Finding Tours in the TSP

David Applegate
Computational and Applied Mathematics
Rice University

Robert Bixby
Computational and Applied Mathematics
Rice University

Vašek Chvátal
Department of Computer Science
Rutgers University

William Cook[*]
Computational and Applied Mathematics
Rice University

ABSTRACT

The traveling salesman problem, or TSP for short, is easy to state: given a finite number of "cities" along with the cost of travel between each pair of them, find the cheapest way of visiting all the cities and returning to your starting point. The travel costs are symmetric in the sense that traveling from city X to city Y costs just as much as traveling from Y to X; the "way of visiting all the cities" is simply the order in which the cities are visited. In this report we consider the relaxed version of the TSP where we ask only for a tour of low cost. This is a preliminary version of a chapter of planned monograph on the TSP.

[*]Supported by ONR Grant N00014-98-1-0014.

CHAPTER 2

Finding Tours

2.1 INTRODUCTION

In this chapter we consider the relaxed version of the TSP where we ask only for a tour of low cost. This task is much easier, but performing it well is an important ingredient in a branch and bound search method for the TSP, as well as being an interesting problem in its own right. Indeed, tour finding is a more popular topic than the TSP itself, having a large and growing literature devoted to its various aspects. Our study will be restricted to the narrow field of tour finding that is applicable to solution methods for the TSP, namely, finding near-optimal tours within a reasonable amount of computing time. Other tour-finding topics (in particular, finding good tours very quickly) can be found in Bentley [1992], Johnson [1990], Johnson, Bentley, McGeoch, and Rothberg [1998], Johnson and McGeoch [1997], and Reinelt [1994].

2.2 LIN-KERNIGHAN

At the heart of the most successful tour-finding approaches to date lies the simple and elegant algorithm of Lin and Kernighan [1973]. This is remarkable, given the wide range of attacks that have been made on the TSP in the past two decades, and even more so when one considers that Lin and Kernighan's study was limited to problem instances having at most 110 cities (very small examples by today's standards). We begin by describing briefly some of the work leading up to their approach.

Shortly after the publication of Dantzig, Fulkerson, and Johnson's [1954] classic paper, Flood [1956] studied their 49-city example from a tour-finding perspective. He began by solving the assignment problem relaxation to the

TSP, obtaining the dual solution $(u_0, \ldots, u_{48})$. He used these values to compute a reduced cost matrix $[c_{ij}]$ by subtracting $u_i + u_j$ from the original cost of travel for each pair of cities (i, j). (Note that this does not alter the set of optimal solutions to the TSP, but it may help in finding a good tour.) Working with these costs, Flood found a *nearest neighbor tour* by choosing a starting city (in his case, Washington, D.C.) and then always proceeding to the closest city that was not already visited. He followed this with a local improvement phase, making use of the observation that in any optimal tour, $(i_0, \ldots, i_{n-1})$, for an n-city TSP, for each $0 \leq p < q < n$ (subscripts will be taken modulo n) we have

$$c_{i_{p-1}i_p} + c_{i_q i_{q+1}} \leq c_{i_{p-1}i_q} + c_{i_p i_{q+1}}. \tag{2.1}$$

A pair (p, q) that violated (2.1) was called an “intersecting pair”, and his method was to fix any such pair until the tour became “intersectionless”.

Croes [1958] used Flood's intersectionless tours as a starting point for an exhaustive search algorithm for the TSP. He also described a procedure for finding an intersectionless tour by a sequence of “inversions”. The observation is that if (p, q) intersect in the tour

$$(i_0, \ldots, i_{p-1}, i_p, \ldots, i_q, i_{q+1}, \ldots, i_{n-1}),$$

then the pair can be fixed by inverting the subsequence $(i_p, \ldots, i_q)$, that is, moving to the tour

$$(i_0, \ldots, i_{p-1}, i_q, i_{q-1}, \ldots, i_{p+1}, i_p, i_{q+1}, \ldots, i_{n-1}).$$

We call this operation *flip*(p, q). (We assume that tours are oriented, so $flip(p, q)$ is well-defined.)

A strengthening of Croes' inversion method was proposed and tested by Lin [1965]. (See also Morton and Land [1955] and Bock [1958].) Rather than simply flipping a subsequence $(i_p, \ldots, i_q)$, he also considered reinserting it (either as is, or flipped) between two other cities that are adjacent in the tour, if such a move would result in a tour of lesser cost. This increases the complexity of the algorithm, but Lin showed that it produces much better tours. (Reiter and Sherman [1965] studied a similar method, but included a specific recipe for which subsequences to consider and allowed arbitrary reorderings of the subsequence, rather than just flips.)

Lin [1965] also provided a common framework for describing intersectionless tours and tours that are optimal with respect to flips and insertion. He called a tour λ-*optimal* if it is not possible to obtain a tour of lesser cost by replacing any λ of its edges (considering the tour as a cycle in a graph) by any other set of λ edges. Thus, a tour is intersectionless if and only if it is 2-optimal. Moreover, it is not difficult to see that a tour is optimal with respect to flips and insertion if and only if it is 3-optimal. Croes' algorithm and Lin's algorithm are commonly referred to as *2-opt* and *3-opt*, respectively.

A natural next step would be to try λ-opt for some greater values of λ, but Lin found that despite the greatly increased computing time for 4-opt, the tours produced were not noticeably better than those produced by 3-opt. As an alternative, Lin and Kernighan [1973] developed an algorithm that is sometimes referred to as "variable λ-opt". The core of the algorithm is an effective search method for tentatively performing a (possibly quite long) sequence of flips such that each initial subsequence appears to have a chance of leading to a tour of lesser cost. (It may help in understanding their algorithm to note that while any λ-opt move can be realised as a sequence of flips, some of the intermediate tours may have cost greater than that of the initial tour, even in an improving λ-opt move.) If the search is successful in finding an improved tour, then the sequence of flips is made and a new search is begun. Otherwise, the tentative flips are discarded before we begin a new search, and we take care not to repeat the same unsuccessful sequence. The procedure terminates when every starting point for the search has proven to be unsuccessful.

We now describe the search method. The algorithm we present differs from the one given in Lin and Kernighan [1973], but the essential ideas are the same. We describe the method in sufficient detail to have a basis for discussing our computational study in later sections.

Suppose we are given a TSP with $c(i,j)$ representing the cost of travel between vertex i and vertex j. Let T be a tour and let *base* be a selected vertex. We will build a sequence of flip operations, and denote by *current_tour* the tour obtained by applying the flip sequence to T. For any vertex v, let $next(v)$ denote the vertex that comes immediately after v in *current_tour* and let $prev(v)$ denote the vertex that comes immediately before v. (Since a tour is oriented, *next* and *prev* are well-defined.) The only flips that will be considered are those of the form $flip(next(base), probe)$, for vertices *probe* that are distinct from *base*, $next(base)$, and $prev(base)$. Such a flip will replace the edges $(base, next(base))$ and $(probe, next(probe))$ by the edges $(next(base), next(probe))$ and $(base, probe)$. (See Figure 2.1.) *current_tour* would be improved by such a flip if and only if

$$\begin{aligned} &c(base, next(base)) + c(probe, next(probe)) > \\ &c(next(base), next(probe)) + c(base, probe), \end{aligned} \tag{2.2}$$

as in the 2-opt algorithm. Rather than demanding such an improving flip, Lin-Kernighan requires that

$$c(base, next(base)) - c(next(base), next(probe)) > 0.$$

This is a greedy approach that trys to improve a single edge in the tour. The idea can be extended as follows. Let *delta* be a variable that is set to 0 at the start of the search and is incremented by

$$c(base, next(base)) - c(next(base), next(probe)) +$$

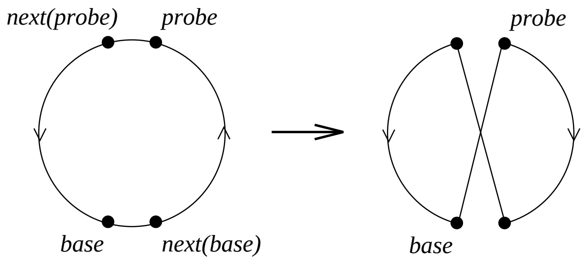


Figure 2.1. $flip(next(base), probe))$

$$c(prob, next(probe)) - c(probe, base)$$

after each $flip(next(base), probe))$. Thus, *delta* represents the amount of local improvement we have obtained thus far with the sequence of flips. (The cost of *current_tour* can be obtained by subtracting *delta* from the cost of the starting tour T.) In a general step, we require that

$$delta + c(base, next(base)) - c(next(base), next(probe)) > 0. \tag{2.3}$$

Thus, we permit *delta* to be negative, as long as it appears that we might be able to later realize an improvement. We call *probe* a *promising vertex* if (2.3) holds.

A rough outline of the search method can be summarized as follows:

```
$delta = 0$
while there exist promising vertices
do    choose a promising vertex $probe$
      $delta = delta + c(base, next(base)) - c(next(base), next(probe)) +$
                  $c(probe, next(probe)) - c(probe, base)$
      add $flip(next(base), probe))$ to the flip sequence
end.
```

If we reach a cheaper tour, we record the sequence of flips, but continue on with the search to see if we might find an even better tour.

Notice that *probe* is promising if and only if the cost of the edge

$$(next(base), next(probe))$$

is small enough. (The other two terms in (2.3) do not depend on *probe*.) So an efficient way to check for a promising vertex is to consider the edges incident with vertex $next(base)$, ordered by increasing costs. When we consider edge $(next(base), a)$, we let $probe = prev(a)$. See Figure 2.2.

Just selecting the first edge that produces a promising *probe* is too short-sighted and often leads to long sequences that do not result in better tours.

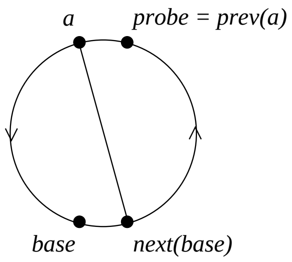


Figure 2.2. Finding a promising vertex

Instead, Lin-Kernighan also considers a third term from (2.2), choosing the edge $(next(base), a)$ that maximizes

$$c(prev(a), a) - c(next(base), a). \tag{2.4}$$

To avoid computing this quantity for all edges incident with $next(base)$, only a prescribed subset of vertices a are considered. We refer to this prescribed subset as the set of *neighbors* of a vertex. (A typical example of a neighbor set is the set of k closest vertices, for some fixed integer k.) The price we pay for using only the neighbors of $next(base)$ is that we may overlook a promising flip operation. This is outweighed, however, by the greatly reduced time of the steps in the search. We call a neighbor, a, of $next(base)$ promising if $probe = prev(a)$ is a promising vertex. The outline of the search becomes:

```
$delta = 0$
while there exist promising neighbors of $next(base)$
do   let $a$ be the promising neighbor of $next(base)$ that
     maximizes (2.4)
     $delta = delta + c(base, next(base)) - c(next(base), a) +$
                    $c(prev(a), a) - c(prev(a), base)$
     add $flip(next(base), prev(a)))$ to the flip sequence
end.
```

To increase the chances of finding an improving sequence, a limited amount of backtracking is used. For each integer $k \geq 1$, let $breadth(k)$ be the maximum number of promising neighbors we are willing to consider at level k of the search. Rather than just adding the flip involving the most promising neighbor, we will consider, separately, adding the flips involving the $breadth(k)$ promising neighbors having the greatest values of (2.4). (Lin and Kernighan set $breadth(1) = 5$, $breadth(2) = 5$, and $breadth(k) = 1$ for all $k > 2$. Setting $breadth(k) = 0$ for some k provides an upper bound on the length of any flip sequence that will be considered.)

Mak and Morton [1993] proposed another method for increasing the breadth of a search. Their idea is to try flips of the form $flip(probe, base)$, as well as those that we normally consider. This can be accomplished by considering the neighbors a of *base* (other than $next(base)$ and $prev(base)$), that satisfy

$$delta + c(base, next(base)) - c(base, a) > 0.$$

In this case, the vertices are ordered by decreasing values of

$$c(a, next(a)) - c(base, a) \tag{2.5}$$

and after a $flip(next(a), base)$, the value of *delta* is incremented by

$$c(base, next(base)) - c(base, a) + c(a, next(a)) - c(next(a), next(base)).$$

These details are analogous to those for the usual flips. The final piece of a Mak-Morton move is to change *base* to be the vertex that was $next(a)$ before the flip. This means that $next(base)$ is the same vertex before and after the flip, analogous to the fact that *base* remains the same in the usual case. See Figure 2.3.

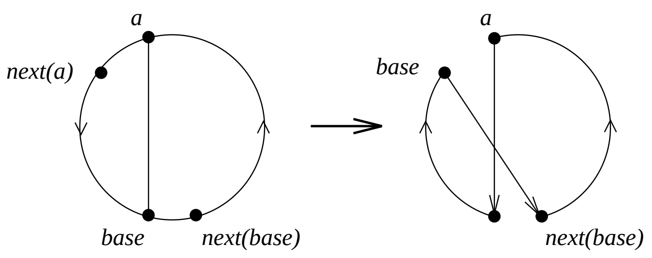


Figure 2.3. A Mak-Morton move

There is no need to consider the Mak-Morton moves separately from the usual flips, so we can create a single ordering consisting of the permitted neighbors of $next(base)$ and *base*, sorted by non-increasing values of (2.4) and (2.5), respectively. (Some vertices may appear twice in the ordering.) Call this the *lk−ordering* for *base*. At each step of the search, the vertices will be processed in this order.

To give an outline of the full search routine (incorporating backtracking and Mak-Morton moves), it is convenient to use the recursive function `step` defined in Algorithm 2.1. This function takes as arguments the current *level* and the current *delta*. A search from *base* is then just a call to `step`(1,0). Note again that at any point, the cost of *current_tour* can be computed using the cost of the initial tour and *delta*. It is easy, therefore, to detect when an improved tour has been found.

