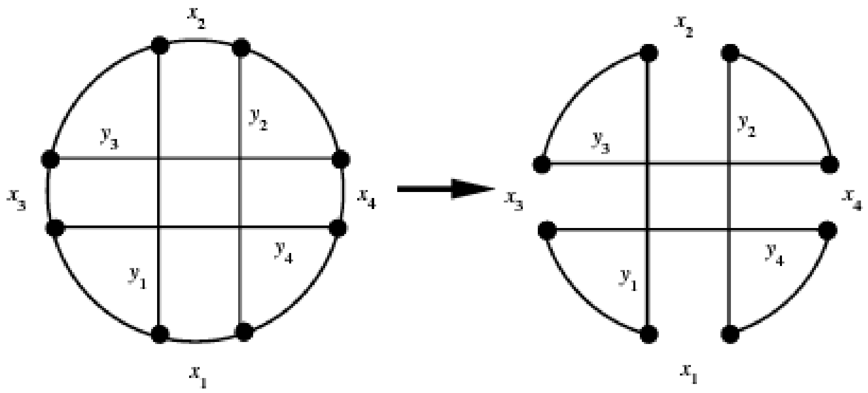

Figure 2.4. Non-sequential exchange (k = 4).

Note that all 2- and 3-opt moves are sequential. The simplest non-sequential move is the 4-opt move shown in Figure 2.4, 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 2.5 shows a sequential 4-opt move performed by a 2-opt move followed by two 2-opt moves.

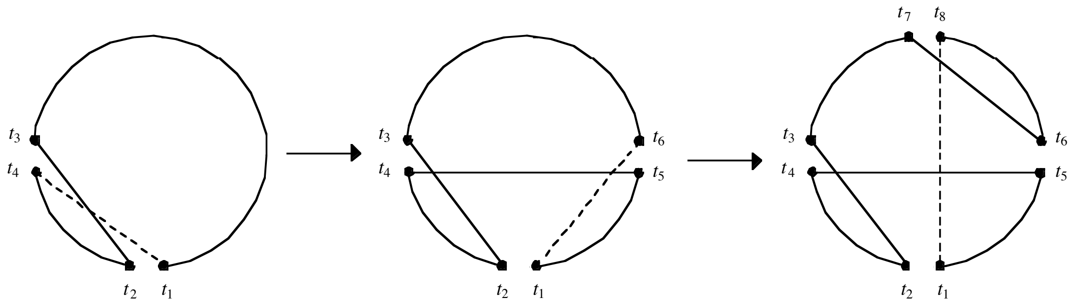

Figure 2.5. Sequential 4-opt move performed by three 2-opt moves. Close-up edges are shown by dashed lines.

(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 $g_1 + g_2 + ... + g_i$. 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 [18] 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 [18]).

(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 sub-move. 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 move 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 sub-moves 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 [16, 17], 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.

1. General K-opt moves

One of the most important means in LKH-2 for obtaining high-quality solutions is its use of general *K*-opt moves. 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 sub-moves, where *K* is any chosen integer greater than or equal to 2 and less than the number of cities. Each sub-move 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 last resort but are integrated into the ordinary search.

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. Even the solution of small problem instances may sometimes benefit from partitioning as this helps to focus the search process. Currently, LKH-2 implements the following six partitioning schemes:

(a) Tour segment partitioning

A given tour is broken up into segments of equal size. Each segment induces a subproblem consisting of all nodes and edges of the segment together with a fixed edge between the segment's two endpoints. When a segment is improved it is put back into the overall tour. After all subproblems of this partition have been treated, a revised partition is used where each new segment takes half its nodes from each of two adjacent old segments. This partitioning scheme may be used in general, whereas the next five schemes require the problem to be geometric.

(b) Karp partitioning

The overall region containing the nodes is subdivided into rectangles such that each rectangle contains a specified number of nodes. The rectangles are found in a manner similar to that used in the construction of the *k-D* tree data structure. Each rectangle together with a given tour induces a subproblem consisting of all nodes inside the rectangle, and with edges fixed between cities that are connected by tour segments whose interior points are outside the rectangle.

(c) Delaunay partitioning

The Delaunay graph is used to divide the set of nodes into clusters. The edges in the Delaunay graph are examined in increasing order of their lengths and added to a cluster if the cluster's size does not exceed a prescribed maximum size. Each cluster together with a given tour induces a subproblem consisting of all nodes in the cluster, and with edges fixed between nodes that are connected by tour segments whose interior points do not belong to the cluster.

(d) *K*-means partitioning

K-means is a least-squares partitioning method that divides the set of nodes into *K* clusters such that the total distance between all nodes and their cluster centroids is minimized. Each cluster together with a given tour induces a subproblem consisting of all nodes in the cluster, and with edges fixed between nodes that are connected by tour segments whose interior points do not belong to the cluster.

(e) Sierpinski partitioning

A tour induced by the Sierpinski spacefilling curve [34] is partitioned into segments of equal size. Each of these segments together with a given tour induces a subproblem. After all subproblems of this partition have been treated, a revised partition of the Sierpinski tour is used where each new segment takes half its nodes from each of two adjacent old segments.

(f) Rohe partitioning [35, 36]

Random rectangles are used to partition the node set into disjoint subsets of about equal size. Each of these subsets together with a given tour induces a subproblem.

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 moves for rather large values of K.

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, Freisleben, Merz and Schreiber [31].

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. This algorithm, which is a simplified version of the algorithm given by Zhang and Looks [38], has proved particularly effective for VLSI instances.

The rest of the report describes and evaluates the implementation of general K-opt moves in LKH-2. Partitioning, tour merging, iterative partial transcription and backbone-guided search will not be described further in this report.

5. Implementation of General *K*-opt Moves

This section describes the implementation of general *K*-opt moves 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 execute a feasible move efficiently.

The involved algorithms are specified by means of the C programming language. The code closely follows the actual implementation in LKH-2.

5.1 Search for sequential moves

A sequential *K*-opt move on a tour *T* may be specified by a sequence of nodes, $(t_1, t_2, ..., t_{2K-1}, t_{2K})$, 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 2*K* that fulfill the two requirements. Generation of such *t*-sequences may, for example, be performed iteratively in 2*K* nested loops, where the loop at level *i* goes through all possible values for t_i.

Suppose the first element, t_1, of the t-sequence has been selected. Then an iterative algorithm for generating the remaining elements may be implemented as shown in the pseudo code below.

```
for (each t[2] in {PRED(t[1]), SUC(t[1])}) {
    G[0] = C(t[1], t[2]);
    for (each candidate edge (t[2], t[3])) {
        if (t[3] != PRED(t[2]) && t[3] != SUC(t[2]) &&
            (G[1] = G[0] - C(t[2], t[3]) > 0) {
            for (each t[4] in {PRED(t[3]), SUC(t[3])}) {
                G[2] = G[1] + C(t[3], t[4]);
                if (FeasibleKOptMove(2) &&
                    (Gain = G[2] - C(t[4], t[1])) > 0) {
                    MakeKOptMove(2);
                    return Gain;
                }
            }
            inner loops for choosing t[5], ..., t[2K]
        }
    }
}
```

Comments:

- The t-sequence is stored in an array, t, of nodes. The three outermost loops choose the elements $t[2]$, $t[3]$, and $t[4]$. The operations *PRED* and *SUC* return for a given node respectively its predecessor and successor on the tour,

- The function $C(t_a, t_b)$, where t_a and t_b are two nodes, returns the cost of the edge (t_a, t_b).

- The array G is used to store the accumulated gain. It is used for checking that the positive gain criterion is fulfilled and for computing the gain of a feasible move.

- The function *FeasibleKOptMove(k)* determines whether a given t-sequence, $(t_1, t_2, ..., t_{2k-1}, t_{2k})$, represents a feasible k-opt move, where $2 \leq k \leq K$. The function *MakeKOptMove(k)* executes a feasible k-opt move. The implementation of these functions is described in Sections 5.3 and 5.4.

- Note that, if during the search a gainful feasible move is discovered, the move is executed immediately.

The inner loops for determination of t_5, ..., t_{2K} may be implemented analogous to the code above. The innermost loop, however, has one extra task, namely to register the non-gainful *K*-opt move that seems to be the most promising one for continuing the chain of *K*-opt moves. The innermost loop may be implemented as follows:

```
for (each t[2 * K] in {PRED(t[2 * K - 1], SUC[t[2 * K - 1]]}) {
    G[2 * K] = G[2 * K - 1] + C(t[2 * K - 1], t[2 * K]);
    if (FeasibleKOptMove(K)) {
        if ((Gain = G[2 * K] - C(t[2 * K], t[1])) > 0) {
            MakeKOptMove(K);
            return Gain;
        }
        if (G[2 * K] > BestG2K &&
            Excludable(t[2 * K - 1], t[2 * K])) {
            BestG2K = G[2 * K];
            for (i = 1; i <= 2 * K; i++)
                Best_t[i] = t[i];
        }
    }
}
```

Comments:

- The feasible *K*-opt move that maximizes the cumulative gain, *G*[2*K*], is considered to be the most promising move.

- The function *Excludabl*e 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 moves (Criterion 4 in Section 3).

Generation of 5-opt moves in LKH-1 was implemented as described above. 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. A recursive implementation of the algorithm is given below.

```
GainType BestKOptMoveRec(int k, GainType G0) {
    GainType G1, G2, G3, Gain;
    Node *t1 = t[1], *t2 = t[2 * k - 2], *t3, *t4;
    int i;

    for (each candidate edge (t2, t3)) {
        if (t3 != PRED(t2) && t3 != SUC(t2) &&
            !Added(t2, t3, k - 2) &&
            (G1 = G0 – C(t2, t3)) > 0) {
            t[2 * k - 1] = t3;
            for (each t4 in {PRED(t3), SUC(t3)}) {
                if (!Deleted(t3, t4, k - 2)) {
                    t[2 * k] = t4;
                    G2 = G1 + C(t3, t4);
                    if (FeasibleKOptMove(k) &&
                        (Gain = G2 - C(t4, t1)) > 0) {
                        MakeKOptMove(k);
                        return Gain;
                    }
                    if (k < K &&
                        (Gain = BestKOptMoveRec(k + 1, G2)) > 0)
                        return Gain;
                    if (k == K && G2 > BestG2K &&
                        Excludable(t3, t4)) {
                        BestG2K = G2;
                        for (i = 1; i <= 2 * K; i++)
                            Best_t[i] = t[i];
                    }
                }
            }
        }
    }
}
```

Comment:

- The auxiliary functions *Added* and *Deleted* are used to ensure that no edge is added or deleted more than once in the move under construction. Possible implementations of the two functions are shown below. It is easy to see that the time complexity for each of these functions is $O(k)$.

```
int Added(Node *ta, Node *tb, int k) {
    int i = 2 * k;
    while ((i -= 2) > 0)
        if ((ta == t[i] && tb == t[i + 1]) ||
            (ta == t[i + 1] && tb == t[i]))
            return 1;
    return 0;
}
```

```
int Deleted(Node * ta, Node * tb, int k) {
    int i = 2 * k + 2;
    while ((i -= 2) > 0)
        if ((ta == t[i - 1] && tb == t[i]) ||
            (ta == t[i] && tb == t[i - 1]))
        return 1;
    return 0;
}
```

Given two neighboring nodes on the tour, t_1 and t_2, the search for a K-opt move is initiated by a call of the driver function *BestKOptMove* shown below.

```
Node *BestKOptMove(Node *t1, Node *t2,
                   GainType *G0, GainType *Gain) {
    t[1] = t1; t[2] = t2;
    BestG2K = MINUS_INFINITY;
    Best_t[2 * K] = NULL;
    *Gain = BestKOptMoveRec(2, *G0);
    if (*Gain <= 0 && Best_t[2 * K] != NULL) {
        for (i = 1; i <= 2 * K; i++)
            t[i] = Best_t[i];
        MakeKOptMove(K);
    }
    return Best_t[2 * K];
}
```

This implementation makes is simple to construct a chain of K-opt moves:

```
GainType G0 = C(t1, t2), Gain;
do
    t2 = BestKOptMove(t1, t2, &G0, &Gain);
while (Gain <= 0 && t2 != NULL);
```

The loop is left as soon as the chain represents a gainful move, or when the chain cannot be prolonged anymore. In theory, there will be up to n iterations in the loop, where n is the number of nodes. In practice, however, the number of iterations is much smaller (due to the positive gain criterion).

The time complexity for each of the iterations, that is, for each call of *BestKOptMove*, may be evaluated from the time complexities for the sub-operations involved. In the following sections it is shown that these sub-operations may be implemented with the time complexities given in Table 5.1.1.

Operation	*Complexity*
PRED	$O(1)$
SUC	$O(1)$
Excludable	$O(1)$
Added	$O(K)$
Deleted	$O(K)$
FeasibleKOptMove	$O(K\log K)$
MakeKOptMove	$O(\sqrt{N})$

Table 5.1.1. Complexities for operations involved in the search for sequential moves.

Let d denote the maximum node degree in the candidate graph. Then it is easy to see that the worst case time complexity for an iteration is $O(d^K K\log K + \sqrt{n})$. If d and K are small compared to n, which usually is the case, then the worst-time complexity is $O(\sqrt{n})$.

Note, however, that the quantity $d^K K log K$ grows exponentially with K if $d \geq 2$, and that this term quickly becomes the dominating one. 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 Figure 2.4).

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 patch these cycles by one or more alternating cycles. Suppose, for example, that execution of a non-feasible k-opt move, $k \geq 4$, would result in four disjoint cycles. As shown in Figure 5.2.1 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.

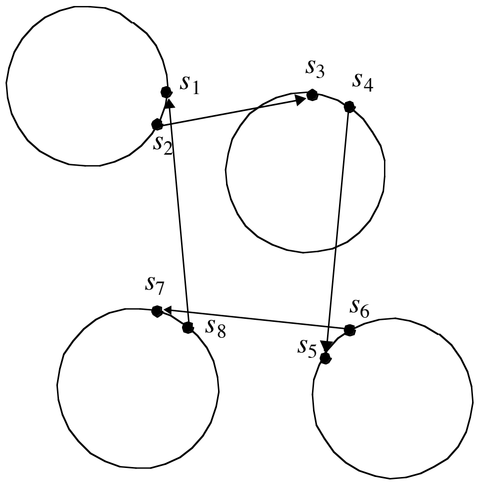

Figure 5.2.1. Four disjoint cycles patched by one alternating cycle.

Figure 5.2.2 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.

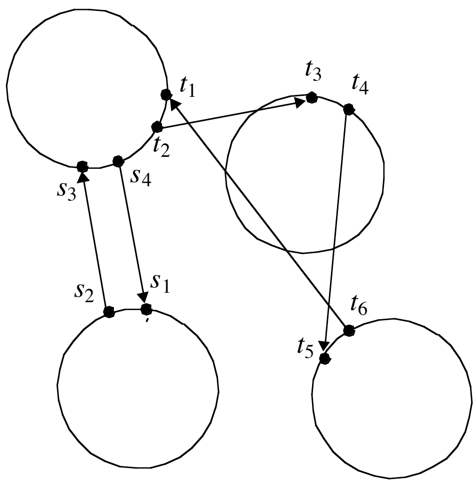

Figure 5.2.2. Four disjoint cycles patched by two alternating cycles.

Figure 5.2.3 illustrates that it is also possible to use three alternating cycles: (s_1, s_2, s_3, s_4, s_6), (t_1, t_2, t_3, t_4, t_1), and (u_1, u_2, u_3, u_4, u_1). In general, K cycles may be transformed into a tour using up to K-1 alternating cycles.

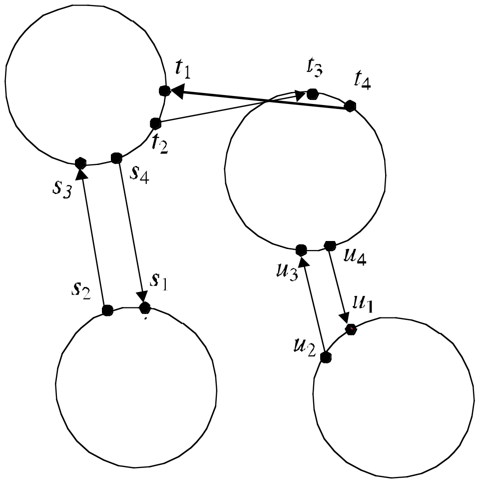

Figure 5.2.3. Four disjoint cycles patched by three alternating cycles.

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. A k-opt move removes k edges from a tour and adds k edges that reconnect k tour segments into a tour. Hence, a k-opt move may be defined as a cyclic permutation of k tour segments where some of the segments may be inverted (swapped). Let one of the segments be fixed. 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:

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

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. Funke, Grünert and Irnich [11, p. 284] give the following recursion formula for $PMT(k)$:

$$PMT(k) = MT(k) - \sum_{i=2}^{k-1} \binom{k}{i} PMT(i) - 1 \text{ for } k \geq 3$$

$$PMT(2) = 1$$

An explicit formula for *PMT*(*k*) may be derived from the formula for series A061714 in The On-Line Encyclopedia of Integer Sequences [40]:

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

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 by Hanlon, Stanley and Stembridge [14]:

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

where

$$c_{a,b}(2) =$$

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

Table 5.2.1 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.

k	2	3	4	5	6	7	8	9	10	50	100
MT(k)	2	8	48	384	3840	46080	645120	1.0E7	1.9E8	3.4E77	5.9E185
PMT(k)	1	4	25	208	2121	25828	365457	5.9E6	1.1E8	2.1E77	3.6E185
SPMT(k)	1	4	20	148	1348	15104	198144	3.0E6	5.1E7	4.3E76	5.9E184
$\frac{SPMT(k)}{PMT(k)}$	1	1	0.80	0.71	0.63	0.58	0.54	0.51	0.48	0.21	0.17

Table 5.2.1. Growth of move types for k-opt 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 given a set of 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:

(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.

(4) The starting node for an alternating cycle must belong to the shortest cycle (the cycle 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 Figures 5.2.1-3. 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 $M \geq 2$ 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 k-opt move, where $K + 2M/2 \leq k \leq K + 4(M - 1)/2$, that is, $K + M \leq k \leq K + 2M - 2$. Since M at most can be K, we can conclude that Rules1-3 permit non-sequential k-opt moves, where $K + 2 \leq k \leq 3K - 2$. For example, if a non-feasible 5-opt move results in 5 cycles, it may be the starting point for finding a non-sequential 7- to 13-opt move.

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 analogue with the positive gain criterion for sequential moves (see Section 2). During the construction of a move, no alternating cycle will be closed unless the cumulated gain plus the cost of 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 $(s_1, s_2, ..., s_{2m-2}, s_{2m-1}, s_{2m}, s_1)$, 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 (*Patching_C*) for the number of cycles that can be patched, and an upper limit (*Patching_A*) for the number of alternating cycles to be used for patching.

