# Chapter 17
# Data Structures for Traveling Salesmen

M. L. Fredman* D. S. Johnson† L. A. McGeoch‡ G. Ostheimer§

**Abstract**

The choice of data structure for tour representation plays a critical role in the efficiency of local improvement heuristics for the Traveling Salesman Problem. The tour data structure must permit queries about the relative order of cities in the current tour and must allow sections of the tour to be reversed. The traditional array-based representation of a tour permits the relative order of cities to be determined in small constant time, but requires worst-case $\Omega(N)$ time (where $N$ is the number of cities) to implement a reversal, which renders it impractical for large instances. This paper considers alternative tour data structures, examining them from both a theoretical and experimental point of view. The first alternative we consider is a data structure based on splay trees, where all queries and updates take amortized time $O(\log N)$. We show that this is close to the best possible, because in the cell probe model of computation any data structure must take worst-case amortized time $\Omega(\log N/\log\log N)$ per operation. Empirically (for random Euclidean instances), splay trees overcome their large constant-factor overhead and catch up to arrays by $N = 10{,}000$, pulling ahead by a factor of 4-10 (depending on machine) when $N = 100{,}000$. Two alternative tree-based data structures do even better in this range, however. Although both are asymptotically inferior to the splay tree representation, the latter does not appear to pull even with them until $N \sim 1{,}000{,}000$.

*Rutgers University, New Brunswick, NJ 08093, and University of California at San Diego, La Jolla, CA 92093.

†Room 2D-150, AT&T Bell Laboratories, Murray Hill, NJ 07974.

‡Department of Mathematics and Computer Science, Amherst College, Amherst, MA 01002.

§Department of Mathematics, Rutgers University, New Brunswick, NJ 08903.

## 1. Introduction

In the Traveling Salesman Problem (TSP) we are given a set of *cities* $c_1, c_2, \ldots, c_N$ and for each pair $c_i, c_j$ of distinct cities a *distance* $d(c_i, c_j)$. Our goal is to find a permutation $\pi$ of the cities that minimizes the quantity $\Sigma_{i=1}^{N-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(N)}, c_{\pi(1)})$. This quantity is referred to as the *tour length*, since it is the length of the tour a salesman would make in visiting the cities in the order specified by the permutation, returning at the end to the starting city. The TSP is one of the most widely known NP-hard problems. Lawler et al. [19] give an excellent introduction to the broad range of work on this problem. In this paper, we concentrate on the symmetric TSP, where $d(c_i, c_j) = d(c_j, c_i)$ for $1 \le i, j \le N$.

Because the TSP is NP-hard, much research has concentrated on approximation algorithms whose goal is to find near-optimal rather than optimal tours. In practice, the best such algorithms have all been based on the principle of local optimization: One obtains a starting tour using some tour-construction heuristic (such as the Greedy algorithm [9]), and then attempts to repeatedly improve it using local modifications. The most commonly used such modifications are 2- and 3-changes, as illustrated in Figures 1 and 2. By themselves, they give rise to the well-known *2-Opt* and *3-Opt* algorithms. In more complicated combinations, they give rise to the famous *Lin-Kernighan* algorithm [21] and provide the basic engines for most applications of simulated annealing [5,15,17], genetic algorithms [4,23,24], and tabu search [10] to the TSP.

Two of the authors of the current paper have been involved in an extended study [3,13] of the 2-Opt, 3-Opt, and Lin-Kernighan algorithms [20,21], and how they can be adapted to very large instances. (Many applications give rise to instances with between 10,000 and 100,000 cities, and VLSI applications with as many as 1.2 million cities have been cited [16].) Table 1 shows the current level of performance we have been able to obtain with these algo-

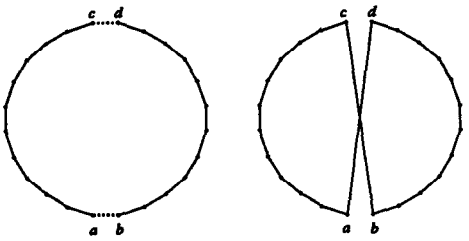


FIGURE 1. A 2-Change.

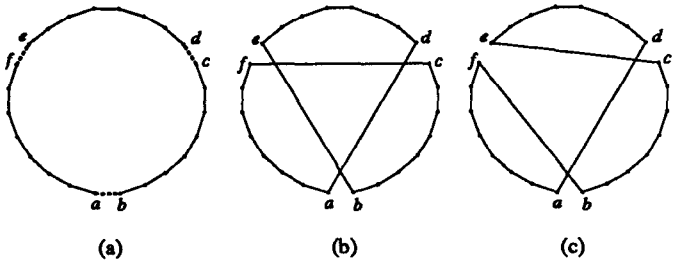


FIGURE 2. Two possible 3-Changes.

rithms for instances consisting of points uniformly distributed in the unit square under the Euclidean metric. Here running times are in *user* cpu-seconds on a 25 MHz MIPS™ processor running in a Silicon Graphics IRIS™ 4D/250 computer. (MIPS is a trademark of MIPS, Inc., IRIS is a trademark of Silicon Graphics, Inc.) A tour's quality is measured by the percentage it exceeds the Held-Karp lower bound [11,12,14] on optimal tour length. (For comparison purposes, note that the famous *Christofides algorithm* [6] only gets within 9-10% of the Held-Karp bound on such instances, and is significantly slower [3].)