Lin-Kernighan increases the breadth of a search in a third way, by considering an alternative first step. The usual $flip(next(base), prev(a))$ removes

```
create the $lk{-}ordering$ for $base$
$i = 1$
while there exist unprocessed vertices in the $lk{-}ordering$
        and $i \leq$breadth($level$)
do      let $a$ be the next vertex in the $lk{-}ordering$
        if $a$ is specified as part of a Mak-Morton move
        then  $g = c(base, next(base)) - c(base, a) +$
                    $c(a, next(a)) - c(next(a), next(base))$
              $newbase = next(a)$
              $oldbase = base$
              add $flip(newbase, base)$ to the flip sequence
              $base = newbase$
              call step($level + 1, delta + g$)
              $base = oldbase$
        else  $g = c(base, next(base)) - c(next(base), a) +$
                    $c(prev(a), a) - c(prev(a), base)$
              add $flip(next(base), prev(a))$ to the flip sequence
              call step($level + 1, delta + g$)
        end
        if an improved tour has been found, then RETURN
        else  delete the added flip from the end of the
              flip sequence
              increment $i$
        end
end.
```

Algorithm 2.1. step(*level*, *delta*) the edge $(prev(a), a)$ from the tour. The alternative (actually a sequence of flips) will remove $(a, next(a))$. To accomplish this, we select a neighbor b of $next(a)$. There are two cases, depending on whether or not b lies on the segment of the tour from $next(base)$ to a. If b lies on this segment, then two sequences of flips are considered, the first removes $(b, next(b))$ from the tour and the second removes $(prev(b), b)$ from the tour. These moves are illustrated

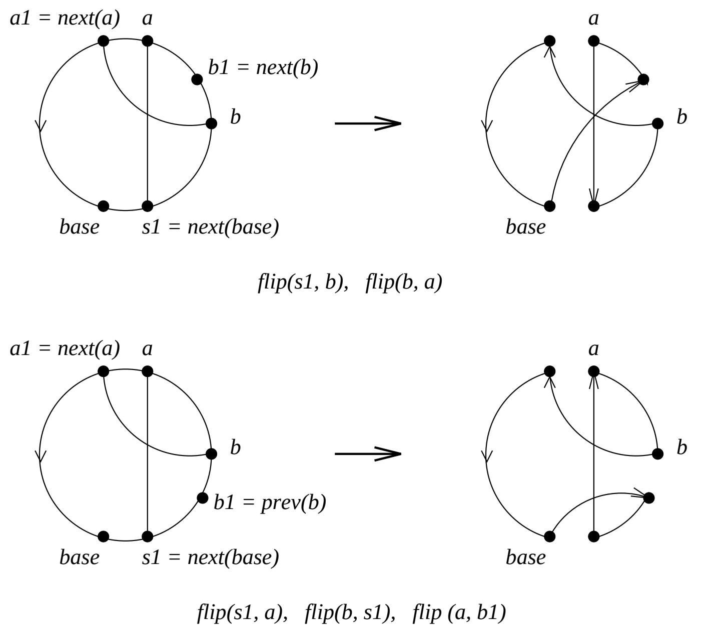
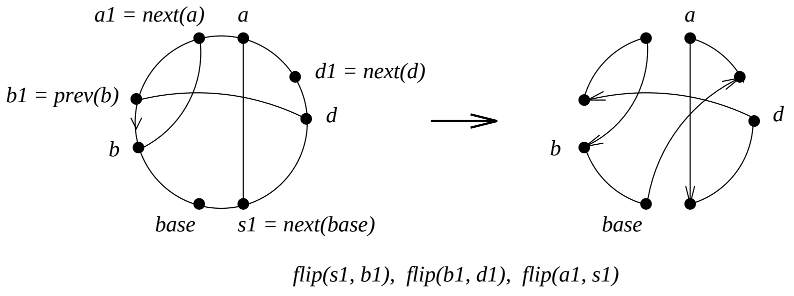


Figure 2.4. Alternative first step, case 1

in Figure 2.4, together with the flips needed to realize them. If b does not lie on the segment from $next(base)$ to a, then we select a neighbor d of $prev(b)$, such that d lies on the segment from $next(base)$ to a. We again consider two sequences of flips, the first removing $(d, next(d))$ and the second removing $(prev(d), d)$. These moves are illustrated in Figure 2.5.

To permit backtracking in this alternate first step, let $breadth_A$ be the maximum number of vertices a that we are willing to try, let $breadth_B$ be the maximum number of pairs $(b, b1)$ (where $b1$ is either $next(b)$ or $prev(b)$), and let $breadth_D$ be the maximum number of pairs $(d, d1)$ (where $d1$ is either $next(d)$ or $prev(d)$). In selecting a, we consider only the promising neighbors of $next(base)$. These are ordered by decreasing value of $c(next(a), a) - c(next(base), a)$. Call this the *A−ordering.* In selecting $(b, b1)$, we consider the neighbors of $next(a)$, distinct from $base$, $next(base)$, and a, that satisfy

$$c(next(a), b) < c(a, next(a)) + c(base, next(base)) - c(next(base), a).$$



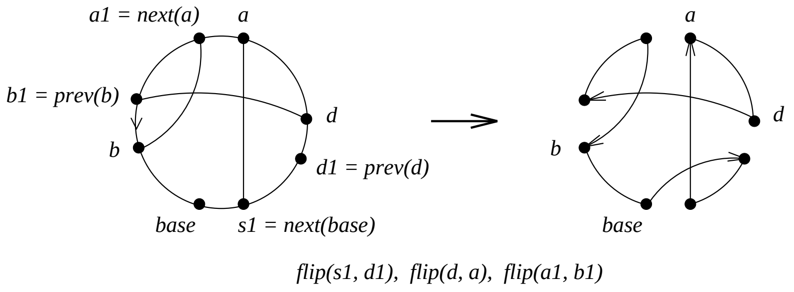


Figure 2.5. Alternative first step, case 2

(This is the analogue of inequality (2.3).) The pairs are ordered by decreasing values of $c(b1, b) - c(next(a), b)$. Call this the $B-ordering$. Finally, in selecting $(d, d1)$, we consider the neighbors of $b1$, distinct from $base$, $next(base)$, a, $next(a)$, and b, that satisfy

$$\begin{aligned} c(b1, d) < c(b, b1) &+ c(base, next(base) - c(next(base), a) \\ &+ c(a, next(a)) - c(next(a), b). \end{aligned}$$

In this case, the pairs are ordered by decreasing values of $c(d1, d) - c(b1, d)$. Call this the $D-ordering$.

With this terminology, we can write the function `alternate_step` as in Algorithm 2.2.

```
$s1 = next(base)$
create the $A-ordering$ of the neighbors of $next(base)$
i = 1
while there exist unprocessed vertices in the $A$-ordering and
      $i \leq breadth_A$
do    let $a$ be next vertex in the $A-ordering$
      $a1 = next(a)$
      create the $B-ordering$ from the neighbors of $next(a)$
      $j = 1$
      while there exists unprocessed pairs in the $B-ordering$ and
            $j \leq breadth_B$
      do    let $(b, b1)$ be the next pair in the $B-ordering$
            if $b$ lies of the tour segment from $next(base)$ to $a$
            then  add the flips listed in Figure 2.4 to the flip sequence
                  and set $delta$ to the difference of the weight of the
                  deleted edges and the weight of the added edges
                  call step(3, $delta$)
                  if an improved tour has been found, then RETURN
                  else  delete the added flips from the flip sequence
            else  create the $D-ordering$ from the neighbors of $b1$
                  k = 1
                  while there exist unprocessed pairs in the
                        $D-ordering$ and $k \leq breadth_D$
                  do    let $(d, d1)$ be the next pair in the $D-ordering$
                        add the flips listed in Figure 2.5 to the flip
                        sequence and set $delta$ as above
                        call step(4, $delta$)
                        if an improved tour has been found
                        then RETURN
                        else  delete the added flips from the
                              flip sequence
                              increment $k$
                        end
                  end
            end
            increment $j$
      end
      increment $i$
end.
```

Algorithm 2.2. alternate_step

Putting the pieces together, we can write the function `lk_search`, that takes as arguments a vertex v and a tour T, as in Algorithm 2.3.

```
initialize current_tour as T
initialize an empty flip sequence
base = v
call step(1, 0)
if an improved current_tour has been found
then  RETURN the improving flip sequence
else  call alternate_step()
      if an improved current_tour has been found
      then  RETURN the improving flip sequence
      else  RETURN with an unsuccessful flag
end.
```

Algorithm 2.3. `lk_search`(v, T)

To apply this, we mark all vertices, then call `lk_search`(v) for some vertex v. If the search is unsuccessful, we unmark v and continue with some other marked vertex. If the search is successful, then it is possible that some unmarked vertices may now permit successful searches. For this reason, Lin and Kernighan again mark all vertices before continuing with the next search. This was an appropriate strategy for the problem instances they studied, but it is too time consuming for larger instances. The trouble is that unmarked vertices that are not close to the improving sequence of flips have little chance of permitting successful searches. To deal with this issue, Bentley [1990] (in the context of 2-opt), proposed to mark only those vertices that are the end vertices of one of the flips in the improving sequence. This leads to somewhat worse tours, but greatly improves the running time of the algorithm. We summarize the method in the function `lin_kernighan` described in Algorithm 2.4; `lin_kernighan` takes as an argument a tour T.

The Lin-Kerngihan routine consistently produces good quality tours on a wide variety of problem instances. Computational results on variations of the algorithm can be found in Bachem and Wottawa [1992], Bland and Shallcross [1989], Codenotti, Manzini, Margara, and Resta [1996], Grötschel and Holland [1991], Johnson [1990], Johnson and McGeoch [1997], Jünger, Reinelt, and Rinaldi [1995], Mak and Morton [1993], Padberg and Rinaldi [1991], Perttunen [1994], Reinelt [1994], Rohe [1997], Schäfer [1994], Verhoeven, Aarts,

```
initialize $lk\_tour$ as $T$
mark all vertices
while there exist marked vertices
do  select a marked vertex $v$
    call lk_search($v$, $lk\_tour$)
    if an improving sequence of flips is found
    then while the flip sequence is nonempty
         do  let $flip(x, y)$ be the next flip in the sequence
             apply $flip(x, y)$ to $lk\_tour$ to obtain a new
             $lk\_tour$
             mark vertices $x$ and $y$
             delete $flip(x, y)$ from the flip sequence
         end
    else unmark $v$
end
RETURN tour $lk\_tour$.
```

Algorithm 2.4. `lin_kernighan`(T)

van de Sluis, and Vaessens [1995], Verhoeven, Swinkels, and Aarts [1995], as well as the original paper of Lin and Kernighan [1973].

An important part of Lin and Kernighan's overall tour-finding scheme is the repeated use of `lin_kernighan`. The idea is simple: for as long as computing time is available, we have a chance of finding a tour that is better than the best we have found thus far by generating a new initial tour and applying `lin_kernighan`. This worked extremely well in the examples they studied and it remained the standard method for producing very good tours for over fifteen years. The situation changed, however, with the publication of the work of Martin, Otto, and Felten [1991] (see also Martin, Otto, and Felten [1992] and Martin and Otto [1996]). Their new idea was to work harder on the tours produced by `lin_kernighan`, rather than starting from scratch on a new tour. They proposed to *kick* the tour found by `lin_kernighan` (that is, to perturb it slightly), and apply `lin_kernighan` to the new tour. Their kick was a sequence of flips that produces the special type of 4-opt move, called a *double-bridge*, that is illustrated in Figure 2.6. (There are many other natural candidates for kicking, but this particular one appears to work quite well.) The resulting algorithm, known as *Chained Lin-Kernighan*, starts with a tour S and proceeds as described in Algorithm 2.5.

Chained Lin-Kernighan is a great improvement over the "Repeated Lin-Kernighan" approach. Computational results comparing the two schemes can be found in Codenotti, Manzini, Margara, and Resta [1996], Hong, Kahng,

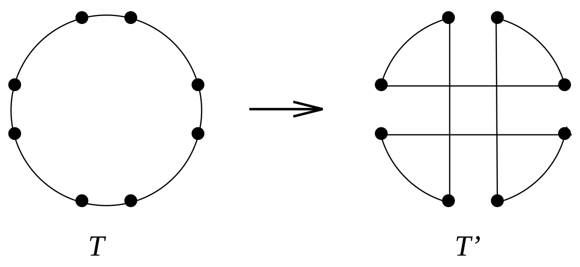


Figure 2.6. A double-bridge

```
call lin_kernighan($S$) to produce the tour $T$
while computing time remains
do    find a kicking sequence of flips
      apply the kicking sequence to $T$
      call lin_kernighan($T$) to produce the tour $T'$
      if $T'$ is cheaper than $T$
      then  replace $T$ by $T'$
      else  use the kicking sequence (in "reverse") to convert
            $T$ back to the old $T$ (the one we had before
            applying the kick)
end
RETURN $T$.
```

Algorithm 2.5. Chained Lin-Kernighan and Moon [1998], Johnson [1990], Johnson and McGeoch [1997], Jünger, Reinelt, and Rinaldi [1995], Martin, Otto, and Felten [1992], Neto [1999], and Reinelt [1994].

In the remainder of the chapter, we will discuss the computational issues involved in implementing and using Chained Lin-Kernighan. It should be remarked that Martin, Otto, and Felten described a more general scheme then the one we have outlined. They proposed to use a simulated annealing-like approach and replace T by T' (even if T' is not a better tour) with a certain probability that depends on the difference in the costs of the two tours.

We call Martin, Otto, and Felten's algorithm "Chained Lin-Kernighan" to match the "Chained Local Optimization" concept introduced in Martin and Otto [1995], and to avoid a conflict with Johnson and McGeoch's [1997] use of the term "Iterated Lin-Kernighan" to mean the special case of the algorithm when random double-bridge moves are used as kicks and no simulated annealing approach is used. (See also Johnson [1990].)

2.3 FLIPPER ROUTINES

The main task in converting the outline of Chained Lin-Kernighan into a computer code is to build data structures to maintain the three tours: the *lk_tour* in `lin_kernighan`, the *current_tour* in `lk_search`, and the overall tour (T in Algorithm 2.5). If these are not implemented carefully, then operations involving them will be the dominant factor in the running time of the code.

Note that the three data structures need not be distinct, since additional flip operations can be used to the undo the flips made during an unsuccessful `lk_search`, as well as to undo all of the flips made during an unsuccessful call to `lin_kernighan`.

An abstract data type sufficient to represent all three tours should provide the functions

- `flip`(a, b) - inverts the segment of the tour from a to b
- `next`(a) - returns the vertex immediately after a in the tour
- `prev`(a) - returns the vertex immediately before a in the tour
- `sequence`(a, b, c) - returns 1 is b lies in the segment of the tour from a to c, and returns 0 otherwise

as well as an initialization routine and a routine that returns the tour represented by the data structure (we are following Applegate, Chvátal, and Cook [1990]). In the outline of Chained Lin-Kernighan, whenever `flip(`a, b`)` needs to be called, $prev(a)$ and $next(b)$ are readily available (without making calls to `prev` and `next`). This additional information may be useful in implementing `flip`, so we allow our tour data structures to use the alternative

- `flip`(x, a, b, y) - inverts the segment of the tour from a to b (where $x = prev(a)$ and $y = next(b)$)

if this is needed.

Asymptotic analysis of two tour data structures can be found in Chrobak, Szymacha, and Krawczyk [1990] and Margot [1992]. In both cases, the authors showed that the functions can be implemented to run in $O(\log n)$ time per function call for an n-city TSP. We present a detailed computational study of practical versions of these two structures as well as several alternatives.

An excellent reference for tour data structures is the paper by Fredman, Johnson, McGeoch and Ostheimer [1995]. Their study works with a slightly different version of `flip`: they allow the function to invert either the segment from a to b or the segment from $next(b)$ to $prev(a)$. Along with computational results, Fredman, Johnson, McGeoch, and Ostheimer [1995] established a lower bound of $\Omega(\log n / \log\log n)$ per function call on the amortized computation time for any tour data structure in the cell probe model of computation.