In the following I will describe how cycle patching is implemented in LKH-2. To make the description more comprehensible I will first show the implementation of an algorithm that performs cycle patching by use of only one alternating cycle. The algorithm is realized as a function, *PatchCycles*, which calls a recursive auxiliary function called *PatchCyclesRec*.

```
GainType PatchCycles(int k, GainType Gain) {
    Node *s1, *s2, *sStart, *sStop;
    GainType NewGain;
    int M, i;

    FindPermutation(k);
    M = Cycles(k);
    if (M == 1 || M > Patching_C)
        return 0;
    CurrentCycle = ShortestCycle(M, k);
    for (i = 0; i < k; i++) {
        if (cycle[p[2 * i]] == CurrentCycle) {
            sStart = t[p[2 * i]];
            sStop = t[p[2 * i + 1]];
            for (s1 = sStart; s1 != sStop; s1 = s2) {
                t[2 * k + 1] = s1;
                t[2 * k + 2] = s2 = SUC(s1);
                if ((NewGain =
                    PatchCyclesRec(k, 2, M,
                                   Gain + C(s1, s2))) > 0)
                    return NewGain;
            }
        }
    }
    return 0;
}
```

Comments:

- The function is called from the inner loop of *BestKOptMoveRec*:

```
if (t4 != t1 && Patching_C >= 2 &&
    (Gain = G2 - C(t4, t1)) > 0 &&   // rule 5
    (Gain = PatchCycles(k, Gain)) > 0)
    return Gain;
```

- The parameter *k* specifies that, given the non-feasible *k*-opt move represented by the nodes in the global array *t*[1..2*k*], the function should try to find a gainful feasible move by cycle patching. The parameter *Gain* specifies the gain of the non-feasible input move.

- 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 in $t[1..2k]$ occur on the tour in a clockwise direction, starting with $t[1]$. For example, p = (1 2 4 3 9 10 7 8 5 6) for the non-feasible 5-opt move shown in Figure 5.2.4. Next, the function *Cycles* is called to determine the number of cycles and to associate with each node in $t[1..2k]$ the number of the cycle it belongs to. Execution of the 5-opt in Figure 5.2.4 move produces two cycles, one represented by the node sequence (t_1, t_{10}, t_7, t_6, t_1), and one represented by the node sequence (t_2, t_4, t_5, t_8, t_9, t_3, t_2). The nodes of the first sequence are labeled with 1, the nodes of the second sequence with 2.

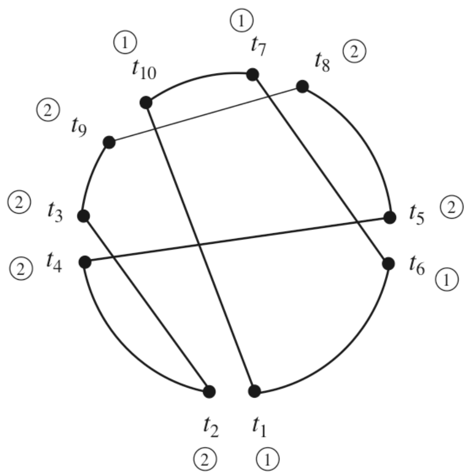

Figure 5.2.4. Non-feasible 5-opt move (2 cycles).

Next, the function *ShortestCycle* (shown below) is called in order to find the shortest cycle. The function returns the number of the cycle that contains the lowest number of nodes.

```
int ShortestCycle(int M, int k) {
    int i, MinCycle, MinSize = INT_MAX;
    for (i = 1; i <= M; i++)
        size[i] = 0;
    p[0] = p[2 * k];
    for (i = 0; i < 2 * k; i += 2)
        size[cycle[p[i]]] +=
            SegmentSize(t[p[i]], t[p[i + 1]]);
    for (i = 1; i <= M; i++) {
        if (size[i] < MinSize) {
            MinSize = size[i];
            MinCycle = i;
        }
    }
    return MinCycle;
}
```

- The tour segments of the shortest cycle are 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 $0 \le i < k$ and $p[0] = p[2k]$. 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.

- Each tour edge of the shortest cycle may now be used as the first out-edge, (s_1, s_2), of an alternating cycle. An alternating cycle is constructed by the recursive function *PatchCyclesRec* shown below.

```
GainType PatchCyclesRec(int k, int m, int M, GainType G0) {
    Node *s1, *s2, *s3, *s4;
    GainType G1, G2, G3, G4, Gain;
    int NewCycle, cycleSaved[1 + 2 * k], i;

    s1 = t[2 * k + 1];
    s2 = t[i = 2 * (k + m) - 2];
    incl[incl[i] = i + 1] = i;
    for (i = 1; i <= 2 * k; i++)
        cycleSaved[i] = cycle[i];
    for (each candidate edge (s2, s3)) {
        if (s2 != PRED(s2) && s3 != SUC(s2) &&
            (NewCycle = FindCycle(s3, k)) != CurrentCycle) {
            t[2 * (k + m) - 1] = s3;
            G1 = G0 – C(s2, s3);
            for (each s4 in {PRED(s3), SUC(s3)}) {
                if (!Deleted(s3, s4, k)) {
                    t[2 * (k + m)] = s4;
                    G2 = G1 + C(s3, s4);
                    if (M > 2) {
                        for (i = 1; i <= 2 * k; i++)
                            if (cycle[i] == NewCycle)
                                cycle[i] = CurrentCycle;
                            if ((Gain =
                                 PatchCyclesRec(k, m + 1, M - 1,
                                                G2)) > 0)
                                return Gain;
                            for (i = 1; i <= 2 * k; i++)
                                cycle[i]= cycleSaved[i];
                    } else if (s4 != s1 &&
                               (Gain = G2 – C(s4, s1)) > 0) {
                        incl[incl[2 * k + 1] = 2 * (k + m)] =
                                               2 * k + 1;
                        MakeKOptMove(k + m);
                        return Gain;
                    }
                 }
            }
        }
    }
    return 0;
}
```

Comments:

- The algorithm is rather simple and very much resembles the *Best-KOptMoveRec* function. The parameter k is used to specify that a non-feasible k-opt move be used as the starting point for finding a gainful feasible move. The parameter m specifies the current number of out-edges of the alternating cycle under construction. The parameter M specifies the current number of cycle components ($M \geq 2$). The last parameter, *G0*, specifies the accumulated gain.

- If a gainful feasible move is found, then this move is executed by calling *MakeKOptMove*($k + m$) before the achieved gain is returned.

- A move is represented by the nodes in the global 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. In order to be able to determine quickly whether an edge is an in-edge or an 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 Figure 5.2.4 *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.

- Let *s*2 be the last node added to the t-sequence. At each recursive call of *PatchCyclesRec* the t-sequence is extended by two nodes, *s3* and *s4,* such that

  (a) (*s2*, *s3*) is a candidate edge,
  (b) *s3* belongs to a not yet visited cycle component,
  (c) *s*4 is a neighbor to *s3* on the tour, and
  (d) the edge (*s3*, *s4*) has not been deleted before.

- Before a recursive call of the function all cycle numbers of those nodes of $t[1..2k]$ that belong to *s*3's cycle component are changed to the number of the current cycle component (which is equal to the number of *s*2's cycle component).

The time complexity for a call of *PatchCyclesRec* 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 5.2.2. Let d denote the maximum node degree in the candidate graph. Then it is easy to see that the worst case time complexity for each call is $O(nd^K K\log K + \sqrt{n})$. The factor n is due to the fact that the shortest cycle in worst case contains at most $n/2$ nodes. The factor $K\log K$ reflects that the permutation p is determined by sorting the nodes 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.

Operation	*Complexity*
PRED	$O(1)$
SUC	$O(1)$
FindPermutation	$O(K\log K)$
Cycles	$O(K)$
ShortestCycle	$O(K)$
FindCycle	$O(\log K)$
Deleted	$O(K)$
MakeKOptMove	$O(\sqrt{n})$

Table 5.2.2. Complexities for the sub-operations of PatchCyclesRec.

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 to the function *patchCyclesRec*.

First, the following code fragment is inserted just after the recursive call:

```
if (Patching_A >= 2 &&
    (Gain = G2 - C(s4, s1)) > BestCloseUpGain) {
    Best_s3 = s3;
    Best_s4 = s4;
    BestCloseUpGain = Gain;
}
```

If the user has decided that two or more alternating cycles may be used for patching (*Patching_A* ≥ 2), this code finds the pair of nodes (*Best_s3*, *Best_s4*) that maximizes the gain of *closing up* the current alternating cycle (Rule 6):

$$- c(s_2, s_3) + c(s_3, s_4) - c(s_4, s_1)$$

The current best close-up gain, *BestCloseUpGain*, is initialized to zero (Rule 5).

Second, the last statement of *PatchCyclesRec* (the *return* statement) is replaced by the following code:

```
Gain = 0;
if (BestCloseUpGain > 0) {
    int OldCycle = CurrentCycle;
    t[2 * (k + m) - 1] = Best_s3;
    t[2 * (k + m)] = Best_s4;
    Patching_A--;
    Gain = PatchCycles(k + m, BestCloseUpGain);
    Patching_A++;
    for (i = 1; i <= 2 * k; i++)
        cycle[i]= cycleSaved[i];
    CurrentCycle = OldCycle;
}
return Gain;
```

Note that this causes a recursive call of *PatchCycles* (not *PatchCyclesRec*).

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 Figure 5.3.1.

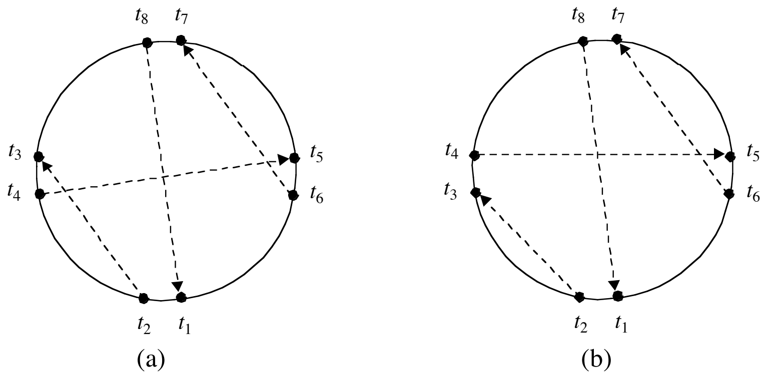

Figure 5.3.1. (a) Feasible 4-opt move. (b) Non-feasible 4-opt move.

Figure 5.3.1a 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 Figure 5.3.1b 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 cycle. 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 Figure 5.3.1a, and jump from t-node to t-node following the direction of the arrows, then all t-nodes are visited in the following sequence: t_6, t_7, t_6, 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 ($1 \le a,b \le 2k$), this fact is registered in the array by setting $incl[a] = b$ and $incl[b] = a$. 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. 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. 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. I will now show 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 Figure 5.3.1 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 $p_{half}[i]$ is odd, then insert $p_{half}[i] + 1$ after $p_{half}[i]$, otherwise insert $p_{half}[i] - 1$ after $p_{half}[i]$.

Let a move be represented by the nodes $t[1..2k]$. Then the function *Find-Permutation* shown below is able 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 function determines $q[1..2k]$ as the inverse permutation to p, that is, the permutation for which $q[p[i]] = i$ for $1 \le i \le 2k$.

```
void FindPermutation(int k) {
    int i,j;
    for (i = j = 1; j <= k; i += 2, j++)
        p[j] = (SUC(t[i]) == t[i + 1]) ? i : i + 1;
    qsort(p + 2, k - 1, sizeof(int), Compare);
    for (j = 2 * k; j >= 2; j -= 2) {
        p[j - 1] = i = p[j / 2];
        p[j] = i & 1 ? i + 1 : i - 1;
    }
    for (i = 1; i <= 2 * k; i++)
        q[p[i]] = i;
}
```

Comments:

- First, the *k* indices to be sorted are placed in *p*. This takes *O*(*k*) time. Next, the sorting is performed. The first element of *p*, *p*[1] is fixed and does not take part in the sorting. Thus, only *k*-1 elements are sorted. The sorting is performed by C's library function for sorting, *qsort*, using the comparison function shown below:

```
int Compare(const void *pa, const void *pb) {
    return BETWEEN(t[p[1]], t[*(int *) pa], t[*(int *) pb])
           ? -1 : 1;
}
```

  The operation *BETWEEN*(*a*, *b*, *c*) determines in constant time whether node *b* lies between node *a* and node *c* on the tour in the SUC-direction. Since the number of comparisons made by *qsort* on average is *O*(*k*log*k*), and each comparison takes constant time, the sorting process takes *O*(*k*log*k*) time on average.

- After the sorting, the full permutation, *p,* and its inverse, *q*, are built. This can be done in *O*(*k*) time.

- From the analysis above we find that the average time for calling *FindPermutation* with argument *k* is *O*(*k*log*k*).

After having determined p and q, we can quickly determine whether a k-opt move represented by the contents of the arrays t and *incl* is feasible. The function shown below shows how this can be achieved.

```
int FeasibleKOptMove(int k) {
    int Count = 1, i = 2 * k;
    FindPermutation(k);
    while ((i = q[incl[p[i]]] ^ 1) != 0)
        Count++;
    return (Count == k);
}
```

Comments:

- In each iteration of the *while*-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 possible t-nodes between $t[incl[p[i]]]$ and the next t-node on the cycle. If the position of $incl[p[i]]$ in p 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 starting value for i is $2k$. The loop terminates when i becomes zero, which happens when node $t[p[1]]$ has been visited (since 1 ^ 1 = 0, where ^ is the exclusive OR operator). The loop always terminates since both $t[p[2k]]$ and $t[p[1]]$ belong to the cycle that is traversed by the loop, and no node is visited more than once.

- It is easy to see that the loop is executed in $O(k)$ time. Since the sorting made by the call of *FindPermutation* on average takes $O(k\log k)$ time, we can conclude that the average-time complexity for the *FeasibleKOptMove* function 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.

- To see a concrete example of how the algorithm works, consider the 4-opt move in Figure 5.3.1a. Before the *while*-loop the following is true:

  $p = (1\ 2\ 4\ 3\ 8\ 7\ 5\ 6)$
  $q = (1\ 2\ 4\ 3\ 7\ 8\ 6\ 5)$
  $incl = (8\ 3\ 2\ 5\ 4\ 7\ 6\ 1)$

The controlling variable *i* is assigned the values 8, 7, 2, 5, 0 in that order, and the variable *Count* is incremented 3 times. Since *Count* becomes equal to 4, the 4-opt move is feasible.

For the 4-opt move in Figure 5.3.b the *while*-loop is started with

p = (1 2 3 4 8 7 5 6)
q = (1 2 3 4 7 8 6 5)
incl = (8 3 2 5 4 7 6 1)

The controlling variable *i* is assigned the values 8, 7, 5, 0 in that order. Since *Count* here becomes 3 (which is ≠ 4), the 4-opt move is not feasible.

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 [9, 28]. 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 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 [30].

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 Figure 5.4.1 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 Figure 5.4.2.

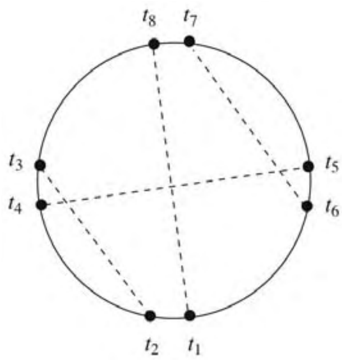

Figure 5.4.1. Feasible 4-opt move.

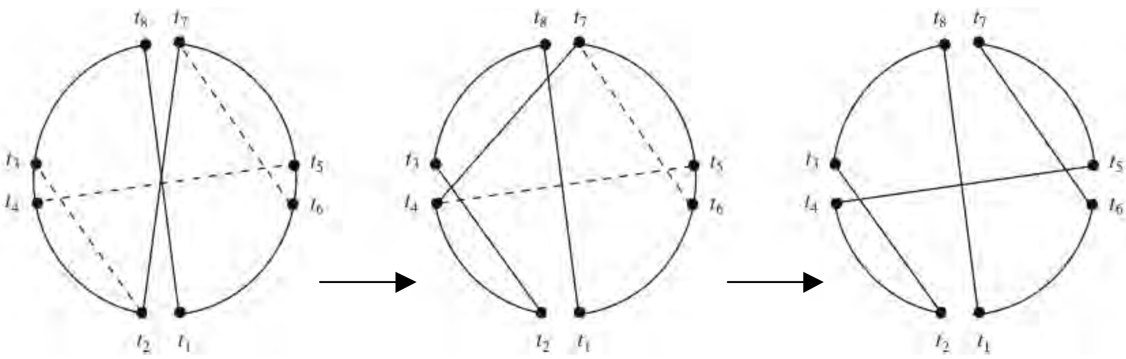

Figure 5.4.2. Execution of the 4-opt move by means of 3 flips.

The 4-opt move of Figure 5.4.1 may be executed by many other flip sequences, for example:

> *FLIP*(t$_2$, t_1, t_6, t_5)
> *FLIP*(t_5, t_1, t_8, t_7)
> *FLIP*(t_3, t_3, t_5, t_7)
> *FLIP*(t_2, t_6, t_7, t_3)

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

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

In the following I will show that an answer to this question can be found by transforming the problem into an equivalent problem, which has a known solution. Consider Figure 5.4.3, 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 to 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).

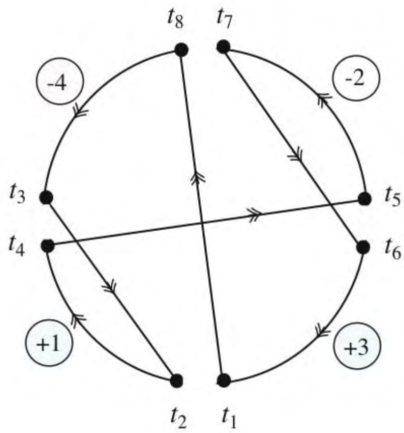

Figure 5.4.3. Segments labeled with orientation and rank.

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)

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 both the order and signs of the elements of a segment of the signed permutation. In Figure 5.4.3 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 -2 +3)
(+1 -4 -3 +2)
(+1 -2 +3 +4)
(+1 +2 +3 +4)

Reversed segments are underlined.

Suppose now that, given a signed permutation of {1, 2, ..., 2k}, we are able to determine the shortest possible sequence of signed reversals that transforms the permutation into the identity permutation(+1, +2, ..., +2k). 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 $\{1, ..., n\}$. 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 $\pi\rho_1 \dots \rho_{d(n)} = (1\ 2 \dots n\text{-}1\ n)$, 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 $\sigma\rho_1 \dots \rho_{d(n)} = (+1\ +2 \dots +(m\text{-}1)\ m)$.

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 $\mathcal{NP}$-hard [7], 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 [15]. Since then faster algorithms have been discovered, among others an $O(n^2)$ algorithm by Kaplan, Shamir and Tarjan [25]. The fastest algorithm for SSBR today has complexity $O(n\sqrt{n \log n})$ [37].

Several of these fast algorithms are difficult to implement. I have chosen to implement a very simple algorithm described by Bergeron [6]. 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 sufficient 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, $O(\sqrt{n})$.

