Data Structures for Traveling Salesmen*

M. L. Fredman[1]
D. S. Johnson[2]
L. A. McGeoch[3]
G. Ostheimer[4]

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 $N \sim 1,000,000$.

---

[1] Rutgers University, New Brunswick, NJ 08903, and University of California at San Diego, La Jolla, CA 92093
[2] Room 2D-150, AT&T Bell Laboratories, Murray Hill, NJ 07974
[3] Department of Mathematics and Computer Science, Amherst College, Amherst, MA 01002
[4] Department of Mathematics, Rutgers University, New Brunswick, NJ 08903
* A preliminary version of this paper appeared under the same title in *Proceedings 4th Ann. ACM-SIAM Symp. on Discrete Algorithms* (1993), 145-154.

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 π 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. [21] 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 [10]) and then repeatedly attempts to improve it using local modifications. The most commonly used such modifications are 2- and 3-changes, as illustrated in Figures 1.1 and 1.2. Here for ease of explanation we view the tour in its alternative guise as a Hamiltonian cycle in the complete graph whose vertices are the cities. The 2- and 3-changes construct new Hamiltonian cycles by deleting and adding edges as shown. By themselves, these operations 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 [23] and provide the basic engines for most applications of simulated annealing [5,17,19], genetic algorithms [4,25,26], and tabu search [11] to the TSP.

Two of the authors of the current paper have been involved in an extended study [3,14] of the 2-Opt, 3-Opt, and Lin-Kernighan algorithms [22,23] 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 [18].) Table 1.1 shows the current level of performance we have been able to obtain with these algorithms 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 [12,13,15] on optimal tour length. (For comparison purposes, note that the famous

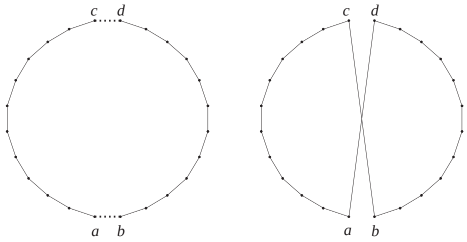

FIGURE 1.1. A 2-change.

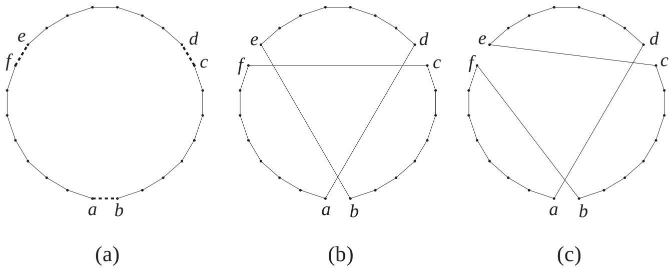

FIGURE 1.2. Two possible 3-changes.

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 1.1, 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.

	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 1.1. Percent excess over Held-Karp bound and running time on a 25 Mhz R3000 MIPS processor.

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 $Next(Prev(a)) = Prev(Next(a)) = a$.)
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 1.1, 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 two disjoint cycles rather than a new tour. This in essence is why the *Next* and *Prev* queries are needed for the datatype. The 3-change operation of Figure 1.2 can be implemented by a sequence of two or three *Flips*, but as with the 2-change, one must obtain additional information if one is to perform the correct sequence and prevent the creation of disjoint cycles. For this the *Between* query is needed in addition to *Next* and *Prev*. (The much more complicated ''λ-changes'' of the Lin-Kernighan algorithm can typically 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 $A[i+1] = Next(A[i])$, $1 \le i < N$, and $A[1] = Next(A[N])$. 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. Unfortunately *Flip* will take time proportional to the length of the segment that is reversed, which in the worst case is $\Theta(N)$ 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 $N^{.7}$ [3], and similar behavior has been observed on instances from the TSPLIB database of Euclidean instances derived from real-world applications [27].

Despite these savings, the costs of tour manipulation grow to dominate overall running time as N increases. Table 1.2 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 $N \geq 10^5$, does not become a significant concern until N approaches 10^6 and more. The results here and in Section 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, yielding 95% confidence intervals for the computed means μ that are typically $[0.96\mu, 1.04\mu]$ or better. (The variances of the experimental values, both between instances and between runs on the same instance, decline as N increases. For instance, the estimated standard deviations for the numbers of *Next*/*Prev* queries are 14% of the reported means at $N = 10^3$, 6% at $N = 10^4$, and 3% at $N = 10^5$. The standard deviations of the microseconds per *Next*/*Prev* query drop from roughly 27% to 10% to 4%.)

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

Note that, contrary to theory, the time for each *Next/Prev* and *Between* operation appears to grow significantly with N, rather than remaining constant. Also, in going from $N = 10,000$ to $N = 100,000$, the time per *Flip* grows by significantly more than the average length of the shorter segment. 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 *Nexts*/*Prevs* were 3.2, 3.5, 3.8 microseconds, and for *Betweens* 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 in theory and in practice. In Section 2, we introduce three new *Tour* representations, all designed to improve on the Array representation, and we analyze their worst-case behavior. Asymptotically, the best of the three is the one based on splay trees [28], 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. We also consider the effects of several practical compromises and simplifications that one can make in implementing the representations, changes that give up the precise asymptotic guarantees highlighted in Section 2, but that may yield codes that are simpler and/or faster in practice. We performed our suite of experiments on the three different computers mentioned above, and our search for conclusions is complicated by the fact that relative results differed substantially from machine to machine. We can conclude, however, that although the Splay Tree representation easily beats Arrays by the time $N = 10,000$, it is *not* the representation of choice for $N \leq 10^6$. It has gained substantially on the best of its competitors by the top of that range, however, and it is clear that it will eventually dominate both.

In Section 4, we show that asymptotically, no significantly better representation than Splay Trees can exist. We prove that in the cell probe model of computation, no *Tour* representation can do better than a worst-case amortized time of $\Omega(\log N / \log\log N)$ per operation. This leaves a theoretical gap of a factor of $\log\log N$, which we discuss in our concluding Section 5, along with other directions for further research. We also describe two proposed *Tour* representations from [7,24] that attain $O(\log N)$ time per operation in a strict worst-case rather than an amortized worst-case sense (and explain why they are unlikely to be competitive with splay trees in practice).

2. Three Alternative Tour Representations

2.1. The *Splay Tree* Representation

All the new representations we propose for the *Tour* datatype are based on trees. A natural first idea is to represent the tour by a binary search tree, with a city at each vertex along with a special *reversal bit* that indicates 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 2.1, 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. We shall refer to this action as *clearing* the reversal bit for v.)

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 then not too 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 can also implement *Flip* to run in a similar time and keep the tree balanced, thus yielding worst-case time $O(\log N)$ per operation [7]. We shall have more to say about this worst-case approach in Section 5. 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

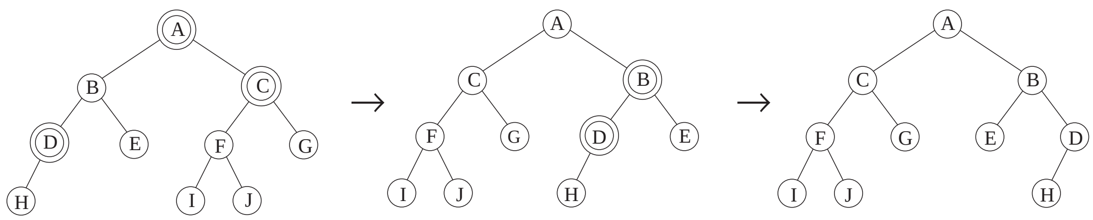

FIGURE 2.1. Three splay trees representing the tour (I,F,J,C,G,A,E,B,H,D).

reference. (The idea of using splay trees is a natural one, and has been independently proposed by several groups of researchers, for instance, Applegate and Cook [2].)

Splay trees were introduced by Sleator and Tarjan [28]. The key idea is that every time a vertex is accessed, it is brought to the root (*splayed*) by a sequence of rotations (local alterations of the tree that preserve the inorder traversal). Each rotation causes the vertex that is accessed to move upward in the tree, until eventually it reaches the root. The precise operation of a rotation depends on whether the vertex is the right or left child of its parent and whether the parent is the right or left child of its own parent. The change does not depend on any global properties of the subtrees involved, such as depth, etc. See [28,29] for detailed descriptions.

Sleator and Tarjan [28] showed that all the standard binary tree operations could be implemented to run in amortized worst-case time $O(\log N)$ using splays. In our Splay Tree *Tour* representation, the process of splaying is made slightly more difficult by the reversal bits. We handle these by preceding each rotation by a step that pushes the reversal bits down out of the affected area. Neither the presence of the reversal bits nor the time needed to clear them affects the amortized time bound for splaying by more than a constant factor. The *Tour* operations are implemented as follows.

To compute *Next(a)*, we begin by locating vertex *a* and splaying it to the root. We then traverse down the tree (taking account of the reversal bits) to find *a*'s successor, and then we splay the successor to the root. The cost of this procedure is clearly proportional to the cost of the splays. The *Prev(a)* query is handled analogously.

To compute *Between*(a,b,c), we locate vertices *b*, *a*, and *c* and splay them to the root one at a time, in that order. Note that, as described by Sleator and Tarjan, a splay operation on a given city will not increase the depth of any other city by more than two. This means that after splaying *b*, *a* and *c*, the depth of *a* is at most 2 and the depth of *b* is at most 4. Thus it now requires only a simple (constant-bounded) case analysis to determine if *b* lies between *a* and *c* in forward cyclic order. Indeed, all we need do is locate *b* in its new position in the tree and traverse up the tree until we encounter either *a* or *c*. The answer to our query is yes if either the first to be encountered is *a* and we arrive from the right, or the first encountered is *c* and we arrive from the left. Otherwise the answer is no. (The proof of this relies on the fact that we have cleared all the relevant reversal bits during the splay operations and, in the case where *a* is now a grandchild of *c*, on the fact that by the nature of the basic splay rotations [28,29], *a* must either be the right child of a right child or the left child of a left child.) The overall cost of the query is no more than the cost of the splays plus a constant.