### 1.1. The *Tour* Datatype

To obtain the performance reported in Table 1, we had to deal with a key implementation detail: how best to represent the current tour. The use of 2- and 3-changes is simplified by requiring that the tour be oriented, meaning that each city has a successor and predecessor in the tour. The labeling of tour neighbors must be consistent, in the sense that it must be possible to start at one city and traverse the entire tour by moving from each city to its successor. In the context of an algorithm based on 2- and 3-changes, the *Tour* datatype must then support four basic operations.

| | |
|---|---|
| *Next(a)* | This is a query that returns the city that follows $a$ in the current tour. |
| *Prev(a)* | This is a query that returns the city that precedes $a$ in the current tour. (It must be the case that $Next(Prev(a)) = a$.) |
| *Between(a,b,c)* | This is a query that returns true or false. Suppose one begins a forward traversal of the tour at city $a$. The query returns true if and only if city $b$ is reached before city $c$. |
| *Flip(a,b,c,d)* | This updates the tour by replacing the edges $(a,b)$ and $(c,d)$ by the edges $(b,c)$ and $(a,d)$. This operation assumes that $a = Next(b)$ and $d = Next(c)$. The orientation of the updated tour is not specified. |

Observe that $Flip(a,b,c,d)$ performs the same surgery as indicated in Figure 1, and hence implements the 2-change operation. Note also that if we had let $b = Next(a)$ instead of vice versa, the indicated surgery would have resulted in

| | Percent Excess | | | | Running Time in Seconds | | | |
|---|---|---|---|---|---|---|---|---|
| $N$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
| 2-Opt | 5.2 | 4.8 | 4.8 | 4.9 | 1.5 | 18 | 266 | 3060 |
| 3-Opt | 2.9 | 2.9 | 2.9 | 2.9 | 1.9 | 22 | 318 | 3720 |
| Lin-Kernighan | 2.0 | 2.0 | 1.9 | 2.0 | 3.1 | 44 | 566 | 9550 |

TABLE 1. Percentage excesses over Held-Karp lower bound and running times on a 25 Mhz MIPS processor.

two disjoint cycles rather than a new tour. This in essence is why the *Next* and *Prev* queries are needed for the datatype. A 3-change operation can be implemented by a series of two or three *Flip*'s, and the *Between* query is necessary to prevent the creation of disjoint cycles here. (Each of the much more complicated "λ-changes" of the Lin-Kernighan algorithm can be expressed as a 3-change followed by a sequence of 2-changes; λ-changes need no additional types of queries.) One more observation, important in what follows: Performing *Flip*($a$,$b$,$c$,$d$) changes the answers to *Next* and *Prev* for more cities than just $a$, $b$, $c$, and $d$. For either the $a$–$c$ or the $d$–$b$ path, all internal vertices must have their values for *Next* and *Prev* interchanged, since one of these two segments ends up reversed with respect to the other. This path-reversal property can result in a major implementation bottleneck.

### 1.2. The *Array* Representation

Consider what is perhaps the most straightforward (and common) implementation of the *Tour* datatype: the *Array* representation. Here the tour is represented by two one-dimensional arrays of length $N$. Array $A$ lists the cities in tour order, with $A[i+1] = Next(A[i])$, $1 \le i < N$. Array $B$ is the inverse of array $A$, with $A[B[i]] = c_i$, $1 \le i \le N$. It is easy to see that for this representation, all three types of queries can be answered in constant time, but *Flip* can take time $\Theta(N)$, even if one always reverses the shorter of the two segments. This worst-case behavior is realized in practice for the theoretically interesting class of instances with random distance matrices (i.e., instances with each $d(c_i,c_j)$ chosen independently and uniformly from the interval (0,1]). For these the average length of the shorter segment is empirically $N/4$ (as one might expect). On the other hand, for more realistic instances, the optimization of reversing the shorter path can reduce the average time per *Flip* to $o(N)$. For the above-mentioned random Euclidean instances, the length of the shorter segment seems to grow roughly as $N^{.7}$ [2], and similar behavior has been observed on instances from the TSPLIB database of Euclidean instances derived from real-world applications [25].

Despite these savings, the costs of tour manipulation grow to dominate overall running time as $N$ increases. Table 2 illustrates this for our implementation of Lin-Kernighan (which we shall treat mostly as a black box in what follows). We concentrate on Lin-Kernighan not only because it constructs the best tours, but also because this is where the effects show up earliest. For 3-Opt, the numbers of calls to *Next*, *Prev*, and *Between* are roughly comparable to those given here, but the number of calls to *Flip* is smaller by a factor of over 100, and so the expense of the latter operation, although noticeable for $N \ge 10^5$, does not become a significant concern until $N$ approaches $10^6$ and more. The results here and in Table 3 are based on averages over 20 runs for each of 5 instances of size $10^3$, over 12 runs for 3 instances of size $10^4$, and 9 or more runs for a single instance of size $10^5$. (The final paper will provide more detailed information about variances, which decline as $N$ increases.)

Note that, contrary to theory, the times per *Next*/*Prev* and *Between* operations appear to grow significantly with $N$, rather than remaining constant. This is likely an artifact of the RISC architecture of the MIPS processor, whose speed depends in large part on the efficient use of its data caches. Presumably the probability of a cache miss increases substantially as $N$ gets larger. We observed similar behavior on a RISC-based SPARCstation™ ELC (SPARC and SPARCstation are trademarks of Sun Microsystems, Inc.), but not on the non-RISC VAX™ 8550 (VAX is a trademark of Digital Equipment Corporation). For the latter, the times per operation for *Next*'s were 3.2, 3.5, 3.8 microseconds, and for *Between*'s were 4.9, 5.6, 5.4, although the overall time for tour operations still grew to 90% by $N = 10^5$.

### 1.3. Outline of What Follows

In this paper we consider alternative *Tour* representations, and the speedups they provide, both in theory and in practice. In Section 2, we introduce three alternative *Tour* representations, all designed to improve on the Array representation, and analyze their worst-case behavior. Asymptotically, the best of the three is the one based on splay trees [26],

| N | | $10^3$ | $10^4$ | $10^5$ |
|---|---|---|---|---|
| Shorter Segment | | 39 | 170 | 864 |
| Number of Calls | *Next* + *Prev* | 160,000 | 1,600,000 | 15,700,000 |
| | *Between* | 21,800 | 199,000 | 1,960,000 |
| | *Flip* | 64,800 | 623,000 | 6,080,000 |
| μSec per Call | *Next* + *Prev* | 0.8 | 1.2 | 1.9 |
| | *Between* | 1.3 | 1.8 | 2.4 |
| | *Flip* | 16.5 | 77.0 | 898.6 |
| Total Seconds for Tour Ops | | 1.2 | 50 | 5500 |
| Percent of Total Time | | 34% | 62% | 92% |