Before describing Bergeron's algorithm and its implementation, some necessary definitions are stated [6].

Let $\pi = (\pi_1 ..., \pi_n)$ be a signed permutation. An *oriented pair* (π_i, π_j) is a pair of adjacent integers, that is $|\pi_i| - |\pi_j| = \pm 1$, with opposite signs. For example, the signed permutation

(+1 -2 -5 +4 +3)

contains three oriented pairs: (+1, -2), (-2, +3), and (-5, +4).

Oriented pairs are useful as they indicate reversals that cause adjacent integers to be consecutive in the resulting permutation. For example, the oriented pair (-2, +3) induces the reversal

(+1 <u>-2 -5 +4</u> +3) $\rightarrow$ (+1 -4 +5 +2 +3)

creating a permutation where +3 is consecutive to +2.

In general, the reversal induced by and oriented pair (π_i, π_j) is

> $\rho(i, j\text{-}1)$, if $\pi_i + \pi_j = +1$, and
> $\rho(i+1, j)$, if $\pi_i + \pi_j = -1$

Such a reversal is called an *oriented reversal*.

The *score* of an oriented reversal is defined as the number of oriented pairs in the resulting permutation. We may now formulate the following simple algorithm for SSBR:

> *Algorithm 1*:
> As long as there are ordered pairs, apply the ordered reversal that has maximum score.

It has been shown that this algorithm is able to optimally sort most random permutations.

Unless all elements of the permutation initially are negative, the algorithm will always terminate with a positive permutation. If the output permutation is the identity permutation (+1 +2 ... +*n*), we have succeeded. Otherwise, we must somehow see to it that an oriented pair is created, so that the algorithm above can be used again.

Consider the positive permutation below.

(+1 +4 +3 +2)

Reversal of the segment (+4) yields a permutation, which the algorithm is able to sort by additional 3 reversals:

(+1 -4 <u>+3</u> +2) → (+1 -4 -3 <u>+2</u>) → (+1 <u>-4 -3 -2</u>) → (+1 +2 +3 +4)

However, if we start by reversing the segment (+4 +3) instead, we save the first of the 3 reversals and are able to finish the sorting process using only two more reversals:

(+1 -4 -3 +2) → (+1 <u>-4 -3 -2</u>) → (+1 +2 +3 +4)

Thus, if we want a minimum number of reversals, we have to be careful in selecting the initial reversal in this case.

A possible strategy for selecting this reversal is the following.

> *Algorithm 2*:
> Find the pair $(+\pi_i, +\pi_j)$ for which $|\pi_j - \pi_i| = 1$, $j \geq i + 3$, and i is minimal. Then reverse the segment $(+\pi_{i+1} \ldots +\pi_{j-1})$.

In the example above, the pair (+1, +2) is found, and the segment (+4 +3) is reversed. In this example, this strategy is sufficient to sort the permutation optimally. However, we cannot be sure that this is always the case. If, for example, a permutation is sorted in reversed order, (+n ... +2 +1), there is no pair that fulfills the conditions above. However, this situation can be avoided by requiring that +1 is always the first element of a permutation. Unless the permutation is sorted, we are always able to find a pair that satisfies the condition.

It is easy to see that this strategy is sufficient to sort any permutation. When a positive permutation is created, we select the first pair $(+\pi_i, +\pi_j)$ for which $+\pi_i$ is in its correct position, and $+\pi_j$ is not. When the segment between these two elements is reversed, at least one oriented pair is created. After making the reversal induced by this oriented pair, the sign of $+\pi_j$ is reversed, so that a subsequent reversal can bring $+\pi_j$ to its correct position (right after $+\pi_i$).

The question now is whether the strategy is optimal. To answer this question, we will use the notion of a *cycle graph* of a permutation [5]. Let π be a signed permutation. The first step in constructing the cycle graph is to frame π by the two extra elements 0 and $n + 1$: $\pi = (0\ \pi_1\ \pi_2\ \ldots\ \pi_n\ n{+}1)$. Next, this permutation is mapped one-to-one into an unsigned permutation π' of $2n+2$ elements by replacing

- each positive element $+x$ in π by the sequence $2x$-1 $2x$,
- each negative element $-x$ in π by the sequence $2x$ $2x$+1, and
- n+1 by $2n$-1.

For example, the permutation

π = (+1 -4 -2 +3 5)

becomes

π' = (0 1 2 8 7 4 3 5 6 9)

Note that the mapping is one-to-one. The reversal $\rho(i, j)$ of π corresponds to the reversal $\rho(2i\text{-}1, 2j)$ of π'.

The cycle graph for π is an edge-colored graph of $2n$+2 nodes {π_0 π_1 ... π_{2n+2}}. For each i, $0 \le i \le n$, we join two nodes π_{2i} and π_{2i+1} by a *black* edge, and with a gray edge, if $|\pi_{2i+1} - \pi_{2i}| = 1$. Figure 5.4.4 depicts an example of a cycle graph. Black edges appear as straight lines. Gray edges appear as curved lines.

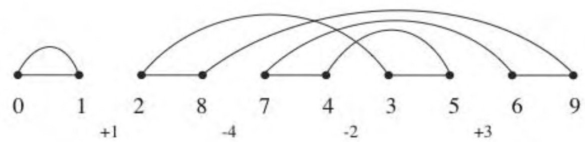

Figure 5.4.4. Cycle graph for π = (+1 -4 -2 +3).

The black edges are called *reality edges*, and the gray edges are called *dream edges* [37]. The reality edges define the permutation π' (what you have), and the dream edges define the identity permutation (what you want).

Every node has degree two, that is, every node has exactly two incident edges. Thus a cycle graph consists of a set of cycles. Each of these cycles is an *alternating* cycle, that is, adjacent edges have different colors.

We use *k-cycle* to refer to an alternating cycle of length k, that is, a cycle consisting of k edges. The cycle graph in Figure 5.5.4 consists of a 2-cycle and an 8-cycle. The cycle graph for the identity permutation (+1 +2 ... +n) consists of n+1 2-cycles (see Figure 5.4.5). Sorting by reversals can be viewed as a process that increases the number of cycles to n + 1.

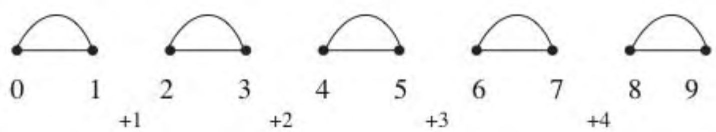

Figure 5.4.5. Cycle graph for π = (+1 +2 +3 +4).

It is easy to see that every reversal induced by an ordered pair increases the number of cycles by one. On the other hand, if there is no ordered pair (because the permutation is positive), then there is no single reversal, which will increase the number of cycles. In this case we will try to find a reversal, which without reducing the number of cycles creates one or more oriented pairs.

A cycle is said to be *oriented* if it contains edges that correspond to one or more oriented pairs; otherwise, it is said to be *non-oriented*. We must, as far as possible, avoid creating non-oriented cycles, unless they are 2-cycles.

Let π be positive permutation, and assume that π is reduced, that is, π does not contain consecutive elements. A *framed interval* in π is an interval of the form

$$i\ \pi_{j+1}\ \pi_{j+2}\ \dots\ \pi_{j+k-1}\ i{+}k$$

such that all its elements belong to the interval [i ... i+k]. In other words, a framed interval consists of integers that can be reordered as i i+1 ... i+k. Keep in mind that since π is a circular permutation, we may wrap around from the end to the start. Consider the permutation below (the plus signs have been omitted):

(1 4 7 6 5 8 3 2 9)

The whole permutation is a framed interval. The segment 4 7 6 5 8 is a framed interval, and the segment 3 2 9 1 4 is also a framed interval, since, by circularity, 1 is the successor of 9, and the permutation can be reordered as 9 1 2 3 4.

Now we can conveniently define a *hurdle*. A hurdle is a framed interval that does not contain any shorter framed intervals. In the example above, the whole permutation is not a hurdle, since it contains the framed interval 4 7 6 5 8. The framed intervals 4 7 6 5 8 and 3 2 9 1 4 are both hurdles, since they do not properly contain any framed intervals.

In order to get rid of hurdles, two operations are introduced: hurdle cutting and hurdle merging. The first one, *hurdle cutting*, consists in reversing the segment between *i* and *i* + 1 of a hurdle:

$$i\ \underline{\pi_{j+1}\ \pi_{j+2} \ldots}\ i{+}1 \ldots\ \pi_{j+k-1}\ i{+}k$$

Given the permutation (1 4 7 6 5 8 3 2 9), the result of cutting the hurdle 4 7 6 5 8 is the permutation (1 4 -6 -7 5 8 3 2 9).

It has been shown that when a permutation contains only one hurdle, one such reversal creates enough oriented pairs to completely sort the resulting permutation by means of oriented reversals. However, when a permutation contains two or more hurdles, this is not always the case. We need one more operation: *hurdle merging*.

Hurdle merging consists of reversing the segment between two end points of two hurdles, inclusive the two end points:

$$i\ \pi_{j+1}\ \pi_{j+2} \ldots\ \underline{i{+}k \ldots i'}\ \ldots\ i{+}k$$

It has been shown [15] that the following algorithm, together with Algorithm 1, can be used to optimally sort any signed permutation:

> *Algorithm 3*:
> If a permutation has an even number of hurdles, merge two consecutive hurdles. If a permutation has an odd number of hurdles, then if it has one hurdle whose cutting decreases the number of cycles, then cut it; otherwise, merge two non-consecutive hurdles, or consecutive hurdles if there are exactly three hurdles.

Since Algorithm 2, described earlier, corresponds to hurdle cutting we are not guaranteed an optimal sorting, unless the number of hurdles is less than 2. However, it is easy to see that those signed permutations we must be able sort in order to find the minimum number of flips for executing a *k*-opt have at most one hurdle. Consider the positive permutation (1 4 7 6 5 8 3 2), which has two hurdles: 7 6 5 8 and 3 2 1 4. Figure 5.4.6 depicts its cycle graph.

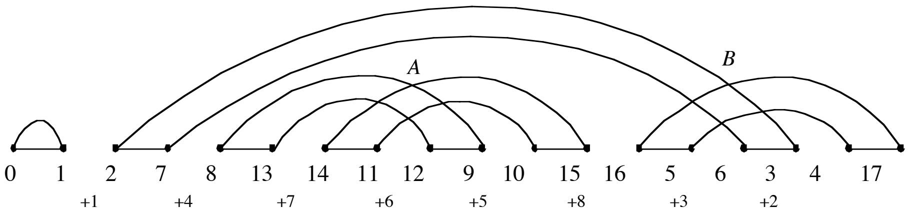

Figure 5.4.6. Cycle graph for a permutation with 2 hurdles (denoted A and B).

Figure 5.5.7 shows the corresponding 6-opt move. As seen this move consists of two independent sequential 3-opt moves. The first one involves the tour segments +1, +2 and +3. The other one involves the tour segments +4, +5 and +6.

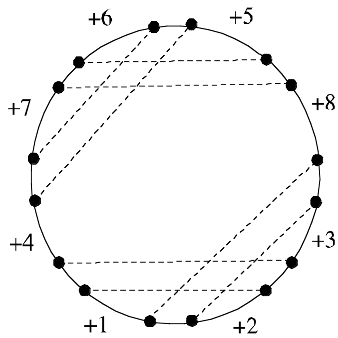

Figure 5.4.7. A 6-opt move that gives rise to two hurdles.

It is easy to see that a hurdle always corresponds to a feasible *k*-opt move. Thus, if there is more than one hurdle, then these hurdles correspond to independent feasible *k*-opt moves. But since independent feasible *k*-opt moves are not considered by LKH-2 when it searches for a move, Algorithm 2, together with Algorithm 1, will sort the involved permutations optimally.

Below is given a function, *MakeKOptMove*, which uses these algorithms for executing a *k*-opt move using a minimum number of *FLIP*-operations.

```
void MakeKOptMove(long k) {
    int i, j, Best_i, Best_j, BestScore, s;
    FindPermutation(k);
  FindNextReversal:
    BestScore = -1;
    for (i = 1; i <= 2 * k - 2; i++) {
        j = q[incl[p[i]]];
        if (j >= i + 2 && (i & 1) == (j & 1) &&
            (s = (i & 1) == 1 ? Score(i + 1, j, k) :
             Score(i, j - 1, k)) > BestScore) {
            BestScore = s; Best_i = i; Best_j = j;
        }
     }
     if (BestScore >= 0) {
         i = Best_i; j = Best_j;
         if ((i & 1) == 1) {
             FLIP(t[p[i + 1]], t[p[i]], t[p[j]], t[p[j + 1]]);
             Reverse(i + 1, j);
         } else {
             FLIP(t[p[i - 1]], t[p[i]], t[p[j]], t[p[j - 1]]);
             Reverse(i, j - 1);
         }
         goto FindNextReversal;
    }
    for (i = 1; i <= 2 * k - 1; i += 2) {
        j = q[incl[p[i]]];
        if (j >= i + 2) {
            FLIP(t[p[i]], t[p[i + 1]], t[p[j]], t[p[j - 1]]);
            Reverse(i + 1, j - 1);
            goto FindNextReversal;
        }
    }
}
```

Comments:

- At entry the move to be executed must be available in the two global arrays *t* and *incl*, where *t* contains the nodes of the alternating cycles, and *incl* represents the in-edges of these cycles, $\{(t[i], t[incl[i]]) : 1 \le i \le 2k\}$.

- First, the function *FindPermutation* is called in order to find the permutation p and its inverse q, where p gives the order in which the t-nodes occur on the tour (see Section 5.3).

- The first *for*-loop determines, if possible, an oriented reversal with maximum score. At each iteration, the algorithm explores the two cases shown in Figures 5.4.8 and 5.4.9.

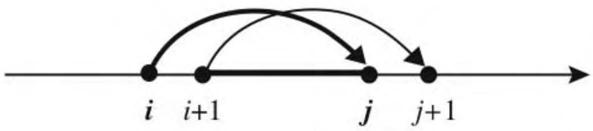

Figure 5.4.8. Oriented reversal: $j \geq i+2 \wedge i$ odd $\wedge j$ odd $\wedge$ $p[j] = incl[p[i]]$.

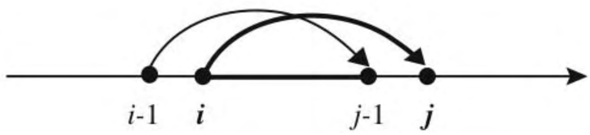

Figure 5.4.9. Oriented reversal: $j \geq i+2 \wedge i$ even $\wedge j$ even $\wedge$ $p[j] = incl[p[i]]$.

- If an oriented reversal is found, it is executed. Otherwise, the last *for*-loop of the algorithm searches for a non-oriented reversal (caused by a hurdle). See Figure 5.4.10.

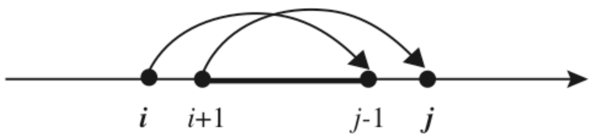

Figure 5.4.10 Non-oriented reversal: $j \geq i+3 \wedge i$ odd $\wedge j$ even $\wedge$ $p[j] = incl[p[i]]$.

- As soon as a reversal has been found, the reversal and the corresponding *FLIP*-operation on the tour are executed. The fourth argument of *FLIP* has been omitted, as it can be computed from the first three. The auxiliary function *Reverse*, shown below, executes a reversal of the permutation segment $p[i..j]$ and updates the inverse permutation q accordingly.

```
void Reverse(int i, int j) {
    for (; i < j; i++, j--) {
        int pi = p[i];
        q[p[i] = p[j]] = i;
        q[p[j] = pi] = j;
    }
}
```

- The auxiliary function *Score*, shown below, returns the score of the reversal of *p*[*left*..*right*].

```
int Score(int left, int right, int k) {
    int count = 0, i, j;
    Reverse(left, right);
    for (i = 1; i <= 2 * k - 2; i++) {
        j = q[incl[p[i]]];
        if (j >= i + 2 && (i & 1) == (j & 1))
            count++;
    }
    Reverse(left, right);
    return count;
}
```

Since the tour is represented by the two-level tree data structure [10], the complexity of the *FLIP*-operation is $O(\sqrt{N})$. The complexity of *Make-KOptMove* is $O(k\sqrt{N} + k^3)$, since there are at most *k* calls of *FLIP,* and at most k^2 calls of *Score*, each of which has complexity *O*(*k*).

The complexity of *Score* may be reduced from *O*(*k*) to O(log*k*) by using a balanced search tree [12]. This will reduce the complexity of *Make-KOptMove* to $O(k\sqrt{n} + k^2 \log k)$. However, this is of little practical value, since usually *k* is very small in relation to *n*, so that the term $k\sqrt{n}$ dominates.

The presented algorithm for executing a move 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. Computational Results

This section presents the results of a series of computational experiments, the purpose of which was to examine LKH-2's performance when general *K*-opt moves are used as submoves. The results include its qualitative performance and its run time efficiency. Run times are measured in seconds on a 2.7 GHz Power Mac G5 with 8 GB RAM.

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

E-instances: Instances consisting of uniformly distributed points in a square.
C-instances: Instances consisting of clustered points in a square.
M-instances: Instances consisting of random distance matrices.
R-instances: Real world instances.

The *E*-, *C*- and *M*-instances are taken from the 8th DIMACS Implementation Challenge [23]. The *R*-instances are taken from the TSP web page of William Cook et al. [39]. Sizes range from 10,000 to 10,000,000 cities.

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 I have selected those instances of 8th DIMACS Implementation Challenge that have 10,000 or more cites. Optima for these instances are currently unknown. I follow Johnson and McGeoch [24] in measuring tour quality in terms of percentage over the Held-Karp lower bound [16, 17] on optimal tours. The Held-Karp bound appears to provide a consistently good approximation to the optimal tour length [22].

Table 6.1 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:

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

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 [21] have estimated C_{OPT} to 0.7124 ± 0.0002. The last column of Table 6.1 contains these ratios. The results are consistent with this estimate for large *n*.

Instance	*n*	*CBT*	*HK bound*	*HK gap (%)*	$\frac{CBT}{\sqrt{n}} 10^{-6}$
E10k.0	10,000	71,865,826	71,362,276	0.706	0.7187
E10k.1	10,000	72,031,896	71,565,485	0.652	0.7203
E10k.2	10,000	71,824,045	71,351,795	0.662	0.7182
E31k.0	31,623	127,282,687	126,474,847	0.639	0.7158
E31k.1	31,623	127,453,582	126,647,285	0.637	0.7167
E100k.0	100,000	225,790,930	224,330,692	0.651	0.7140
E100k.1	100,000	225,662,408	224,241,789	0.634	0.7136
E316k.0	316,228	401,310,377	398,582,616	0.684	0.7136
E1M.0	1,000,000	713,192,435	708,703,513	0.633	0.7132
E3M.0	3,162,278	1,267,372,053	1,260,000,000	0.585	0.7127
E10M.0	10,000,000	2,253,177,392	2,240,000,000	0.588	0.7125

Table 6.1. Tour quality for E-instances.