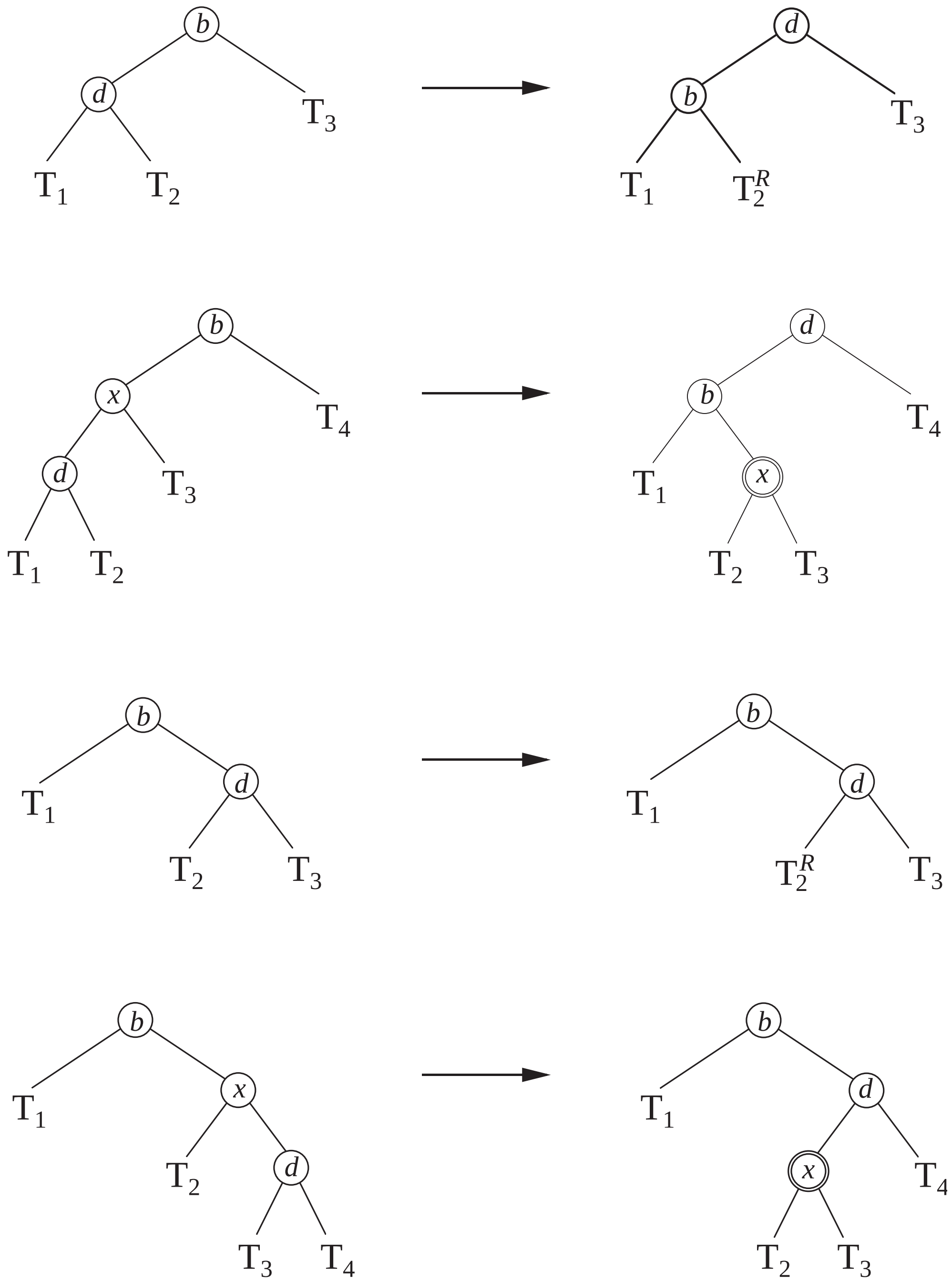

FIGURE 2.2. The four possible cases for *Flip* in the Splay Tree representation.

The *Flip* operation is the most complicated. To perform the operation *Flip(a,b,c,d)*, we begin by locating vertex *d* and splaying it to the root and then locating and splaying vertex *b*. This leaves the tree in one of the configurations shown on the left side of Figure 2.2, where the T_i's denote rooted subtrees. Because reversal bits were cleared before the last rotation, neither *b*, *d*, nor the intermediate node *x* (if it exists) has its reversal bit on after the splays. The definition of *Flip* requires the reversal of either the path forward from vertex d to vertex b or the path forward from a to c. In the first two cases in Figure 2.2, vertex d precedes vertex b in the inorder traversal, and we can flip the $d-b$ path by doing the rearrangement shown. (Here T_i^R denotes subtree T_i with the reversal bit for its root complemented. As in Figure 2.1, vertices with their reversal bits on are represented by double circles.) In the second two cases, vertex d follows vertex b, and the $a-c$ path is precisely the segment following b and preceding d. We can flip this path by doing the rearrangement shown.

Bounding the cost of the *Flip* operation is more problematic than was bounding the costs of the *Next*, *Prev* and *Between* queries, as this operation actually alters the tree by operations other than splays. We shall thus have to go into more of the details of the amortized time bound proof of [28]. That proof uses a potential function Φ, which is always nonnegative. The value of this function is initially $\Theta(N)$. Thereafter, for each splay operation the amount of work done plus the increase in Φ is bounded by $3\log N + 1$. (Throughout this paper, all logarithms are assumed to have base 2.) That is, if the potential function increases, it can increase by no more than the amount by which $3\log N + 1$ exceeds the work actually performed. If the work performed exceeds $3\log N + 1$, the potential function must drop by at least a corresponding amount. We thus can view $\Phi(T)$ as the current balance in a back account. Each splay operation that uses more than its quota of time can be viewed as drawing credits from the account to cover the excess, using credits initially in the account or saved on previous operations. At the end of any sequence of m operations, the total time spent on splay operations will thus be at most $m(3 log\ N + 1)$ plus the initial potential. The initial potential is proportional to the cost of constructing the initial tree, so that part of the total cost can be assigned to the tree construction. The remaining cost can be spread among the m operations, so the amortized cost per operation is $O(\log N)$.

The function Φ is actually fairly simple. For each vertex v in the tree, let $w(v)$ be the number of vertices in the subtree rooted at v and $rank(v) = \log(w(v))$. Then for a given tree T,

$$\Phi(T) \quad = \quad \sum_{v \in T} rank(v)\,.$$

It is easy to see that if T is an N-vertex balanced binary tree, then $\Phi(T) = \Theta(N)$ under this definition. (The argument is essentially the same as the one used in [1] to show that a heap can be constructed in linear time.) Furthermore, since no vertex can have rank exceeding $\log N$ and no surgery on the tree can affect the rank of any vertex below the point of surgery, none of the changes we make to the tree as illustrated in Figure 2.2 can increase the value of Φ by more than log N. The cost of the surgery itself is bounded by a fixed constant B. If we charge log $N + B$ for the surgery, this is enough to cover both the actual cost and any change in potential. So the amortized cost of the surgery is $O(\log N)$, just as it was for the splays. The amortized cost for a complete *Flip* is then also $O(\log N)$, since the time needed is simply that for the surgery plus two splays and a constant amount of overhead. We conclude that the amortized cost for any sequence of *Next*, *Prev*, *Between*, and *Flip* operations in our Splay Tree representation will be $O(log\ N)$ per operation.

2.2. The *Two-Level Tree* Representation

The idea of using a tree with just two levels to represent the *Tour* datatype was suggested by Tom Leighton [20], 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 also occurred independently to Chrobak, Szmacha, and Krawczyk at about the same time [7].) In particular, the *Two-Level Tree* representation is designed to take advantage of the fact that our algorithm makes significantly more calls to *Prev* and *Next* than to *Flip*. All the query operations are performed in constant time (as in the Array representation, albeit with slightly larger constants), whereas *Flip* operations have a worst-case cost of $O(\sqrt{N})$ per flip.

The tour is divided into roughly $\sqrt{N}$ segments, each of length in the range $[\sqrt{N}/2, 2\sqrt{N}]$. Each segment is maintained as a doubly-linked list (using pointers labeled *previous* and *next*). See Figure 2.3. 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. By using the reversal bit, we can reverse the orientation of a whole segment in constant time, leading to an efficient approach to the *Flip* operation. In addition to the *next* and *previous* pointers, each member of a segment contains the index of the city it represents and a sequence number that gives 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 the segment uses its *previous* pointer to point to its tour neighbor that is not in the segment, and the last vertex uses its *next* pointer similarly. Also, as in our Splay Tree representation, the structures representing the cities actually reside in an array, so that the location of a city in the two-level tree can be found in constant time by indexing on its name.

The parent nodes of the segments are themselves connected in a doubly-linked list, and each node contains a sequence number that represents its position in the list. (As with the vertices on the lower level, the sequence numbers are consecutive but need not start with 1.) Each parent also contains a count of the number of cities in the segment it represents and pointers to the segment's two endpoints. (As described here, the Two-Level Tree representation is significantly more elaborate than those envisioned in [7,20], which did not consider the issue of efficiently implementing the *Between* query or attempt to reduce the constant bound on the time for *Nexts* and *Prevs*.)

The *Tour* operations are implemented as follows.

To compute *Next(a)*, we begin by locating vertex *a*. We then follow the pointer to *a*'s parent and look at its reversal bit. If the reversal bit is off, then *Next(a)* is found by following *a*'s *next* pointer; otherwise it is found by following *a*'s *previous* pointer. This operation takes constant time.