**TABLE 2.** Counts and times for *Tour* operations performed by the Lin-Kernighan algorithm (*Array* representation).

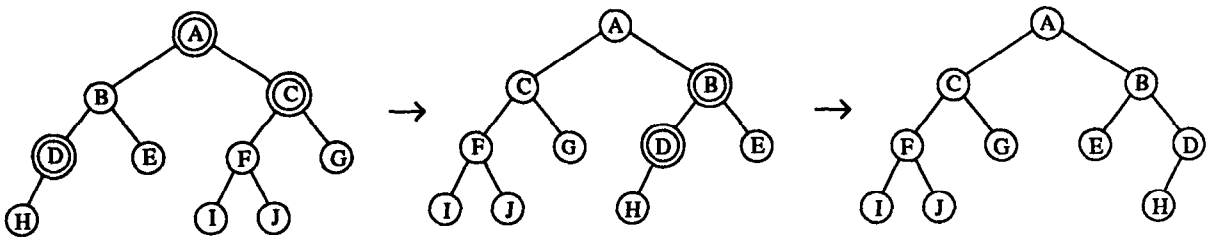


FIGURE 3. Three splay trees representing the tour $(I,F,J,C,G,A,E,B,H,D)$.

which has an amortized worst-case running time of $O(\log N)$ per operation. The constants of proportionality are high, however, leaving a surprisingly large window of opportunity for the other two representations. Section 3 summarizes experiments aimed at obtaining a detailed understanding of the empirical behavior of the various *Tour* representations and of the crossover points between them. Although the splay tree representation still hasn't quite caught the best of the alternatives by $N = 10^6$, it is clear that it will eventually dominate both. In Section 4, we show that no significantly better representation can exist. We prove that in the cell probe model of computation, no *Tour* representation can do better than a worst-case amortized bound of $\Omega(\log N/\log\log N)$ per operation.

## 2. Three Alternative Tour Representations

### 2.1. The *Splay Tree* Representation

Here the tour is represented by a binary search tree, with a city at each vertex along with a special *reversal bit*, indicating whether the subtree rooted at that vertex should be traversed in inorder, i.e., from left to right (reversal bit off) or in reversed inorder, i.e., from right to left (reversal bit on). Reversal bits lower down in the tree can then locally undo (or redo) the effect of bits higher up. See Figure 3, where vertices with their reversal bits on are represented by double circles. A simple way to determine the tour represented by a given tree is to push the reversal bits *down the tree* until they disappear, as is done in the figure, at which point the tour can be read off the tree by a simple inorder traversal of its vertices. (If a vertex $v$ has its reversal bit on, one can obtain an equivalent tree by turning the bit off, interchanging $v$'s left and right children, and complementing the reversal bit for each child.)

The vertices of the binary search tree are stored in an array indexed by the cities, so that given a city's name, one can find the city in the tree in constant time. It is thus not difficult to see how the three query operations can be implemented to run in worst-case time proportional to the depth of the tree. With effort one could probably implement *Flip* to run in a similar time and keep the tree balanced, thus yielding worst-case time $O(\log N)$ per operation. (A related structure with this behavior was recently proposed in [22].) Our *Splay Tree* representation settles for *amortized* worst-case time $O(\log N)$, but benefits from programming simplicity and the ability to take advantage of locality of reference.

Splay trees were introduced by Sleator and Tarjan [26]. The key concept is that every time an item is accessed, whether in a query or an update operation, it is brought to the root (splayed) by a sequence of rotations (local alterations of the tree that preserve the inorder traversal). In the splay operation, each rotation only affects a vertex, its parent, and its grandparent, and the precise change made depends only on whether the current vertex and its parent are left or right children, not on any global properties of the subtrees involved, such as depth, etc. Sleator and Tarjan [26] showed that all the standard binary tree operations could be implemented to run in amortized worst-case time $O(\log N)$ using splays.

For our Splay Tree *Tour* representation, the main added complication is the reversal bits. To deal with these, we precede each rotation by an initial pass that pushes the reversal bits down out of the affected area, after which we can perform an ordinary rotation. The implementations of our four operations all then follow the same general pattern: First we splay the relevant cities to the top of the tree, and then we handle the remaining work by a simple case analysis on the structure of the top few levels of the tree. (This is not quite true for $Prev(a)$ and $Next(a)$: after splaying $a$ to the root, one must traverse down the tree to identify the predecessor or successor, which itself is then splayed up to the top of the tree as a final clean-up to maintain the amortized guarantees.) In the full paper we shall give more details, and show how the analysis of [26] applies to give the claimed bounds.

### 2.2. The *Two-Level Tree* Representation

This approach was suggested by Tom Leighton [18], who observed that for $N \le 10^6$, it might well be that $c\sqrt{N}$ with a small $c$ is better than $d\log N$ where $d$ represents the overhead factor for splay trees. It is also designed to take advantage of the fact that our algorithm makes significantly more calls to *Prev* and *Next* than to *Flip*. In this representation, all the query operations are performed in constant time (as in the Array representation, albeit with slightly larger constants), while the *Flip* operation takes amortized time $O(\sqrt{N})$.