Using cutting-plane methods Concorde [1] has found a lower bound of 713,003,014 for the 1,000,000-city instance *E1M.0* [4]. The bound shows that LKH's tour for this instance has a length at most 0.027% greater than the length of an optimal tour.

6.1.1 Results for E10k.0

The first test instance is *E10k.0*. Figure 6.1 depicts the current best tour for this instance.

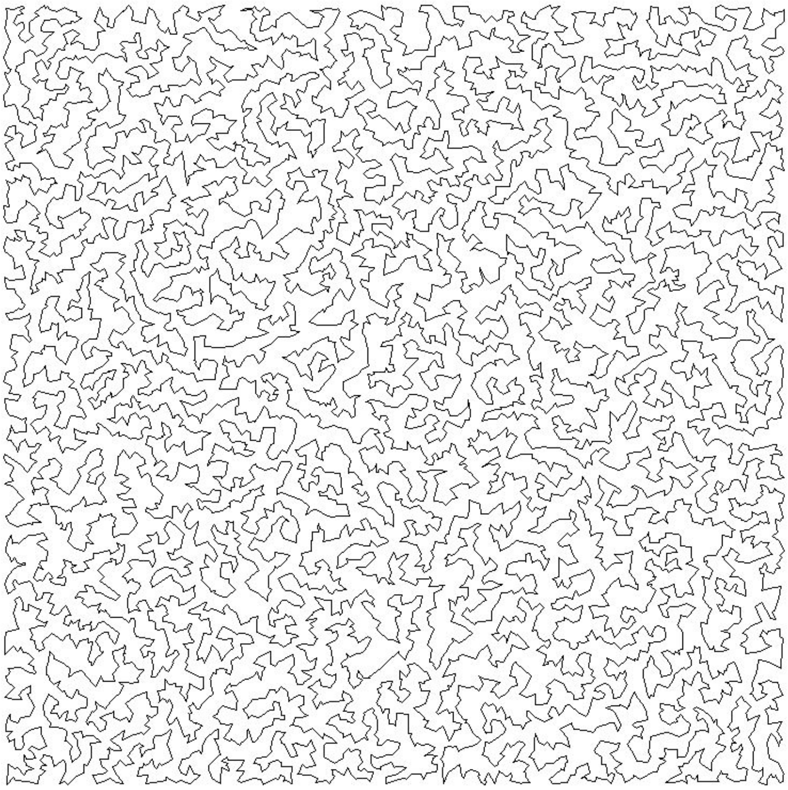

Figure 6.1. Best tour for E10k.0

First we will examine, for increasing values of K, how tour quality and CPU times are affected if we use non-sequential K-opt moves as submoves. The 5 α-nearest edges incident to each node are used as candidate set (see Figure 6.2). As many as 99.5% of the edges of the best tour belong to this set.

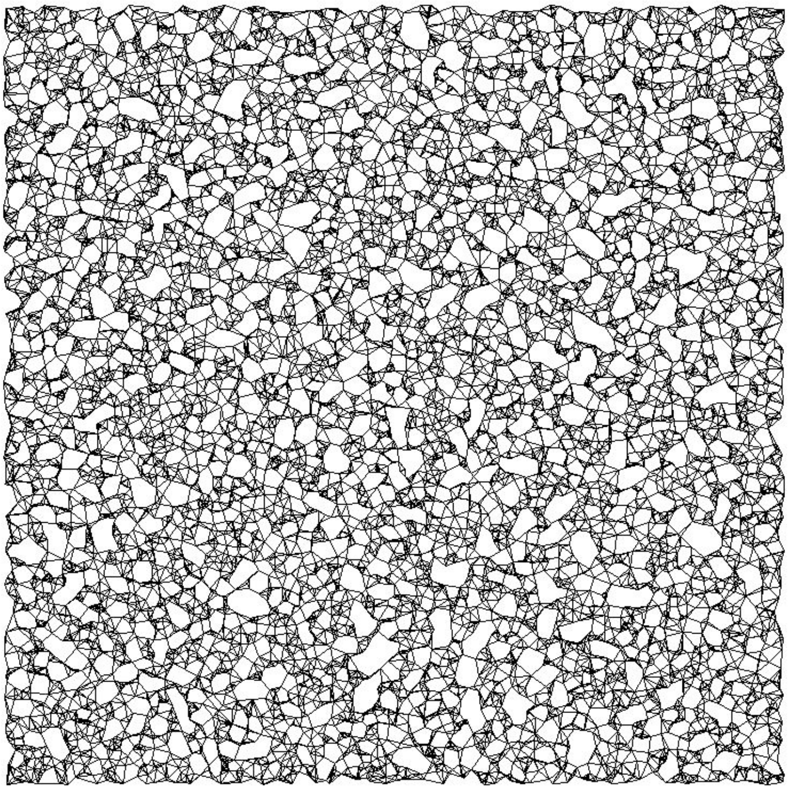

Figure 6.2. 5 α-nearest candidate set for E10k.0.

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 [18].

The results from these experiments are shown in Table 6.2. 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.

	HK gap (%)			Time (s)		
K	*min*	*avg*	*max*	*min*	*avg*	*max*
2	1.785	1.936	2.040	1	2	2
3	1.123	1.220	1.279	1	1	2
4	0.946	0.987	1.023	1	1	2
5	0.824	0.885	0.966	2	3	3
6	0.789	0.816	0.861	9	18	27
7	0.793	0.811	0.847	43	69	96
8	0.780	0.797	0.827	100	207	361

Table 6.2. Results for E10k.0, no patching
(1 trial, 10 runs, MOVE_TYPE = K, PATCHING_C = 0).

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 (Figures 6.3 and 6.4).

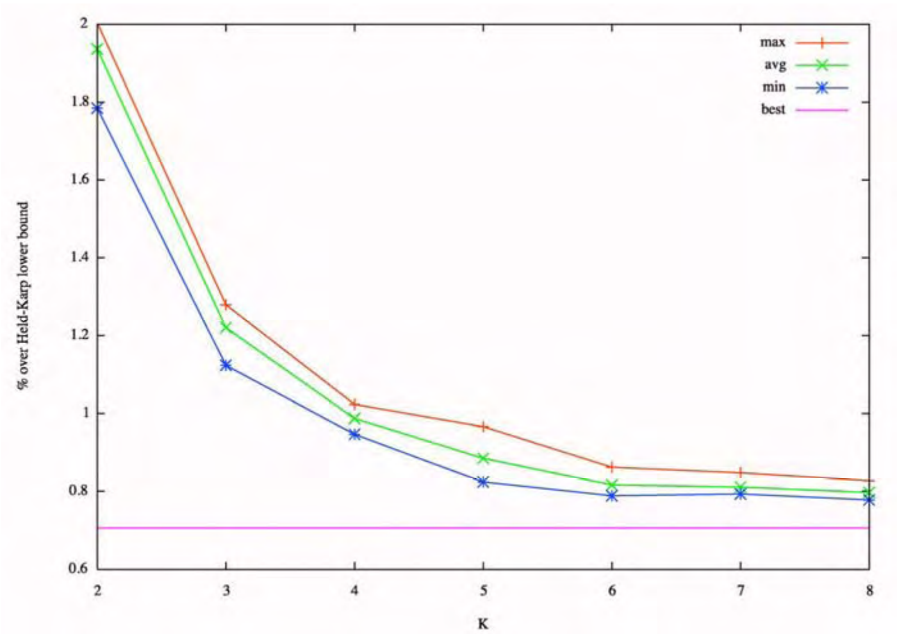

Figure 6.3. E10k.0: Percentage excess over HK bound
(1 trial, 10 runs, MOVE_TYPE = K, PATCHING_C = 0).

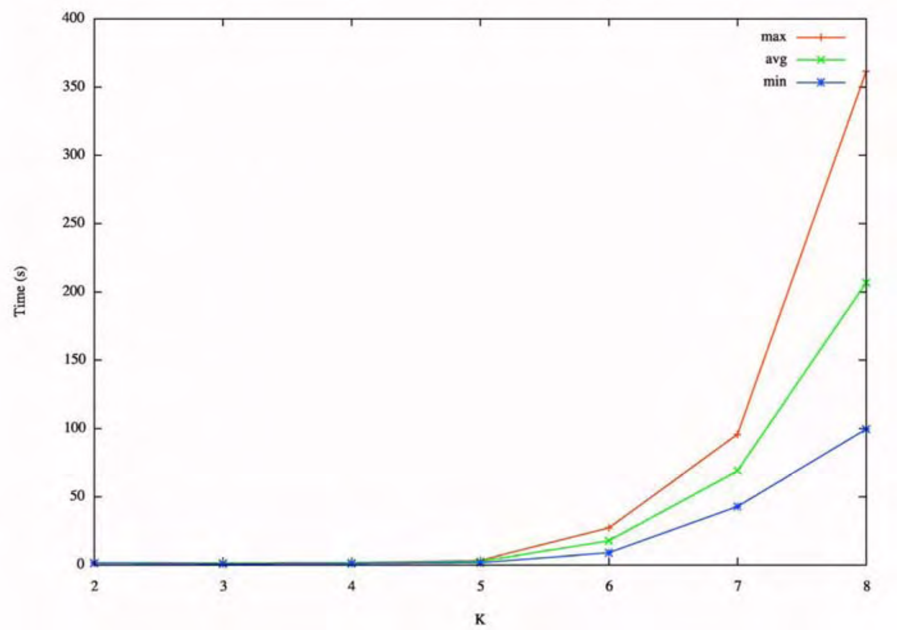

Figure 6.4. E10k.0: CPU time
(1 trial, 10 runs, MOVE_TYPE = K, PATCHING_C = 0).

As expected, CPU time grows exponentially with *K*. The average time per run grows as $0.02 \cdot 3.12^K$ (RMS error: 2.9). 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 quality-time assessment of *K* could be defined as the product of time and excess over optimum (*OPT*):

$$Time(K) \cdot \frac{Length(K) - OPT}{OPT}$$

The smaller this value is for *K* the better. The measure gives an equal weight to time and quality. Figure 6.5 depicts the measure for this experiment, where *OPT* has been replaced by the length of the current best tour. As can be seen, 4 and 5 are the two 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.

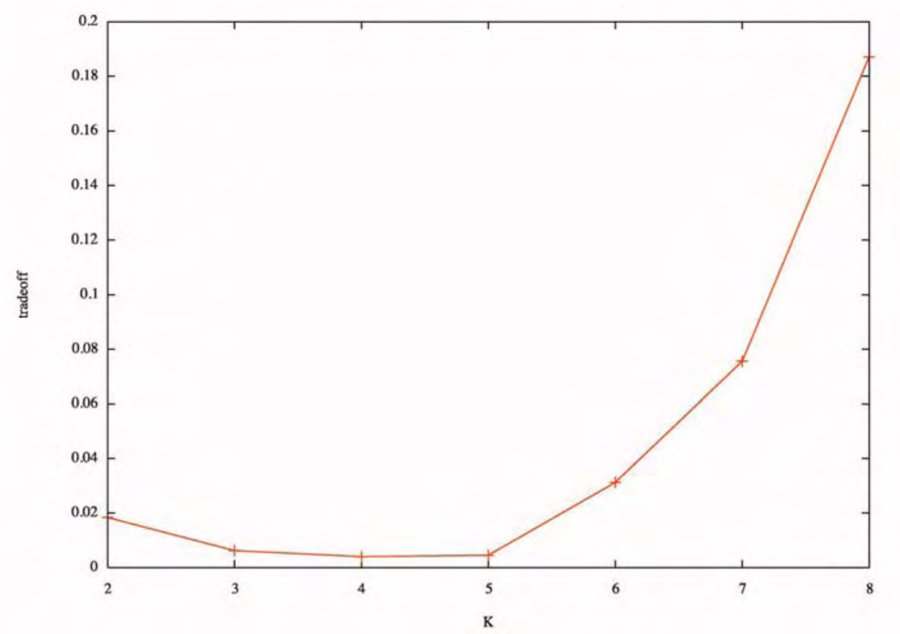

Figure 6.5. E10k.0: Quality-time tradeoff
(No patching, 1 trial, 10 runs).

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 cycle 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. Search for non-sequential moves will be made only if PATCHING_C ≥ 2 and *PATCHING_A* ≥ 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, I 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 are patched with as many alternating cycles as possible.

Tables 6.3 and 6.4 report the results from the experiments with simple patching and full patching.

	HK gap (%)			*Time (s)*		
K	*min*	*avg*	*max*	*min*	*avg*	*max*
2	1.573	1.743	1.859	1	1	2
3	1.031	1.093	1.217	1	2	2
4	0.884	0.926	0.980	2	3	3
5	0.824	0.854	0.966	4	6	7
6	0.789	0.822	0.870	12	15	18
7	0.783	0.803	0.824	33	56	112
8	0.774	0.794	0.804	142	192	273

Table 6.3. Results for E10k.0 (Simple patching, 1 trial, 10 runs).

	HK gap (%)			*Time (s)*		
K	*min*	*avg*	*max*	*min*	*avg*	*max*
2	1.573	1.743	1.859	1	2	2
3	1.039	1.098	1.185	1	2	2
4	0.863	0.914	0.950	2	3	3
5	0.795	0.866	0.923	4	6	8
6	0.792	0.826	0.843	12	19	38
7	0.768	0.792	0.822	33	62	139
8	0.757	0.770	0.781	89	198	340

Table 6.4. Results for E10k.0 (Full patching, 1 trial, 10 runs).

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.

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 6.5-7. As can be seen from these tables the same conclusions may be drawn. It pays off to use non-sequential moves. Note, however, for this instance there seems to be no reason to prefer full patching to simple patching.

	HK gap (%)			*Time (s)*		
K	*min*	*avg*	*max*	*min*	*avg*	*max*
2	1.779	1.839	1.941	88	99	137
3	1.126	1.151	1.184	22	33	43
4	0.920	0.931	0.951	25	31	34
5	0.818	0.837	0.846	48	61	75
6	0.781	0.791	0.804	276	341	395
7	0.754	0.759	0.770	1051	1409	1720
8	0.721	0.736	0.749	3529	5315	7351

Table 6.5. Results for E100k.0 (No patching, 1 trial, 10 runs).

	HK gap (%)			*Time (s)*		
K	*min*	*avg*	*max*	*min*	*avg*	*max*
2	1.627	1.667	1.716	23	29	34
3	1.010	1.034	1.092	23	29	37
4	0.851	0.873	0.899	33	40	48
5	0.785	0.799	0.813	83	91	111
6	0.735	0.751	0.778	253	293	398
7	0.725	0.731	0.742	698	992	1550
8	0.709	0.715	0.721	2795	4323	7017

Table 6.6. Results for E100k.0 (Simple patching, 1 trial, 10 runs).

	HK gap (%)			*Time (s)*		
K	*min*	*avg*	*max*	*min*	*avg*	*max*
2	1.627	1.667	1.716	23	29	34
3	0.998	1.020	1.184	20	23	31
4	0.838	0.857	0.874	29	34	40
5	0.748	0.801	0.809	78	85	101
6	0.746	0.755	0.766	293	291	322
7	0.727	0.734	0.747	793	1138	1506
8	0.706	0.714	0.719	2432	5722	9058

Table 6.7. Results for E100k.0 (Full patching, 1 trial, 10 runs).

6.1.3 Comparative results for E-instances

In order to examine the performance of the implementation as *n* grows, I 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 6.8-10. Figures 6.6-11 provide a graphical visualization of the results. As can be seen the algorithm is very robust for this problem type.

	Average HK gap (%)						
K	*E10k.0*	*E31k.0*	*E100k.0*	*E316k.0*	*E1M.0*	*E3M.0*	*E10M.0*
4	0.987	0.909	0.931	0.976	0.931	0.896	0.892
5	0.885	0.821	0.837	0.881	0.830	0.791	0.780
6	0.816	0.769	0.791	0.829	0.771	0.737	0.720
7	0.811	0.741	0.759	0.785	0.743	0.699	0.681
	Time (s)						
K	*E10k.0*	*E31k.0*	*E100k.0*	*E316k.0*	*E1M.0*	*E3M.0*	*E10M.0*
4	2	6	31	290	3116	22930	120783
5	3	11	61	343	2340	25893	113674
6	17	79	341	1640	6741	39906	143704
7	69	331	1409	4919	21969	88021	330188

Table 6.8. Results for E-instances (No patching, 1 trial, 1 run).

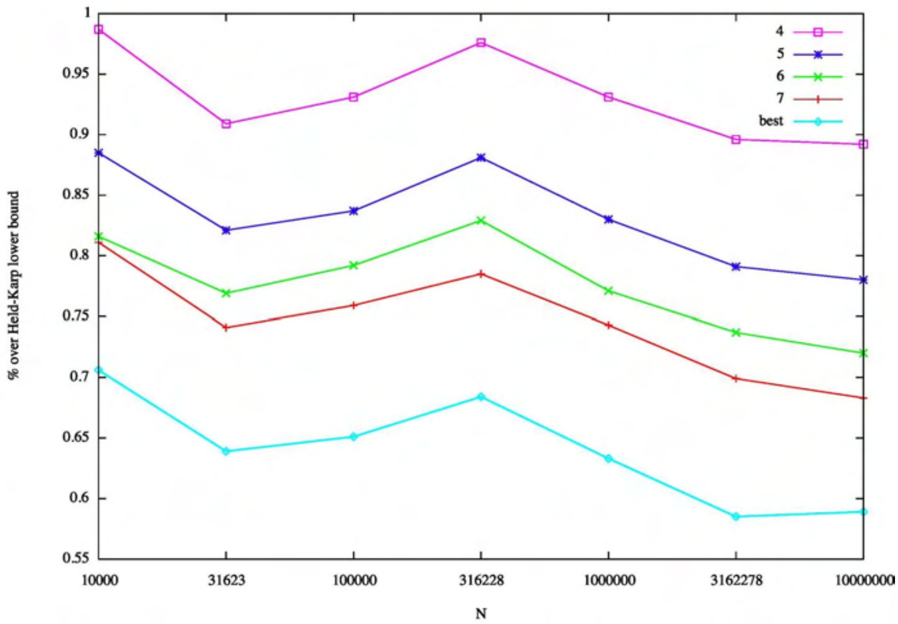

Figure 6.6. E-instances: Percentage excess over HK bound (No patching, 1 trial, 1 run).