To compute *Between*(a,b,c), we locate the three vertices and their parents. If all three parents are distinct, the answer can be determined from the sequence numbers of the parents. If two or more of the vertices have the same parent (i.e. lie in the same seg-

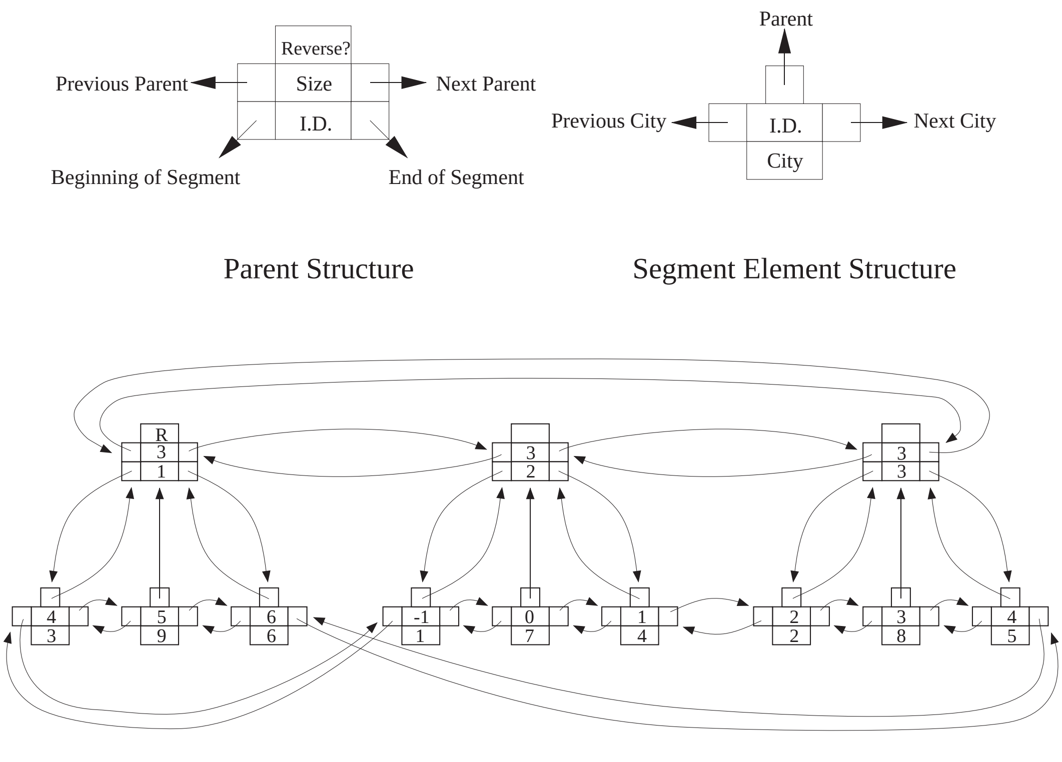

FIGURE 2.3. The Two-Level Tree *Tour* representation.

ment), the answer depends on the sequence numbers within that segment and on the parent's reversal bit. (If all three vertices lie in the segment, the sequence numbers of all three are significant; if only two lie in the segment, the location of the third is irrelevant.) For example, suppose that *a* and *b* lie in the same segment, *c* lies in some other segment, and that *a* has a lower sequence number than *b* within their common segment. If the reversal bit of the parent is off, then the answer is *true*, because *b* follows *a* within the segment and therefore *b* precedes *c* in a forward traversal from *a*. If the reversal bit is on, then *a* follows *b* within the segment and the result is *false*. The other cases are similarly straightforward, but there are quite a few of them, and so this implementation of *Between* is somewhat more expensive than the simple test used in the Array representation. The worst-case time is still bounded by a constant, however.

The *Flip(a,b,c,d)* operation is the most complicated. The two simplest cases are when the path to be flipped lies entirely in one segment or when it is made up entirely of complete segments. First suppose that either the $d-b$ path or the $a-c$ path lies within one segment of the two-level tree. In this case we do the flip locally within the segment containing the path. The *next* and *previous* pointers are swapped within each internal vertex of the path to be flipped, and appropriate changes are also made to the pointers in the path endpoints and in their non-path neighbors. The total time is proportional to the length of the path and hence $O(\sqrt{N})$.

Suppose next that the $d-b$ and $a-c$ paths each consist of a sequence of consecutive segments. (Either both must, or neither.) In this case *a* and *b* must lie in distinct segments, as must *c* and *d*. The problem is thus reduced to that of reversing a sequence of segments. Either path may be flipped, and one can use the sequence numbers of the parent nodes of *a*, *b*, *c*, and *d* to choose the path that involves the smaller number of segments. The actual flip involves (1) reversing the *next* and *previous* pointers in each internal parent node along the path, (2) making appropriate changes to the pointers in the first and last parent nodes in the path and in their non-path neighbors, (3) flipping the reversal bit of each affected parent node, (4) appropriately adjusting the *next* and *previous* pointers for the cities at the ends of the two paths, and (5) updating the sequence numbers of the affected parent nodes. The total time is proportional to the number of segments in the path and hence is $O(\sqrt{N})$.

If neither of the above two cases apply, we can rearrange the segments to ensure that the second case does. At least one of the pairs (b,a), (c,d) must lie within a single segment of the two-level tree. Suppose that *b* and *a* do. Since they are tour neighbors, they are also neighbors within the segment, so we can split the segment between *b* and *a*, creating two new segments, at a cost proportional to the length of the shorter of the two new segments. (We create a new parent node and make it the parent of this shorter segment, leaving the members of the longer segment with their original parent.) If *c* and *d* also lie within a single segment, we similarly split that segment between *c* and *d*. Given the upper bound on the sizes of the segments before splitting, the total time for splitting is $O(\sqrt{N})$. We are now back to the situation where both paths are composed entirely of complete segments, and we already know how to handle this, again in time $O(\sqrt{N})$ because of the bound on the total number of segments.

At this point, our two-level tree may have become unbalanced and the top sequence numbers may have been corrupted. This can all be fixed with an additional $O(\sqrt{N})$ work as follows: First, we consider in turn each of the (possibly 4) newly created segments. If the size of that segment is less than $\sqrt{N}/2$, we merge it into its smaller neighboring segment in time proportional to its length. At the end of this merging process all segments will be of size at least $\sqrt{N}/2$, but we may now have one of the merged segments larger than $2\sqrt{N}$. If any such large segment has been created, it can be no larger than $3\sqrt{N}$, the worst case being when components almost as large as $\sqrt{N}/2$ were merged into it from both sides. If we split it in half, we will thus obtain a two components of size between $\sqrt{N}$ and $1.5\sqrt{N}$, which will be within our required range. The splitting process will take time $O(\sqrt{N})$, and then all that remains is a pass over the entire set of parent nodes to fix up their sequence numbers, again an $O(\sqrt{N})$ operation.

Thus the worst-case time for each operation is $O(\sqrt{N})$. This worst-case bound is attained at a cost of considerable overhead in the rebalancing operations, however. Given the conventional wisdom that data structures often stay roughly balanced in practice, it might be worthwhile first to consider variants on this representation that give up the worst-case guarantees in exchange for lower overhead, and only implement full-scale rebalancing should it be seen to be necessary. Given that our main goal in this study was to find representations that worked well in practice, this is the approach we have taken. (As we shall see, it turns out that thorough rebalancing is *not* necessary in practice.)

Here is how our implementation differs from the idealized one given above. Rather than rebalance as indicated above, we simply maintain a fixed number of segments, chosen so that the average segment size is close to a specified number *groupsize* of cities. When we have to split a group in the course of doing a flip, we immediately (before flipping) merge the smaller half with its neighboring segment, thus keeping the number of groups fixed. Note that this raises the possibility that, as a result of the merge, one of our two choices of a path to flip may now be entirely contained in the new merged segment, in which case we can perform the flip entirely within that segment.

Our action when one of the paths to be flipped is entirely contained within a single segment is also more complicated than in the original plan. (The added complication is for the purpose both of saving work and of obtaining a bit of implicit rebalancing.) If the path to be flipped is of length no more than $(3/4)$ *groupsize* we proceed as before. If it is longer, however, we first split off the two ends of the segment so that all that is left is the path, and we then merge the split off parts with their neighboring segments. This reduces us to a trivial instance of the case where the path to be flipped consists entirely of complete segments (in this case just one such segment), and we proceed as in that case. Note that, assuming our tree is relatively well-balanced, the cost of this alternative should not be too much more than performing the reversal without doing the splits, and it may be much less. Moreover, we may get a bit of rebalancing as a side-effect, since overly long segments should be more likely to completely contain paths and thus be eligible for splitting. Despite this implicit rebalancing, however, there is no guarantee that a few segments might not grow very large, causing *Flip* operations to take $\Omega(N)$ time. Thus we have indeed given up our $O(\sqrt{N})$ worst-case guarantee, and the question to be considered is whether we will pay the price in practice.

2.3. The *Segment Tree* Representation

The idea of a *Segment Tree* representation was first proposed by David Applegate and Bill Cook [2] to exploit a feature of the Lin-Kernighan algorithm. In the inner loop of this algorithm, a sequence of tentative *Flips* is applied to the current best tour (the *permanent* tour), with the hope of finding an improved tour. If a shorter tour is found, an initial substring of the tentative *Flips* is applied to the permanent tour, otherwise they are all discarded. In general there are many more tentative *Flips* than permanent ones. The *Tour* representations discussed so far do not take advantage of the distinction between tentative and permanent *Flips*; they perform all *Flips*, both tentative and permanent, in the structure they use to represent the permanent tour. In order to discard a sequence of tentative *Flips,* these implementations perform a second (reverse) sequence of *Flips* that step-by-step undoes the effects of the first sequence.

One advantage of the Segment Tree representation is that it avoids this work of undoing tentative flips. A second advantage comes if one imposes a fixed bound on the allowed length of a sequence of tentative *Flips*. As defined in [23], the pure Lin-Kernighan algorithm imposes only an implicit bound of N on the number of tentative flips in a sequence. Imposing a fixed bound typically does not have a significant effect on the length of the output tour, however, so long as the bound is relatively large, say 50, as in the Applegate-Cook implementation [2]. (For the random Euclidean instances profiled in Table 1.2, only about 0.6% of the improving moves found when no bound was imposed involved sequences of more than 50 tentative *Flips*; the average improving move involved just four or five *Flips*.) Imposing a fixed bound on the number of tentative *Flips* in a sequence gives a benefit to the Segment Tree representation, because under this constraint Segment Trees can implement those tentative *Flips* at a cost of $O(1)$ per operation. There is a drawback, in that the cost of permanent *Flips* increases to $\Omega(N)$, but in practice there are far more of the former than the latter. (For our random Euclidean instances, the number of permanent *Flips* grows roughly as $0.6N$, compared to a growth rate of roughly $30N$ for the tentative ones.)