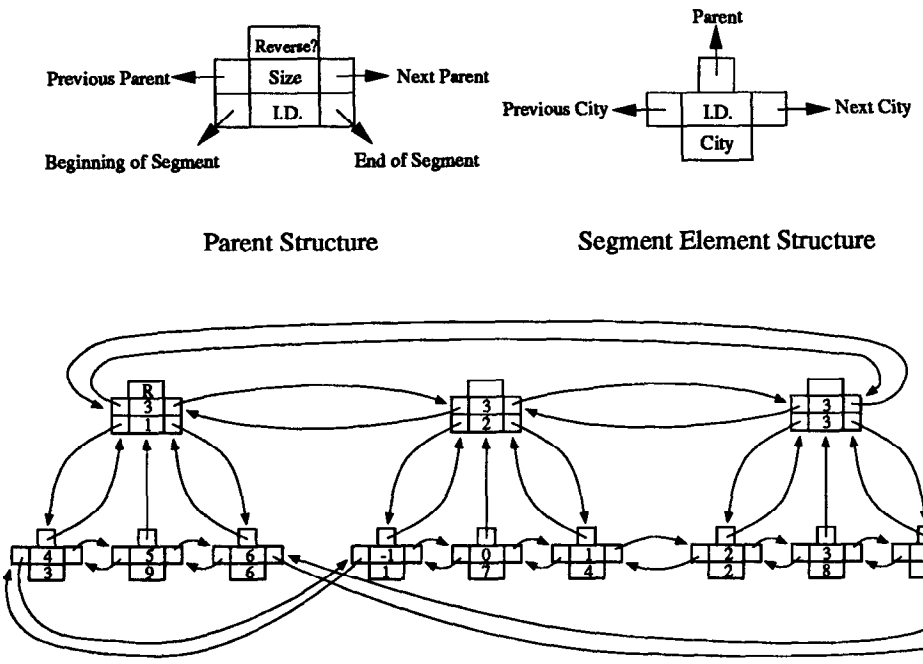


FIGURE 4. The Two-Level *Tour* representation.

The tour is divided into $\lceil N/g \rceil$ consecutive subsegments of approximately equal length, for a given *group-size* parameter $g$. Each segment is maintained as a doubly-linked list (using pointers labeled *prev* and *next*). See Figure 4. All members of the segment also contain a pointer to a parent node representing the entire segment. The parent node contains a *reversal* bit to indicate whether the segment should be traversed in forward or reverse direction. In addition, each member of the segment contains the index of the city it represents and an identification number that represents its position within the segment, so as to facilitate answering *Between* queries. (This numbering is consecutive within the segment, but need not start with 1.) The first vertex in a segment uses its *prev* pointer to point to its tour neighbor that is not in the segment, and the last vertex uses its *next* pointer similarly.

The parent nodes of the segments are themselves connected in a doubly-linked list. In addition, each parent node contains an identification number of its own (with the numbers running in sequence from 1 to $\lceil N/g \rceil$), a count of the number of cities in the segment it represents, and pointers to the segment's two endpoints. It is easy to see that all three types of queries can be answered in constant time, although the constant for *Between* is somewhat higher than it was for arrays. There is a real savings over arrays in the subtour reversal operation, however. All we need do is perform some surgery on the segments containing the endpoints of the subtour to be reversed (time $O(g)$, assuming segment sizes stay relatively balanced), and then re-order and re-number all the parents of segments that currently lie between those endpoints, at the same time flipping their reversal bits (total time $O(N/g)$). If we set $g \sim \sqrt{N}$ and also spend a bit of time (when needed) rebalancing the segment lengths so that they stay within a factor of two of this target value, we obtain an amortized time of $O(\sqrt{N})$ per *Flip*. Once again, full details will appear in the final paper.

### 2.3. The *Segment Tree* Representation

This form of representation was proposed by David Applegate and Bill Cook [1] to exploit an implementation of Lin-Kernighan that imposes a hard-wired bound (typically 50) on the maximum depth of tentative *Flips* that can be performed. (In pure Lin-Kernighan, the search for an improving move can involve making up to $N$ tentative *Flip*'s, and our implementation normally follows this scheme. Bounding the depth at 50 typically does not have a significant effect on the length of the tour output by Lin-Kernighan, but it can affect overall running times, either speeding them up or slowing them down depending on the instance.)

Our *Segment Tree* representation is a hybrid, using an Array representation for the current "permanent" tour, and an auxiliary tree and list to keep track of the "tentative" tour derived from the current sequence of tentative *Flip*'s. (See Figure 5.) Each vertex of the auxiliary tree corresponds to a segment of the permanent tour, identifying the segment by its endpoints in the array. The tree is structured so that an inorder traversal gives the segments in the order in which they occur in the permanent tour. The auxiliary list then gives the ordering of the segments as they occur in the tentative tour, with a reversal bit for each. Given a bound of

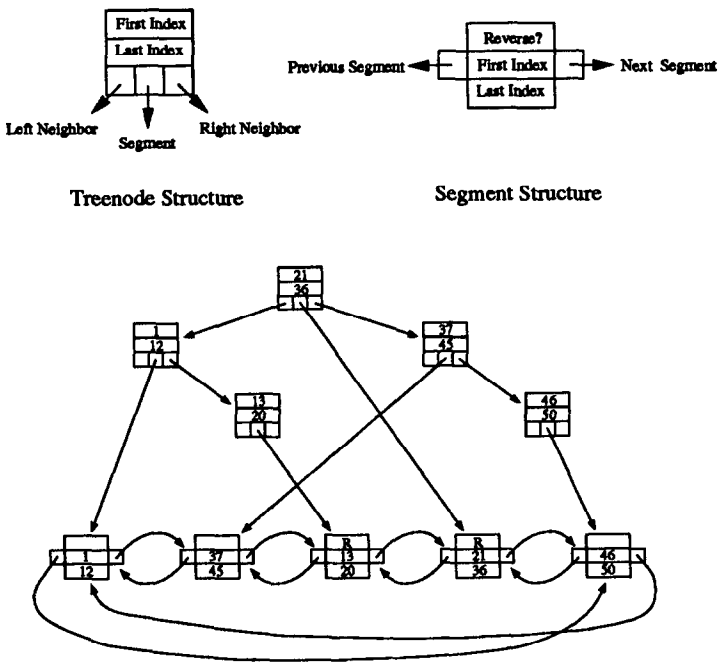


FIGURE 5. The Segment Tree *Tour* representation.

$k$ on the number of tentative *Flip*'s that can be performed, the length of this list is bounded by $2k+1$, as is the depth of the tree. The representation can thus be shown to take time $O(k)$ per query and tentative *Flip*, which is constant for fixed $k$. (Following Applegate and Cook, we take $k = 50$ in our implementation.)