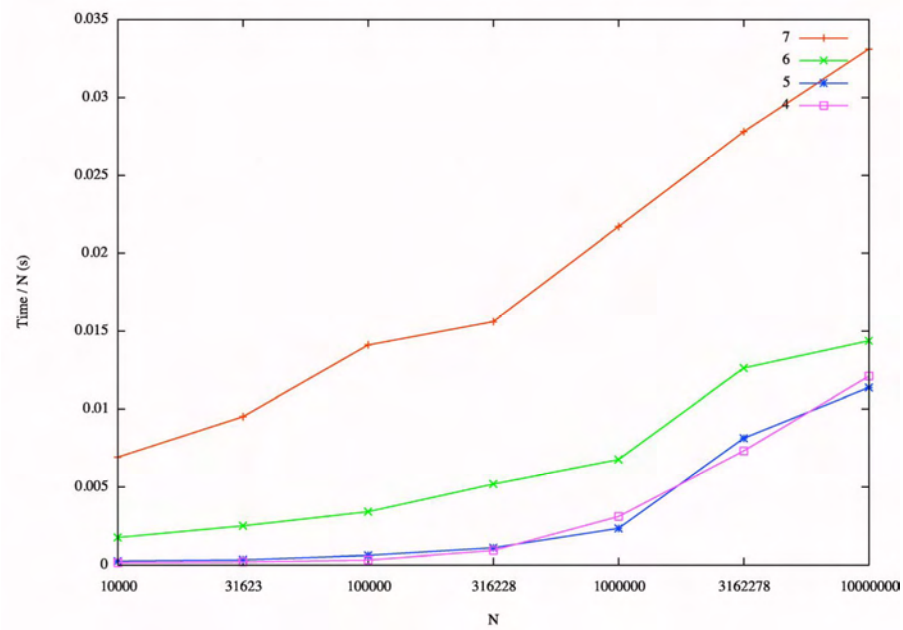

Figure 6.7. E-instances: Time per node (No patching, 1 trial, 1 run).

K	Average HK gap (%)						
K	*E10k.0*	*E31k.0*	*E100k.0*	*E316k.0*	*E1M.0*	*E3M.0*	*E10M.0*
4	0.926	0.854	0.873	0.902	0.855	0.815	0.816
5	0.854	0.785	0.799	0.834	0.787	0.736	0.736
6	0.822	0.748	0.751	0.791	0.736	0.695	0.688
7	0.803	0.724	0.731	0.766	0.712	0.668	0.660
	Time (s)						
K	*E10k.0*	*E31k.0*	*E100k.0*	*E316k.0*	*E1M.0*	*E3M.0*	*E10M.0*
4	3	9	40	206	1281	5536	18232
5	6	23	90	371	1688	11823	24050
6	15	70	291	1049	4354	20277	58623
7	56	290	987	4151	15639	69706	239741

Table 6.9. Results for E-instances (Simple patching, 1 trial, 1 run).

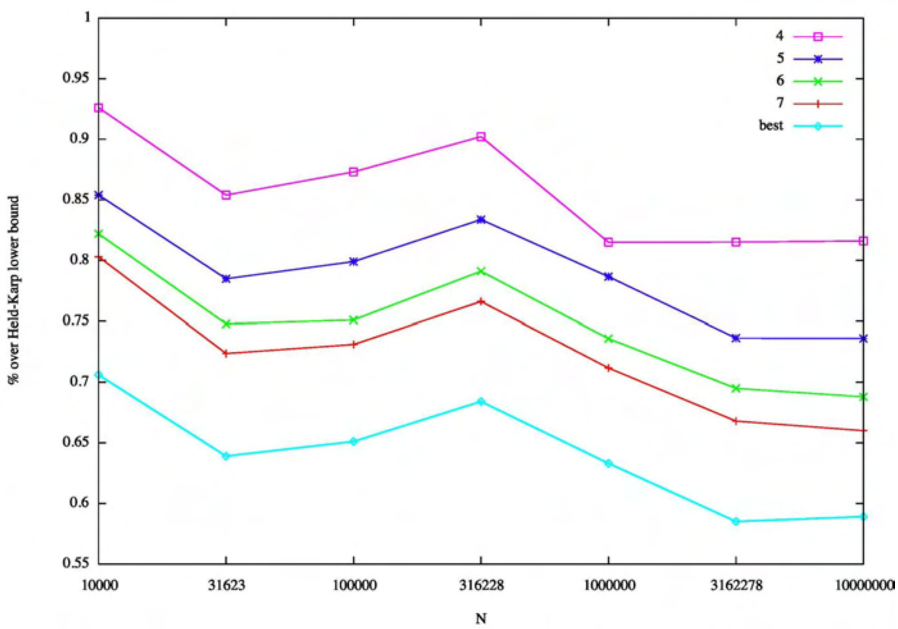

Figure 6.8. E-instances: Percentage excess over HK bound (Simple patching, 1 trial, 1 run).

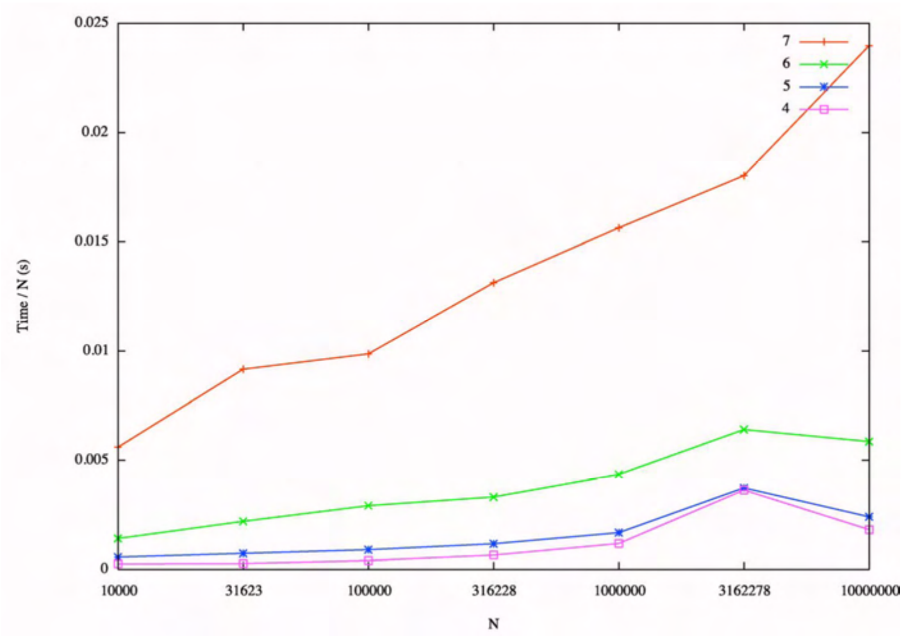

Figure 6.9. E-instances: Time per node (Simple patching, 1 trial, 1 run).

	Average HK gap (%)						
K	*E10k.0*	*E31k.0*	*E100k.0*	*E316k.0*	*E1M.0*	*E3M.0*	*E10M.0*
4	0.926	0.851	0.857	0.889	0.850	0.808	0.808
5	0.854	0.784	0.801	0.829	0.780	0.732	0.728
6	0.826	0.744	0.755	0.786	0.742	0.689	0.639
7	0.792	0.721	0.734	0.765	0.719	0.665	0.657
	Time (s)						
K	*E10k.0*	*E31k.0*	*E100k.0*	*E316k.0*	*E1M.0*	*E3M.0*	*E10M.0*
4	3	10	33	216	1062	4503	19225
5	6	25	85	390	1567	6797	23563
6	19	77	291	1115	4039	21277	74892
7	63	309	1138	4688	17360	57010	245502

Table 6.10. Results for E-instances (Full patching, 1 trial, 1 run).

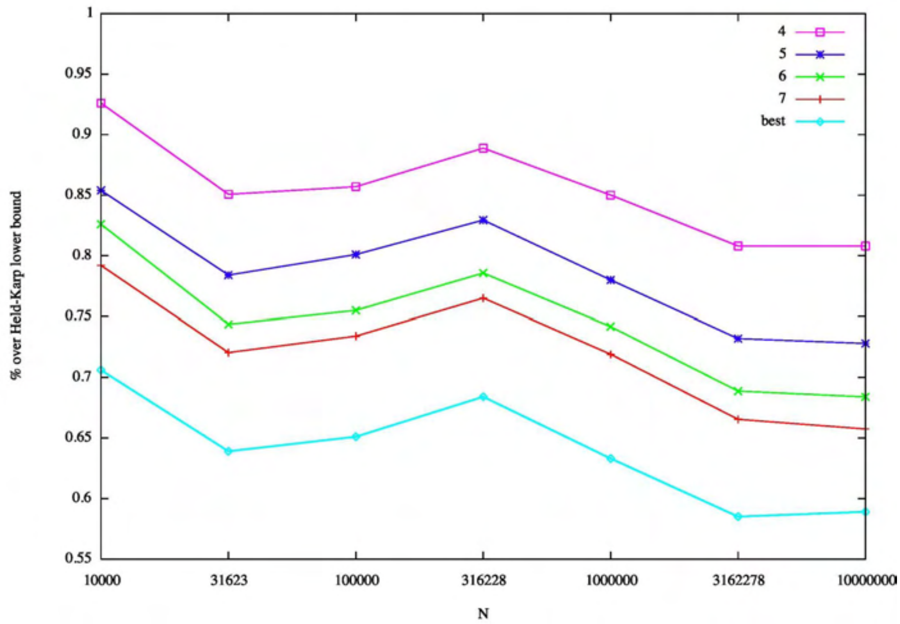

Figure 6.10. E-instances: Percentage excess over HK bound (Full patching, 1 trial, 1 run).

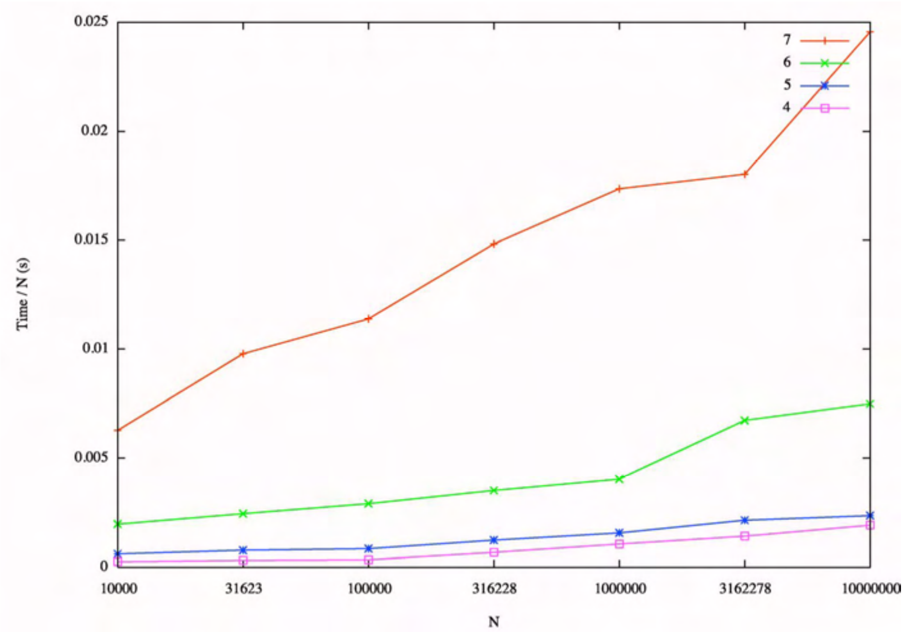

Figure 6.11. E-instances: Time per node (Full patching, 1 trial, 1 run).

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 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 Figure 2.4) is often used [2, 3, 29].

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:

> 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.

I 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 6.11-13 report the results for 1000 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 (simple patching seems to be a better choice than full patching).

	HK gap (%)			*Time (s)*		
K	*min*	*avg*	*max*	*min*	*avg*	*max*
4	0.712	0.728	0.741	512	569	640
5	0.711	0.720	0.745	444	564	750
6	0.711	0.716	0.724	1876	2578	4145

Table 6.11. Results for E10k.0 (No patching, 1000 trials, 10 runs).

K	HK gap (%)			Time (s)		
	min	*avg*	*max*	*min*	*avg*	*max*
4	0.716	0.728	0.742	387	464	618
5	0.710	0.724	0.747	464	559	689
6	0.706	0.714	0.729	634	798	924

Table 6.12. Results for E10k.0 (Simple patching, 1000 trials, 10 runs).

K	HK gap (%)			Time (s)		
	min	*avg*	*max*	*min*	*avg*	*max*
4	0.712	0.724	0.740	427	511	562
5	0.709	0.721	0.731	473	567	689
6	0.707	0.720	0.742	805	1054	1468

Table 6.13. Results for E10k.0 (Full patching, 1000 trials, 10 runs).

Figures 6.12-14 illustrate the convergence of the tour length for each of the three patching cases.

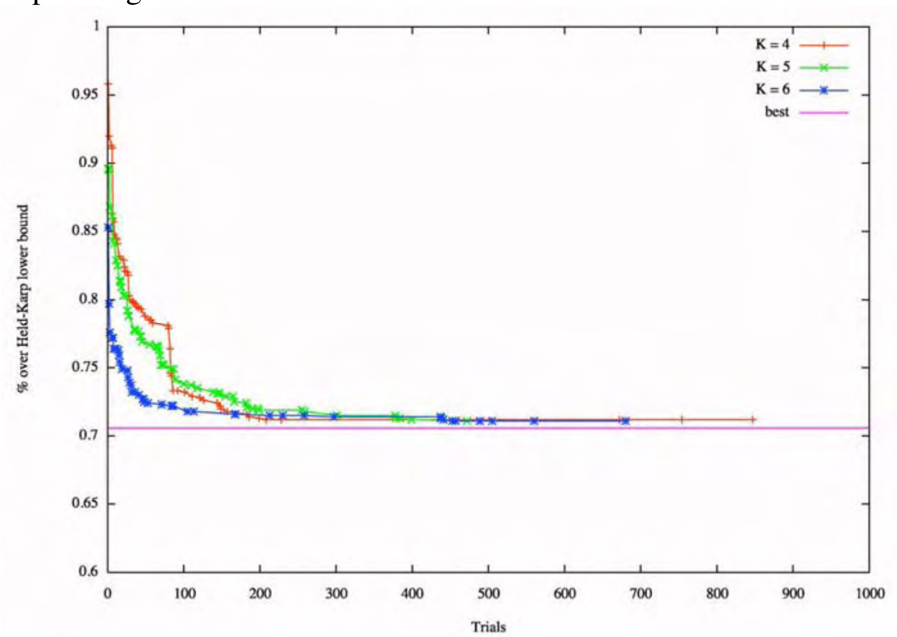

Figure 6.12. Convergence for E10k.0
(No patching, 1000 trials, best of 10 runs).

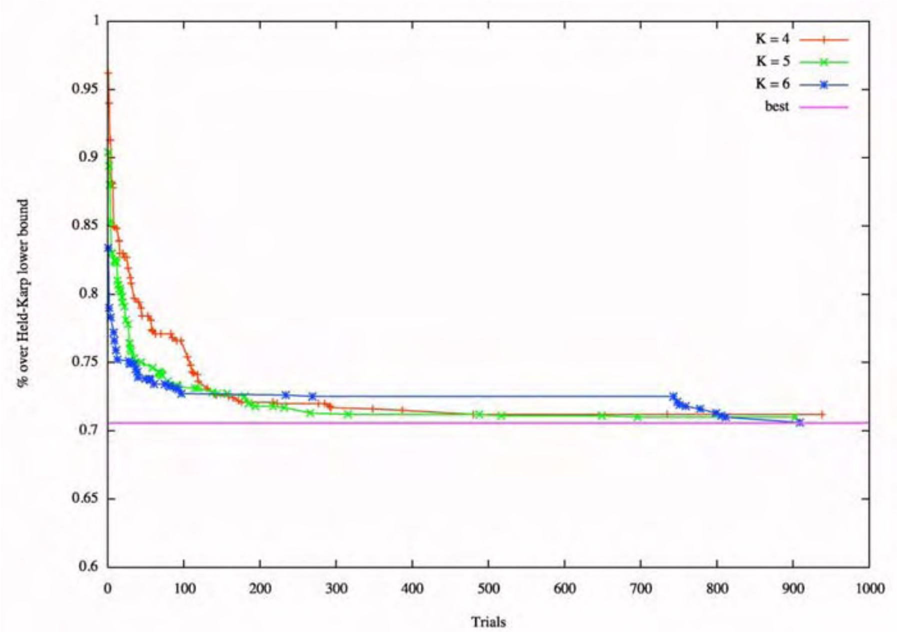

Figure 6.13. Convergence for E10k.0
(Simple patching, 1000 trials, best of 10 runs).

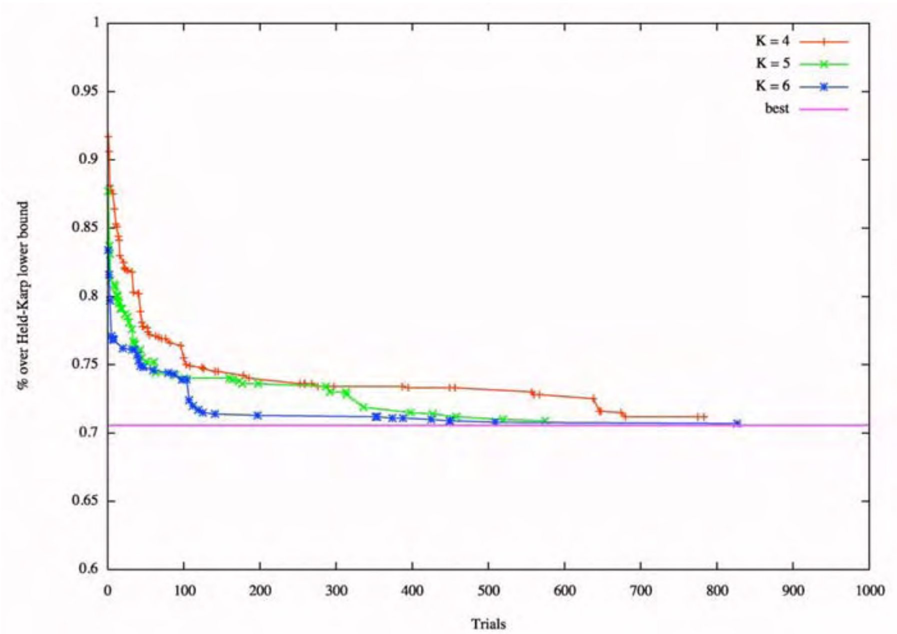

Figure 6.14. Convergence for E10k.0
(Full patching, 1000 trials, best of 10 runs).

Tables 6.15 and Figures 6.15-17 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*.

	HK gap (%)			*Time (s)*		
K	*No*	*Simple*	*Full*	*No*	*Simple*	*Full*
4	0.798	0.709	0.694	27023	9304	10675
5	0.707	0.674	0.672	32740	12043	13873
6	0.703	0.663	0.665	189911	22202	25266

Table 6.14. Results for E100k.0 (1000 trials, 1 run).