Here are the details: Our Segment Tree representation is a hybrid, using our standard Array representation for the permanent tour and an auxiliary *segment tree* and *segment list* to keep track of the ''tentative'' tour derived from the current sequence of tentative *Flips*. (See Figure 2.4.) Each vertex of the segment tree corresponds to a segment of the permanent tour, identified with a closed interval of positions in that tour, for instance $[j,k]$, $1 \le j \le k \le N$. The indices j and k are stored in the structure representing the vertex. The segment tree is structured so that an inorder traversal gives the segments in the order in which they occur in the permanent tour. The segment list is a doubly-linked list, each member of which corresponds to one of the segments and contains the indices of the segment's endpoints as well as a reversal bit. The representative of a segment in the segment tree contains a pointer to the representative in the segment list. (Alternatively, we can have a single structure as the segment's representative, one that contains both tree and list pointers, although conceptually it is easier to think of the representation with the two structures separate, as in Figure 2.4.) The tentative tour is derived from the segment list in the obvious way: one goes through the segments in the order they occur in the list, with the direction of traversal through an individual segment determined by its reversal bit.

As with our other representations, we need a way to find a given city in the representation of the tentative tour. This is made more complicated here since our representation does not contain objects that correspond to individual cities. Thus we need to look for the segment that contains the city, not the city itself, and the identity of this segment is subject to change. The purpose of the segment tree is to facilitate this search. We first look up the city in the array B of the Array representation for the permanent tour, which gives the city's index in the permanent tour. We then traverse downward from the root of the segment tree until we find the segment containing the index of the desired city. The time required is no more than the total number of segments, which as we shall see is bounded by $2K+1$ (where K is the maximum length allowed for a sequence of tentative flips) and therefore is bounded by a constant (albeit a large one) when $K=50$. As we

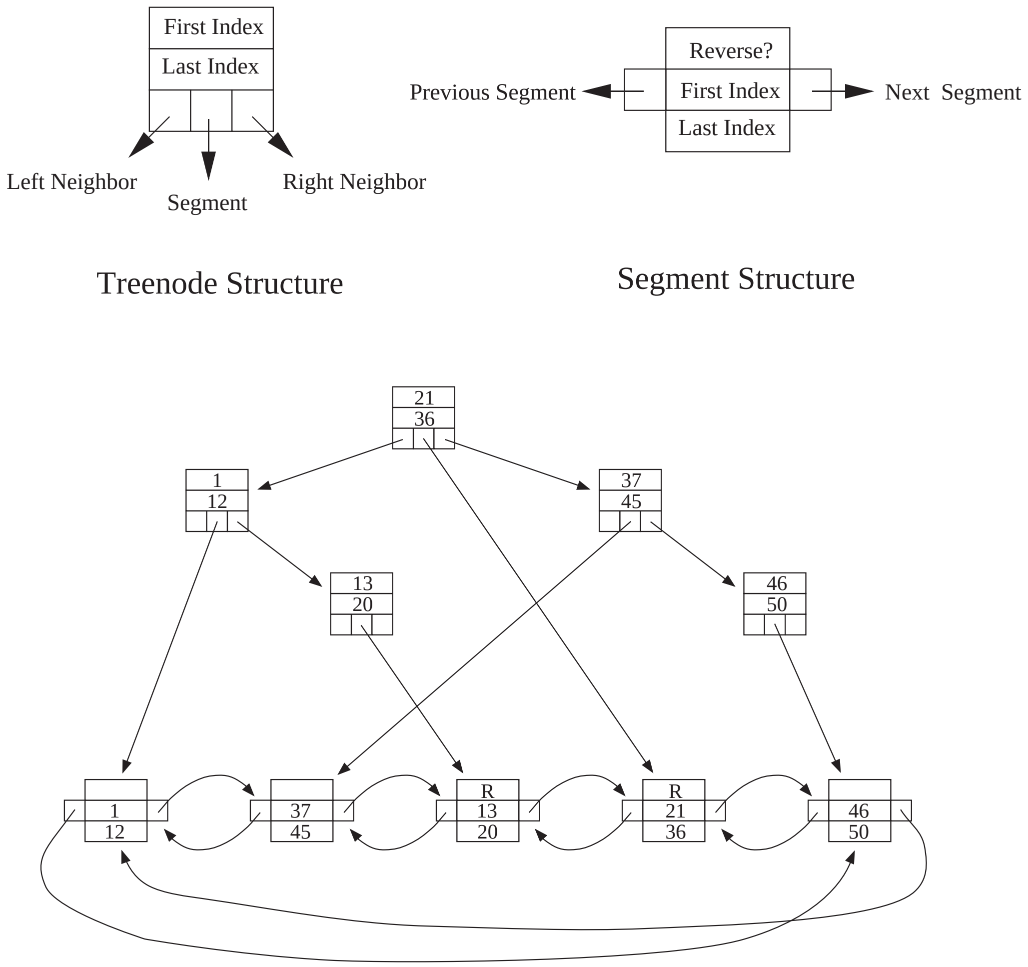

FIGURE 2.4. The Segment Tree *Tour* representation.

shall see later, the time for a search can be considerably reduced in an amortized sense by using an auxiliary data structure.

We shall now discuss how the Segment Tree representation works. Our discussion has three parts. First we describe how the segment tree and list are initialized each time we begin a new sequence of tentative flips. Then we describe how the various operations are implemented when they are applied to the tentative tour. Finally, we describe how we update the permanent tour when a permanent change is required.

To prepare for a new sequence of tentative flips, we simply create a tree consisting of a single root vertex that represents the segment $[1,N]$ (the entire tour). The corresponding (single) member of the doubly-linked list has its reversal bit off and is its own successor and predecessor. This takes constant time.

To compute $Next(a)$ (or $Prev(a)$) for a given city a, we first find the tree node for the segment currently containing that city. We then go to the corresponding member of the linked list of segments and take note of the reversal bit for that segment and the indices of its endpoints. Suppose that city a's index in the permanent tour is $B[a] = i$, and its current segment is $[j,k]$. If the reversal bit for the segment is off and $i < k$, $Next(a) = A[i+1]$, the city following a in the current tour; if the reversal bit is on and $i > j$ and $Next(a) = A[i-1]$. If the reversal bit is off and $i = k$, then we must look at the next segment, taking its left endpoint to be $Next(a)$ if that segment's reversal bit is off, and otherwise taking its right endpoint. If the reversal bit is on and $i = j$, we must look at the previous segment and proceed analogously. The answers to $Prev(a)$ are determined similarly. The time in all cases is constant, although a bigger constant than that for Arrays or Two-Level Trees.

In our implementation of Lin-Kernighan, the $Between(a,b,c)$ operation is only applied to the permanent tour, and so in the Segment Tree representation it has the same implementation as it did in the Array representation. It thus takes time $O(1)$.

To perform $Flip(a,b,c,d)$ in the tentative tour, we locate the tree nodes for each of the cities involved. If a and b lie in the same segment, we split the segment between them so that they lie in adjacent segments of the tentative tour. In particular, suppose the segment was $[j,k]$ and the indices for a and b were i and $i+1$. (They must be consecutive since the two cities are tour neighbors.) The new segments are $[j,i]$ and $[i+1,k]$. The member of the doubly-linked list for $[j,k]$ is replaced by the two new members, with reversal bits inherited and with the two segments themselves ordered according to the original reversal bit. The vertex of the tree for $[j,k]$ is replaced by the vertex for the larger of the two subsegments. If that segment is $[j,i]$, then the new vertex created for $[i+1,k]$ becomes its right child and has for its right child the original right child of $[j,k]$. Otherwise $[i+1,k]$ replaces $[j,k]$ and $[j,i]$ becomes its left child, inheriting the left child of $[j,k]$. The split takes only constant time. Similarly we force c and d into separate segments if they aren't already. After these splits, the paths $d-b$ and $a-c$ do not share any segments. We then reverse the order of segments in the segment list for one of these paths. The order of cities within the affected segments is reversed by flipping the reversal bits on the corresponding segment list nodes.

Note that each *Flip* operation creates at most two new segments, and so, as claimed, the total number of segments can never grow to more than $2K+1$, where K is a bound on the number of tentative *Flips* that can be performed in a sequence. Thus the worst-case time to perform a *Flip* is also $O(1)$.

Let us now explain how we can speed up the process of finding a city in the segment tree. We use an additional array of *location pointers*, one for each city. Initially, before a sequence of tentative flips begins, all location pointers are set to the single vertex representing $[1,N]$ at the root of the segment tree. (We use a time-stamping procedure so as to avoid having to initialize all the pointers individually. By default any location pointer points to the root unless it was reset more recently than the last initialization.) Whenever we look for a city in the tree, we start at the vertex to which its location pointer points. If the corresponding segment still contains our city, we are done; otherwise we traverse downwards until we find the vertex for the segment currently containing the city, and set the location pointer to point at this vertex. (Our vertex-splitting procedure insures that a vertex's current segment is always in the subtree rooted at the vertex that represented its previous segment.) Thus the total time spent on searches for a given city during a sequence of tentative flips is at most proportional to K plus the number of searches made for that city. This amortizes the $O(K)$ cost over the entire sequence of searches rather than charging it to each search individually.

We now discuss what is to be done when a sequence of tentative *Flips* has been completed. If an initial substring of this sequence yielded an improving move, we need to update the arrays representing the permanent tour. We shall call this the *MakePermanent* operation. Typically, we implement this by re-enacting the relevant sequence of *Flips*, this time by using the *Flip* operation in our Array representation, a process that in the worst case can take time $\Theta(KN)$. Applegate and Cook [2] implemented an alternative method that would be preferable if a large number of *Flips* were required or if the instances were such that *Flips* came close to their worst-case time bounds. Their idea is simply to undo all the tentative flips that follow the improving initial substring, a process that takes time at most $O(K^2) = O(1)$, and then to read off the new permanent tour directly from the segment list in time $\Theta(N)$.

3. Experimental Comparisons

3.1. Implementation Details

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 R3000 MIPS processor as

CPU. Roughly 24 megabytes of memory were required to run a 100,000-city instance without excessive paging. Instances with 1,000,000 cities required 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. (This profiling does not exact a significant penalty on running times.) 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.