Our implementation contains several enhancements to the basic scheme sketched above. For instance, for each city we keep a pointer to the treenode whose interval contained the city the last time we looked, thus speeding the segment lookup process. Also, in practice one would not use separate structures for the elements of the auxiliary tree and list, but would have one structure that contains both the tree and list pointers. (The redundancy in Figure 5 was added to highlight the fact that both a tree and a list were present.) Another optimization is applied when a particular sequence of tentative *Flip*'s does not lead to an improving move. Rather than undo all the flips, as we would in other representations, we can in constant time simply delete the current auxiliary tree and list, and start building a new one from scratch.

When the result of the tentative *Flip*'s is an improving move, however, we need to do real work, as the array containing the permanent tour must be updated. We shall call this the *Make Permanent* operation. Typically, we implement this by re-enacting the sequence of *Flip*'s, this time by physically interchanging cities in the permanent tour, a process that in the worst case can take time $\Theta(kN)$. In some situations a better alternative is simply to read off the new permanent tour from the auxiliary list representing the tentative tour, which (always) takes time $\Theta(N)$.

In a worst-case sense, this Segment Tree representation is asymptotically at least as good as the Array representation, and, depending on the frequency with which improving moves are permanently made, can be much faster. For instance, for the random Euclidean instances profiled in Table 2, the number of permanent improving moves (*Make Permanent* operations) grows roughly as $N/6$, compared to a growth rate of roughly $60N$ for overall *Flip*'s. This should yield a hefty advantage for Segment Trees over Arrays. The question is whether there is a window in which Segment Trees also beat our other representations, and in the next section we summarize experiments that address this question as well as others.

## 3. Experimental Comparisons

We have implemented the three alternative *Tour* representations (along with the original Array representation) in the C programming language. Our experiments were performed under variants of the UNIX™ operating system (UNIX is a trademark of Unix Systems Laboratories, Inc.) on three different machines. The first (and slowest) was a DEC VAX 8550 computer with 128 megabytes of main memory. The second was a SPARCstation ELC with 64 megabytes of main memory. The third was a Silicon Graphics IRIS 4D/250 computer with 256 megabytes of main memory and a 25 MHz MIPS processor as CPU. Roughly 24 megabytes of memory were required to run the 100,000 cities without excessive paging. Instances with 1,000,000 cities required

| | μSec per Call | | | | Total Seconds | | | |
|---|---|---|---|---|---|---|---|---|
| N | $10^3$ | $10^4$ | $10^5$ | $10^6$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
| Array: | | | | | | | | |
| *Next + Prev* | 0.8 | 1.2 | 1.9 | | 0.1 | 1.9 | 29 | |
| *Between* | 1.3 | 1.8 | 2.4 | | 0.0 | 0.4 | 5 | |
| *Flip* | 16.5 | 77.0 | 898.6 | | 1.1 | 48.0 | 5466 | |
| Total | | | | | 1.2 | 50.3 | 5501 | |
| Splay Tree: | | | | | | | | |
| *Next + Prev* | 5.2 | 7.6 | 10.5 | 16.2 | 0.8 | 12.3 | 169 | 2574 |
| *Between* | 13.4 | 17.6 | 23.8 | 31.5 | 0.3 | 3.6 | 47 | 609 |
| *Flip* | 10.5 | 11.7 | 13.3 | 15.3 | 0.7 | 7.4 | 82 | 925 |
| Total | | | | | 1.8 | 23.2 | 298 | 4108 |
| Two-Level Tree: | | | | | | | | |
| *Next + Prev* | 1.0 | 1.9 | 2.2 | 3.9 | 0.2 | 3.2 | 37 | 637 |
| *Between* | 2.3 | 3.3 | 3.5 | 4.6 | 0.1 | 0.7 | 7 | 89 |
| *Flip* | 10.2 | 14.5 | 21.5 | 54.1 | 0.7 | 9.1 | 132 | 3256 |
| Total | | | | | 0.9 | 13.0 | 175 | 3983 |
| Segment Tree: | | | | | | | | |
| *Next + Prev* | 3.2 | 3.8 | 4.4 | 5.1 | 0.5 | 6.0 | 66 | 748 |
| *Between* | 1.3 | 1.9 | 2.5 | 2.8 | 0.0 | 0.4 | 5 | 55 |
| *Flip* | 14.0 | 15.0 | 15.8 | 16.2 | 0.5 | 4.8 | 49 | 487 |
| *MakePermanent* | 92.7 | 584.9 | 7776.4 | 60550.5 | 0.0 | 1.1 | 134 | 10290 |
| Total | | | | | 1.0 | 12.3 | 254 | 11580 |

**TABLE 3.** Counts and times on an IRIS 4D/250 for *Tour* operations performed by the Lin-Kernighan algorithm.

roughly 244 megabytes, and hence were only run on the IRIS. Times reported here are *user* times, and do not include time spent in paging, except insofar as that time leaks into the figures the machine reports as *user* time. (Such leakage did occur for our million-city instances when other large jobs were sharing the machine, so the million-city experiments were performed when we were essentially the only user of the machine.)

The *Tour* representations were plugged into our Lin-Kernighan code via a standardized interface that included calls to the four standard *Tour* operations, augmented to include necessary calls such as *Initialize* and *OutputTour*, along with other calls that allow us to treat permanent and tentative changes differently (as required by the Segment Tree representation). The code performs its own profiling, using the UNIX system command `profil(2)`, which allows us to obtain consistent profiling results across different machines. Compilation was done using the `-O` optimization flag on all machines, although since each machine had a different compiler, the extent of optimization differed from machine to machine.

In Table 3, we summarize the profiling information on the IRIS 4D/250 for our three new *Tour* representations, including for comparison purposes the data from Table 2 for the Array representation. For reasons of efficiency and programming simplicity, our implementations depart in several ways from the idealized ones for which the theoretical worst-case results were presented in the previous section. For instance, no rebalancing of groups was done under the Two-Level Tree representation, and the initial *groupsize* was fixed at 100 for all instances with $N \leq 100{,}000$, increasing to 200 for $N = 1{,}000{,}000$. A *groupsize* of 50 would have provided a small improvement in running time for $N \leq 10{,}000$, but the Two-Level representation proved remarkably robust with respect to *groupsize*, at least until the million-city level. (Here, the time per flip would have increased by 36% had we left *groupsize* at 100.)