Test Data

To provide a test for the tour data structures, we need to specify both a problem instance and a particular implementation of Chained Lin-Kernighan. Problem instances are readily available: Reinelt [1991,1991a,1995] has collected over 100 of them, with sizes ranging from 14 to 85,900 cities, in a library called TSPLIB. From this library, we have selected two instances arising in applications and one instance obtained from the locations of cities on a map. These examples are listed in Table 2.1. The version of Chained Lin-

Table 2.1. Problem Instances

Name	Size	Cost Function	Target Tour
pcb3038	3,038	Rounded Euclidean	139070
usa13509	13,509	Rounded Euclidean	20172983
pla85900	85,900	Ceiling Euclidean	143564780

Kernighan we use is one that is typical of those studied in Section 3.4 of this chapter. The various parameters that must be set in Chained Lin-Kernighan do have an impact on the relative performance of the tour data structures, but the trends will be apparent with this test version. For each test problem, we run Chained Lin-Kernighan until a tour is found that is at least as good as the "Target Tour" listed in Table 2.2. These values are 1% greater than known lower bounds for the problem instances. Since the operation of Chained Lin-Kernighan is independent of the particular tour data structure that is used, each of our test runs will be faced with exactly the same set of `flip`, `next`, `prev`, and `sequence` operations. Some important statistics for these operations are given in Table 2.2. The `lin_kernighan`, `lk_search`, `flip`, `next`, `prev`, and `sequence` rows give the number of calls to the listed function; "`lin_kernighan` winners" is the number of calls to `lin_kernighan`

Table 2.2. Statistics for Test Data

Function	pcb3038	usa13509	pla85900
`lin_kernighan`	141	468	1,842
`lin_kernighan` winners	91	261	1,169
flips in a `lin_kernighan` winner	61.0	99.1	108.3
flips in a `lin_kernighan` loser	42.5	88.2	86.4
`lk_search`	19,855	95,315	376,897
`lk_search` winners	1,657	9,206	29,126
flips in an `lk_search` winner	4.7	4.8	6.3
`flip`	180,073	1,380,545	5,110,340
undo flips	172,396	1,336,428	4,925,574
size of a flip	75.6	195.3	607.9
flips of size at most 5	67,645	647,293	1,463,090
`next`	662,436	6,019,892	14,177,723
`prev`	715,192	4,817,483	13,758,748
`sequence`	89,755	773,750	263,7757

that resulted in a tour that was at least as good as the current best tour; "flips in a `lin_kernighan` winner" is the average number of flips performed on *lk_tour* in `lin_kernighan` winners; "flips in a `lin_kernighan` loser" is the average number of flips performed on *lk_tour* in calls to `lin_kernighan` that resulted in a tour that was worse than the current best tour; "`lk_search` winners" is the number of calls to `lk_search` that returned an improving sequence; "flips in an `lk_search` winner" is the average length of an improving sequence of flips; "undo flip" is the number of flip operations that are deleted in `lk_search`; "size of a flip" is the average, over all calls `flip`(a, b), of the smaller of the number of cities we visit when we travel in the tour from a to b (including a and b) and the number of cities we visit when we travel from $next(b)$ to $prev(a)$ (including $next(b)$ and $prev(a)$); "flips of size at most 5" is the total number of flips of size 5 or less.

The codes tested in this section are written in the C Programming Language (Kernighan and Ritchie [1978]), and run on a workstation based on a Digital Equipment Corporation Alpha 21164a microprocessor, running at 500 Mhz, with 2 MByte external cache. The times reported are in seconds of computing time, including the time spent in computing the starting tour and the neighbor sets.

Arrays

A natural candidate for a tour data structure is to keep an array, called *tour*, of the cities in the order they appear in the tour. To locate a given city in *tour*, we use another array, called *inverse*, where the ith item in *inverse* contains the location of city i in *tour*. The top pair of arrays in Figure 2.7 illustrate the data structure for the tour 9-0-8-5-7-2-6-1-4-3.

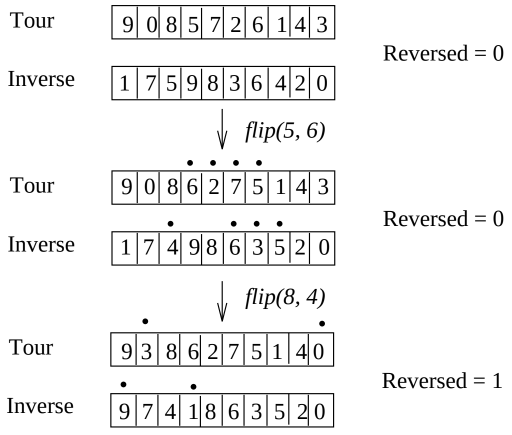


Figure 2.7. Array flipper

This data structure is particularly easy to implement. The functions `next` and `prev` are provided by the formulas

$$next(v) = tour(inverse(v) + 1)$$
$$prev(v) = tour(inverse(v) - 1)$$

where the indices are taken modulo n. To provide `sequence`, we can determine whether or not city b lies on the tour segment from a to c by examining the values of $inverse(a)$, $inverse(b)$, and $inverse(c)$. The time consuming operation is `flip`. To carry out `flip`(a, b), we need to work our way through the a to b segment of the tour array, swapping the positions of a and b, $next(a)$ and $prev(b)$, and so on, until the entire segment has been inverted. At the same time, we need to swap $inverse(a)$ and $inverse(b)$, $inverse(next(a))$ and $inverse(prev(b))$, and so on. This operation is illustrated in the middle pair of arrays in Figure 2.7. (The entries with the dots are the ones that were swapped.)

It is clearly too time consuming to create new copies of *tour* and *inverse* each time `lk_search` is called, so we will use the same pair of arrays to represent both *lk_tour* and *current_tour*. To do this, we us an additional call to `flip` whenever we delete an item from the current flip sequence. (To delete $flip(a, b)$, we call `flip`(b, a).)

The performance of this array data structure is quite poor. For the three test problems, the Alpha workstation times (in seconds) are

pcb3038	usa13509	pla85900
7.2	246.6	10422.5

respectively.

Not surprisingly, the above running times are dominated by the times for the flip operations: 91.7% for pcb3038, 97.4% for usa13509 and 99.4% for pla85900. A simple way to improve this is to maintain a *reversed* bit, that indicates the orientation of the tour. If *reversed* is set to 0, then *tour* gives the proper order of the cities, and if it is set to 1, then *tour* gives the cities in the reverse order. The big advantage is that we can carry out `flip`(a, b) by inverting the tour segment from *next*(b) to *prev*(a) and complementing the *reversed* bit, if the segment from *next*(b) to *prev*(a) is shorter than the segment from a to b. The *reversed* bit must be examined when we answer `next`, `prev`, and `sequence`, but this extra computation is more than compensated by the much lower time for `flip`. The resulting code improves the running times to

pcb3038	usa13509	pla85900
1.6	21.6	265.9

on our test set. In this version, `flip` is fast enough that we can lower the running times a bit more by also representing the overall best tour with the same pair arrays that is used for *lk_tour* and *current_tour*. This means that we must keep a list of all of the successful flip sequences made during a given call to `lin_kernighan`, and “undo” all of these flips: working in reverse order, we call `flip`(b, a) for each *flip*(a, b). The slightly better running times are

pcb3038	usa13509	pla85900
1.6	21.5	242.2

for our test problems.

A breakdown of the time spent in the tour operations for this last version of the array based code is given in Table 2.3. The rapidly growing time for

Table 2.3. Profile for Arrays

Function	pcb3038	usa13509	pla85900
`flip`	51%	74%	88%
`next`	1%	1%	0%
`prev`	1%	1%	0%
`sequence`	0%	0%	0%

`flip` means that the data structure is probably not acceptable for instances that are much larger than pla85900. On the other hand, given the ease in which the computer code can be written, the good performance of arrays on the two smaller problems indicates that they may be the method of choice in many situations.

Linked Lists

A second natural tour data structure is a doubly linked list, where each city has pointers to its two neighbors in the tour. (See Figure 2.8.) With this

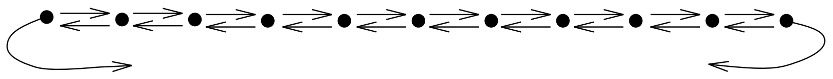


Figure 2.8. Linked list tour

structure, it is convenient to use the alternative `flip`(x, a, b, y) form of the `flip` function, since this allows us to implement `flip` by manipulating four pointers:

$$\begin{gathered}\text{replace } x \to a \text{ by } x \to b,\\ \text{replace } a \to x \text{ by } a \to y,\\ \text{replace } y \to b \text{ by } y \to a,\\ \text{replace } b \to y \text{ by } b \to x.\end{gathered}$$

The orientation of the tour can be maintained by choosing one or more cities and marking which of their two neighbors is *next* and which is *prev*. Call such a marked city a *guide*. Initially we can select any subset of the cities as guides, since the orientation of the starting tour is known, and after `flip`(x, a, b, y) we can directly store the information to use any one or more of a, b, x, and y as a guide (the old guides are invalidated by the flip operation). To implement `next`(v), we start at city v and trace the tour until we arrive at a guide, which will indicate whether or not the first city we visited after v was $next(v)$. If it was indeed $next(v)$, then we return it. Otherwise we return the other neighbor of v. The same procedure can be used to implement `prev`(v). To implement `sequence`$(a,\ b,\ c)$, we first trace the tour from a until we reach a guide, in order to learn the orientation of the neighbors of a, then trace the tour in the forward direction from a. If we reach c before we reach b, then we return 0. Otherwise, we return 1.

A difficulty with this straightforward implementation is that we will often traverse large portions of the tour in calls to `next`, `prev`, and `sequence`. Margot's [1992] answer is to include additional information in the linked list to allow the traversals to skip over large blocks of cities. (A similar method was proposed by Shallcross [1990].) The simplest version is to include a second doubly linked list consisting of a subset of $\sqrt{n}$ of the cities, and require that each of these cities be guides. (See Figure 2.9.) As long as the selected cities are spread out roughly evenly, this requires only $O(\sqrt{n})$ additional work in *flip*: we trace the tour from a in the direction of y until we reach one of the selected cities and fix its orientation, we then use the extra double linked

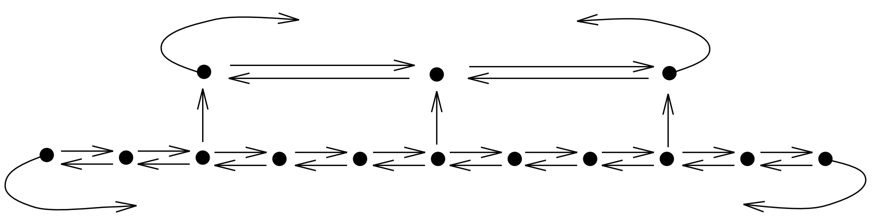


Figure 2.9. Linked list with a second level

list to fix the orientation of each of the other selected cities. Furthermore, with the large supply of guides, `next` and `prev` will run in $O(\sqrt{n})$ time, and with some additional work, `sequence` can also be implemented in $O(\sqrt{n})$ time (using the extra doubly linked list to trace from a to c).

Margot [1992] takes this idea to its natural conclusion, adding not one extra doubly linked list, but $\log n$ additional lists, each one a subset of the previous list. He also gives a construction for explicitly maintaining a balance condition that keeps the cities spread out roughly evenly, and thus obtains an $O(\log n)$ running time bound for each of the tour functions. We will study how this idea of having additional linked lists works in practice.

To begin, let us consider the straightforward implementation of a single doubly linked list. Given the low cost (in terms of CPU time) for the flip operations, we will use the same list to represent the overall tour, *lk_tour*, and *current_tour*. As guides, we use the two ends of the most recent `flip`.

The running times are

pcb3038	usa13509	pla85900
50.8	5,929.4	$> 50,000$

for our test problems. (The run on pla85900 was not completed after 50,000 seconds on the Alpha workstation.) This performance is even worse than the original (no *reversed* bit) array implementation. The profile of the runs on the two smaller problems, given in Table 2.4, indicates that the search for guides

Table 2.4. Profile for Linked Lists

Function	pcb3038	usa13509
`flip`	0%	0%
`next`	41%	37%
`prev`	51%	56%
`sequence`	6%	6%

is taking nearly all of the CPU time. The main reason for this is simply that the tour segments that need to be traced can be quite long. To attack this issue, we modified the code to carry out the search for a guide simultaneously in both directions around the tour. If we are working from random cities, then this change should not have much of an affect on the running times, since, on average, we would be still be visiting the same number of cities per search. The important point, however, is that we are not working from random locations: the `next`, `prev`, and `sequence` queries tend to be from cities that are near to the previous `flip` (due to the use of neighbor sets). Indeed, the change improves the running times to

pcb3038	usa13509	pla85900
15.7	426.3	24,047.9

for our test problems.

The profile for the improved code is given in Table 2.5. Although the runing

Table 2.5. Profile for Linked Lists with Simultaneous Searches

Function	pcb3038	usa13509	pla85900
`flip`	2%	1%	0%
`next`	31%	35%	28%
`prev`	32%	35%	23%
`sequence`	19%	24%	49%

times are better, it is clear that guide searching is still taking too long. A brute force way to deal with this problem is to give up the constant-time flip operations, and explicitly store the two neighbors of each city as *prev* and *next*. To maintain this information, we will need to swap *prev* and *next* for each city along the segment that is flipped, so flips will become much more expensive. As in the array implementation, it is important to use a *reversed* bit to permit us to flip the shorter of the two segments in a flip operation. This give the more respectable

pcb3038	usa13509	pla85900
2.8	113.6	2,109.5

performance.

A difficulty with this code (as opposed to the array implementation) is that we need to do extra traversals of the tour in order to determine whether it is better to invert the segment from a to b or the segment from $next(b)$ to $prev(a)$, in a call to `flip`(a, b). We can get around this by maintaining an *index* for each city, that gives its relative position in the tour. The indices are consecutive integers, starting from some city. With such indices, we can immediately determine which of the two possible segments in a flip operation is the shorter. As a byproduct, we can also use the indices to implement `sequence` without traversing the tour. The downside is that the index for each city in the tour segment that is inverted in a flip operation needs to be updated. The resulting code is indeed faster, however, as is shown by the running times

pcb3038	usa13509	pla85900
1.8	65.6	697.3

for our test instances.

The profile for these runs is given in Table 2.6. It indicates that the only

Table 2.6. Profile for Linked Lists with Explicit *next* and *prev*

Function	pcb3038	usa13509	pla85900
`flip`	54%	79%	94%
`next`	1%	1%	0%
`prev`	1%	1%	0%
`sequence`	0%	0%	0%

way to make substantial improvements in the code is to somehow speed up the flip operations, without completely sacrificing `next` and `prev` as we did earlier. Fortunately, this is exactly the type of improvement provided by Margot's idea of keeping additional linked lists as "express lanes." In our implementation, we follow the lessons learned from the single list implementations: we use simple linked lists for all lower levels of the data structure and a linked list with explicit *next* and *prev*, reversed bit, and indices, for the top level. Rather than choosing $n^{(D-k)/D}$ cities for the kth level in a D-level data structure (where k runs from 0 up to $D-1$), we allow a bit more flexibility, choosing first some constant γ and then selecting $\gamma^k n^{(D-k)/D}$ cities at the kth level. In Table 2.7, we report the running times using two, three, and four levels. The

Table 2.7. Running Times for Multi-level Linked Lists

Structure	pcb3038	usa13509	pla85900
2 Levels	1.7	16.0	104.1
3 Levels	2.3	18.5	81.4
4 Levels	2.5	20.3	82.1

values of γ were 8, 5 and 3, respectively. (These were the (positive integer) values giving the best results for the particular data structure.) The profile

Table 2.8. Profile for 3-Level Linked Lists

Function	pcb3038	usa13509	pla85900
`flip`	42%	37%	38%
`next`	5%	8%	10%
`prev`	6%	10%	9%
`sequence`	2%	3%	4%

for the 3-level data structure is given in Table 2.8. The running times are a big improvement over the best times for our single list implementations, and are also significantly better that the array implementation (with a *reversed* bit). On the other hand, the times point out that the theoretically attractive method of using $\log n$ levels is probably too aggressive to use in practice, for example, at 85,900 cities, three levels is preferable to four or more levels.

Two-Layered Lists

Multi-level linked lists traded off very fast `next` and `prev` times for improved performance in `flip` operations. A different approach, proposed by Chrobak, Szymacha, and Krawczyk [1990], aims at getting improvements in `flip` while maintaining the constant time operation of `next` and `prev`. The idea is to divide the tour into blocks of size roughly $\sqrt{n}$. Each block has an associated *parent* node, and the parents are organized in a cycle that gives the ordering of the blocks in the tour. An important concept is that each parent p has a bit, *p.reversed*, that indicates whether or not the tour segment in the associated block is reversed in the tour represented by the data structure. These bits allow us to invert a block of cities in constant time and this leads to a fast implementation of `flip`. Chrobak, Szymacha, and Krawczyk call their data structure a *two-layered list.* It is studied in detail in Fredman, Johnson, McGeoch, and Ostheimer [1995] (under the name "two-level trees"), and we follow their implementation.