Before presenting our experimental comparisons, we must first say a few more words about the actual implementations, since certain key details were left open in the previous section. In deciding on these details for each representation, we were guided by the goal of determining how well the representation could do ‘‘in practice’’ for instances with N in the range covered by our study. Thus decisions that could be justified by our asymptotic analysis were sometimes abandoned if they appeared to have no practical impact (or a negative one) on instances of the size and type being considered. We were able, however, to gain some insight into how large N must be for asymptotic effects to become visible, and we will discuss this issue as well.

The Segment Tree representation is of course an embodiment of this approach, since asymptotically it can be as bad as Arrays (and the results we shall present indicate that asymptopia has already begun to set in by $N = 1,000,000$). There are two details that were left open in the previous section, however. First the lesser one: we mentioned in Section 2 that we could either have two representatives for each segment, one in the segment tree and one in the segment list, with a pointer from the first to the second (as in Figure 2.4), or we could have one common structure for the segment, containing both the tree and list pointers and not duplicating the other information. The latter approach is more space-efficient and should in principle be slightly faster, as we would no longer have to manipulate the pointer linking the two representations. Based on limited experimentation, however, the speedup is negligible, so the results we report should not depend on which approach was taken. (They were in fact generated using the former approach. The extra space required was not a handicap on our machines, and we did not get around to testing the latter until the main experiments were completed.)

The second choice we had to make in our Segment Tree implementation was the method for performing the *MakePermanent* operation. For random Euclidean instances, our implementation makes changes in the permanent tour by performing *Flips* in that tour's representation, rather than by rebuilding that representation from scratch based on the tentative tour's segment list, the other method we discussed in Section 2. This choice is based on (1) our earlier observation that the actual number of *Flips* performed averages about 5, (2) the fact that the time for a *Flip* tends to grow as $o(N)$ for instances of this type, and (3) the fact that the alternative method always takes time $\Theta(N)$. We experimented with the hybrid method mentioned in Section 2, where one uses the *Flip*-based approach so long as the number of *Flips* to be performed is below a threshold and otherwise takes the $\Theta(N)$ approach. For these instances, however, performance was degraded unless the threshold was set so high that the second option practically never occurred.

As already mentioned in Section 2, our Two-Level Tree implementation also has the potential of being as bad as the Array representation, for we have chosen to forego the overhead of explicitly rebalancing the segments, thus giving up the asymptotic worst-case guarantee of $O(\sqrt{N})$ time per operation. Instead, we simply maintain a fixed number of segments that we hope (but cannot guarantee) will stay reasonably balanced. The main detail to be specified here is the value of *groupsize*, the average size of a segment. In initial experiments, we were surprised to discover how insensitive our results were to the actual value of *groupsize*, at least for $N \leq 100{,}000$. For instance, roughly the same overall running times for *Tour* operations were obtained for values of *groupsize* ranging from 40 to 120, and only when *groupsize* was as large as 200 was real performance degradation noticeable. For simplicity, we thus fixed *groupsize* at 100 for all instances with $N \leq 100{,}000$. For $N = 1{,}000{,}000$, however, *groupsize* $=$ 100 was clearly too small, and the time per *Flip* was reduced by 27% when we increased *groupsize* to 200, which is the value used for $N = 10^6$ in the table. (Further increases in *groupsize* did not offer significant improvements, with performance remaining relatively stable for values up to 800.)

It would be nice if, given N, we could predict the optimal value for *groupsize*, but the observed insensitivity suggests that our trial-and-error approach is the best that can currently be hoped for. We did compile statistics on the average number of vertices touched per flip on the top and bottom levels of the trees, and these varied in predictable ways with *groupsize*. (The value for the top level, for instance, roughly tracking the number $\lceil N/groupsize \rceil$ of segments.) Actual performance was relatively insensitive to these values, however, beyond the fact that it appeared to be a bad idea to let either value be a large multiple of the other (say 3 or more).

Our biggest compromises involved the Splay Tree representation, with many of the results we report being for a version that differed significantly from the theoretical model presented in Section 2. In particular, no splays were performed in the *Next* and *Prev* operations under the Splay Tree representation; a simple traversal scheme was used instead. For instance, to compute $Next(a)$ we first find a in the tree and then walk up the tree to the root keeping track of reversal bits so as to discover the direction of the tour at city a. We then traverse in the appropriate direction from a to find a's successor. We thus replace the splay of a by a (significantly faster) traversal and omit the splay of $Next(a)$ entirely. In addition, *Between* was implemented with the first splay, that of city b, omitted. (The remaining two splays in the *Between* implementation described in Section 2 are crucial to its operation and could not be omitted, as are the two splays in our implementation of *Flip*.)

For the range of N being studied here, it turns out that the extra overhead involved in splaying (versus a simple traversal, or nothing at all) is substantial, and the resulting rebalancing of the tree does not justify the cost. The nature of the speedups due to foregoing splay operations is revealed in Table 3.1, which gives profiling information on the IRIS 4D/250 for four different versions of the implementation on the random Euclidean instances discussed in Section 1. *Splay Tree-3* performs all the splays specified in Section 2, *Splay Tree-2* omits only the splay of city b in $Between(a,b,c)$, *Splay Tree-1* omits in addition the splays of the cities $Next(a)$ and $Prev(a)$ in *Next* and *Prev* queries, and *Splay Tree-0* in addition omits the splay of a in those operations.

Observe that each removal of a splaying operation yields a decrease in the total time for *Tour* operations, with the operation from which the splay was removed benefiting the most. For instance, the cost per *Between* operation drops by roughly 30% when the splay of city b is omitted. The cost per *Next*/*Prev* at $N = 100,000$ drops by 10% when the splay on the result is omitted, and by more than 30% when the splay on a is replaced by a traversal. It is perhaps surprising that the latter change has more of an effect than the former. A major reason is that the splay on the result can be (and was) much more efficiently implemented than the splay on a, and so there is less to save by omitting it. For instance, while we are searching down the tree for $Next(a)$ we can clear reversal bits as we go down. This is more efficient than clearing them during the splay rotations on the way back up, as a bit cleared during one rotation may be reset during the next. Fur-

		μSec per Call			Total Seconds		
	N	10^3	10^4	10^5	10^3	10^4	10^5
Splay Tree-3:	*Next* + *Prev*	10.6	13.3	16.8	1.8	21.4	262
	Between	17.3	19.4	22.7	0.4	3.9	44
	Flip	11.4	12.3	13.7	0.8	7.7	83
	Total				3.0	33.0	389
Splay Tree-2:	*Next* + *Prev*	10.6	13.4	17.3	1.7	21.4	268
	Between	12.5	13.7	16.0	0.3	2.7	31
	Flip	11.4	12.5	13.7	0.7	7.8	83
	Total				2.7	31.9	382
Splay Tree-1:	*Next* + *Prev*	10.0	12.5	15.4	1.6	20.3	242
	Between	11.8	13.8	15.0	0.3	2.8	30
	Flip	10.5	11.4	12.2	0.7	7.2	74
	Total				2.6	30.3	346
Splay Tree-0:	*Next* + *Prev*	5.2	7.6	10.5	0.8	12.3	169
	Between	13.4	17.6	23.8	0.3	3.6	47
	Flip	10.5	11.7	13.3	0.7	7.4	82
	Total				1.8	23.2	298

TABLE 3.1. Times on an IRIS 4D/250 for various Splay Tree implementations of the *Tour* operations performed by the Lin-Kernighan algorithm.

thermore, clearing the bits on the way down will typically leave the city *Next*(a) as the last in a long sequence of left children of the right child of a, and so in splaying *Next*(a) up the tree we can in each rotation omit the tests for the local structure of the tree.

It should be pointed out, however, that the omission of the splays in one operation can have a negative impact on the other operations. For instance, omitting the splay in *Between* causes the *Next*/*Prev* operations to slow down slightly. Omitting the splay on a in *Next*(a) and *Prev*(a) causes the time for *Between* to increase greatly. This is due not so much to the rebalancing effect of the splays as to the fact that they enhance the ability of the *Tour* representation to take advantage of locality of reference. Typically when we invoke *Between*(a,b,c), we have recently performed one or more *Next* and *Prev* operations in which a, b, or c was involved. Thus if they were splayed to the top of the tree during the *Next*/*Prevs*, they will still be near the top of the tree when *Between*(a,b,c) is invoked. This is illustrated in Figure 3.1, which reports the average depth at which the relevant vertices are encountered in the tree as a function of the operation and of the number of cities under Splay Tree-0 (the dashed lines) and Splay Tree-2 (the solid lines), with *N*, *B*, *F* representing *Next*/*Prev*, *Between* and *Flip* operations respectively.

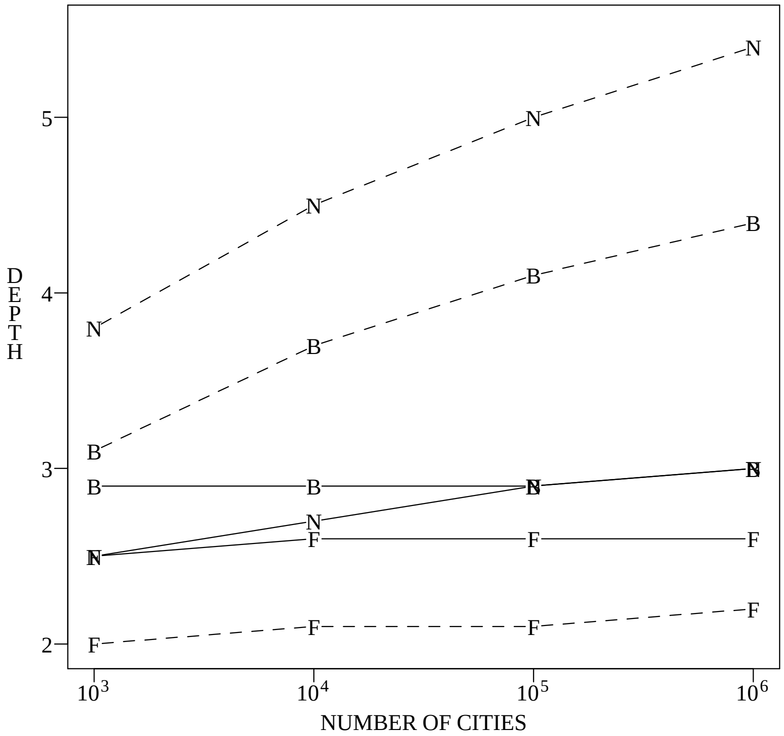

FIGURE 3.1. Growth rates for the depth of cities.