Our Splay Tree representation also differed significantly from the theoretical model presented in the previous section. In particular, no splays were performed in the *Next* and *Prev* operations under the Splay Tree representation; a simple traversal scheme was used instead, resulting in significant speedups. (We also implemented *Between* with one splay replaced by a traversal, but this was more for programming simplicity than speed, given the relative infrequency of calls to *Between*.) The nature of the speedups due to our splay-less implementation of *Next* and *Prev* is revealed in Table 4, which gives average overall running times, normalized by dividing through by $N$, on all three machines. The first row of times presents the initial time spent by the algorithm before the *Tour* representation code is invoked (initial preprocessing plus the construction of the

| | (Running Time in Milliseconds)/$N$ | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | VAX-8550 | | | SPARCstation ELC | | | IRIS 4D/250 | | | |
| $N$ | $10^3$ | $10^4$ | $10^5$ | $10^3$ | $10^4$ | $10^5$ | $10^3$ | $10^4$ | $10^5$ | $10^6$ |
| Preprocessing | 5.0 | 5.4 | 6.0 | 1.6 | 1.9 | 2.1 | 1.3 | 1.6 | 2.1 | 2.7 |
| Array | 7.4 | 18.0 | 87.7 | 2.6 | 9.4 | 40.3 | 2.2 | 6.5 | 57.3 | - |
| Splay Tree-2 | 13.4 | 14.1 | 15.5 | 4.3 | 4.6 | 4.9 | 3.7 | 4.7 | 5.9 | 7.8 |
| Splay Tree-0 | 10.9 | 13.2 | 12.8 | 3.1 | 3.3 | 3.6 | 2.8 | 3.8 | 5.1 | 7.1 |
| Two-Level Tree | 7.4 | 7.7 | 9.1 | 2.0 | 2.2 | 2.9 | 1.9 | 2.8 | 3.6 | 6.7 |
| Segment Tree | 7.2 | 7.2 | 8.7 | 2.0 | 2.1 | 2.9 | 2.0 | 2.6 | 4.2 | 13.9 |

**TABLE 4.** Overall running times for *Tour* representations (Normalized).

starting tour). The remaining rows give the time spent in local optimization, where the *Tour* representation code is active. Here the row labelled *Splay Tree-2* corresponds to the case when both splays *are* performed in *Next* and *Prev* (the option covered in Table 3), whereas the row labelled *Splay Tree-0* corresponds to the case where neither are, and is consistently faster, although the gap appears to be closing.

Further evidence of the closing gap is presented in Figure 6, which reports the average depth at which the relevant vertices are encountered in the tree as a function of $N$ and of the operation under the two different Splay Tree regimes. Here the dashed lines correspond to Splay Tree-0 and the solid lines to Splay Tree-2, with $N$, $B$, $F$ representing *Next*, *Between* and *Flip* operations respectively. All these figures support the hypothesis that the representations are taking advantage of the locality of reference within Lin-Kernighan, but note that under full splaying (Splay Tree-2) this exploitation is so strong that the average depths are essentially constant from 1,000 to 1,000,000 cities. The average depths for *Flip*'s under Splay Tree-0 is less than that under Splay Tree-2 because typically most *Flip*'s are restricted to the same small key set of endpoints, and performing splays on *Next*/*Prev*'s often splays non-key cities to the top of the tree, thus pushing the key cities further down in the tree. The advantage of Splay Tree-0 on *Flips*'s, however, will eventually be dominated by its growing disadvantage on the more frequently invoked *Next*'s.

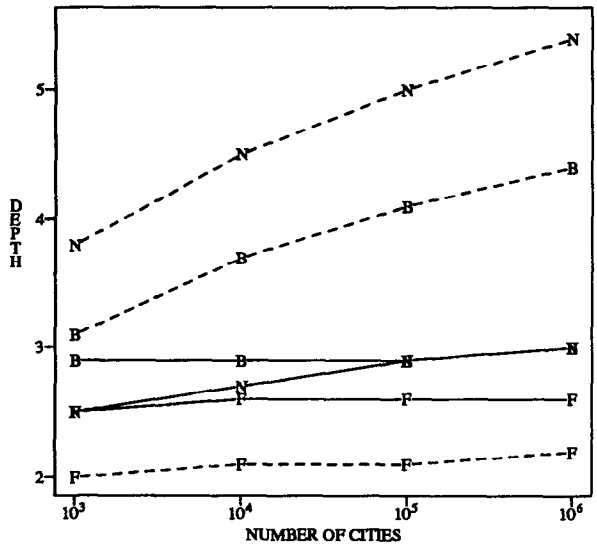


**FIGURE 6.** Growth rates for the depth of cities.

Having discussed some of the details, let us now step back and view the overall picture. A first observation is that the relative efficiencies of the *Tour* representations are machine dependent, with Segment Trees and Arrays performing worst on the IRIS. (Another IRIS anomaly: although faster than the SPARCstation for $N = 1{,}000$, it drops further and further behind for larger $N$.) The full paper will discuss other factors that effect the relative efficiencies of the representations. For instance, turning off the optimization flag on the IRIS hurts the Two-Level representation the most, causing it to fall significantly behind Segment Trees for all instances with $N \leq 100{,}000$. Nonetheless, it is clear that *all* our new representations are significant improvements over Arrays in the range from 10,000 to 100,000 cities, offering major speedups in the latter case. Extrapolations suggest that Arrays would have taken over 150 hours on the IRIS, as compared less than three for Splay Trees and Two-Level Trees. The average overall times for our representations on a million cities were 2.65 hours for Two-Level Trees, 2.69 for Splay Tree-0, 2.87 for Splay Tree-2, and 4.60 for Segment Trees. (As the profiling information in Table 3 makes clear, the growing cost of the *MakePermanent* operation is likely doom the Segment Tree representation as $N$ increases beyond 1,000,000, no matter what machine and what optimization flag.)