The tour segments making up the blocks in the data structure are represented as doubly linked lists with explicit *next* and *prev* pointers and with indices giving the relative location of the cities in the segment. The cities that are ends of a block also have a pointer to the neighboring city in the tour that is not a member of their block. Each city has a pointer to the parent of its block and each parent has pointers to the two cities that are the ends of its associated tour segment. The cycle of parents is represented by a doubly linked list with explicit *next* and *prev*, location indices, and a *reversed* bit. A sketch of the data structure is given in Figure 2.10.

The structure does indeed allow constant time `next` and `prev` operations, since, for example, $next(a)$, for a given city a, is the city in a's *next* pointer if *reversed* and the reversed bit of a's parent are equal, and otherwise $next(a)$ is the city in a's *prev* pointer. (The "if" test can be avoided if we store the *next* and *prev* pointers as a two element array, and index the array by the exclusive or of *reversed* and a's parent's reversed bit.) The `sequence` operations can

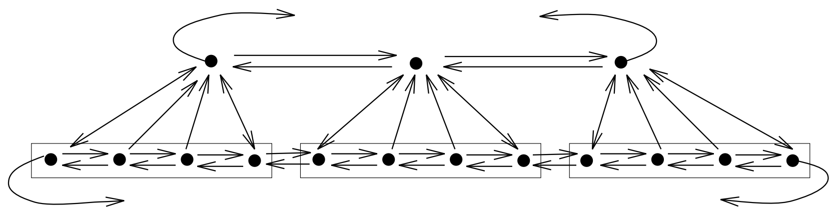


Figure 2.10. Two-Layered List

also be provided in constant time, making use of the indices on the cities to determine the relative order within a segment and the indices for the parents to determine the relative order of the blocks. If the size of the blocks are kept to be roughly $\sqrt{n}$, then `flip` can be implemented to run in $O(\sqrt{n})$. Chrobak, Szymacha, and Krawczyk accomplish this in the following way. At the start, each block is of size between $\sqrt{n}$ and $2\sqrt{n}$. To perform `flip`(a, b) we examine city a. If a is not the first city in its block (or the last city if the block is reversed), then we remove the portion of the block that precedes a and merge it with the preceding block. If the merged block now has size larger than $2\sqrt{n}$, then it is split into two blocks of (nearly) equal size. Next, we merge a's block with the block following it, and again split the merged block in two if it is too large. In a similar way, we make city b the last city in its block (or first city, if the block is reversed). Now the tour segment from a to b can be inverted by inverting the segment in the parent cycle from the parent of a to the parent of b, and complementing the reversed bit of each parent node that is involved in the flip.

Fredman, Johnson, McGeoch, and Ostheimer speed up the practical performance of this `flip` procedure in three ways. Firstly, they give up the idea of explicitly keeping the size of the segments balanced; this could have a detrimental effect on the data structure in the long run, but it seems to be the appropriate choice in practice. Secondly, they make a the first city in its block by either merging the portion of the segment preceding a with the preceding block or merging the portion of the segment starting at a with the following block (depending on which of the two segments is shorter), rather than performing two merges. Thirdly, if a and b are in the same block and are not too far apart in the tour segment (this can be determined using the indices), then the segment from a to b is inverted directly in the linked list structure for the block, rather than performing the merges that make a and b the ends of the block.

In the Fredman, Johnson, McGeoch, and Ostheimer implementation, the initial blocks are chosen to contain approximately K cities each, where K is some constant. If cities a and b are in the same block and no more than $.75K$ cities apart, then `flip`(a, b) is performed directly in the linked list for the block. Although the running times are affected by the choice of K, the dependence is fortunately not that strong. For this reason, Fredman, Johnson, McGeoch, and Ostheimer choose to use $K = 100$ in their code for all problem instances having at most 100,000 cities. In our implementation, we set $K = \alpha\sqrt{n}$ to allow the code to be somewhat more robust over a wider range of problem sizes. The default value in our code is $\alpha = 0.5$.

Fredman, Johnson, McGeoch, and Ostheimer point out that the choice of $.75K$ as the cutoff for performing `flip` operations directly in a block helps keep the size of the blocks roughly in balance, since only the larger blocks are likely to be split by `flip` operations involving two cities in the same block. In our computer code, the operations necessary to perform a merge are considerably faster than those for inverting a segment of the linked lists, so it is worthwhile to consider cutoffs βK for smaller values of β. Based on a series of tests, we have chosen $\beta = 0.3$ as our default value.

Using the same data structure for *lk_tour*, *current_tour*, and the overall tour, the code gives the running times

pcb3038	usa13509	pla85900
1.2	10.1	43.9

for our test instances. A profile of the test runs is given in Table 2.9.

Table 2.9. Profile for Two-Layered Lists

Function	pcb3038	usa13509	pla85900
`flip`	19%	19%	26%
`next`	4%	4%	3%
`prev`	3%	4%	3%
`sequence`	1%	1%	1%

The performance of two-layered lists is very good—the test results are nearly a factor of 2 better than those for 3-level linked lists. Furthermore, the profile reports that on the three test instances, the time spent on the tour operations is less than one third of the total running time of the code. Nonetheless, it certainly should be possible to improve the performance of the data structure with some additional tweaks or new ideas. One possibility would be to replace the linked list used to represent the parent cycle by something more effective. We made one attempt in this direction, using an array with *reversed* bit for the parents, but the running times

pcb3038	usa13509	pla85900
1.2	10.3	46.0

are not as good as those for the linked list structure (due to the extra dereferencing that was needed).

Binary Trees

Along with two-layered lists, Chrobak, Szymacha, and Krawczyk [1990] proposed a elegant method for obtaining an O(log n) bound on tour operations using binary trees. The key idea is to attach a *reversed* bit to each node of the tree indicating how the subtree rooted at that node should be interpreted.

Let B be a rooted binary tree on n vertices, with each vertex associated with a unique city. If all of the *reversed* bits are set to 0, then the tour represented by B is that given by an inorder traversal of the cities. Setting the *reversed* bit of a vertex v to 1 inverts the tour segment associated with the subtree rooted at v.

In general, suppose some subset of the *reversed* bits are set to 1. For each vertex v, let $parity(v)$ denote the parity of the number of *reversed* bits that are equal to 1 on the path from v up to the root of B (including both v and the root). The tour represented by B is given by traversing the tree, starting at the root, using a rule that depends on $parity(v)$ for each vertex v. If $parity(v)$ is 0 then we traverse the subtree rooted at v by first traversing the subtree rooted at the left child of v, then v itself, followed by a traversal of the subtree rooted at the right child of v. If $parity(v)$ is 1 then we traverse the subtree in the reverse order, first traversing the subtree rooted at the right child of v, then v itself, followed by the subtree rooted at the left child of v. As an example, the tree given in Figure 2.11 represents the tour 9-0-8-5-7-2-6-1-4-3.

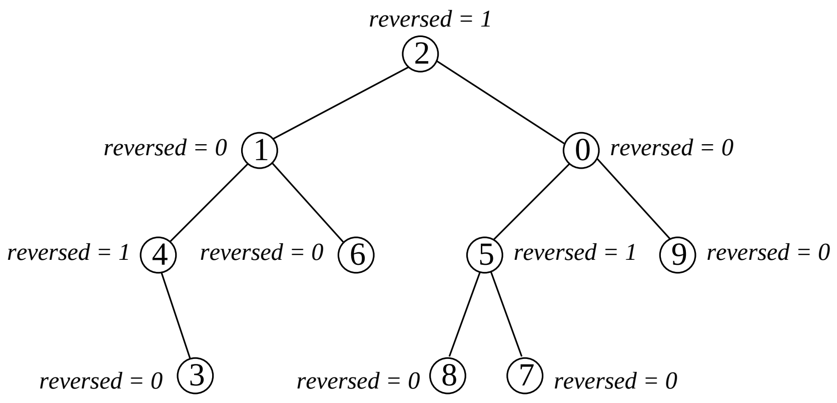


Figure 2.11. Binary Tree for 9-0-8-5-7-2-6-1-4-3

It is clear from this definition that the tour represented by B is unchanged if we pick some vertex v, complement its *reversed* bit and the *reversed* bits of its children, and swap v's left child with v's right child. We call this procedure *pushing* v's *reversed* bit.

The role of the *reversed* bit is to allow us to implement `flip` efficiently. To carry out `flip`(a, b), we partition B into into several components to isolate the $[a, b]$ segment, we flip this segment by complementing a *reversed* bit, and we glue the tree back together. The partitioning and gluing can be handled by the standard *split* and *join* operations for binary trees, as described, for example, in Knuth [1973] or Tarjan [1983]. A simple way to handle the complications of the *reversed* bits is to push any bit that is set to one on the path from x to the root of the tree, starting at the root and working backwards. (It is more efficient not to do this, but the difference is quite small.)

Similarly, we can implement `sequence`(a, b, c) by splitting B into components to isolate $[a, c]$, and checking which component contains b.

Finally, given a tree B, we can find *next*(a) and *prev*(a) by searching B starting at the vertex associated with a. The amount of work needed for the search is bounded by the maximum depth of the tree.

To obtain the $O(\log n)$ result, Chrobak, Szymacha, and Krawczyk [1990] used balanced binary trees (AVL trees in their implementation). We did not implement this feature, but our code does not appear to be hurt by the fact that the trees are not being explicitly balanced. Indeed, for each of the four tour operations we computed the depth of the vertices corresponding to the cities involved in the operation (at the time the function was called). The average value over all operations was

pcb3038	usa13509	pla85900
6.0	6.3	6.7

for our set of test problems.

The running times for the implementation were

pcb3038	usa13509	pla85900
1.4	12.6	52.9

respectively. In this test, we used the same data structure to maintain *lk_tour*, *current_tour*, and the overall tour. Although the times are slightly worse than the times for the two-layered list implementation, the binary-tree data structure is much more natural to code than two-layered lists, as well as being more efficient on larger instances. A profile for the runs is given in Table 2.10.

An alternative approach for implementing `flip` was proposed by Applegate, Chvátal, and Cook [1990] and Fredman, Johnson, McGeoch, and Ostheimer [1995]. Rather than splitting the tree into components, we can perform *splay* operations (see Sleator and Tarjan [1983] and Tarjan [1983]) on B, to make the vertices in the flipping segment appear together as a single

Table 2.10. Profile for Binary Trees

	pcb3038	usa13509	pla85900
`flip`	17%	16%	17%
`next`	7%	10%	13%
`prev`	9%	12%	11%
`sequence`	5%	6%	6%

subtree of the tree. We then complement the *reversed* bit associated with the root of the subtree. A detailed treatment of this approach can be found in Fredman, Johnson, McGeoch, and Ostheimer [1995]. The running times for our implementation were

pcb3038	usa13509	pla85900
1.5	12.7	50.5

for the test instances. These times are similar to those for the split and join version of the data structure.

Summary

From the tests, it is clear that the simple array data structure is adequate for many applications, particularly when the instances are relatively small (say at most 10,000 cities). The multi-level linked-list implementations perform better than arrays on larger instances, but they are dominated in performance by the two-layered list data structure. Binary trees (including the splay tree variants) perform slightly worse than two-layered lists on TSPLIB problems, but they are very easy to code and should be the data structure of choice in applications involving large instances. Indeed, on an example having 5 million cities, the binary-tree codes were nearly a factor of 2 faster than two-layered lists.

We did not include, in our tests, the *segment-tree* data structure proposed by Applegate, Chvátal, and Cook [1990]. Details of this implementation can be found in Fredman, Johnson, McGeoch, and Ostheimer [1995] and Verhoeven, Swinkels, and Aarts [1995]. Segment trees suffer from the same drawbacks as two-layered lists: they are not easy to code efficiently and they do not scale up well to large problem instances. Moreover, in our implementations, segment trees performed somewhat worse than two-layered lists on TSPLIB instances.

2.4 ENGINEERING LIN-KERNIGHAN

With a tour data structure in hand, it is not difficult to get a working version of Chained Lin-Kernighan. There are, however, a wide variety of implementations that are consistent with the basic algorithm that we have outlined.

The decisions that must be made in an implementation can, moreover, make or break the performance of the algorithm. We discuss these design issues in this section.

To keep our presentation down to a manageable size, we will limit our computational reports to the single TSPLIB instance usa13509. In each of our tables, we give results on running specific implementations of Chained Lin-Kernighan until we reach a tour of value at most 20082519, which is within 0.5% of the optimal value for usa13509. Reports on other TSPLIB instances and other tour qualities will be presented in the next section.

Breadth of `lk_search`

In our description of `lk_search`, we included the parameters *breadth*(k), $breadth_A$, $breadth_B$, and $breadth_D$, specifying the maximum number of promising neighbors to be considered. These parameters are the principal means for controlling the breadth of the search, and it is obvious that some care must be taken in selecting their values.

In Table 2.11, we report results for several choices of *breadth*. Each row of the table summarizes 10 runs using distinct random seeds. The column labeled "Main" contains the values $(breadth(k), k = 1, \ldots, t)$, where $breadth(k) \leq 1$ for all $k > t$. The column labeled "Alternate" contains the triple ($breadth_A$, $breadth_B$, $breadth_D$). The running times are reported in seconds on the 500 Mhz Alpha workstation described in the previous section. The "Mean Steps" column gives the average number of calls to `lin_kernighan` in the 10 runs.

Table 2.11. Varying the Breadth of the Search (10 Trials)

Main	Alternate	Mean Time	Max Time	Mean Steps
(1)	(1, 1, 1)	99.9	138.0	14,659
(3)	(2, 1, 1)	40.1	51.1	5,442
(10)	(10, 1, 1)	39.4	62.8	3,854
(5, 2)	(5, 2, 1)	31.9	40.1	2,867
(4, 3, 2)	(5, 2, 1)	34.9	42.5	2,389
(10, 5)	(10, 5, 2)	41.1	73.2	2,519
(10, 5, 3)	(10, 5, 2)	41.2	55.5	1,734
(12, 8, 4, 2)	(10, 5, 2)	56.3	72.2	1,468
(4, 3, 2, 2, 2, 2, 2)	(5, 2, 2)	53.0	71.4	1,372
(8, 6, 4, 2, 2, 2)	(10, 5, 2)	75.8	97.8	1,108

The rows of Table 2.11 are ordered according to the total breadth of the search. Not surprisingly, the number of steps required is almost uniformly decreasing as we move down the table. The running times, however, favor a modest amount of backtracking, spread out over the first two or three levels of the search.

The algorithm used to obtain these results includes Mak-Morton moves, but only for levels k such that $breadth(k) = 1$. We can further manipulate the breadth of the search by either including Mak-Morton moves at all levels or by excluding them entirely. Moreover, we have the option of performing another type of move developed by Reinelt [1994]. His moves are called *vertex-insertions* since they correspond to taking a vertex from the tour and inserting it at another point in the tour. In Table 2.12, we report on a number of combinations of these moves, with *breadth* set at (5, 2) and the alternate *breadth* set at $(5, 2, 1)$.

Two things are apparent in Table 2.12. Firstly, vertex-insertion moves decrease the number of steps required, but in our implementation this is more than offset by the extra time needed to handle these moves. Secondly, Mak-Morton moves appear to be a good idea later in a search, but not at the first several levels. We must remark that Mak and Morton [1993] originally developed their moves as an alternative to using the `alternate_step` function (motivated by the complication this function adds to the implementation of `lk_search`), and thus it is not surprising that using both `alternate_step` and early Mak-Morton moves is not advantageous.