Note that even when no splaying is done on *Next* and *Prev*, the growth in the depth still seems to be $O(\log N)$, which is the worst case when splaying *is* performed. Under (almost) full splaying (Splay Tree-2) the exploitation of locality of reference is so strong that the average depths are essentially constant from 1,000 to 1,000,000 cities. (The only difference for Splay Tree-3 would be that the line for *Between* would be higher and the one for *Next/Prev* would be slightly lower.) The average depths for *Flips* under Splay Tree-0 is less than that under Splay Tree-2 because typically most *Flips* are restricted to the same small key set of endpoints, and performing splays on *Next/Prevs* 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*, however, will eventually be dominated by its growing disadvantage on the more frequently invoked *Nexts* and *Prevs*.

3.2. Comparisons on Random Euclidean Instances

We are now ready to look at our data comparing the Splay Tree representation and its competitors. We begin with the random Euclidean instances described in Section 1. In Tables 3.2 through 3.4, we summarize the profiling information on our three machines for the Array, Splay Tree-0, Two-Level Tree, and Segment Tree representations.

	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-0:								
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
Tentative 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 3.2. Times on an IRIS 4D/250 for *Tour* operations.

Let us begin by looking at the results for the IRIS (Table 3.2) and then consider how the results for the other machines differ. Note that on the IRIS the Array representation as expected has the most efficient implementation of *Next/Prev*, although Two-Level Trees are close. Even without splaying on *Next* and *Prev*, our Splay Tree-0 implementation is by far the worst for these operations (as it is for *Betweens*, where Arrays are again the winner). For *Flips*, on the other hand, Splay Trees show their expected dominance over Arrays and Two-Level Trees, and indeed the times for their *Flips* even beat out the times for tentative *Flips* in Segment Trees, which theoretically are $O(1)$ and hence should asymptotically be faster. (Theory, however, does not take into account the great benefit we obtain for *Flip* under Splay Tree-0 from locality of reference, as illustrated in Figure 3.1.) Indeed, in the Splay Tree representation *Flips* are even less expensive than *Betweens* (a fact that held true for all four alternatives in Table 3.1). One other remark about *Flips* concerns the Two-Level Tree results. Note that despite the lack of explicit rebalancing in our implementation of this representation, the time per *Flip* is still growing more slowly than $\sqrt{N}$ for $N \le 10^6$.

Another observation is that the *MakePermanent* operation for Segment Trees is even worse than expected. The time per operation is growing at a superlinear rate (although the number of calls is so small that the effect on overall running time is effectively masked until $N > 10,000$). This is because of the phenomenon, already observed in Section 1, that the MIPS processor of our IRIS 4D/250 has rapidly degrading performance for *Flips* in the Array representation as N gets large, presumably because of an increasing percentage of cache misses. (This phenomenon hits the alternative $\Theta(N)$ rebuild-from-scratch method even harder, however, so that updating the permanent tour using *Flips* is still to be preferred.) Nevertheless, Segment Trees are reasonably competitive with Two-Level Trees in the total time spent on *Tour* operations when $N \le 10,000$. We should also note that the `-O` optimization compile flag on the IRIS was crucial to the dominance of Two-Level Trees over Segment Trees when $N \le 100,000$. Compiling without optimization degrades the performance of Two-Level Trees much more than it does the performance of Segment Trees, and their ranking reverses in this case. As our profiling information makes clear, however, the growing cost of the *MakePermanent* operation is likely to doom the Segment Tree representation as N increases substantially beyond 100,000, no matter what optimization flags we use.

Neither of our two other machines had enough memory to run our million-city instance, so their tables only go up to $N = 100,000$. Perhaps the most interesting thing to note about the VAX results (Table 3.3) is that the times per *Next*/*Prev*, *Between*, and *Flip* in the Segment Tree representation all appear to be essentially constant as N increases, perhaps because of the VAX's much more limited reliance on caches. Note that the time for *Flips* in the Splay Tree representation also seem to be essentially constant. This is inconsistent with theory, but it mirrors our experimental results on splay depth (Figure 3.1) more closely than do our results for the other two (faster) machines. One anomaly for which we have no good explanation is the fact that on this machine *Betweens* are consistently more expensive for Segment Trees than they are for Arrays, even though our Segment Tree and Array representations share the same code for implementing this operation. (This phenomenon does not appear to occur on the other two machines, where the differences between times for *Betweens* seem more to be the results of random fluctuations.)

Finally we should note that while both Two-Level Trees and Segment Trees do significantly better than Splay Trees in overall *Tour* operation time, here Segment Trees significantly outperform Two-Level Trees at $N = 100,000$, a reversal of the situation on

	N	μSec per Call 10^3	10^4	10^5	Total Seconds 10^3	10^4	10^5
Array:	*Next + Prev*	3.3	3.5	3.8	0.5	5.8	62
	Between	4.9	5.6	5.4	0.1	1.2	11
	Flip	51.0	214.5	1347.8	3.4	137.7	8353
	Total				4.0	144.7	8426
Splay Tree-0:	*Next + Prev*	25.9	33.1	34.9	4.0	52.5	513
	Between	50.2	57.9	64.9	1.1	12.2	132
	Flip	36.7	40.0	40.0	2.4	25.6	243
	Total				7.5	90.3	888
Two-Level Tree:	*Next + Prev*	4.6	5.0	5.5	0.8	8.4	91
	Between	10.8	11.6	12.3	0.3	2.3	24
	Flip	43.6	49.4	70.5	2.9	30.8	431
	Total				4.0	41.5	546
Segment Tree:	*Next + Prev*	13.0	12.8	12.7	2.1	19.6	195
	Between	6.1	6.3	6.5	0.1	1.3	13
	Tentative Flip	47.6	49.7	47.2	1.6	15.6	148
	MakePermanent	320.9	1238.8	9374.4	0.1	2.2	163
	Total				3.9	38.7	518

TABLE 3.3. Times on an VAX 8550 for *Tour* operations.

the IRIS. These distinctions are significant, as variance analysis implies that the averages reported here are probably within ±3% of the true values (based on 95% confidence intervals).

Note that the comparison between Two-Level Trees and Segment Trees is somewhat problematic, given that the two are embedded in slightly different versions of the Lin-Kernighan algorithm, one with no bound on the length of a sequence of *Flips*, and one with a bound of 50. This does not appear to be the source of the running time distinctions we have made, however. When we impose the bound of 50 on the Two-Level Tree representation, running times for random Euclidean instances change only by 2-4%, which is not a significant difference in this context.

For the RISC-based SPARCstation results (Table 3.4) one sees much the same qualitative behavior as we saw for the IRIS. The SPARCstation seems to be susceptible to the same cache-miss problem we discussed for the IRIS, although perhaps not as badly. Note that although the SPARCstation results tend to be slower than those for the IRIS for $N = 1,000$, they are significantly faster for $N = 100,000$. This effect seems to benefit Segment Trees more than Two-Level Trees, so that the lead of the latter over the former at $N = 100,000$ on the IRIS shrinks considerably.

		μSec per Call			Total Seconds		
	N	10^3	10^4	10^5	10^3	10^4	10^5
Array:	*Next* + *Prev*	1.0	1.3	1.3	0.2	2.0	20
	Between	2.0	1.9	2.3	0.0	0.4	4
	Flip	22.3	127.7	642.1	1.4	79.9	3876
	Total				1.6	82.3	3900
Splay Tree-0:	*Next* + *Prev*	5.9	7.1	7.9	1.0	11.3	125
	Between	16.2	18.5	21.2	0.8	3.7	42
	Flip	12.3	13.4	14.2	0.4	8.4	87
	Total				2.2	23.4	254
Two-Level Tree:	*Next* + *Prev*	0.9	1.1	1.2	0.2	1.8	20
	Between	3.0	3.3	3.6	0.1	0.7	7
	Flip	11.9	14.3	23.8	0.8	8.9	145
	Total				1.1	11.4	172
Segment Tree:	*Next* + *Prev*	3.1	3.3	3.5	0.5	5.1	52
	Between	1.6	2.1	2.2	0.0	0.4	4
	Tentative Flip	13.9	14.5	15.4	0.5	4.6	47
	MakePermanent	120.6	726.6	4734.5	0.0	1.3	81
	Total				1.0	11.4	184

TABLE 3.4. Times on an SPARCstation ELC for *Tour* operations.

One general conclusion that one can draw from Tables 3.2 through 3.4 is that the relative efficiencies of the *Tour* representations are indeed machine-dependent. To get a more direct view of this difference, and perhaps a better insight into the asymptotics of our implementations, see Table 3.5. This table presents the average overall running times, normalized by dividing through by N, for Splay Tree-2, Splay Tree-0, and our other three *Tour* representations 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 starting tour). The remaining rows give the time spent in local optimization, the phase in which the *Tour* representation code is active.

Note 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. For a million cities, extrapolations suggest that Arrays would have taken over 150 hours on the IRIS, as compared to less than three for Splay Trees and Two-Level Trees. The overall times (including preprocessing) for a million cities were 2.62 hours for Two-Level Trees, 2.70 hours for Splay Tree-0, 2.88 hours for Splay Tree-2, and 4.60 hours for Segment Trees. As to trends in the data, it is clear that Segment Trees have run out of gas by 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	7.6
Splay Tree-0	10.9	13.2	12.8	3.1	3.3	3.6	2.8	3.8	5.1	7.0
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 3.5. Overall running times for *Tour* representations (Normalized).

time $N = 10^6$. The Two-Level Tree representation is also beginning degrade, although not nearly as catastrophically. Its normalized time goes up by a factor of almost two as N increases from 100,000 to 1,000,000, while our normalized Splay Tree times go up by less than 40%. Indeed, it seems likely that Splay Trees will be the representation of choice as soon as N gets much larger than 10^6.