In addition to testing our *Tour* representations on random Euclidean instances, we have also tested them on random distance matrices with $N$ ranging from 1,000 to 31,622 (approximately $10^{4.5}$), and on structured Euclidean instances from real-world applications with up to 85,900 cities obtained from Gerd Reinelt's TSPLIB [25]. For the former class of instances, which do not obey the triangle inequality, Segment Trees are the clear winner on the IRIS (taking advantage of the fact that far fewer changes to the

permanent tour occur for such instances). Moreover, at the top end of the range, the Splay Tree-0 representation has also passed Two-Level Trees. For large real-world instances, Segment Trees are also the winner, although Two-Level Trees here remain in second. Details will be provided in the full paper.

The main conclusion one draws from all the above experimental results is that a robust implementation of Lin-Kernighan might need to include *all* the *Tour* representations we have discussed. (Arrays continue to be the structure of choice for instances with $N < 1{,}000$.) However, it should be pointed out that within the range of $N$ covered in our experiments, the running times we report for all the representations (other than arrays) all remain roughly within a factor of two of each other. They are thus well within the range where additional code-tuning and minor algorithmic modifications might well have a major impact on the locations of the various crossover points. Consequently, the results we have presented should for now be taken only as preliminary guides.

## 4. Lower Bounds

In this concluding section, we return to the safer world of worst-case asymptopia, and sketch our proof that any *Tour* representation must take amortized time $\Omega(\log N/\log\log N)$ per operation in the worst case. We prove our amortized lower bounds in the *cell probe* model of computation, introduced by A. Yao (see [7] for a precise description). In this model there is a single parameter $b$, the number of bits in single word of memory, and the cost of a computation is simply the number of words accessed. This is a very general model, encompassing even such baroque data structures as the *fusion trees* of [8]. Our proof relies on a generalization of Theorem 3′ in [7] (proved by a straightforward generalization of the proof of that result). We shall first describe the generalization, and then say a bit about how it is applied to the *Tour* datatype. We begin with some definitions.

**Definition.** A *dynamic query problem* consists of a finite state space $S$ with a distinguished start state $s_0$, an ordered finite set Q of *query operations* $Q\colon S \to \{0,1\}$, and a finite set U of *update operations* $U\colon S \to S$. A *solution* to a dynamic query problem is an on-line algorithm for maintaining the current state while supporting the update and query operations. (The update operations change the state, whereas the query operations return an answer but leave the state unchanged.) Note that no assumption is made about the manner in which states are to be represented or whether there are multiple representations of a given state. Also, we do not preclude the possibility that the implementation of a query operation may change the representation of the current state.

**Definition.** Let $\Delta$ be a dynamic query problem, and let $F^{(k)} \subseteq \mathbf{U}^*$ be a collection of update sequences of length $k$. Let $M = |\mathbf{Q}|$. For any state $s \in S$ and for each sequence $\lambda$ in $F^{(k)}$ let $\lambda(s)$ be the state obtained by starting with $s$ and sequentially applying the $k$ update operations of $\lambda$. For any state $s$, let $v_s \in \{0,1\}^M$ be the $M$-dimensional vector determined by the responses to the $M$ queries in Q from the state $s$. Let $\delta$ denote the Hamming distance function between vectors. For positive real numbers $\gamma$ and $\rho$ we say that $F^{(k)}$ is $(\gamma, \rho)$-*dispersing* provided that for all $u \in \{0,1\}^M$ and all $s \in S$, $|\{\lambda \in F^{(k)}\colon \delta(v_{\lambda(s)}, u) \le \gamma M\}| \le |F^{(k)}|/2^{\rho k}$.

Intuitively, this definition asserts (when $\gamma < 1$) that the operation sequences in $F^{(k)}$ have sufficiently varying impact so as to prevent clustering of the query responses.

**Definition.** Let $\Delta$ and $M$ be as above. Let $F \subseteq \mathbf{U}^*$ be a collection of update sequences of length $m$ having the product form $F = F_1 F_2 \cdots F_m$, where each $F_j \subseteq \mathbf{U}$. Given a positive integer $t$ we say that $F$ is $(t, \gamma, \rho)$-*dispersed* if the following holds. Let $F^{(k)}$ be the product of any $k$ consecutive terms from the product defining $F$. We require that $F^{(k)}$ be $(\gamma, \rho)$-dispersing for each $k$, $t \le k \le \sqrt{M}$.

**Theorem A.** *Let $\Delta$ be a dynamic query problem and let $F$ be a set of length-$m$ update sequences for $\Delta$ that is $(t,\gamma,\rho)$-dispersed, where $m \ge 2\sqrt{M}$, $t \le M^{1/4}$ and $\rho \ge (\log M)^{-\kappa}$. Let $H$ be the set of all sequences of operations of the form $U_1\, Q_1 \cdots U_m\, Q_m$, where the $Q_i \in \mathbf{Q}$ and update subsequences, $(U_1 \cdots U_m) \in F$. Let $A$ be an on-line algorithm for $\Delta$ in the cell probe model of computation with $b$-bit word size, where $b \le (\log M)^{\kappa}$. Then for at least one of the sequences $\alpha$ in $H$, $A$ performs $\Omega(\gamma \cdot m \cdot \log M/(\kappa \log\log M))$ memory accesses when executing the operations in $\alpha$.*

In order to apply Theorem A to the problem of maintaining a *Tour* representation, we first show that our *Tour* datatype can be used to simulate a related dynamic query problem. The *Reversible String* problem is defined as follows. The states are all permutations of a length-$N$ string $s_0$ consisting of $N$ distinct symbols. Let $\Sigma$ be the corresponding $N$-symbol alphabet. Queries are of the form *Precedes*$(x,y)$, where $x,y \in \Sigma$, where *Precedes*$(x,y)$ applied to state $s$ returns 1 if and only if $y$ is the immediate successor of $x$ in $s$. Updates are of the form *Reverse*$(x,y)$ and cause the current string $s$ to be modified by reversing the substring of $s$ that runs between symbols $x$ and $y$ (inclusive).