Table 2.12. Mak-Morton Moves and Node-Insertions (10 Trials)

Mak-Morton	Vertex-Insertion	Mean Time	Max Time	Mean Steps
no	no	43.7	60.2	5,329
partial	no	31.9	40.1	2,867
yes	no	42.9	62.2	2,986
no	yes	48.1	60.6	4,607
partial	yes	38.1	45.0	2,756
yes	yes	50.2	64.1	2,925

For the remaining tests in this section, we use partial Mak-Morton moves, we use no vertex-insertion moves, and we set *breadth* and the alternate *breadth* at (5, 2) and (5, 2, 1), respectively.

The Neighbor Sets

Our description of `lk_search` makes use of a prescribed set of neighbors for each vertex in the TSP. The choice of these neighbor sets directly effects the quality of the moves in `lk_search`, since we consider only flips that involve a vertex and one of its neighbors.

Rather than treating the neighbors as subsets of vertices, we can consider the *neighbor graph* consisting of the vertex set of the TSP, with edges joining vertices to their neighbors. Indeed, we define our neighbor sets in terms of this graph: if two vertices are adjacent, then we make each a neighbor of the other.

There are many choices for the neighbor graph. An obvious candidate is the *k-nearest* graph, consisting of the k least costly edges meeting each vertex. This works well on many examples, but it can cause problems on geometric instances where the points are clustered, since it does not tend to choose inter-cluster edges. To help in these cases, Miller and Pekney [1995] proposed the *k-quad-nearest* graph, defined as the k least costly edges in each of the four geometric quadrants (for 2-dimensional geometric instances) around each vertex. Miller and Pekney studied this graph in the context of 2-matching algorithms and Johnson and McGeoch [1997] have shown that it is an effective neighbor graph for Chained Lin-Kernighan.

For geometric instances, another good choice is the *Delaunay triangulation* of the point set, as described, for example, in Aurenhammer [1991], Edelsbrunner [1987], and Mehlhorn [1984]. This triangulation has been proposed as a neighbor graph in Jünger, Reinelt, and Rinaldi [1995] and Reinelt [1992, 1994]. It has the nice property of being very sparse, while still capturing well the structure of the point set.

The results reported in Table 2.11 and in Table 2.12 were obtained with the 3-quad-nearest graph. In Table 2.13, we report results for a number of other choices.

The point set for usa13509 is reasonably well distributed, and thus the k-nearest graph works well for modest choices of k. Superior results were obtained, however, using the Delaunay graph. The triangulation was computed using the computer code "sweep2" by Fortune [1994], based on the sweepline algorithm described in Fortune [1987].

Table 2.13. Choosing the Neighbor Graph (10 Trials)

Neighbor Graph	Mean Time	Max Time	Mean Steps
5-nearest	53.6	114.0	7,148
10-nearest	31.7	41.6	3,142
20-nearest	35.5	43.7	3,152
1-quad-nearest	36.0	50.1	5,392
2-quad-nearest	33.6	42.1	3,257
3-quad-nearest	31.9	40.1	2,867
4-quad-nearest	35.9	43.9	3,131
5-quad-nearest	40.4	52.7	3,266
Delaunay	26.8	35.2	3,546

In the remainder of this section, we will use the Delaunay graph to determine our neighbor sets.

Depth of `lk_search`

In `lk_search` we attempt to construct a sequence of flips that results in an improved tour. Lin and Kernighan [1973] proposed a straightforward method for ensuring that these sequences are bounded in length: they forbid flips that add edges to *current_tour* that have previously been deleted in the search, as well as forbidding flips that delete edges that have previously been added. We have incorporated this idea into our implementation, but we have found it useful to take further measures to limit the depth of the search, as we describe below.

To begin, we can use the *breadth* parameters to set a hard limit on the depth by fixing $breadth(k) = 0$ for some k. In the previous tables, our implementations had $breadth(50) = 0$; in Table 2.14, we compare this choice with several others. For our test instance, it appears that a smaller bound performs better and we will set $breadth(25) = 0$ for the remaining tables in this section.

Table 2.14. Varying the Maximum Depth (10 Trials)

Max Depth	Mean Time	Max Time	Mean Steps
5	37.6	67.5	9.110
10	28.7	37.0	5,111
25	22.1	33.9	3,093
50	26.8	35.2	3,546
100	26.5	34.8	3,248
∞	25.8	31.9	3,205

The advantage of a bounded depth search is that it prevents us from considering long sequences of flips that eventually involve vertices that are quite distant from the original *base* vertex. A particularly disturbing case of this is when we have already found an improved tour on the given search, but we continue even though

$$delta + c(base, next(base)) - c(next(base), next(probe)) \tag{2.6}$$

is less than the amount of the improvement. To handle this situation, we tighten the definition of a promising vertex by insisting that (2.6) be at least as large as any improvement we have found thus far in `lk_search`, rather than requiring only that it be nonnegative. As indicated in Table 2.15, this gives

Table 2.15. Restrictions on Promising Neighbors (10 Trials)

Restriction	Mean Time	Max Time	Mean Steps
nonnegative	25.4	34.2	3,457
max improvement	22.1	33.9	3,093

slightly better performance for the algorithm.

Vertex Marking

The number of times we call `lk_search` in a single run of `lin_kernighan` is controlled by the strategy we adopt in our Bentley-marking scheme. Recall that we have marks on our vertices; we begin searches only from marked vertices; we unmark a vertex after an unsuccessful search; and we mark the vertices that are the ends of the flips in the sequences found by successful searches. By marking additional (or fewer) vertices after a successful search, we can increase (or decrease) the number of `lk_search` calls.

In Table 2.16, we consider several possibilities: we mark either just the flip ends, the flip ends plus their adjacent vertices in *current_tour*, the flip ends plus the vertices that are at most two away from them in *current_tour*, or the flip ends plus their neighbor sets. The running times indicate that the best choice is to simply mark the flip ends as Bentley [1992] proposed. We carried out a final experiment, where we marked each of the flip ends with probability 0.5, but this performed very poorly.

Table 2.16. Marking Vertices (10 Trials)

Marks	Mean Time	Max Time	Mean Steps
flip ends	22.1	33.9	3,093
tour 1-neighbors	24.7	31.6	3,303
tour 2-neighbors	26.1	32.1	3,458
graph neighbors	28.3	42.3	3,296
flip ends (probability 0.5)	47.4	72.7	7,323

In contrast to these results, it does appear to be worthwhile to mark more than just the flip ends after applying a kicking sequence in Chained Lin-Kernighan. Our default strategy is to mark, after a kick, the flip ends as well as their neighbor sets and the vertices that are at most 10 away in T, the overall tour. A comparison of this approach with several other strategies is given in Table 2.17.

Table 2.17. Marking Nodes After Kicks (10 Trials)

Marks	Mean Time	Max Time	Mean Steps
flip ends	26.4	37.6	3,777
graph neighbors	25.0	35.0	3,280
graph & tour 5-neighbors	24.6	28.7	3,090
graph & tour 10-neighbors	22.1	33.9	3,093
graph & tour 25-neighbors	23.7	28.8	2,289
graph & tour 50-neighbors	31.3	46.9	2,548

Complementing the marking strategy, we need to determine the order in which we process the marked vertices. Two simple choices are to use a stack ("last-in-first-out") or a queue ("first-in-first-out") to store the marked vertices and thus control the processing order. Another possibility, used successfully by Rohe [1997] in a Lin-Kernighan heuristic for matching problems, is to order the marked vertices by some measure of the likelihood that a search from the vertex will be successful. The measure proposed by Rohe is to compute the nearest neighbor, $near(v)$, to each vertex v, and order the marked vertices by decreasing values of

$$c(v, next(v)) - c(v, near(v)). \tag{2.7}$$

The motivation is that vertices with high values of (2.7) appear to be out of place in the tour (they travel along an edge of cost much greater than the cost to visit their nearest neighbor). To implement this ordering, we store the marked vertices in a priority queue.

Table 2.18. Processing Marked Vertices (10 Trials)

Marks	Mean Time	Max Time	Mean Steps
stack	29.9	40.2	3,635
queue	22.1	33.9	3,093
priority queue	33.3	47.5	3,489

In Table 2.18, we compare the three different approaches for processing the vertices. Both the running times and the number of steps clearly favor the queue implementation. One factor contributing to this is that a queue will tend to distribute the searches around the tour, rather than concentrating the effort on a small tour segment where recent successes have occurred (as in the stack approach) or on a set of vertices with consistently high values of (2.7) (as in the priority queue approach).

Initial Tour

Lin and Kernighan [1973] use (pseudo) random tours to initialize their search procedure. This remains a common choice in implementations of Lin-Kernighan and Chained Lin-Kernighan. Random starting tours have the nice feature of permitting repeated calls to `lin_kernighan` without explicitly building randomization into the algorithm. It is possible, however, to initialize `lin_kernighan` with tours produced by any tour construction heuristic, and for very large examples (over a million cities) the choice can have a great impact on the performance of the algorithm. For smaller instances, however, random tours perform nearly as well as any other choice we have tested.

In Table 2.19, we report results for a number of different initial tours. "Farthest Addition" and "Spacefilling Curve" are tour construction heuristics proposed by Bentley [1992] and Bartholdi and Platzman [1982], respectively. "Greedy" is a heuristic developed by Bentley [1992] (he called it "multiple fragment" and used it in implementations of 2-opt and 3-opt; it was used in Chained Lin-Kernighan by Codenotti, Manzini, Margara, and Resta [1996]; a description of the algorithm can be found in Johnson and McGeoch [1997]). "Greedy + 2-opt" and "Greedy + 3-opt" are Greedy followed by 2-opt and 3-opt, respectively, implemented as described in Bentley [1992].

Table 2.19. Initial Tours (10 Trials)

Tour	Mean Time	Max Time	Mean Steps
Random	23.4	28.2	3,175
Nearest-Neighbor	23.4	31.4	3,369
Farthest-Addition	24.0	51.9	3,151
Spacefilling Curve	34.2	71.5	4,987
Greedy	23.6	40.7	3,331
Greedy + 2-opt	21.6	28.1	3,038
Greedy + 3-opt	22.2	32.0	3,081
Quick-Borůvka	22.1	33.9	3,093

The final line in Table 2.19 reports results for our default initial tour. This heuristic is called *Quick-Borůvka*, since it is motivated by the minimum-weight spanning tree algorithm of Borůvka [1926]. In Quick-Borůvka, we build a tour edge by edge. The construction begins (for geometric instances) by sorting the vertices of the TSP according to their first coordinate. We then process the vertices in order, skipping those vertices that already meet two edges in the partial tour we are building. To process vertex x, we add to the partial tour the least costly edge meeting x that is permissible (so we do not consider edges that meet vertices having degree 2 in the partial tour, nor edges that create subtours). This procedure can be implemented efficiently using *kd*-trees. (For a discussion of *kd*-trees, see Bentley [1992]).

Quick-Borůvka gives tours of slightly worse quality than Greedy, but it requires less time to compute and it appears to work well together with Chained Lin-Kernighan.

For further results on the initial tour selection, see Bland and Shallcross [1989], Codenotti, Manzini, Margara, and Resta [1996], Johnson [1990], Pertunen [1994], and Rohe [1997].

Kicking Strategy

A standard choice for a kicking sequence is the double-bridge kick we described in Section 2.2. This sequence was proposed in the original Chained Lin-Kernighan papers of Martin, Otto, and Felten [1991, 1992]. In their computations, Martin, Otto, and Felten generated double-bridges at random, but they used only those that involved pairs of edges of relatively small total cost.

Johnson [1990] and Johnson and McGeoch [1997] dropped this restriction on the cost of the double-bridge and simply generated them at random. An argument in favor of this later strategy is that using purely random kicks permits Chained Lin-Kernighan to alter the global shape of the tour on any iteration, whereas most of the cost-restricted kicks will tend to be local in nature and might cause the algorithm to get stuck in some undesirable global configuration. On the other hand, we can expect that `lin_kernighan` will be much faster after a restricted move than after a random move, and thus in the same amount of computing time we can perform many more iterations of the algorithm.

It is important to notice that for instances with as many vertices as usa13509, finding a cheap double-bridge by taking random samples is inefficient—most kicks will be rejected at any reasonable cut off point. To get around this, we consider below two alternative methods for obtaining cheap double-bridges. In both of our procedures, we employ a method proposed by Rohe [1997] for selecting the first edge of a kick. Like the `lk_search` method we described earlier, Rohe's idea is to start the double-bridge at a vertex v that appears to be out of place in the tour. This is accomplished by considering a small fraction of the vertices as candidates for v and choosing the one that maximizes (2.7). The first edge of the double-bridge will be $(v, next(v))$. To complete the construction, we choose the remaining three edges to be close to v, as we describe below.

Our first selection procedure examines, for some constant β, a random sample of βn vertices. We then attempt to build a double-bridge using three edges of the form $(w, next(w))$, for vertices w that are amongst the 6 nearest neighbors of v, distinct from $next(v)$, in the random sample. We call the double-bridges found by this procedure *close kicks.* Note that as we increase β, the kicks we obtain with this method are increasingly local in nature.

A second, perhaps more natural procedure, is to complete the double-bridge from edges of the form $(w, next(w))$, where w is chosen at random amongst the k vertices nearest to v. By varying k, we can get very local kicks or kicks similar to those generated purely at random. Notice, however, that these kicks are time-consuming to compute in general instances, since we would be required to examine every vertex in order to obtain the k nearest vertices. In geometric instances, however, we can use *kd*-trees to efficiently examine the nearest sets. We call the double-bridges found in this way *geometric kicks.*

In Table 2.20, we compare random, close, and geometric kicks, as well as random kicks where we use Rohe's rule for choosing the initial edge. The results indicate a clear preference for the restricted-cost kicks. We shall see in the next section, however, that for some small instances the situation is reversed.

There is no strong argument for favoring double-bridges over other kicking sequences, but in our tests they appear to work at least as well as any alternatives that we have tried. For some interesting studies of general kicks,

Table 2.20. Kicks (10 Trials)

Tour	Mean Time	Max Time	Mean Steps
random	99.6	116.3	4,056
random, long first edge	103.6	146.5	4,167
close ($\beta = 0.01$)	24.3	36.0	3,366
close ($\beta = 0.03$)	23.5	28.7	4,469
close ($\beta = 0.10$)	52.0	75.0	10,916
geometric ($k = 50$)	32.6	44.6	9,712
geometric ($k = 250$)	22.1	33.9	3,093
geometric ($k = 1,000$)	23.5	27.7	3,078

see Codenotti, Manzini, Margara, and Resta [1996] and Hong, Kahng, and Moon [1998].

Summary

We have, of course, not discussed all of the decisions that must be made in a computer implementation of Chained Lin-Kernighan. Some further details (albeit difficult to gleam) can be found in the source code to our implementation. Of the topics we have treated, the ones that appear to hold the most promise for further improvement are the choices of the *breadth* values and the choice of the kicking strategy. These subjects are discussed in more detail in Applegate, Cook, and Rohe [1999].

For other discussions of Chained Lin-Kernighan, we refer the reader to Johnson and McGeoch [1997] and Johnson, Bentley, McGeoch, and Rothberg [1999].

2.5 TSPLIB TESTS

Chained Lin-Kernighan performs very well over a wide variety of problem instances and target tour qualities. We must remark, however, that several of the design decisions we discussed in the previous section need to be altered as we go from small to large instances. In this section, we report on tests involving Reinelt's TSPLIB, and make observations on the choices of neighbor sets and kicking strategies for instances of varying size.