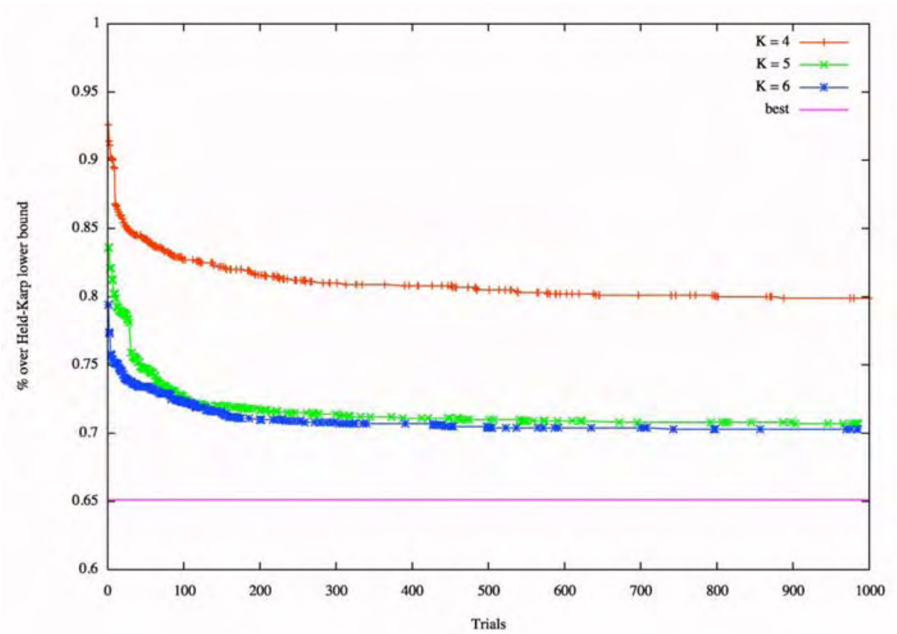

Figure 6.15. Convergence for E100k.0
(No patching, 1000 trials, 1 run).

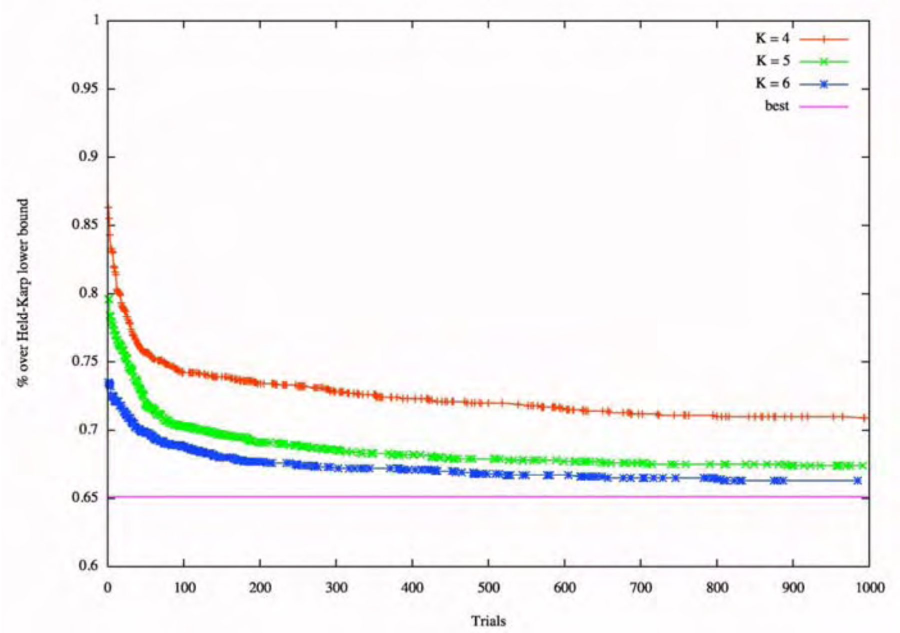

Figure 6.16. Convergence for E100k.0
(Simple patching, 1000 trials, 1 run).

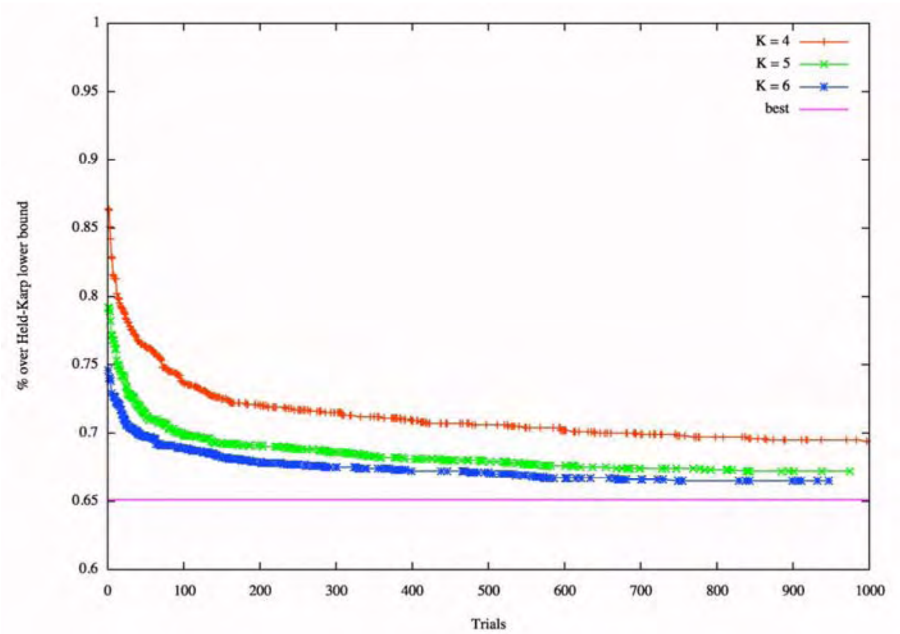

Figure 6.17. Convergence for E100k.0
(Full patching, 1000 trials, 1 run).

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 [32].

To examine LKH's performance for clustered problems I performed experiments on the eight largest *C*-instances of the 8th DIMACS TSP Challenge. Table 6.15 covers the lengths of the current best tours for these instances. These tours have all been found by LKH. My experiments with *C*-instances are very similar to those performed with the *E*-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	59,545,428	59,169,193	0.636
C31k.1	31,623	59,293,266	58,840,096	0.770
C100k.0	100,000	104,646,656	103,916,254	0.703
C100k.1	100,000	105,452,240	104,663,040	0.754
C316k.0	316,228	186,936,768	185,576,667	0.733

Table 6.15. Tour quality for C-instances.

6.2.1 Results for C10k.0

Figure 6.18 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.

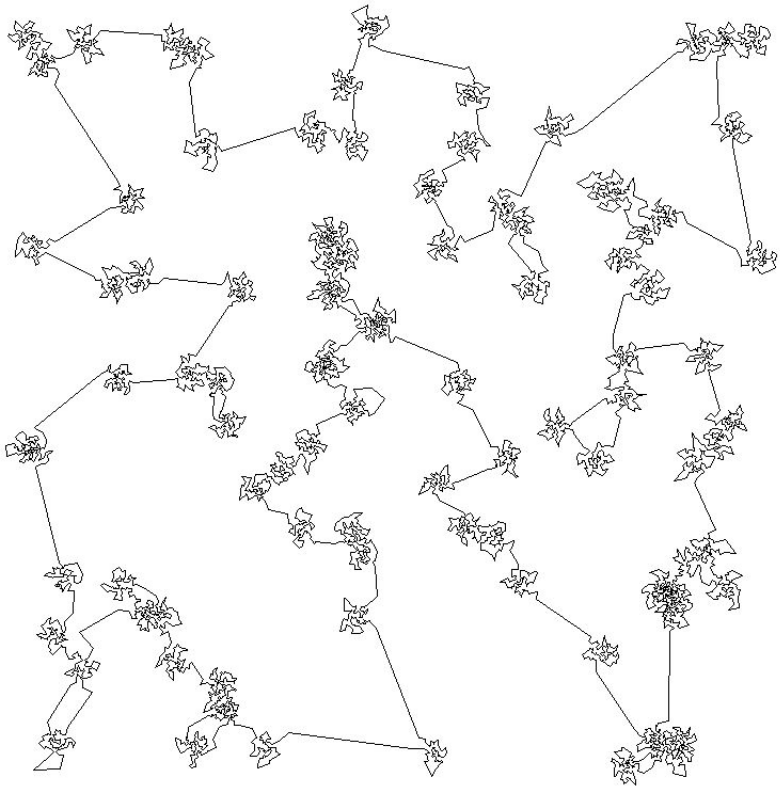

Figure 6.18. Best tour for C10k.0.

As in the experiments with *E*-instances a candidate set based on the α-measure could be used. Figure 6.19 depicts the 5 α-nearest candidate set for *C10k.0*. Since the 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.

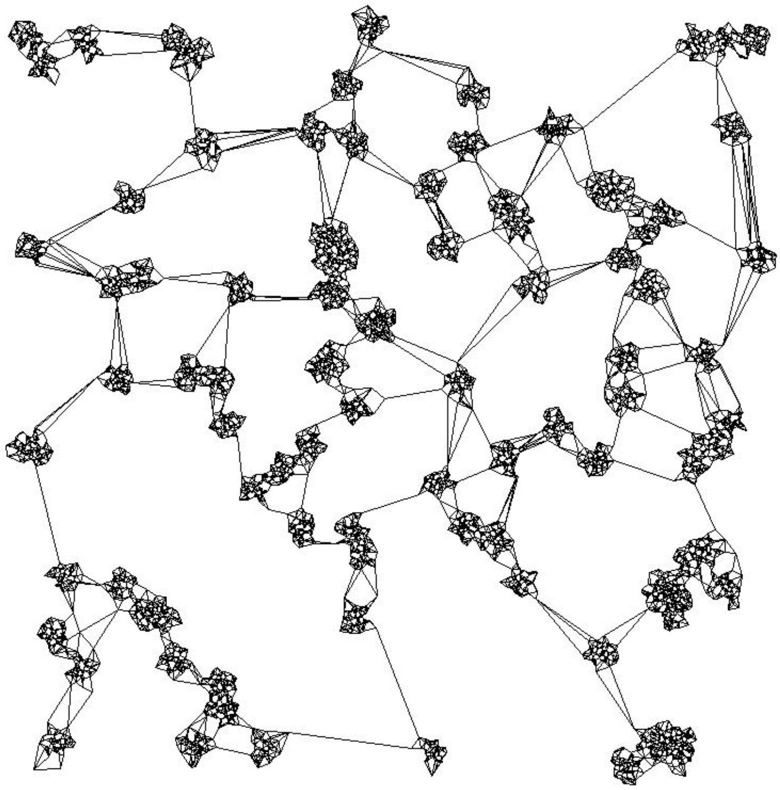

Figure 6.19. 5 α-nearest candidate set for C10k.0.

For geometric instances, Johnson [20] 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 5 closest cities in each of its four surrounding quadrants.

For clustered instances I 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 6.20 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.

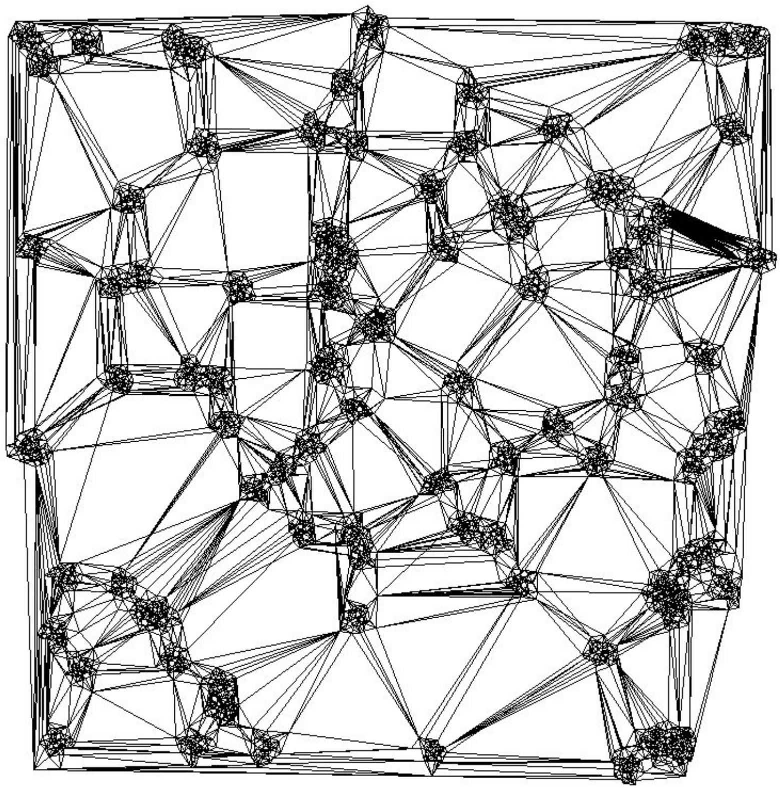

Figure 6.20. 4 α-nearest + 4 quadrant nearest candidate set for C10k.0.

However, even if this sparse candidate set is used, my experiments with cycle patching on clustered instances have revealed that the search space is large. To prune the search space I 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 Sections 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 6.16-18. 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.

	HK gap (%)			*Time (s)*		
K	*min*	*avg*	*max*	*min*	*avg*	*max*
3	2.129	2.771	3.316	1	2	2
4	1.003	2.020	3.356	2	2	3
5	1.131	1.472	2.332	7	9	11
6	0.963	1.289	1.767	122	167	256

Table 6.16. Results for C10k.0 (No patching, 1 trial, 10 runs).

	HK gap (%)			*Time (s)*		
K	*min*	*avg*	*max*	*min*	*avg*	*max*
3	1.837	2.805	4.248	2	2	2
4	1.461	1.952	2.935	4	5	6
5	1.022	1.255	1.650	18	23	33
6	0.995	1.259	1.918	111	144	194

Table 6.17. Results for C10k.0 (Simple patching, 1 trial, 10 runs).

	HK gap (%)			*Time (s)*		
K	*min*	*avg*	*max*	*min*	*avg*	*max*
3	1.612	2.038	2.472	1	2	2
4	1.524	1.973	2.654	4	5	6
5	1.019	1.402	1.962	19	25	30
6	0.896	1.218	1.782	93	143	236

Table 6.18. Results for C10k.0 (Full patching, 1 trial, 10 runs).

6.2.2 Results for C100k.0

Looking at the results for 1-trial solution of *C10k.0* (Tables 6.16-18), there seems to be only little advantage in using cycle patching. To see whether this also applies to larger instances I made the same experiments with the 100,000-city instance *C100k.0*. The results of these experiments are covered in Tables 6.19-21. As can be seen, it is also questionable whether patching is useful for this instance. In addition, the runtime penalty for increasing *K* from 5 to 6 is even more conspicuous for this instance.

	HK gap (%)			*Time (s)*		
K	*min*	*avg*	*max*	*min*	*avg*	*max*
3	2.981	3.320	4.076	33	38	48
4	2.163	2.372	2.572	34	39	46
5	1.643	1.786	2.009	94	112	140
6	1.382	1.563	1.699	1519	1697	2097

Table 6.19. Results for C100k.0 (No patching, 1 trial, 10 runs).

	HK gap (%)			*Time (s)*		
K	*min*	*avg*	*max*	*min*	*avg*	*max*
3	2.814	3.203	3.905	29	33	40
4	2.057	2.282	2.681	65	74	99
5	1.495	1.740	2.026	306	362	440
6	1.229	1.381	1.381	1578	1949	2348

Table 6.20. Results for C100k.0 (Simple patching, 1 trial, 10 runs).

	HK gap (%)			*Time (s)*		
K	*min*	*avg*	*max*	*min*	*avg*	*max*
3	2.868	3.244	3.656	31	52	79
4	1.823	2.175	2.570	73	82	101
5	1.523	1.783	2.174	309	392	491
6	1.280	1.437	1.588	1721	2272	3392

Table 6.21. Results for C100k.0 (Full patching, 1 trial, 10 runs).

6.2.3 Comparative results for C-instances

In order to evaluate the robustness of the implementation we also performed experiments with the instances *C31k.0* and *C316k.0*. Tables 6.22-24 contain comparative results for the four chosen *C*-instances.

	Average HK gap (%)				*Time (s)*			
K	*C10k.0*	*C31k.0*	*C100k.0*	*C316k.0*	*C10k.0*	*C31k.0*	*C100k.0*	*C316k.0*
3	2.771	2.327	3.320	4.419	2	16	38	416
4	1.453	1.987	2.287	3.001	2	8	51	329
5	1.472	1.404	1.786	2.068	17	30	112	673

Table 6.22. Results for C-instances (No patching, 1 trial, 1 run).

	Average HK gap (%)				*Time (s)*			
K	*C10k.0*	*C31k.0*	*C100k.0*	*C316k.0*	*C10k.0*	*C31k.0*	*C100k.0*	*C316k.0*
3	2.805	2.860	3.203	2.747	2	6	33	444
4	1.952	1.998	2.282	1.779	5	19	74	669
5	1.255	1.522	1.740	1.870	23	89	362	2038

Table 6.23. Results for C-instances (Simple patching, 1 trial, 1 run).

	Average HK gap (%)				*Time (s)*			
K	*C10k.0*	*C31k.0*	*C100k.0*	*C316k.0*	*C10k.0*	*C31k.0*	*C100k.0*	*C316k.0*
3	2.038	1.772	3.244	3.718	2	21	52	507
4	1.973	1.725	2.175	2.655	5	21	82	453
5	1.402	1.439	1.783	1.618	25	90	392	1881

Table 6.24. Results for C-instances (Full patching, 1 trial, 1 run).

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 1000-trial experiments with *C10k.0* and *C100k.0*, high-quality solutions may be achieved using few trials. It is interesting that for this instance type it does not seem to pay off to set *K* to other values than 4, and that only little is gained by using non-sequential moves.

	HK gap (%)			*Time (s)*		
K	*No*	*Simple*	*Full*	*No*	*Simple*	*Full*
3	0.669	0.669	0.669	617	741	758
4	0.668	0.688	0.668	768	1180	1395
5	0.668	0.668	0.668	2049	3227	3797

Table 6.25. Results for C10k.0 (1000 trials, best of 10 runs).

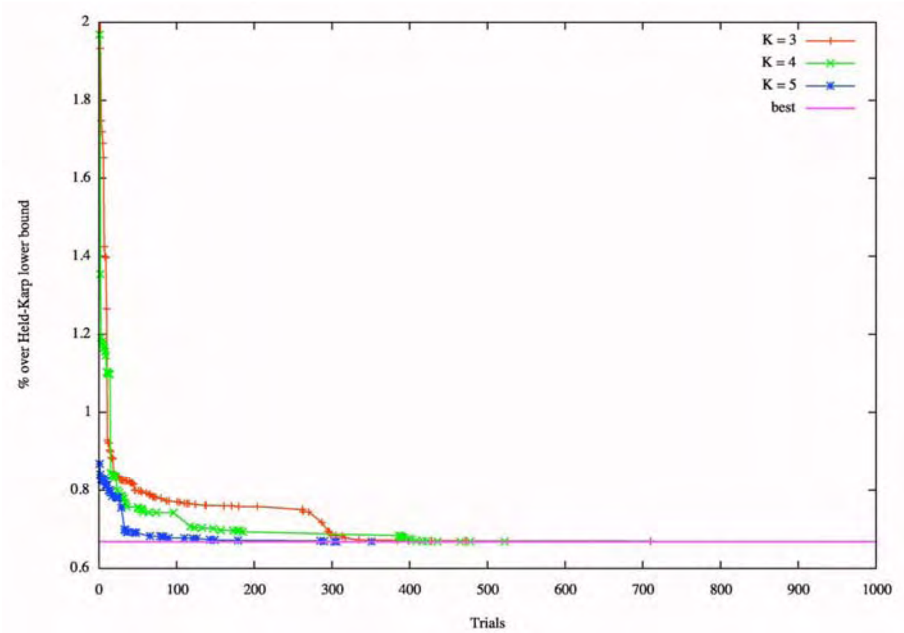

Figure 6.21. Convergence for C10k.0
(No patching, 1000 trials, best of 10 runs).