It is not difficult to show that any implementation of the *Tour* datatype can be adapted to handle the Reversible String problem. More specifically, any sequence of $m$ operations for the latter problem can be implemented by a sequence of at most $5m$ *Tour* operations plus $O(m)$ additional computation. The string can be turned into a tour by considering each symbol to be a city, and adding two additional cities $c$ and $d$ where $c$ will always be linked to $d$ and the last symbol in $s$, and $d$ is in addition linked to the first symbol in $s$. We will then have that *Precedes*$(x,y) = 1$ iff either (a) *Next*$(c) = d$ and *Next*$(x) = y$, or (b) *Next*$(c) \ne d$ and *Prev*$(x) = y$, so *Precedes*$(x,y)$ can be implemented with two calls to *Prev*/*Next*. *Reverse*$(x,y)$ is a bit more

complicated, but can be done with three *Prev/Next*'s, one *Between*, and a *Flip*. Thus if we can show that the Reversible String problem requires worst-case amortized time $\Omega(\log N/\log\log N)$ time per operation, the same conclusion will follow for our *Tour* datatype.

We actually apply Theorem A to a highly restricted version of the Reversible String Problem. First, we assume $N$ is a power of 2, that $\Sigma$ consists of the symbols $a_i, b_i$, $1 \le i \le N/2$, and that $s_0 = a_1 b_1 a_2 b_2 \cdots a_{N/2} b_{N/2}$. Then we restrict our updates to those that, when performed, reverse a substring of length $2^k$ starting at a position $i \equiv 1 (mod 2^k)$ for some $k \le \log N$. Among other things, this will insure that every pair $(a_i, b_i)$ will remain contiguous, although their order may be reversed. Our restriction on queries assumes this property: All queries we ask will be of the form $Precedes(a_i, b_i)$ for some $i \le N/2$. We leave to the full paper the details of how the hypotheses of Theorem A can be shown to apply under these restrictions.

## References


1. D. APPLEGATE AND W. COOK, private communication (1990).
2. J. L. BENTLEY, "Fast algorithms for geometric traveling salesman problems," *ORSA J. Comput.*, to appear.
3. J. L. BENTLEY, D. S. JOHNSON, L. A. MCGEOCH, AND E. E. ROTHBERG, "Near-optimal solutions to very large traveling salesman problems," in preparation.
4. R. M. BRADY, "Optimization strategies gleaned from biological evolution," *Nature* **317** (October 31, 1985), 804-806.
5. V. CERNY, "A Thermodynamical Approach to the Travelling Salesman Problem: An Efficient Simulation Algorithm," *J. Optimization Theory and Appl.* **45** (1985), 41-51.
6. N. CHRISTOFIDES, "Worst-case analysis of a new heuristic for the travelling salesman problem," Report No. 388, GSIA, Carnegie-Mellon University, Pittsburgh, PA, 1976.
7. M. L. FREDMAN AND M. E. SAKS, "The cell probe complexity of dynamic data structures," in *Proceedings 21st Ann. ACM Symp. on Theory of Computing*, Association for Computing Machinery, New York, 1989, 345-354.
8. M. L. FREDMAN AND D. E. WILLARD, "BLASTING through the information theoretic barrier with FUSION TREES," in *Proceedings 22nd Ann. ACM Symp. on Theory of Computing*, Association for Computing Machinery, New York, 1990, 1-7.
9. A. M. FRIEZE, "Worst-case analysis of algorithms for travelling salesman problems," *Methods of Operations Research* **32** (1979), 97-112.
10. F. GLOVER, "Tabu search – Part I," *ORSA J. Comput.* **1** (1989), 190-206.
11. M. HELD AND R. M. KARP, "The traveling-salesman problem and minimum spanning trees," *Operations Res.* **18** (1970), 1138-1162.
12. M. HELD AND R. M. KARP, "The traveling-salesman problem and minimum spanning trees: Part II," *Math. Programming* **1** (1971), 6-25.
13. D. S. JOHNSON, "Local optimization and the traveling salesman problem," *in Proc. 17th Colloq. on Automata, Languages, and Programming*, Lecture Notes in Computer Science **443**, Springer-Verlag, Berlin, 1990, 446-461.
14. D. S. JOHNSON AND E. E. ROTHBERG, "Asymptotic experimental analysis of the Held-Karp lower bound for the traveling salesman problem," in preparation.
15. S. KIRKPATRICK, "Optimization by simulated annealing: Quantitative studies," *J. Stat. Physics* **34** (1984), 976-986.
16. B. KORTE, "Applications of combinatorial optimization," talk at the 13th International Mathematical Programming Symposium, Tokyo, 1988.
17. J. LAM AND J.-M. DELOSME, "An efficient simulated annealing schedule: implementation and evaluation," manuscript (1988).
18. F. T. LEIGHTON, private communication (1989).
19. E. L. LAWLER, J. K. LENSTRA, A. H. G. RINNOOY KAN, AND D. B. SHMOYS, *The Traveling Salesman Problem*, John Wiley & Sons, Chichester, 1985.
20. S. LIN, "Computer solutions of the traveling salesman problem," *Bell Syst. Tech. J.* **44** (1965), 2245-2269.
21. S. LIN AND B. W. KERNIGHAN, "An Effective Heuristic Algorithm for the Traveling-Salesman Problem," *Operations Res.* **21** (1973), 498-516.
22. F. MARGOT, "Quick updates for $p$-opt TSP heuristics," *Operations Res. Lett.* **11** (1992), 45-46.
23. O. MARTIN, S. W. OTTO, AND E. W. FELTEN, "Large-step Markov chains for the traveling salesman problem," manuscript (1989).
24. H. MÜHLENBEIN, M. GORGES-SCHLEUTER, AND O. KRÄMER, "Evolution algorithms in combinatorial optimization," *Parallel Comput.* **7** (1988), 65-85.
25. G. REINELT, "TSPLIB—A traveling salesman problem library," *ORSA. J. Comput.* **3** (1991), 376-384.
26. D. D. SLEATOR AND R. E. TARJAN, "Self-adjusting binary search trees," *J. Assoc. Comput. Mach.* **32** (1985), 652-686.