Quick Solutions

We begin our discussion with the modest goal of obtaining tours that are at most 1% more costly than an optimal tour. In Table 2.21, we compare three versions of our code, reporting results for each of the TSPLIB instances having at least 1,000 cities. In the first two versions of the code, we used the 3-quad-nearest graph to define our neighbor sets, but varied the kicking strategy, using random kicks in one case and geometric kicks (or close kicks

Table 2.21. 1% Optimality Gap (10 Trials)

Instance	Random	Geometric/Close	Split-Delaunay
dsj1000	0.97	0.51	0.89
pr1002	0.70	0.64	0.77
si1032	3.14	3.19	non-geometric
u1060	0.70	0.57	0.71
vm1084	0.60	0.68	0.41
pcb1173	1.03	31.04 [3]	0.55
d1291	5.07	14.07 [1]	1.62
rl1304	2.76	32.71 [3]	1.10
rl1323	0.68	0.58	0.43
nrw1379	0.67	0.39	0.42
fl1400	1.49	10.82	1.35
u1432	0.69	1.00	1.07
fl1577	0.27	0.26	0.13
d1655	4.84	26.37 [2]	4.54
vm1748	0.70	0.64	0.51
u1817	1.83	1.50	1.22
rl1889	2.65	5.88	1.70
d2103	4.93	48.93 [4]	6.12
u2152	2.22	1.96	4.52
u2319	0.46	0.42	0.30
pr2392	1.90	2.22	2.73
pcb3038	2.58	1.67	2.42
fl3795	28.67 [2]	38.10 [3]	10.48
fnl4461	2.84	1.67	3.17
rl5915	14.45	15.17	5.33
rl5934	6.59	3.42	6.05
pla7397	6.25	14.88 [1]	6.71
rl11849	30.13	12.96	9.45
usa13509	23.29	12.94	7.74
brd14051	21.95	7.89	7.10
d15112	25.19	10.77	8.35
d18512	26.02	10.96	8.17
pla33810	56.66 [1]	32.11	28.83
pla85900	78.08 [1]	40.17	34.38

for non-geometric instances) in the other. The table entries give the average CPU time (using the 500 Mhz Alpha 21164a workstation) over 10 distinct runs of Chained Lin-Kernighan. For these tests we set a CPU limit of 100 seconds; an entry followed by "[k]" means that k of the trials failed to produce an acceptable tour in the allotted time.

As we mentioned in the previous section, a number of the smaller instances do not behave well with the local kicks provided by the geometric or close procedures. Indeed, on the instances having less than 10,000 cities, we reached the time limit in 17 cases when using geometric kicks, while only 2 cases failed when using random kicks. On the other hand, the average running time for each of the larger instances was significantly better when using geometric kicks. We therefore propose a hybrid method where we use random kicks on instances having less than 10,000 cities, but we switch to geometric kicks on instances having more than 10,000 cities. We report on this approach in the third set of results in Table 2.21, using the Delaunay triangulation as our nearest-neighbor graph. (There is no entry for si1032 since this is not a Euclidean example.) We recorded no failures in this third set of tests, and, for the larger instances, the average running times were slightly better than the best of the 3-quad-nearest results.

In Figure 2.12, we plot the running times for the entire set of 110 TSPLIB instances. Each of the marks in the figure represents the mean CPU time over 10 independent runs of Chained Lin-Kernighan. In these tests, we adopted the hybrid random/geometric approach, and used the Delaunay graph for Euclidean instances, the 3-quad-nearest graph for non-Euclidean geometric instances, and the 12-nearest graph for non-geometric instances. The target tour was obtained in less than 0.1 seconds for all instances having at most 200 cities and, with the exception of the instance a280, the target tour was obtained in under 1.0 seconds for all instances having at most 1,000 cities.

High Quality Solutions

The above discussion points out that Chained Lin-Kernighan is a suitable method for obtaining reasonable quality tours in small amounts of CPU time. The real strength of the algorithm, however, is that is can effectively use larger amounts of CPU time to obtain much higher quality results. This feature of Chained Lin-Kernighan distinguishes it from other tour-finding procedures that have been proposed thus far in the literature.

In Table 2.22, we report on 10 longer runs for the three TSPLIB instances pcb3038, usa13509, and pla85900. These examples are representative of the larger instances in the library. We set time limits for the tests at 250 seconds for pcb3038, 1,000 seconds for usa13509, and 5,000 seconds for pla85900. These limits correspond to roughly 100 times the CPU time that is needed to obtain 1%-quality tours for each of the instances. The average costs of the tours obtained were within 0.17% of the optimal value for pcb3038, within 0.18% of the optimal value for usa13509, and within 0.30% of a known lower

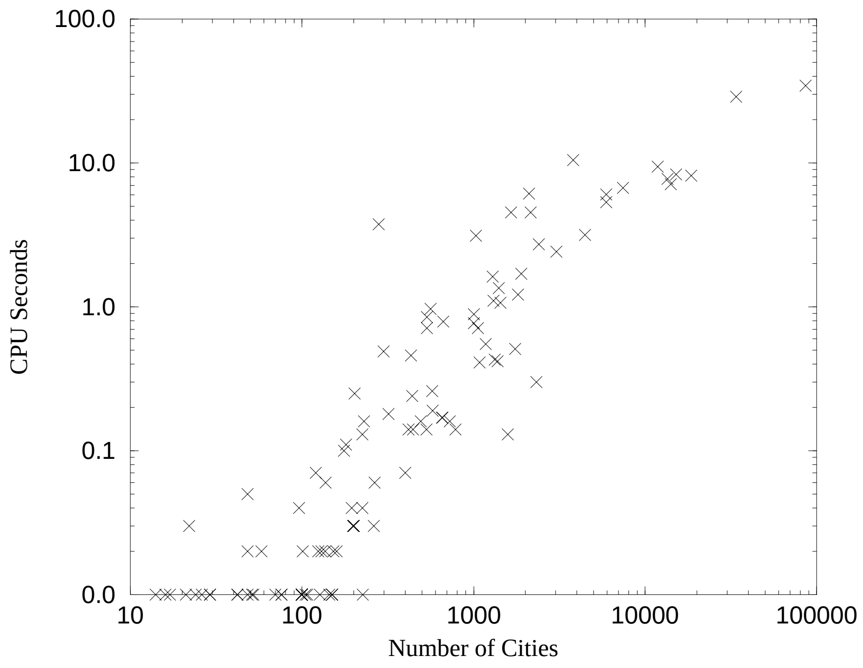


Figure 2.12. 1% of Optimality for TSPLIB

bound for pla85900. If we take the best of the 10 tours for each of the instances, then the costs are within 0.09% for pcb3038, 0.16% for usa13509, and 0.26% for pla85900. (The degraded tour quality for pla85900 may be due, in part, to the weakness of the lower bound.)

Table 2.22. Longer Runs (10 Trials)

Instance	CPU Seconds	Mean Cost	Max Cost	Min Cost
pcb3038	250	137932	138036	137815
usa13509	1,000	20019576	20032653	20014105
pla85900	5,000	142675621	142744967	142608512

To exhibit how the quality of the tour improves over time, in Figure 2.13, we plot the cost of the tour against the number of CPU seconds used by Chained Lin-Kernighan, over a single run of pla85900. The curve shown in the figure is typical for runs on TSPLIB instances: we get sudden jumps in the tour quality when a kicking sequence permits a global change in the tour structure, and these are followed by a number of smaller improvements as the kicks provide additional local optimization. At the conclusion of the 50,000 second run, the tour obtained was within 0.19% of the lower bound.

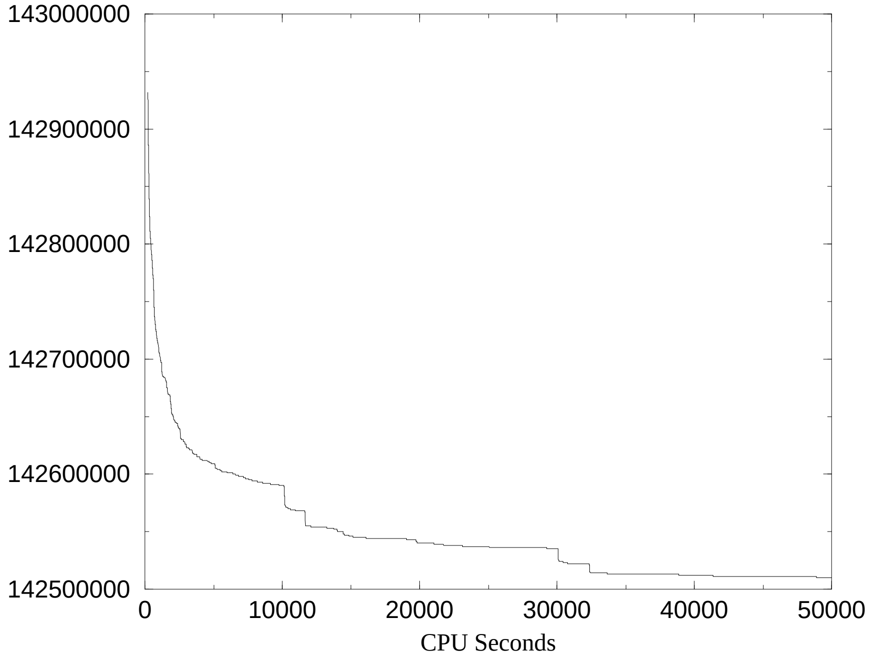


Figure 2.13. Chained Lin-Kernighan for pla85900

Optimal Solutions

One of the points stressed by Lin and Kernighan [1973] is that, in many cases, their algorithm can be adopted to find optimal TSP solutions. Indeed, in the abstract to their classic paper, the authors write: “The procedure produces optimum solutions for all problems tested, ‘classical’ problems appearing in the literature, as well as randomly generated test problems, up to 110 cities.” Their results were obtained using the “Repeated Lin-Kernighan” approach of making independent runs on a given problem instance, and selecting the best of the tours. We have of course argued that this process is inferior to Chained Lin-Kernighan's approach of iterating the calls to `lin_kernighan`. But if we examine the cost curve given in Figure 2.13, it is clear that with our implementation of Chained Lin-Kernighan, there comes a point where additional running time is unlikely to lead to significant further improvements in the quality of the tour produced. Johnson [1990] and Johnson and McGeoch [1997] reached this same conclusion with their implementations; they proposed to go back to making independent runs, but this time using Chained Lin-Kernighan as the core algorithm. In Table 2.22, we have seen examples of how this technique can lead to improved tours, and we will now discuss this in detail.

Let us begin with the very small examples from TSPLIB. For each of the 49 instances having at most 200 cities, we made 10 independent runs of Chained Lin-Kernighan, with the optimal values as the target tours and with a CPU limit of 1 second. We performed the test three times, using different selections for the neighbor graph. In the first test, we used the Delaunay graph for Euclidean instances, the 3-quad-nearest graph for non-Euclidean geometric instances, and the 12-nearest graph for non-geometric instances. In the second test, we used the 3-quad-nearest graph for all geometric instances and the 12-nearest graph for non-geometric instances. In the final test, we used the 5-quad-nearest graph for geometric instances and the 20-nearest graph for non-geometric instances. The number of times the codes failed to find optimal tours were

Delaunay	3-Quad Nearest	5-Quad Nearest
20	33	48

for the sets of 490 test runs. The failures were spread across a number of instances, with every example being solved at least 3 times in the Delaunay runs and at least 3 times in the 3-quad-nearest runs. In the 5-quad-nearest runs, the instance d198 was not solved in any of the 10 trials. This instance was also the worst example in both the Delaunay and 3-quad-nearest tests, so we choose it to discuss in greater detail.

In Table 2.23, we report on a series of additional tests on d198. In these runs, we varied the CPU limit from 0.1 seconds up to 10 seconds. For each value, we made 1,000 independent trials of Chained Lin-Kernighan, recording the mean CPU times between occurrences of optimal solutions. These times are reported in seconds in Table 2.23. On this instance, the Delaunay graph performed better than the (denser) 3-quad-nearest graph. Using the Delaunay graph, the number of times we actually found optimal solutions in the 1,000 trials ranged from 13, with the 0.1 second limit, up to 967, when we allotted 10 seconds for the runs. With the 3-quad-nearest, we were unable to obtain an optimal solution when we limited the runs to 0.1 seconds, but with a 10 second limit we achieved the optimal value in 730 of the trials.

Table 2.23. Optimal Solutions for d198 (1,000 Trials)

Time Limit	Time per Tour (Delaunay)	Time per Tour (3-Quad)
0.1	7.68	NO TOUR
0.2	3.12	19.98
0.5	1.97	8.76
1.0	1.97	5.94
2.0	2.19	5.68
5.0	2.51	6.76
10.0	2.64	7.19

Table 2.24. Optimal Solutions with 10 Second Limit (1,000 Trials)

Instance	Time per Tour (Delaunay)	Time per Tour (Quad-3)
gr202	non-Euclidean	3.69
ts225	0.69	0.14
tsp225	0.24	0.35
pr226	0.08	0.98
gr229	non-Euclidean	17.25
gil262	1.37	0.80
pr264	0.27	0.56
a280	93.83	0.17
pr299	33.84	3.46
lin318	11.11	7.92
rd400	42.28	75.04
fl417	6.69	14.00
gr431	non-Euclidean	74.72
pr439	6.69	4.26
pcb442	7.96	6.93
d493	179.23	245.68
att532	non-Euclidean	95.03
ali535	non-Euclidean	44.02
si535	non-geometric	1427.14
pa561	non-geometric	44.89
u574	17.23	28.33
rat575	829.64	551.97
p654	13.78	21.79
d657	380.73	117.37
gr666	non-Euclidean	1425.67
u724	318.45	412.64
rat783	18.15	19.43

We consider next the remaining TSPLIB instances having less than 1,000 cities. These examples range in size from 202 cities up to 783 cities, including 27 instances in total. For examples in this range, it becomes difficult to produce optimal solutions using the straight version of Lin-Kernighan. For example, Johnson [1990] reported that a good implementation of Lin-Kernighan was unable to find an optimal solution to att532 in 20,000 independent trials. Using the extra power of Chain Lin-Kernighan, however, Johnson [1990] and Johnson and McGeoch [1997] report that optimal solutions for a number of the examples could be found in reasonable amounts of CPU time, including the instances lin318, pcb442, and att532.

In our tests on these mid-sized instances, we set a 10 second time limit on the individual calls to Chained Lin-Kernighan. We carried out 1,000 independent runs on each instance, using both the Delaunay and 3-quad-nearest graphs. In Table 2.24, we again report the mean time between occurrences of optimal solutions. Although we could indeed find optimal solutions for each of the test instances, we need to point out that in a number of cases the required CPU times are far above those required for solving the given instances with our linear programming based branch-and-bound code.

In several of the cases requiring large amounts of computation, the problem actually lies with the choice of the neighbor graph. For these instances, to keep the algorithm from frequently getting trapped in nonoptimal solutions, we need a broader collection of neighbors than those provided by our relatively sparse graphs. For example, with the non-geometric instance si535, if we switch from using the 12-nearest graph to using the 40-nearest graph, then the average time to find an optimal tour drops from 1427.14 seconds down to 185.81 seconds.

In other cases, such as gr666, the difficulty appears to be more fundamental. Increasing the size of the neighbor sets in gr666 does not improve the time needed for obtaining optimal solutions. We will come back to this example in the next section, when we discuss a more robust procedure for using multiple runs of Chained Lin-Kernighan.