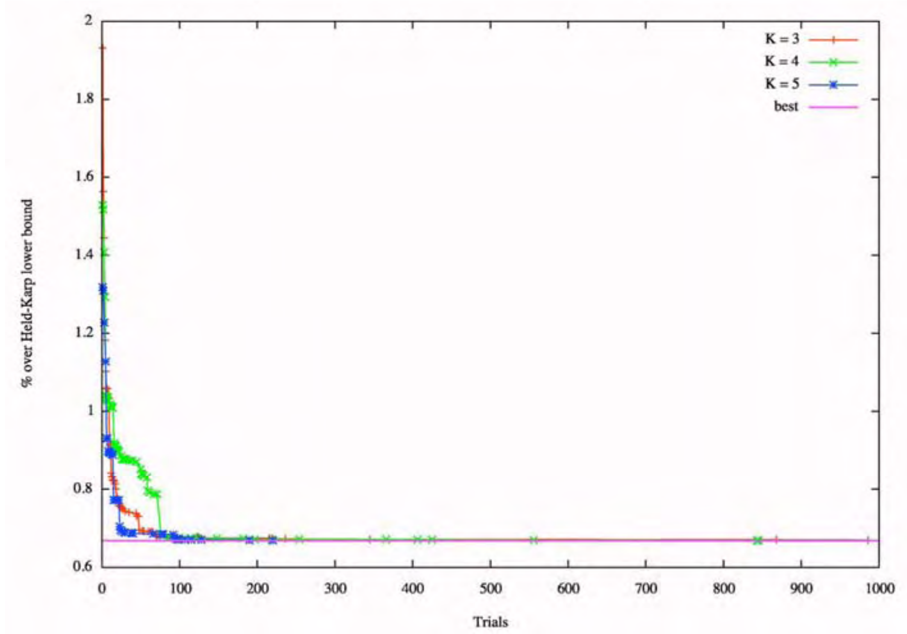

Figure 6.22. Convergence for C10k.0
(Simple patching, 1000 trials, best of 10 runs).

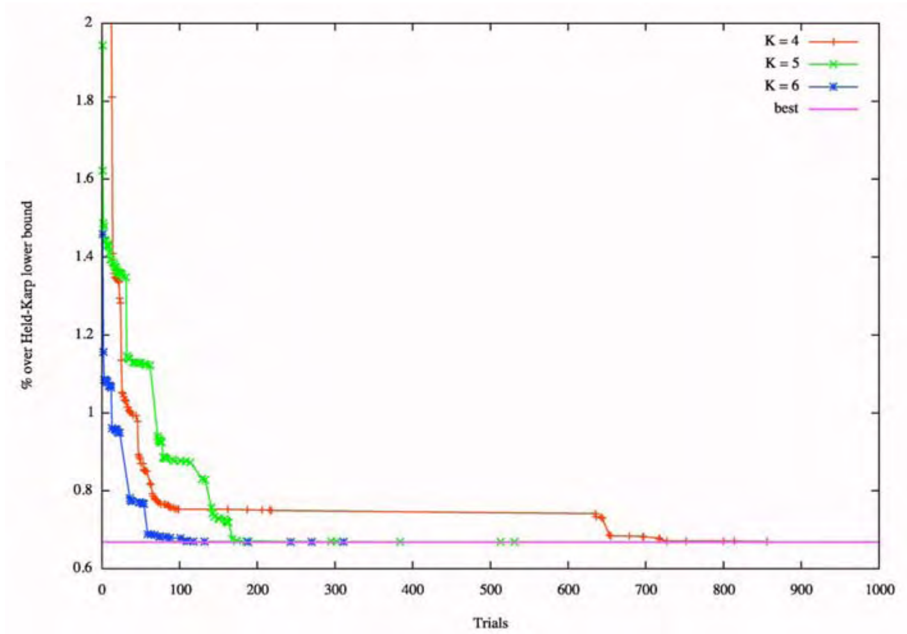

Figure 6.23. Convergence for C10k.0
(Full patching, 1000 trials, best of 10 runs).

	HK gap (%)			*Time (s)*		
K	*No*	*Simple*	*Full*	*No*	*Simple*	*Full*
3	0.895	1.011	1.045	23592	9553	13901
4	0.829	0.794	0.792	11700	12039	21711
5	0.823	0.829	0.806	14632	28639	34010

Table 6.26. Results for C100k.0 (1000 trials, 1 run)

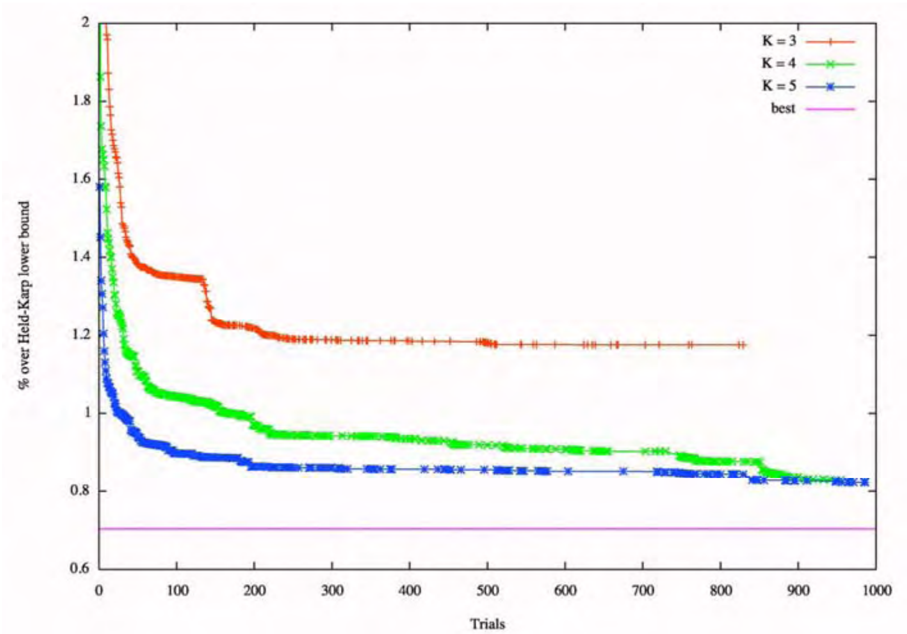

Figure 6.24. Convergence for C100k.0
(No patching, 1000 trials, 1 run).

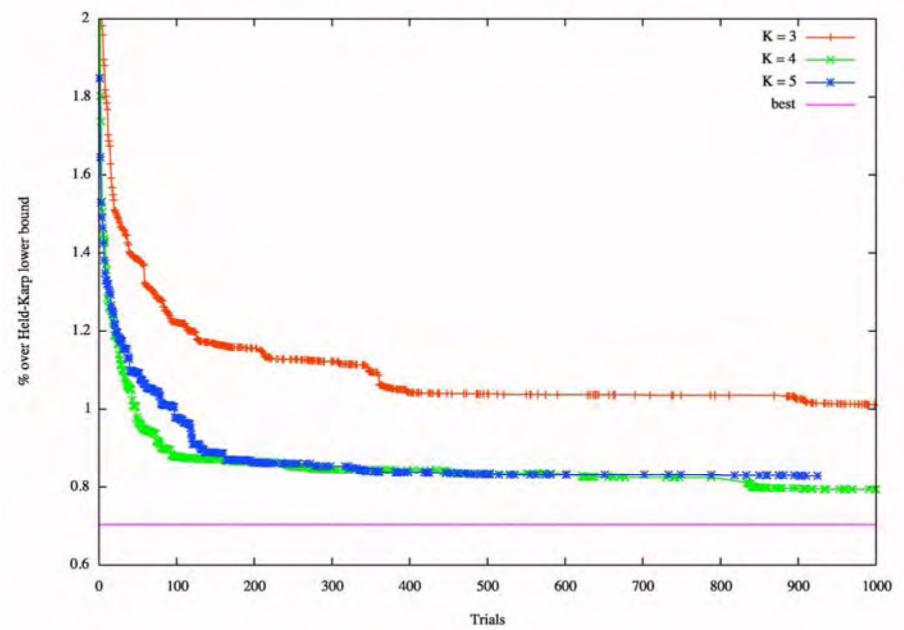

Figure 6.25. Convergence for C100k.0
(Simple patching, 1000 trials, 1 run).

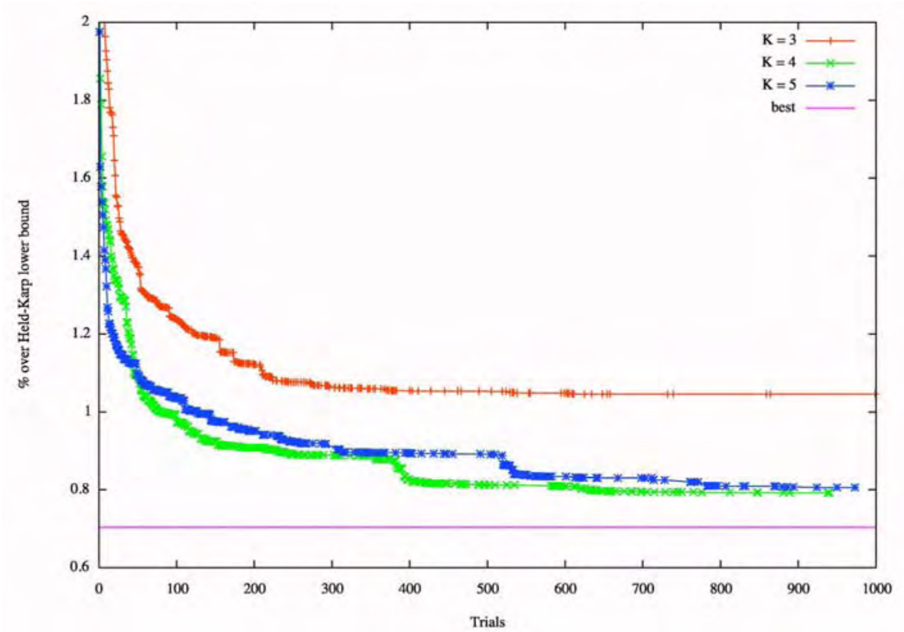

Figure 6.26. Convergence for C100k.0
(Full patching, 1000 trials, 1 run).

6.3 Performance for M-instances

The testbed of the 8th DIMACS TSP Challenge also contains seven instances specified by distance matrices where each entry is chosen independently and uniformly from the integers in (0,1000000]. Sizes range from 1,000 to 10,000. All have been solved to optimality.

This type of instances appears to be easy to LKH-2. I have chosen only to report the results of my experiments with the largest of these instances, the 10,000-city instance *M10k.0*.

6.3.1 Results for M10k.0

Tables 6.27-29 report the results of the 1-trial runs. As optimum is known for this instance, I will measure tour quality as the percentage excess over optimum. The tables show that high-quality solutions can be obtained using only one trial, and with running times that are small, even for $K = 8$.

	OPT gap (%)			*Time (s)*		
K	*min*	*avg*	*max*	*min*	*avg*	*max*
3	0.167	0.200	0.238	1	1	1
4	0.038	0.053	0.077	1	1	1
5	0.013	0.022	0.031	1	1	1
6	0.006	0.012	0.016	2	3	3
7	0.003	0.007	0.010	4	4	5
8	0.001	0.005	0.009	6	8	10

Table 6.27. Results for M10k.0 (No patching, 1 trial, 10 runs).

K	*OPT gap (%)*			*Time (s)*		
	min	*avg*	*max*	*min*	*avg*	*max*
3	0.136	0.183	0.207	2	3	4
4	0.036	0.049	0.057	3	4	6
5	0.012	0.022	0.034	7	8	9
6	0.004	0.012	0,019	11	13	16
7	0.003	0.006	0.009	16	19	24
8	0.001	0.004	0.008	22	26	29

Table 6.28. Results for M10k.0 (Simple patching, 1 trial, 10 runs).

K	*OPT gap (%)*			*Time (s)*		
	min	*avg*	*max*	*min*	*avg*	*max*
3	0.136	0.167	0.192	2	3	4
4	0.031	0.054	0.067	3	4	6
5	0.016	0.024	0.040	6	7	10
6	0.004	0.011	0.016	10	12	16
7	0.002	0.006	0.012	14	18	27
8	0.001	0.006	0.008	19	24	27

Table 6.29. Results for M10k.0 (Full patching, 1 trial, 10 runs).

Tables 6.27-29 suggest that there is no advantage in using cycle patching. This is even more evident from Table 6.30, which reports the results from 1000-trial runs. Adding cycle patching does not result in better tours. It only increases running time considerably.

K	*OPT gap (%)*			*Time (s)*		
	No	*Simple*	*Full*	*No*	*Simple*	*Full*
3	0.051	0.047	0.091	321	2119	2149
4	0.013	0.012	0.010	297	2533	2601
5	0.004	0.004	0.005	340	3633	3490
6	0.001	0.002	0.002	641	6367	5923
7	0.001	0.001	0.001	900	10583	10284
8	0.000	0.000	0.000	1372	15301	16152

Table 6.30. Results for M10k.0 (1000 trials, 1 run).

6.4 Performance for R-instances

The preceding sections have presented computational results from experiments with synthetic generated test instances. In order to study the performance of LKH-2 for real world instances I made experiments with three large-scale instances from the TSP web page of Applegate, Bixby, Chvátal and Cook [39]. These instances are listed in Table 6.31. The column labeled *CBT* contains the length of the current best tour for each of the instances.

Instance	*n*	*CBT*
ch71009	71,009	4,566,506
lrb744710	744,710	1,611,420
World	1,904,711	7,515,964,553

Table 6.31. R-instances.

Optima for these instances are unknown. The current best tours have been found by LKH. Since the Held-Karp lower bounds are not available, I will measure tour quality as percentage excess over these tours (*CBT gap*).

6.4.1 Results for ch71009

The TSP web page of Applegate et al. contains a set of 27 test instances derived from maps of real countries, ranging in size from 28 cities in Western Sahara to 71,009 cities in China. I have selected the largest one of these instances, *ch71009*, for my experiments. Figure 6.27 depicts the current best tour for this instance (found by LKH).

Figure 6.27. Current best tour for ch71009.

The Concorde package [1] was used to compute a lower bound for *ch71009* of 4,565,452. This bound shows that LKH's best tour is at most 0.024% greater than the length of an optimal tour.

Table 6.32 lists the computational results for one trial. The results indicate that "Simple patching" or "Full patching" is to be preferred to "No patching". In this respect *ch71009* resembles *E*-instances. The advantage of using patching is further emphasized by the 1000-trial results shown in Table 6.33. Note that the 1000-trial run with $K = 6$ and full patching results in a tour that deviates only 0.004% from the current best tour.

K	CBT gap (%)			Time (s)		
	No	*Simple*	*Full*	*No*	*Simple*	*Full*
4	0.188	0.213	0.229	16	27	30
5	0.118	0.126	0.127	25	93	95
6	0.086	0.091	0.092	204	471	380
7	0.059	0.042	0.053	847	1394	1443
8	0.057	0.040	0.041	5808	7626	7398

Table 6.32. Results for ch71009 (1 trial, 1 run).

K	CBT gap (%)			Time (s)		
	No	*Simple*	*Full*	*No*	*Simple*	*Full*
4	0.084	0.066	0.055	9545	10246	10390
5	0.032	0.014	0.019	10682	15661	17896
6	0.010	0.008	0.004	71468	38664	36686

Table 6.33. Results for ch71009 (1000 trials, 1 run).

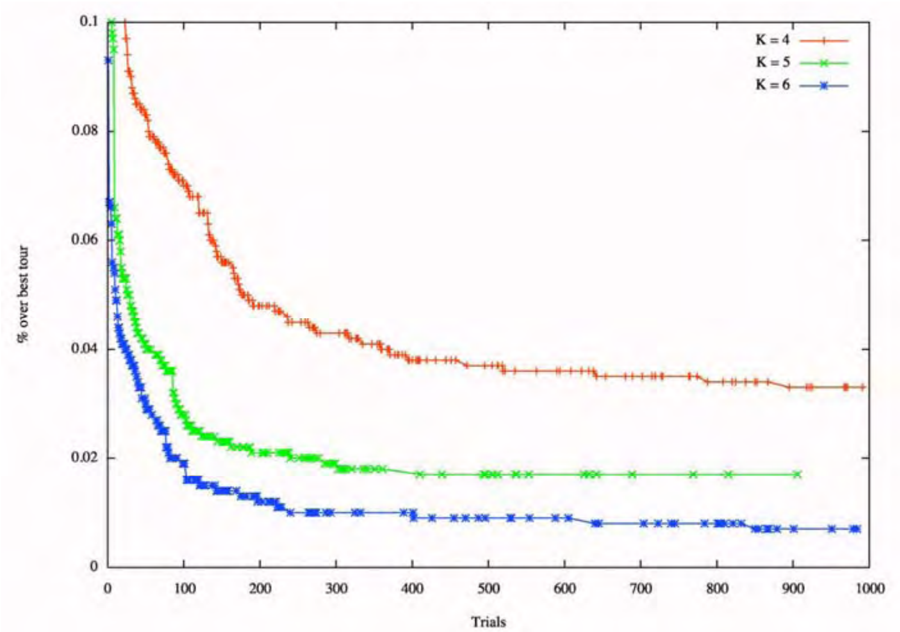

Figure 6.28. Convergence ch71009
(Full patching, 1000 trials, 1 run).

6.4.2 Results for lrb744710

The TSP web page of Applegate et al. contains a set of 102 instance based on VLSI data. The instances range in size from 131 cities up to 744,710 cities. I have selected the largest one of these instances, *lrb744710*, for my experiments. Figure 6.29 depicts the current best tour for this instance (found by LKH).