3.3. Comparisons on ‘Real World’ Instances

So far we have reported results only for random Euclidean instances. It has been observed for many optimization problems that random instances bear almost no relation to instances arising in the ‘‘real world,’’ but this is fortunately not the case for the TSP. Lessons learned from random Euclidean TSP instances do give insight into algorithmic behavior on more realistic geometric instances, although the correspondence is of course not complete. In this section we shall compare our results for random Euclidean instances to those we obtained on more realistic instances from Gerd Reinelt’s TSPLIB [27], currently the best public domain source of real-world instances, available via anonymous FTP from `softlib.rice.edu`. Most of the instances in this collection are too small (5000 or fewer cities) for the issues we are studying here to have major impact. There is, however, a set of three related instances that range from 7397 to 85,900 cities. These are identified as *pla*7397, *pla*33810, and *pla*85900 in TSPLIB, and arose in an AT&T application in which a laser was used to customize programmable logic arrays.

Table 3.6 presents a comparison between our result for these *pla* instances and our already reported results for random Euclidean instances. The entries in the table are all expressed as ratios of the results for the *pla* instances (lengths, counts, times) to extrapolations of what the corresponding results for random Euclidean instances of the same size would be. The top of the table corresponds to the top of Table 1.2 for random Euclidean instances. It presents the ratios for the average length of the shorter segment when a *Flip*

	N	7397	33810	85900	7397	33810	85900
Shorter Segment (Ratio)		1.4	1.7	2.2			
Number of Calls (Ratio)	*Next* + *Prev*	9.4	11.3	8.0			
	Between	0.7	0.6	0.4			
	Flip	3.2	3.7	2.7			
		IRIS 2D/450			VAX 8550		
Array							
Time per Call (Ratio)	*Next* + *Prev*	1.0	0.8	0.6	0.9	0.9	0.8
	Between	0.9	0.8	0.7	1.1	1.0	1.0
	Flip	1.8	1.3	1.4	1.3	1.7	1.6
Total Time for *Tour* Ops. (Ratio)		4.7	2.1	3.3	3.1	2.8	4.0
Splay Tree-0							
Time per Call (Ratio)	*Next* + *Prev*	1.2	1.0	0.8	0.8	0.8	0.7
	Between	1.0	0.9	0.8	0.9	0.8	0.9
	Flip	1.3	1.3	1.2	1.1	1.1	0.7
Total Time for *Tour* Ops. (Ratio)		6.1	5.8	4.5	5.2	6.5	3.8
Two-Level Tree							
Time per Call (Ratio)	*Next* + *Prev*	0.6	0.5	0.5	0.9	1.0	0.9
	Between	0.8	0.7	0.7	0.9	0.8	0.9
	Flip	0.9	1.0	1.1	1.2	1.3	1.5
Total Time for *Tour* Ops. (Ratio)		2.4	3.4	3.4	3.7	4.8	4.0
Segment Tree							
Time per Call (Ratio)	*Next* + *Prev*	0.9	0.9	0.8	0.9	0.9	0.9
	Between	0.9	0.7	0.7	0.8	1.0	1.0
	Tentative Flip	1.2	1.2	1.2	1.0	1.5	1.1
	MakePermanent	1.1	1.3	1.3	1.0	2.0	2.1
Total Time for *Tour* Ops. (Ratio)		4.4	3.0	2.1	4.6	4.8	2.9

TABLE 3.6. Comparison of results for *pla* instances to extrapolated results for random Euclidean instances of the same size.

is performed and the number of calls to the *Tour* operations under the Array, Splay Tree, and Two-Level representations. (Segment Trees, because they are used in a modified version of the algorithm, have fewer calls to *Flip* and *Next/Prev*, although their other overheads typically counterbalance this saving.) In computing the ratios here, we used the extrapolations that for random Euclidean instances the shorter segment length grows at approximately $0.38N^{.67}$, and that the *Next*/*Prev*, *Between*, and *Flip* call counts grow as roughly $160N$, $20N$, and $62N$, respectively. These are rough approximations, and we only compute the ratios to the nearest tenth, but this is enough to give a general sense of the differences between instances. Note that for these instances the number of calls to *Next*/*Prev* goes up by an average factor of roughly 10 over the number for random

Euclidean instances, whereas the number of calls to *Flip* goes up by a factor of 3, and the number of calls to *Between* goes down. This situation would seem to bode worst for our Splay Tree representation, which has by far the most expensive *Nexts* and *Prevs*. On the other hand, the length of the shorter segment on *Flips* also goes up significantly (although the actual average length still seems to be growing sublinearly with *N*). This should negatively affect our Array representation and (to a lesser extent) the Two-Level Tree representation and (in its *MakePermanent* operation) the Segment Tree representation.

The actual effects on the representations are summarized in the rest of the table, which reports results for all four representations, and for our two most disparate machines (the IRIS 2D/450 and the VAX 8550). Here the numerators of the ratios for each *pla* instance are averages over 6 runs on the IRIS and 3 runs on the VAX, and the denominators are derived from our data on random Euclidean instances by simple linear interpolation (between the results for $N = 1000$ and $N = 10{,}000$ for *pla*7397, and between the results for $N = 10{,}000$ and $N = 100{,}000$ for *pla*33810 and *pla*85900). Again, these are very rough estimates, but enough to give some feeling for behavior. Note that in most cases the times per operation do not differ greatly from those for the random instances. Times for *Next*, *Prev*, and *Between* are often slightly faster, increasingly so as the instances get larger, and times for *Flip* are typically slower, but rarely is the difference as much as a factor of 2. (The reason that the ratios for *Next*, *Prev*, and *Between* tend to decrease as *N* increases is that for these instances those operations tend to take the constant time predicted by theory, rather than the slowly growing time observed for our random Euclidean instances.) Consequently, the main reason for the major increases in the total time spent on *Tour* operations reported in the table is simply that many more *Tour* operations are being performed.

The net effect of all these contending factors in revealed in Table 3.7, which reports overall running times (not normalized) for the three instances, two machines, and four

	Running Time in Seconds					
	IRIS 4D/250			VAX-8550		
N	7397	33810	85900	7397	33810	85900
Preprocessing	11	53	142	39	180	499
Array	221	3390	15885	463	7432	30228
Splay Tree-0	148	796	1646	498	2854	4300
Two-Level Tree	53	414	1014	249	1745	3465
Segment Tree	77	434	835	251	1571	2389

TABLE 3.7. Overall running times for *pla* instances.

representations. As in the random Euclidean case, the competition is between Two-Level Trees and Segment Trees, with Splay Trees far back and Arrays out of the running. Note, however, that for these instances Segment Trees outperform Two-Level Trees on the larger instances even on the IRIS. Further experiments indicate, however, that this gain is entirely the result of the modification one has to make in the Lin-Kernighan algorithm in order to use the Segment Tree representation (the imposition of a bound of 50 on the length of an allowable sequences of *Flips*). For these instances, if one imposes the same bound of 50 while running the Two-Level Tree representation, one obtains a significant speedup, more than enough to put the Two-Level Tree representation back in the lead. There is a price to pay for this speedup in either case, however: the average length of the resulting tour increases by roughly 0.2%, significant when as here the tours are already within 2-3% of the Held-Karp lower bound. (Recall that such an increase was not observed in the random Euclidean case.) One possible explanation is that for these instances the number of improving sequences of *Flips* longer than 50 in unconstrained Lin-Kernighan is 2-3%, whereas it is less than 1% in the random Euclidean case.

3.4. Comparisons for Random Distance Matrices

As our final set of test instances, we consider a type of instance that is about as far from ‘‘real world’’ as possible: random distance matrices. These are instances in which each inter-city distance is chosen independently from a uniform distribution on [0,1]. Note that such instances do not obey the triangle inequality, and indeed lack all the correlations one might expect from instances that arise in practice. They are, however, a substantial challenge to our *Tour* representations and are interesting for that reason. For this class of instances we restricted our main tests to the IRIS 2D/450. We considered values of N that grew by factors of $\sqrt{10}$ from 1,000 to 31,623 (two instances for each size). The instances were generated in such a way that the $\Theta(N^2)$ inter-city distances did not need to be explicitly stored (which would have been impossible even for our IRIS when $N = 31,623$). Distances were instead generated on the fly in a reproducible way, using a special random number generator that took three seeds, one identifying the instance and the other two corresponding to the cities whose distance was being computed (see [3] for a description). Our results are averages over 10 runs on each of the instances except for those of the largest size, where we average over 3 runs. This yields confidence intervals of roughly ±4%.

Table 3.8 presents results analogous to those of Table 3.6 for the *pla* instances, with entries expressed as ratios to the corresponding results (or extrapolated results when $N = 3,162$ or $N = 31,623$) for random Euclidean instances of the same size. Note that the number of calls to *Prev*/*Next* and to *Flip* are here higher than for random Euclidean

	N	1,000	3,162	10,000	31,623
Shorter Segment (Ratio)		6.4	9.4	14.7	20.1
Number of Calls (Ratio)	*Next* + *Prev*	1.1	1.5	2.0	2.7
	Between	0.7	0.7	0.8	0.8
	Flip	1.4	1.7	2.0	2.4
		IRIS 2D/450			
Array					
Time per Call (Ratio)	*Next* + *Prev*	1.0	1.5	1.8	-
	Between	1.0	1.4	1.6	-
	Flip	5.6	10.2	14.0	-
Total Time for *Tour* Ops. (Ratio)		6.9	8.2	26.6	-
Splay Tree-0					
Time per Call (Ratio)	*Next* + *Prev*	1.3	1.6	2.3	3.8
	Between	1.5	1.8	2.3	3.5
	Flip	1.5	1.8	2.2	3.1
Total Time for *Tour* Ops. (Ratio)		1.6	2.2	4.0	7.1
Two-Level Tree					
Time per Call (Ratio)	*Next* + *Prev*	1.3	1.4	1.9	2.8
	Between	1.2	1.1	1.3	1.7
	Flip	2.2	2.5	4.2	7.4
Total Time for *Tour* Ops. (Ratio)		1.9	2.8	6.5	12.2
Segment Tree					
Time per Call (Ratio)	*Next* + *Prev*	1.1	1.7	2.0	2.5
	Between	1.4	1.2	1.5	1.8
	Tentative Flip	1.1	1.4	1.8	1.8
	MakePermanent	18.5	34.7	50.6	57.2
Total Time for *Tour* Ops. (Ratio)		1.5	2.6	5.0	6.7

TABLE 3.8. Comparison of results for random distance matrices to extrapolated results for random Euclidean instances of the same size.

instances, and are growing at a faster (and hence superlinear) rate. The average length of the shorter segment in *Flips* is also growing much faster. Indeed, for these instances the average length of the shorter segment is extremely close to the $N/4$ value predicted in Section 1, and thus it grows much more rapidly than the sublinear rates we saw for random Euclidean instances and for *pla* instances. Turning to the behavior of the four representations, we see that all are losing ground to their performance on random Euclidean instances as N increases, both with respect to time per operation and to overall time for operations. In the case of Splay Trees, this is presumably because random distance matrices don't provide nearly as much locality of reference for the splay operations to capitalize on. Lack of locality of reference may also be one reason why the other representations are degrading, although here it would be because of increasing probability of cache-misses, a machine-dependent factor. (Limited experiments on the VAX-8550 suggest this is the case. For the latter machine the time per *Next*/*Prev* and *Between* operations under Array, Two-Level Tree, and Segment Tree representations remained relatively constant from $N = 1,000$ up to $N = 10,000$, the largest random distance matrix which that machine could reasonably handle.)