We now move on to the TSPLIB instances in the range of 1,000 cities up to 2,000 cities. In Table 2.25, we report results for this set of 17 instances, running 100 trials of Chained Lin-Kernighan with a time limit of 100 seconds. As before, we report the number of seconds between occurrences of optimal tours. Unlike the tests on small instances, however, in a number of these runs we were not successful in producing the optimum. Indeed, we failed in 5 cases when we used the Delaunay graph and in 4 cases when we used the 3-quad-nearest graph. Moreover, increasing the density of the neighbor sets by moving to the 10-quad-nearest graph only made matters worse—the number of failures went up to 6 out of the 17 instances. Nonetheless, it is interesting that such a straightforward use of Chained Lin-Kernighan can produce optimal solutions on more than three-quarters of the instances of this size.

Table 2.25. Optimal Solutions with 100 Second Limit (100 Trials)

Instance	Time per Tour (Delaunay)	Time per Tour (3-Quad)
dsj1000	2457.30	4960.29
pr1002	98.65	60.53
si1032	non-geometric	26.79
u1060	2457.01	2467.15
vm1084	947.04	463.87
pcb1173	367.34	1621.58
d1291	397.37	1198.94
rl1304	182.83	704.50
rl1323	2453.77	4936.07
nrw1379	NO TOUR	9974.98
fl1400	75.88	49.67
u1432	1964.72	663.03
fl1577	NO TOUR	NO TOUR
d1655	441.85	1077.62
vm1748	9985.21	9980.64
u1817	NO TOUR	NO TOUR
rl1889	NO TOUR	NO TOUR

When we proceed to even larger instances, we encounter even more failures, naturally. In Table 2.26, we report on the remaining instances having less than 4,000 cities. We did not include d2103 in this set of tests, since the optimal

Table 2.26. Optimal Solutions with 250 Second Limit (100 Trials)

Instance	Time per Tour (3-Quad)
u2152	24798.65
u2319	NO TOUR
pr2392	3082.64
pcb3038	NO TOUR
fl3795	NO TOUR

value for this instance has not (as of this writing) been determined. Using 100 trials with the 3-quad-nearest graph, and with a 250 second time limit, we found optimal solutions for 2 of the 5 instances.

We remark that, using ad hoc methods, it is possible to coax Chained Lin-Kernighan into producing optimal tours for those instances in Tables 2.25 and 2.26 where our multiple short runs failed. (For example, increasing the *breadth* parameters, using longer runs, using Martin, Otto, and Felten's simulated annealing-like approach for accepting tours, etc.) We were not able to do this, however, on any of the examples having greater than 4,000 cities. For each of these larger instances, the optimal solutions (when known) were found either by a branch-and-bound search or by the method we present in the next section.

2.6 TOUR MERGING

A valid criticism of the multiple-run Chained Lin-Kernighan procedure we described at the end of the previous section is that in choosing only the best of a large collection of tours, we may be discarding a great deal of valuable information. A challenge is to utilize the combined content of the tours to produce a final tour that is superior to any single one in the collection. This is the same task that confronts designers of genetic algorithms for the TSP, as described, for example, in Oliver, Smith, and Holland [1987], Mühlenbein, Gorges-Schleuter, and Krämer [1988], and Ulder, Aarts, Bandelt, Laarhoven, and Pesch [1991]. In our context, however, the quality of the population of tours is such that the simple randomized techniques used in successful genetic algorithms are very unlikely to produce tour improvements. Instead, we propose a *tour-merging* procedure that attempts to produce the best possible combination of tours by solving a TSP restricted to the edges that are present in at least one tour in the collection.

To illustrate our procedure, consider the 1577-city TSPLIB instance fl1577. In Table 2.25 we reported that 100 trials of Chained Lin-Kernighan with a time limit of 100 seconds failed to produce the optimal fl1577-tour on even a single occasion. This instance was the smallest for which we were unable to obtain an optimal tour in our 100-trial tests. It is, however, possible to find an optimal tour with the process we describe in this section. For example, making 25 runs of Chained Lin-Kernighan with a 10-second time limit, we assemble the graph represented in Figure 2.14. The edge set of this graph is the union of the 25 edge sets of the tours; the graph has 2391 edges in total. Solving a TSP restricted to such a sparse edge set can be much simpler than solving the original problem. Indeed, using our linear-programming-based TSP code (see Applegate, Bixby, Chvátal, and Cook [1998]), the optimal solution to the restricted problem was found in 469.8 seconds on a 500 Mhz Alpha 21164a processor workstation. In this example, moreover, the tour produced is also optimal for the original TSP instance. The total time to obtain this optimal solution was approximately 12 minutes.

Tour-merging is a powerful method for obtaining tours of quality exceeding that of the standard multiple-run Chained Lin-Kernighan procedure. It is not only an effective way of expending large amounts of computational resources to obtain the best quality tours, but also a very competitive procedure for modest length computational runs on a wide range of problem instances. As we shall describe, however, care needs to be taken so that the solve times of the sparse TSPs do not become unacceptably long.

We begin our discussion with the collection of TSPLIB instances having less than 1,000 cities. From this set, all instances having less than 200 cities are

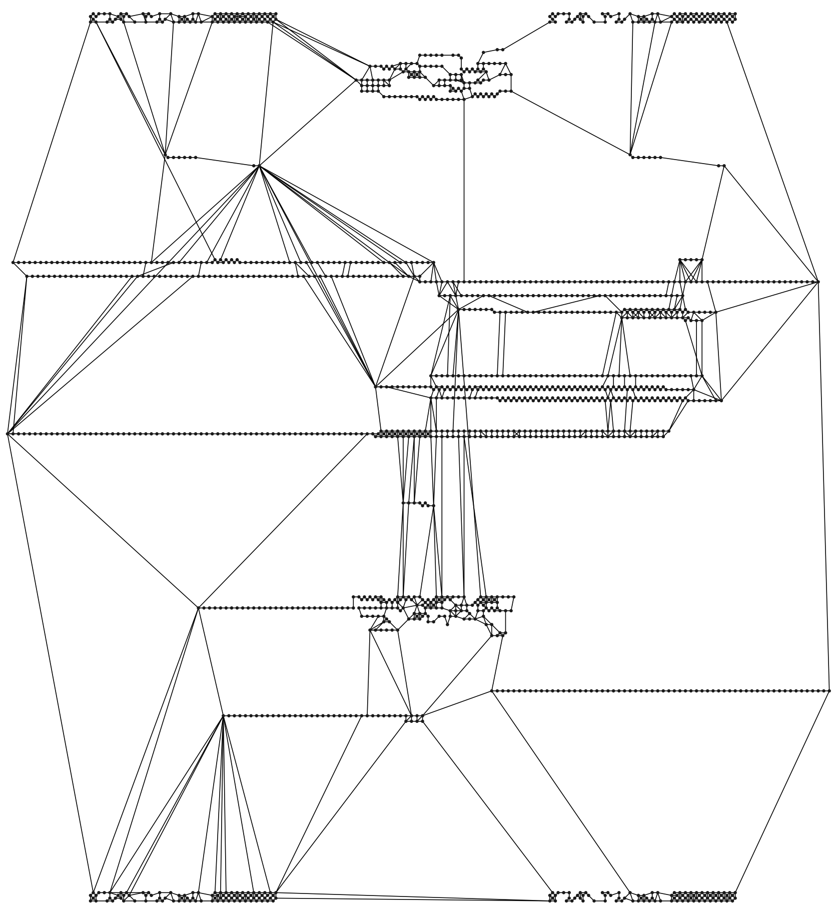


Figure 2.14. Union of 25 Chained Lin-Kernighan Tours for fl1577

solved handily with Chained Lin-Kernighan, so we will not consider further these instances. In Table 2.24, we reported results for the remaining examples using multiple 10-second runs of Chained Lin-Kernighan. Of the 27 instances considered in Table 2.24, only 5 required an average time of greater than 2 minutes to find optimal solutions. The average times for these 5 most difficult instances ranged from 245 seconds for d493 to 1,427 seconds for si535. We will use these 5 examples as our initial testbed for tour merging.

For each of the 5 instances, we made 10 trials of merging 25 tours found with Chained Lin-Kernighan, giving each Chained Lin-Kernighan run a 2.5 second time limit. In these tests, we used the 3-quad-nearest graph to determine the neighbor sets for Chained Lin-Kernighan, and we used the nearest-neighbor algorithm to produce the starting tours. The results are reported in Table 2.27 (again using a 500 Mhz Alpha 21164a workstation). The column labeled "LK-

Table 2.27. Tour Merging with 25 Tours and 2.5 Second Limit (10 Trials)

Instance	LK-quality	Merge-quality	LK-opt	Merge-opt	Time (seconds)
d493	1.00039	1.00000	0	10	167.76
si535	1.00120	1.00007	0	5	85.02
rat575	1.00025	1.00009	1	4	125.38
gr666	1.00061	1.00006	0	8	97.90
u724	1.00044	1.00000	0	10	135.37

quality" gives the average ratio of the cost of the best of the 25 Chained Lin-Kernighan tours in each trial, to the cost of an optimal tour; the column labeled "Merge-quality" gives the average ratio of the cost of the final merged tour for each trial, to the cost of an optimal tour; "LK-opt" reports the number of trials for which the best Chained Lin-Kernighan tour was optimal, and "Merge-opt" reports the number of times the final merged tour was optimal.

In 37 of the 50 trials the merged tour was indeed optimal, and the average tour quality over all trials was 1.00004, that is, on average the tours produced were within 0.004% of the optimal values. To compare these results with those given in Table 2.24, note that the average time between appearances of optimal solutions ranged from 135 seconds for u724 to 313 seconds for rat575.

In the above tests, the sparse TSP instances were solved in an average time of under 50 seconds with the linear-programming-based code. Although these solve times are quite acceptable, in many cases alternative methods can treat the sparse problems more effectively than the general purpose code of Applegate, Bixby, Chvátal, and Cook [1998]. One attractive alternative is to use the *branch-width* graph invariant proposed by Robertson and Seymour [1991] to solve the sparse TSPs via dynamic programming. This idea is treated in Cook and Seymour [1999].

The combination of $N = 25$ tours and a time limit of $T = 2.5$ seconds on Chained Lin-Kernighan gives a reasonable compromise between tour quality and total running time for the under 1000-city examples. In Table 2.28 we compare these results (over the 50 total trials), with the results for several other combinations of N and T. Note that the average running time for

Table 2.28. Tour merging with 5 instances having less than 1000 cities

Number of Tours	Time Limit (seconds)	Tour Quality	Time (seconds)
5	5	1.00042	39.24
10	2.5	1.00045	54.92
10	5	1.00014	74.36
25	1	1.00017	137.04
25	2.5	1.00004	122.29
25	5	1.00003	170.20

$N = 25$ and $T = 1$ is higher than the time for $N = 25$ and $T = 2.5$, despite the much lower time for computing the collection of tours. The explanation is that the lower quality Chained Lin-Kernighan tours lead to a lessor number of repeated edges in the tour population, and therefore a denser (more difficult) TSP instance to be solved.

Let us now consider the TSPLIB instances in the range of 1,000 cities up to 2,000 cities. The results for multiple 100-second Chained Lin-Kernighan runs for this class of 17 examples is presented in Table 2.25. The three instances fl1577, u1817, and rl1889 were not solved in the 100 trials considered in those tests, while the average running times for the remaining instances varied from 27 seconds up to 9980 seconds (to obtain optimal solutions). In our tour-merging tests, we will treat the 13 instances that required greater than 600 seconds, including the 3 examples that were not solved. The results for this problem set, using $N = 25$ and $T = 10$, are given in Table 2.29. With these settings, optimal solutions were found in 95 of the 130 trials, and every instance was solved at least 2 times. The running times are modest, with the exception of u1817, which required an average of nearly 11,000 seconds to solve the sparse TSPs. A direct precaution against such behavior is to set an upper bound on the time allowed in the TSP solver, reporting a failure if the bound is reached. For example, setting a bound of 5,000 seconds lowers the average running time to 4451.83 seconds for u1817, while still reaching the optimal solution on 5 of the 10 trials.

In Table 2.30 we compare several different choices of N and T for the set of 13 instances considered in Table 2.29. The values reported are the averages over all 130 trials. The quality of the final tour drops quickly as we decrease the size of the tour population, but there is a corresponding decrease in the time required to solve the resulting sparse TSP instances.

Proceeding to larger examples, let us again consider the 5 TSPLIB instances under 4,000 cities that we treated in Table 2.26. These are all TSPLIB instances in the range of 2,000 to 4,000 cities, with the exception of d2103 (the

Table 2.29. Tour Merging with 25 Tours and 10 Second Limit (10 Trials)

Instance	LK-quality	Merge-quality	LK-opt	Merge-opt	Time (seconds)
dsj1000	1.00094	1.00032	0	4	356.14
u1060	1.00070	1.00000	0	10	519.00
pcb1173	1.00013	1.00000	4	8	385.89
d1291	1.00091	1.00000	0	10	415.38
rl1304	1.00143	1.00038	0	7	263.78
rl1323	1.00114	1.00011	0	6	324.58
nrw1379	1.00046	1.00000	0	8	458.37
u1432	1.00099	1.00000	0	10	1337.15
fl1577	1.00123	1.00021	0	2	633.36
d1655	1.00102	1.00002	0	8	450.12
vm1748	1.00057	1.00000	0	10	347.71
u1817	1.00253	1.00017	0	7	11231.06
rl1889	1.00214	1.00002	0	5	451.42

Table 2.30. Tour merging with 13 instances having between than 1000 and 2000 cities

Number of Tours	Time Limit (seconds)	Tour Quality	Time (seconds)
5	50	1.00098	339.44
10	10	1.00045	422.25
10	25	1.00045	476.22
25	10	1.00009	1321.07

optimal value for d2103 has not yet been determined). In the tests reported in Table 2.26, optimal solutions were found only for instances u2152 and pr2392, requiring an average of 24,799 seconds and 3,083 seconds, respectively. In Table 2.31, we report tour-merging results for 10 trials on each of the 5 test instances, using $N = 25$ and $T = 10$. Optimal solutions were obtained in 41 of

Table 2.31. Tour Merging with 25 Tours and 10 Second Limit (10 Trials)

Instance	LK-quality	Merge-quality	LK-opt	Merge-opt	Time (seconds)
u2152	1.00199	1.00000	0	10	6026.40
u2319	1.00172	1.00000	0	10	28097.92
pr2392	1.00124	1.00000	0	10	303.68
pcb3038	1.00190	1.00001	0	8	16196.12
fl3795	1.00592	1.00031	0	3	4808.10

the 50 trials, and each example was solved at least 3 times. This is certainly an improvement over our multiple Chained Lin-Kernighan runs, but the running times for two of the instances, u2319 and pcb3038, exceeded 10,000 seconds. The u2319 example can be solved easily using other methods (for example, the branch-width technique we mentioned briefly above), so we select pcb3038 for further discussion.