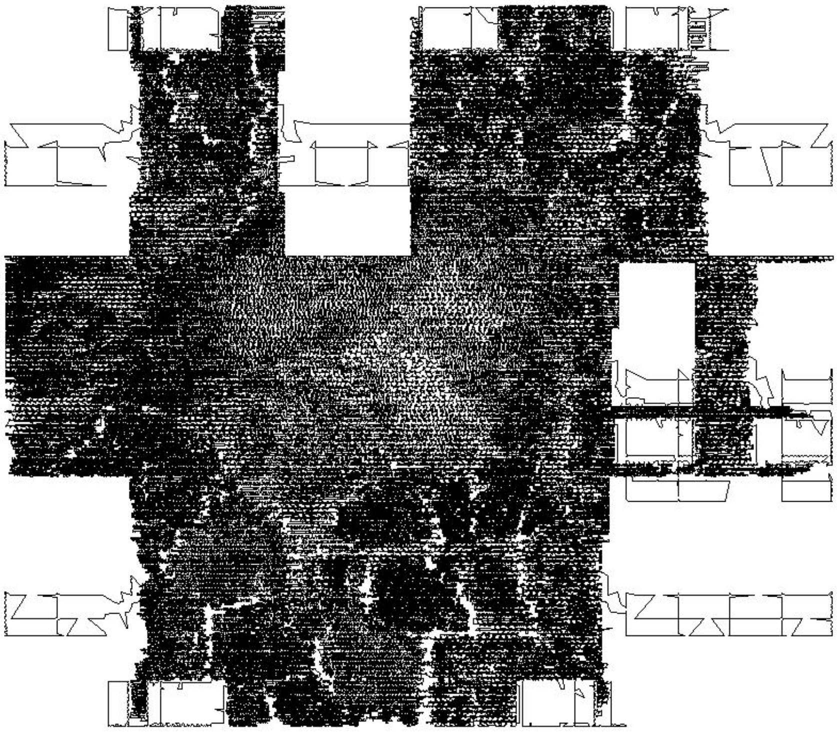

Figure 6.29. Current best tour for lrb744710.

To give a clearer picture of what a VLSI-tour look like I have depicted the optimal solution of the instance *xqc2175* in Figure 6.30.

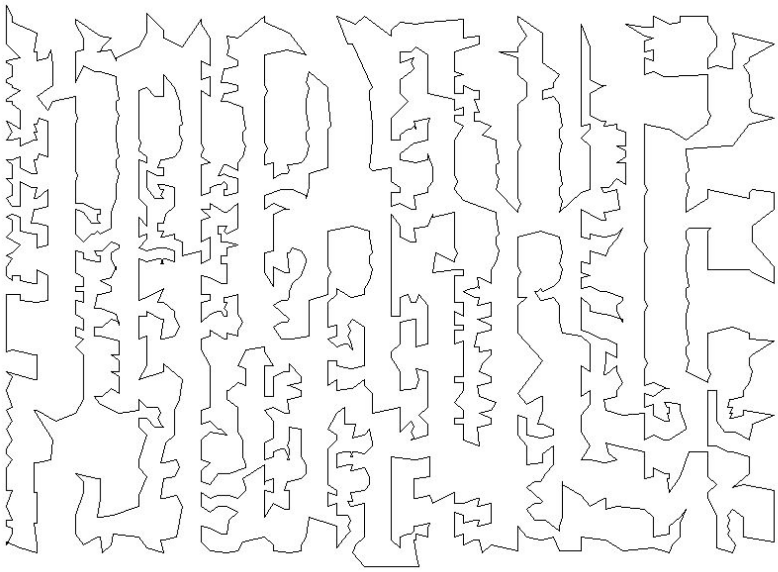

Figure 6.30. Optimal tour for xqc2175.

Table 6.34 lists the computational results for one trial. Again we see the same picture. “Simple patching” or “Full patching” is to be preferred to “No patching”. The running times show that this is a tough instance for LKH.

	CBT gap (%)			*Time (s)*		
K	*No*	*Simple*	*Full*	*No*	*Simple*	*Full*
3	1.414	1.263	1.027	758	381	423
4	0.658	0.418	0.437	530	401	405
5	0.357	0.266	0.253	639	957	926
6	0.308	0.207	0.189	5711	5137	3931
7	0.203	0.135	0.142	33464	24331	17230

Table 6.34. Results for lrb744710 (1 trial, 1 run).

The 1-trial results indicate that using “Full patching” is a little more advantageous than using “Simple patching”. Thus, to limit the number of experiments I used “Full patching” in my 1000-trial runs. The results are given in Table 6.35. Figure 6.31 depicts the change of the tour cost over the 1000 trials.

K	*CBT gap (%)*	*Time (s)*
3	0.392	89016
4	0.132	88256
5	0.081	228559

Table 6.35. Results for lrb744710 (Full patching, 1000 trials, 1 run).

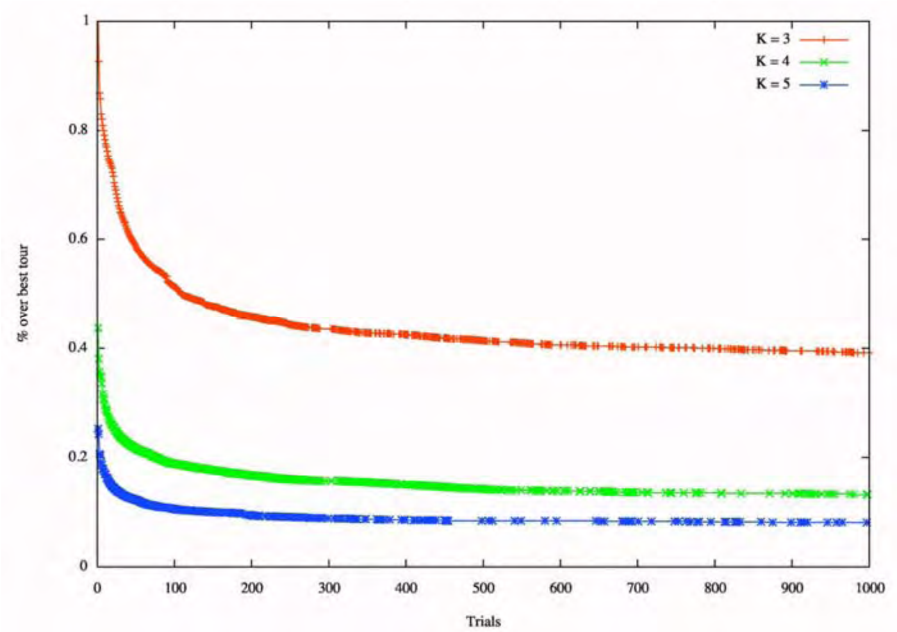

Figure 6.31. Convergence for lrb744710
(Full patching, 1000 trials, 1 run).

The current best tour for this instance was found with additional runs that exploit the subproblem partitioning and tour merging facilities of LKH-2.

6.4.3 Results for World

The largest test instance on the TSP web page of Applegate et al. is a 1,904,711-city instance of locations throughout the world, named *World*. Figure 6.32 depicts the current best tour for this instance (found by LKH).

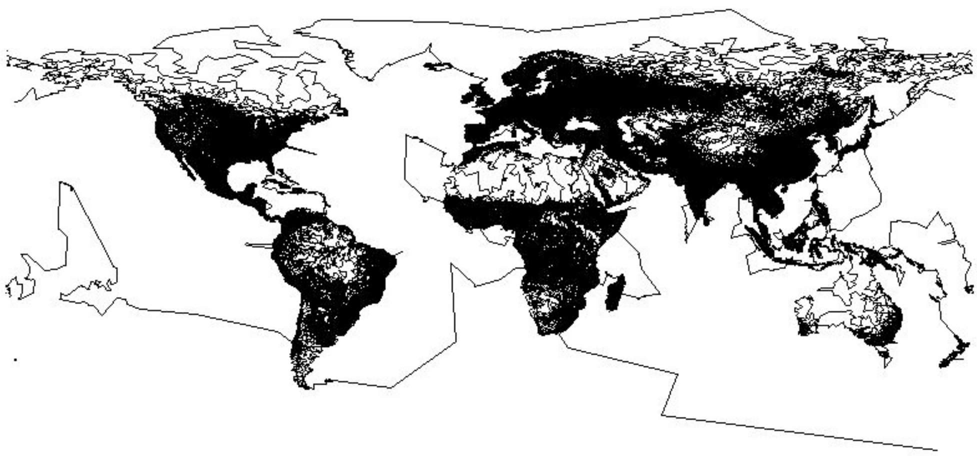

Figure 6.32. Projection of current best tour for World.

The Concorde package [1] was used to compute a lower bound of 7,512,276,947 for this instance (August 2007). This bound shows that LKH-2's best tour has length at most 0.049% greater than the length of an optimal tour.

As can be seen from Figure 6.32, this instance is clustered. So in my experiments I used the same candidate neighborhood as was used for the *C*-instances, i.e., the 4 α-nearest neighbors supplemented with the 4 quadrant-nearest neighbors. In addition, the so-called Quick-Boruvka algorithm [3] was used to construct the initial tour. I found that this tour construction algorithm improved the tour quality considerably in my 1-trial runs. Table 6.36 lists the computational results from these runs.

	CBT gap (%)			*Time (s)*		
K	*No*	*Simple*	*Full*	*No*	*Simple*	*Full*
3	1.072	0.915	1.011	4911	867	974
4	0.658	0.567	0.505	2881	1508	1459
5	0.490	0.301	0.316	3802	3736	7485
6	0.275	0.235	0.228	20906	15788	61583

Table 6.36. Results for World (1 trial, 1 run).

Table 6.37 reports the results from my experiments with multi-trial runs. I have limited the experiments to only cover runs with 100 trials and simple patching. Figure 6.33 depicts the change of the tour cost over the 100 trials.

K	*CBT gap (%)*	*Time (s)*
3	0.453	55962
4	0.189	41570
5	0.114	68581
6	0.077	206226

Table 6.37. Results for World (Simple patching, 100 trials, 1 run).

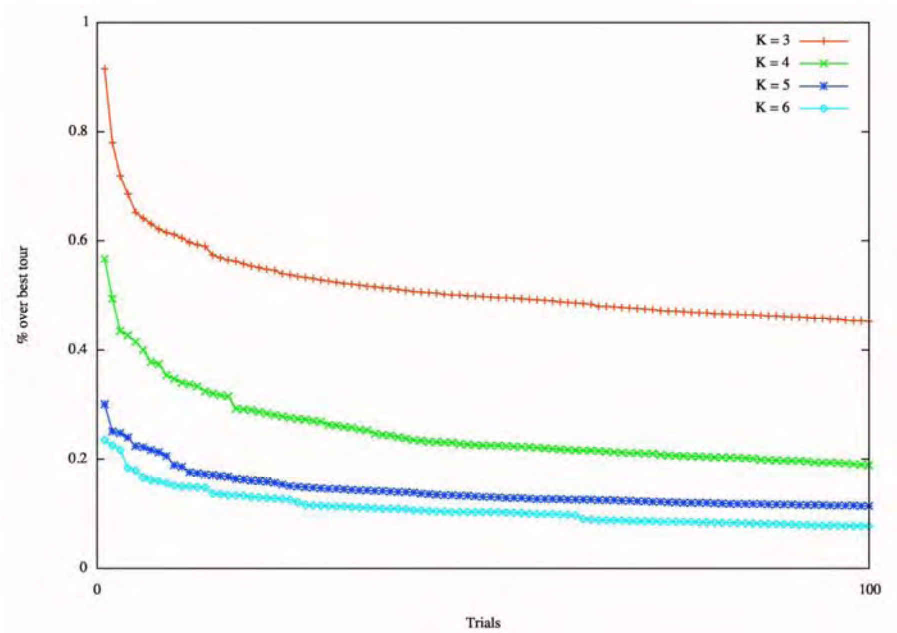

Figure 6.33. Convergence for World (Simple patching, 100 trials, 1 run).

The current best tour for this instance was found with additional runs that exploit the subproblem partitioning and tour merging facilities of LKH-2.

7. Conclusions

This report has described the implementation of a general *k*-opt submove for the Lin-Kernighan heuristic. My 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.

References

[1] D. Applegate, R. Bixby, V. Chvátal, and W. Cook,
"*Concorde: A code for solving Traveling Salesman Problems*" (1999),
http://www.tsp.gatech.edu/concorde.html

[2] D. Applegate, R. Bixby, V. Chvátal, and W. Cook,
"*Finding tours in the TSP*",
Technical Report 99885, Forschungsinstitut für Diskrete Mathematik,
Universität Bonn (1999).

[3] D. Applegate, W. Cook, and A. Rohe,
"*Chained Lin-Kernighan for Large Traveling Salesman Problems*",
Technical Report 99887, Forschungsinstitut für Diskrete Mathematik,
Universität Bonn (2000).

[4] D. Applegate, R. Bixby, V. Chvátal, and W. Cook,
"Implementing the Dantzig-Fulkerson-Johnson Algorithm for Large
Traveling Salesman Problems",
Mathematical Programming, **97**, pp. 91-153 (2003).

[5] V. Bafna and P. Pevzner,
"Genome rearrangements and sorting by reversals",
SIAM Journal on Computing, **25**(2), pp. 272-289 (1996).

[6] A. Bergeron,
"A Very Elementary Presentation of the Hannenhalli-Pevzner Theory",
Lecture Notes in Computer Science, **2089**, pp. 106-117 (2001).

[7] A. Caprara,
"Sorting by reversals is difficult",
In *Proceedings of the First International Conference on Computational Molecular Biology,* pp. 75-83 (1997).

[8] B. Chandra, H. Karloff and C. Tovey,
"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).

[9] N. Christofides and S. Eilon,
"Algorithms for large-scale traveling salesman problems",
Operations Research Quarterly, **23**, pp. 511-518 (1972).

[10] M. L. Fredman, D. S. Johnson, L. A. McGeoch, and G. Ostheimer,
"Data Structures for Traveling Salesmen",
J. Algorithms, **18**(3) pp. 432-479 (1995).

[11] B. Funke, T. Grünert and S. Irnich,
"Local Search for Vehicle Routing and Scheduling Problems: Review and Conceptual Integration".
Journal of Heuristics, **11**, pp. 267-306 (2005).

[12] D. Garg and A. Lal,
"CS360 - *Independent Study Report*",
IIT-Delhi (2002).

[13] G. Gutin and A. P. Punnen,
"Traveling Salesman Problem and Its Variations",
Kluwer Academic Publishers (2002).

[14] P. J. Hanlon, R. P. Stanley, and J. R. Stembridge,
"Some combinatorial aspects of the spectra of normally distributed random matrices".
Contemporary Mathematics, **138**, pp. 151-174 (1992).

[15] S. Hannenhalli and P. A. Pevzer,
"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).

[16] M. Held and R. M. Karp,
"The traveling-salesman problem and minimum spanning trees",
*Operations Res*earch, **18**, pp. 1138-1162 (1970).

[17] M. Held and R. M. Karp,
"The traveling-salesman problem and minimum spanning trees: Part II",
Math. Programming, pp. 6-25 (1971).

[18] K. Helsgaun,
"An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic",
European Journal of Operational Research, **12**, pp. 106-130 (2000).

[19] H. H. Hoos and T. Stützle,
"*Stochastic Local Search: Foundations and Applications*",
Morgan Kaufmann (2004).

[20] D. S. Johnson,
"Local optimization and the traveling salesman problem",
Lecture Notes in Computer Science, **442**, pp. 446-461 (1990).

[21] D. S. Johnson, L. A. McGeoch, and E. E. Rothberg,
"Asymptotic Experimental Analysis for the Held-Karp Traveling Salesman Bound",
Proc. 7th Ann. ACM-SIAM Symp. on Discrete Algorithms, pp. 341-350 (1996).

[22] D. S. Johnson and L. A. McGeoch,
"The Traveling Salesman Problem: A Case Study in Local Optimization"
in *Local Search in Combinatorial Optimization*, E. H. L. Aarts and J. K. Lenstra (editors), John-Wiley and Sons, Ltd., pp. 215-310 (1997).

[23] D. S. Johnson, L. A. McGeoch, F. Glover, and C. Rego,
"*8th DIMACS Implementation Challenge: The Traveling Salesman Problem*" (2000),
http://www.research.att.com/~dsj/chtsp/

[24] D. S. Johnson and L. A. McGeoch,
"Experimental Analysis of Heuristics for the STSP",
In *The Traveling Salesman Problem and Its Variations*, G. Gutin and A. Punnen, Editors, pp. 369-443 (2002).

[25] H. Kaplan, R. Shamir, and R. E. Tarjan,
"Faster and simpler algorithm for sorting signed permutations by reversals",
In *Proc. 8th annual ACM-SIAM Symp. on Discrete Algorithms (SODA 97)*, pp. 344-351, 1997.

[26] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys (eds.),
"The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization",
Wiley, New York (1985).

[27] S. Lin and B. W. Kernighan,
"An Effective Heuristic Algorithm for the Traveling-Salesman Problem",
Operations Research, **21**, pp. 498-516 (1973).

[28] K. T. Mak and A. J. Morton,
"Distances between traveling salesman tours",
Discrete Applied Mathematics, **58**, pp. 281-291 (1995).

[29] O. Martin, S.W. Otto, and E.W. Felten,
"Large-step markov chains for the tsp incorporating local search heuristics",
Operations Research Letters, **11**, pp. 219-224 (1992).

[30] D. Mendivil, R.Shonkwiler, and M. C. Spruill,
"An analysis of Random Restart and Iterated Improvement for Global Optimization with an application to the Traveling Salesman Problem",
Journal of optimization theory and applications, **124** (4), pp. 407-433 (2005).

[31] A. Möbius, B. Freisleben, P. Merz, and M. Schreiber,
"Combinatorial Optimization by Iterative Partial Transcription",
Physical Review E, **59** (4), pp. 4667-4674 (1999).

[32] D. Neto.
"*Efficient Cluster Compensation For Lin-Kernighan Heuristics*",
PhD thesis, University of Toronto (1999).

[33] M. Okada,
"*Studies on Probabilistic Analysis of λ–opt for Traveling Salesperson Problems*",
Doctor's thesis, Nara Institute of Science and Technology (1999).

[34] L. K. Platzman and J. J. Bartholdi,
"*Spacefilling curves and the planar traveling salesman problem*",
Journal of the ACM, **36**(4), pp. 719-737 (1989).

[35] A. Rohe,
"*Parallele Heuristiken fur sehr große Traveling Salesman Probleme*",
Diplomarbeit, Research Institute for Discrete Mathematics, Universität Bonn (1997).

[36] A. Rohe,
"Parallel Lower and Upper Bounds for Large TSPs",
ZAMM, **77**(2), pp. 429-432 (1997).

[37] E. Tannier, M. Sagot,
"*Sorting by reversals in subquadratic time*",
Rapport de recherche Nº 5097, l'INRIA (2004).

[38] W. Zhang and M. Looks,
"A Novel Local Search Algorithm for the Traveling Salesman Problem that Exploits Backbones",
IJCAI 2005, pp. 343-350 (2005).

[39] "*Traveling salesman problem web*",
http://www.tsp.gatech.edu/

[40] "*The On-Line Encyclopedia of Integer Sequences",*
http://www.research.att.com/~njas/sequences/A061714