In Table 2.32 we report results of 10 trials on pcb3038 using a variety of settings for N and T. These results illustrate several common properties of

Table 2.32. Tour Merging on pcb3038 (10 Trials)

Settings	LK-quality	Merge-quality	Merge-opt	Time (seconds)
$N = 25,\ T = 10$	1.00190	1.00001	8	16196.12
$N = 25,\ T = 50$	1.00103	1.00001	7	10042.43
$N = 25,\ T = 100$	1.00096	1.00002	5	7006.48
$N = 25,\ T = 250$	1.00071	1.00000	10	11318.06
$N = 25,\ T = 500$	1.00063	1.00002	7	16145.61
$N = 25,\ T = 1000$	1.00059	1.00000	9	28781.31
$N = 10,\ T = 100$	1.00145	1.00024	0	3631.65
$N = 10,\ T = 250$	1.00090	1.00009	1	4029.43
$N = 10,\ T = 500$	1.00095	1.00008	3	6526.59
$N = 10,\ T = 1000$	1.00059	1.00008	3	10963.02

tour-merging runs. Firstly, the quality of the final tour depends heavily on the size of the population of tours, but only to a lessor degree on the time limit given to Chained Lin-Kernighan (as long as the limit is reasonably large). Secondly, the solution time for the sparse TSP decreases dramatically as the quality of the tours in the population is improved, that is, for fixed N, the sparse TSP time decreases as T is increased. To select the best settings of N and T, we must trade off the time used to produce the tour population versus the time needed to solve the sparse TSPs. This is a difficult task on large instances, since the performance of the linear-programming-based code is difficult to predict without running experiments.

To apply tour merging to even larger instances, we need to take great care in the selection of the tour population. For example, 10 trials on the instance fnl4461 produced the following results

LK-quality	Merge-quality	LK-opt	Merge-opt	Time
1.00078	1.00000	0	10	85920.98

using $N = 25$ and $T = 100$. The fact that the optimal tour was obtained in each of the trials is a very positive sign, but the average running time for the sparse TSPs is very high (the full instance was solved in less than 150,000 seconds by Applegate, Bixby, Chvátal, and Cook [1998]). It possible, however, to use Chained Lin-Kernighan to obtain tour populations that are much more likely to be amenable to sparse TSP solution routines. What is needed is more homogeneous collection of tours. To achieve this, rather than using the nearest-neighbor algorithm to produce different starting tours for each run, we can start the Chained Lin-Kernighan runs from a common starting tour that is itself the result of a run of Chained Lin-Kernighan. This adds a third setting for our runs, namely the number of seconds S used in the initial run to build the common starting tour.

In Table 2.33 we report results for fnl4461, where we fix N and T, and permit S to vary between 5, 10, and 25. As one would expect, as we increase

Table 2.33. Tour Merging on fnl4461 with Fixed Start (10 Trials)

Settings	LK-quality	Merge-quality	Time (seconds)
$N = 25,\ T = 100,\ S = 5$	1.00099	1.00065	3911.41
$N = 25,\ T = 100,\ S = 10$	1.00106	1.00074	2828.08
$N = 25,\ T = 100,\ S = 25$	1.00111	1.00087	2690.53
$N = 25,\ T = 50,\ S = 5$	1.00128	1.00076	3875.04
$N = 25,\ T = 50,\ S = 10$	1.00121	1.00082	1681.37
$N = 25,\ T = 50,\ S = 25$	1.00123	1.00092	1352.63

the quality of the common starting tour, the time needed to solve the sparse TSP drops, but at the expense of lowering the average quality of the final merged tour.

The added flexibility of using a high quality common starting allows us to apply tour-merging to a very wide range of instances. To illustrate this, we consider pla85000, the largest instance in the TSPLIB. On an instance of this size, there is a large variance in the quality of tours produced by individual runs of Chained Lin-Kernighan, so before we invest the time to obtain an entire collection of tours, it is important to make an effort to have a common starting tour that is likely to lead to a high quality population. To achieve this, we can use a short multiple Chained Lin-Kernighan run to obtain the common tour, rather than simply using a single run. In our case, making 5 runs of Chained Lin-Kernighan, with a time limit of 1,000 seconds, we obtain tours of the following costs

$$142880000, 142744923, 142732749, 142754147, 142866379.$$

Now, making 5 further Chained Lin-Kernighan runs using the best of these (142732749) as a starting tour, and using a 5,000 second time limit, we obtain the population

$$142628651, 142603285, 142613880, 142625393, 142624904.$$

With this collection, the merged tour can be computed in 153,034 additional seconds, resulting in a tour of cost 142575637. The TSP solve time is extremely high due to the fact that 85,900 cities is well beyond the size of instances targeted by the Applegate, Bixby, Chvátal, and Cook [1998] code, but this example does indicate that tour-merging is a viable approach, even for instances of this size.

Tour-merging can be used in a wide variety of ways, and it is likely that ideas drawn from the field of genetic algorithms can be combined with tour merging to produce a powerful class of heuristics. On large examples, our use of the procedure has been confined primarily to ad hoc methods, but, as we report in the next section, we have nonetheless been able to obtain the best currently known results for each of the unsolved TSPLIB instances. It would certainly appear that tour merging is an area that holds promise for further improvement in tour finding algorithms.

2.7 TSPLIB TOURS

In Table 2.34, we report the costs of the cheapest tours we have found, with our heuristic methods, for the instances in TSPLIB that were unsolved as of 1994. The tours for four of the instances have been shown to be optimal, and the tour for usa13509, although not optimal, is only 0.0002% more costly than the optimal tour. In each of the other cases, the tours give the best values that have been reported to G. Reinelt [1999].

In each instance, the tour was obtained by using Chained Lin-Kernighan to produce a collection of tours, using tour merging to refine the collection, and repeating the process.

Table 2.34. Unsolved TSPLIB Problems

Name	Size	Tour Cost	Lower Bound	Gap
fl1577	1,577	22249	22249	OPTIMAL
d2103	2,103	80450	80330	0.15%
fl3795	3,795	28772	28772	OPTIMAL
rl5915	5,915	565530	565530	OPTIMAL
rl5934	5,934	556045	556045	OPTIMAL
rl11849	11,849	923368	923132	0.03%
usa13509	13,509	19982889	19982859	0.00%
brd14051	14,051	469445	469272	0.04%
d15112	15,112	1573152	1572863	0.02%
d18512	18,512	645300	645198	0.02%
pla33810	33,810	66116530	65970716	0.22%
pla85900	85,900	142473573	142253424	0.16%

The lower bounds were computed using our linear-programming-based code for the TSP, as described in Applegate, Bixby, Chvátal, and Cook [1998]. Again, in each case the lower bounds are the best bounds that have been reported to G. Reinelt [1999].

Bibliography

D. Applegate, V. Chvátal, and W. Cook (1990). "Lower bounds for the travelling salesman problem", in: "TSP '90", *CRPC Technical Report* CRPC-TR90547, Center for Research in Parallel Computing, Rice University.

D. Applegate, R. Bixby, V. Chvátal, and W. Cook (1998), "On the solution of traveling salesman problems", *Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung, International Congress of Mathematicians*, 645–656.

D. Applegate, W. Cook, and A. Rohe (1999). "Chained Lin-Kernighan for large traveling salesman problems", in preparation.

F. Aurenhammer (1991). "Voronoi diagrams: a survey of a fundamental data structure", *ACM Computational Surveys* **23**, 345–405

A. Bachem and M. Wottawa (1992). "Parallelisierung von heuristiken für grosse traveling-salesman-probleme", *Report Number* 92.119, Mathematisches Institut, Universität zu Köln.

J. J. Bartholdi III and L. K. Platzman (1982). "An O(NlogN) planar travelling salesman heuristic based on spacefilling curves", *Operations Research Letters* **1**, 121–125.

J. L. Bentley (1990). "Experiments on traveling salesman heuristics", in: *First Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, pp. 91–99.

J. L. Bentley (1992). "Fast algorithms for geometric traveling salesman problems", *ORSA Journal on Computing* **4**, 387–411.

R. G. Bland and D. F. Shallcross (1989). "Large traveling salesman problems arising from experiments in X-ray crystallography: a preliminary report on computation", *Operations Research Letters* **8**, 125–128.

F. Bock (1958). Research Report, Armour Research Foundation. (Presented at the Operations Research Society of America Fourteenth National Meeting, St. Louis, October 24, 1958.)

O. Borůvka (1926). "On a certain minimal problem" (in Czech), *Práce Moravské Přírodovědecké Společnosti* **3**, 37–58.

N. Christofides (1976). "Worst-case analysis of a new heuristic for the travelling salesman problem", *Report Number* 388, Graduate School of Industrial Administration, Carnegie Mellon University.

M. Chrobak, T. Szymacha, and A. Krawczyk (1990). "A data structure useful for finding Hamiltonian cycles", *Theoretical Computer Science* **71**, 419–424.

B. Codenotti, G. Manzini, L. Margara, and G. Resta (1996). "Perturbation: an efficient technique for the solution of very large instances of the Euclidean TSP", *INFORMS Journal on Computing* **8**, 125–133.

W. Cook and P. D. Seymour (1999). "A branch-decomposition heuristic for the TSP", in preparation.

G. A. Croes (1958). "A method for solving traveling-salesman problems", *Operations Research* **5**, 791–812.

H. Edelsbrunner (1987). *Algorithms in Combinatorial Geometry*, Springer, Heidelberg.

M. M. Flood (1956). "The traveling-salesman problem", *Operations Research* **4**, 61–75.

S. J. Fortune (1987). "A sweepline algorithm for Voronoi diagrams", *Algorithmica*, 153–174.

S. J. Fortune (1994). "sweep2", computer code available via anonymous ftp from `netlib.att.com`.

M. L. Fredman, D. S. Johnson, L. A. McGeoch, and G. Ostheimer (1995). "Data structures for traveling salesmen", *Journal of Algorithms* **18**, 432–479.

M. Grötschel and O. Holland (1991). "Solution of large-scale symmetric travelling salesman problems", *Mathematical Programming* **51**, 141–202.

I. Hong, A. B. Kahng, and B. Moon (1998). "Improved large-step markov chain variants for the symmetric TSP", to appear in *Journal of Heuristics.*

B. W. Kernighan and D. M. Ritchie (1978). *The C Programming Language*, Prentice Hall, Englewood Cliffs.

D. S. Johnson (1990). "Local optimization and the traveling salesman problem", in: *Proceedings 17th Colloquium of Automata, Languages, and Programming*, Lecture Notes in Computer Science, Volume 443, Springer-Verlag, Berlin, pp. 446–461.

D. S. Johnson, J. L. Bentley, L. A. McGeoch, and E. E. Rothberg (1999). In preparation.

D. S. Johnson and L. A. McGeoch (1997). "The traveling salesman problem: a case study in local optimization", in: *Local Search in Combinatorial Optimization* (E. H. L. Aarts and J. K. Lenstra, eds.), Wiley, New York, pp. 215–310.

M. Jünger, G. Reinelt, and G. Rinaldi (1995). "The traveling salesman problem", in: *Handbook on Operations Research and Management Sciences: Networks* (M. Ball, T. Magnanti, C. L. Monma, and G. Nemhauser, eds.), North-Holland, pp. 225–330.

D. E. Knuth (1973). *Sorting and Searching, The Art of Computer Programming, Volume 3*, Addison Wesley, Reading.

E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, eds. (1985). *The Traveling Salesman Problem*, Wiley, Chichester.

S. Lin (1965). "Computer solutions of the traveling salesman problem", *The Bell System Technical Journal* **44**, 2245–2269.

S. Lin and B. W. Kernighan (1973). "An effective heuristic algorithm for the traveling-salesman problem", *Operations Research* **21**, 498–516.

K.-T. Mak and A. J. Morton (1993). "A modified Lin-Kernighan traveling-salesman heuristic", *Operations Research Letters* **13**, 127–132.

F. Margot (1992). "Quick updates for p-opt TSP heuristics", *Operations Research Letters* **11**, 45–46.

O. Martin, S. W. Otto, and E. W. Felten (1991). "Large-step Markov chains for the traveling salesman problem", *Complex Systems* **5**, 299–326.

O. Martin, S. W. Otto, and E. W. Felten (1992). "Large-step Markov chains for the TSP incorporating local search heuristics", *Operations Research Letters* **11**, 219–224.

O. C. Martin and S. W. Otto (1996). "Combining simulated annealing with local search heuristics", *Annals of Operations Research* **63**, 57–75.

K. Mehlhorn (1984). *Multi-dimensional Searching and Computational Geometry*, Springer, Berlin.

G. Morton and A. H. Land (1955). "A contribution to the 'travelling-salesman' problem", *Journal of the Royal Statistical Society, Series B,* **17**, 185–194.

H. Mühlenbein, M. Gorges-Schleuter, and O. Krämer (1988). "Evolution algorithms in combinatorial optimization", *Parallel Computing* **7**, 65–85.

D. M. Neto (1999), *Efficient Cluster Compensation for Lin-Kernighan Heuristics*, Ph.D. Thesis, Department of Computer Science, University of Toronto.

I. M. Oliver, D. J. Smith, and J. R. C. Holland (1987). "A study of permutation crossover operators on the traveling salesman problem", J.J. Grefenstette (ed.), *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, pp. 224–230.

M. Padberg and G. Rinaldi (1991). "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems", *SIAM Review* **33**, 60–100.

J. Perttunen (1994). “On the significance of the initial solution in travelling salesman heuristics”, *Journal of the Operational Research Society* **45**, 1131–1140.

G. Reinelt (1991). “TSPLIB - A traveling salesman library”, *ORSA Journal on Computing* **3**, 376–384.

G. Reinelt (1991a). “TSPLIB - Version 1.2”, Report Number 330, Schwerpunktprogramm der Deutschen Forschungsgemeinschaft, Universität Augsburg.

G. Reinelt (1992). “Fast heuristics for large geometric traveling salesman problems”, *ORSA Journal on Computing* **4**, 206–217.

G. Reinelt (1994). *The Traveling Salesman: Computational Solutions for TSP Applications*, Springer-Verlag, Berlin.

G. Reinelt (1995). “TSPLIB95”, Interdisziplinäres Zentrum für Wissenschaftliches Rechnen (IWR), Heidelberg. (Manuscript and problem instances available via anonymous ftp:
`ftp://ftp.iwr.uni-heidelberg.de/pub/tsplib`.)

G. Reinelt (1999). Personal communication.

S. Reiter and G. Sherman (1965). “Discrete optimizing”, *SIAM Journal on Applied Mathematics* **13**, 864–889.

N. Robertson and P. D. Seymour (1991). “Graph minors. X. Obstructions to tree-decomposition”, *Journal of Combinatorial Theory, Series B*, **52**, 153–190.

A Rohe (1997). *Parallele Heuristiken für sehr grosse Traveling Salesman Probleme*, Diplomarbeit, Research Institute for Discrete Mathematics, Universität Bonn.

M. Schäfer (1994). *Effiziente Algorithmen für sehr grosse Traveling Salesman Probleme*, Diplomarbeit, Research Institute for Discrete Mathematics, Universität Bonn.

D. Shallcross (1990). Personal communication.

D. D. Sleator and R. E. Tarjan (1983). “Self-adjusting binary trees”, *Proceedings of the Fifteenth Annual SCM Symposium on Theory of Computing*, ACM, pp. 235–245.

R. E. Tarjan (1983). *Data Structures and Network Algorithms*, SIAM, Philadelphia.

N. L. J. Ulder, E. H. L. Aarts, H.-J. Bandelt, P. J. M. Laarhoven, and E. Pesch (1991). “Genetic local search algorithms for the traveling salesman problem”, in: *Parallel Problem Solving from Nature* (H.-P. Schwefel and R. Männer, eds.), Elsevier Science Publishers, Amsterdam, pp. 65–74.

M. G. A. Verhoeven, P. C. J. Swinkels, and E. H. L. Aarts (1995). “Parallel local search for the traveling salesman”, Working Paper, Philips Research Laboratories, Eindhoven.

M. G. A. Verhoeven, E. H. L. Aarts, E. van de Sluis, and R. J. M. Vassens (1995). "Parallel local search and the travelling salesman problem", in: *Parallel Problem Solving from Nature 2* (R. Männer and B. Manderick, eds.), North-Holland, Amsterdam, pp. 543–552.