The results for *Flip* provide the most insight. As could be expected from the average length of the shorter segment, the Array representation degrades substantially more rapidly as N grows than it did for the random Euclidean instances (and it was already degrading rapidly for those). This effect is also seen for Two-Level Trees, where the cost of a *Flip* in our implementation is also dependent on the length of the shorter segment, especially when we fix *groupsize* at 100, as we did for these experiments. The significant jump in the time per operation as we go from $N = 10,000$ to $N = 31,623$ suggests that a bigger value for *groupsize* might have yielded better results for instances of the latter size. The one other operation that is affected by the increased shorter segment length for these instances is the *MakePermanent* operation under Segment Trees. Here the effect is by far the greatest, suggesting that our decision always to implement the operation by doing *Flips* in the permanent tour may be misguided for this type of instance. We thus re-ran the experiments using a hybrid scheme in which *MakePermanent* operation was implemented using *Flips* only if 20 or fewer flips were needed. If more than 20 flips were needed, *MakePermanent* was implemented by rebuilding the permanent tour from scratch based on the segment list. Using this threshold each method was called about the same number of times (to within a factor of 2) and took roughly the same time per operation. Moreover, as seen below, the overall time for *Tour* operations was significantly reduced for the larger instances.

IRIS 4D/250	Running Time in Seconds			
N	1,000	3,162	10,000	31,623
Preprocessing	4.6	45.4	465	4651
Array	10.0	115.1	1371	-
Splay Tree-0	4.7	24.7	144	918
Two-Level Tree	4.0	20.3	138	900
Segment Tree	3.2	19.2	116	777
Segment Tree+	3.3	18.1	96	658

TABLE 3.9. Overall running times for random distance matrices.

Table 3.9 summarizes the overall running time results for the various *Tour* representations on random distance matrices, with a separate entry (*Segment Tree+*) for Segment Trees using the hybrid *MakePermanent* operation. Note that for these instances Two-Level Trees are barely better than Splay Trees, although Arrays are far behind both. Segment Trees are the clear winner in the range up to $N = 31,623$, with the hybrid *MakePermanent* scheme to be preferred. (This despite the fact that for these instance, bounding the length of *Flip* sequences at 50, as we do for Segment Trees, actually slows the algorithm down a bit, rather than speeding it up as it did for the *pla* instances.) Extrapolating the growth of *MakePermanent* work, however, even under the hybrid scheme, suggests that Segment Trees could well lose out to Splay Trees at $N = 100,000$, should we ever have cause to try random distance matrices that large. Note, however, that the real bottleneck for our Lin-Kernighan code here is not the *Tour* operations but the preprocessing. For instances with uncorrelated distances, as we have here, the preprocessing phase must look at all pairs of cities and hence must take time $\Omega(N^2)$, as opposed to roughly linear time on our Euclidean instances (random and *pla*-based). Thus the total time for the algorithm is so dominated by the preprocessing time that the relative difference in overall running time attributable to our choice of *Tour* representation is quite minor (assuming we do not choose Arrays).

4. Lower Bounds

In this section, we show that the Splay Tree implementation of the *Tour* datatype is almost the best possible (in an asymptotic worst-case sense), by proving 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 [30] (see also [8]). In this model there is a single parameter *b*, the number of bits in single word of shared memory, and the cost of a computation is essentially the number of words accessed. This is a very general model, encompassing even such baroque data structures as the *fusion trees* of [9]. Our proof relies on a rather technical generalization of Theorem 3′ from [8], and we shall state this generalization here without proof, calling it *Theorem A*. The proof of Theorem A, itself a straightforward generalization of the proof of Theorem 3′ in [8], will be included in the journal version of that paper, and in the meantime is available from the authors.

We will not be applying Theorem A directly to our problem, but rather to a mathematically simpler problem that we can show reduces to our problem in a two-stage process. The first step in formalizing these reductions is the following definition.

Definition. *A* dynamic query problem *is a quadruple* $(\mathbf{S}, S_0, \mathbf{Q}, \mathbf{U})$*, where* **S** *is a finite state space,* $S_0 \in \mathbf{S}$ *is distinguished start state,* **Q** *is an ordered finite set of* query operations $Q\colon \mathbf{S} \to \{0,1\}$*, and* **U** *is a finite set of* update operations $U\colon \mathbf{S} \to \mathbf{S}$. *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.*

The reductions in our lower bound proof involve the *Reversible String* problem, a dynamic query problem that is defined as follows. The states are all permutations of a length-N string s_0 consisting of N distinct symbols. Let Σ be the corresponding N-symbol alphabet. Queries are of the form $Precedes(x,y)$, where $x,y \in \Sigma$. The answer to $Precedes(x,y)$ applied to string s is defined to be 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 solve the Reversible String problem. More specifically, any sequence of m operations for the latter problem can be implemented by a sequence of at most $4m$ *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 and d will always be linked together, c will always in addition be linked to the first symbol in s, and d will always be linked to the last symbol in s. See Figure 4.1. We will then have that $Precedes(x,y) = 1$ if and only if either (a) $Next(d) = c$ and $Next(x) = y$, or (b) $Next(d) \neq c$ and $Prev(x) = y$. Thus $Precedes(x,y)$ can be implemented with two calls to *Prev*/*Next*. $Reverse(x,y)$ is a bit more complicated, but can be done with two *Prev*/*Next*'s, one *Between*, and a *Flip*, as follows: If $x = y$, nothing need be done. If x and y are distinct, ask a $Between(c,x,y)$ query. If the answer is yes, let $u = Prev(x)$, $v = Next(y)$, and perform $Flip(u,x,v,y)$. Otherwise, let $u = Prev(y)$, $v = Next(x)$, and perform $Flip(u,y,v,x)$.

Thus if we can show that the Reversible String problem requires worst-case amortized time $\Omega(\log N / \log\log N)$ per operation, the same conclusion will follow for our *Tour* datatype. To show that the conclusion holds for the Reversible String problem, we will use the latter to solve yet a third problem, the *Bit-Vector Complementation* problem. Theorem A will then be used to show that this last problem has the required lower bound.

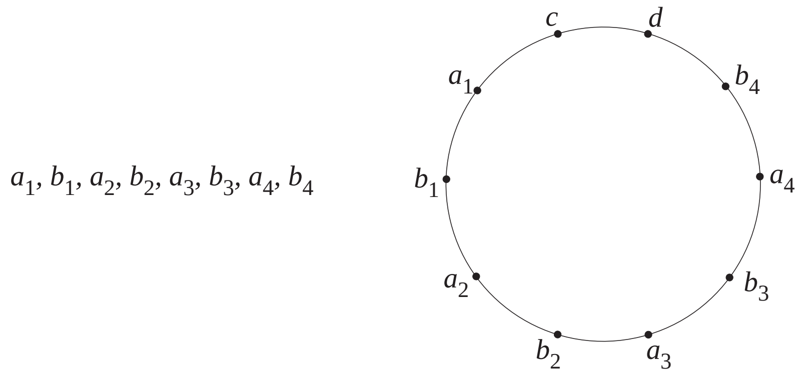

FIGURE 4.1. Correspondence between the Reversible String problem and the Tour datatype.

The Bit-Vector Complementation problem is a dynamic query problem defined as follows. The state set **S** consists of all length-M binary strings, where $M = 2^k$ for some fixed k. The start state S_0 is the all-1 vector. The set **Q** of query operations consists of one query $Q[i]$ for each integer i, $1 \le i \le M$. The answer to query $Q[i]$ for a given string S is simply the value of the ith bit of S. The set **U** of update operations contains one update $U[h,j]$ for each pair of integers h,j such that $0 \le h \le k$ and $1 \le j \le 2^{k-h} = M/2^h$, where update $U[h,j]$ says to complement the bits in positions $(j-1)2^h + 1$ through $j2^h$. For instance, $U[0,j]$ says to complement the jth bit, $U[k,1]$ says to complement the entire string, and $U[3,4]$ says to complement the bits in positions 25 through 32. For technical reasons we also include a vacuous update $U[0]$, whose effect is to leave the current bit-vector unchanged, for an overall total of $2M$ update operations. In what follows it will be useful to think of each update operation $U[h,j]$ as corresponding to a binary vector $\sigma_{U[h,j]}$ which has 1's only in positions $(j-1)2^h + 1$ through $j2^h$. (The vector $\sigma_{U[0]}$ will by definition be the all-zero vector.) An update can then be performed simply by adding the corresponding binary vector to the current vector S (*mod* 2). Note that the update operations are all nicely nested: the domains of two updates (in terms of positions of S effected) are either disjoint or else one is entirely contained in the other.

Let us now show how the Bit-Vector Complementation problem can be reduced to the Reversible String problem, so that each operation of the former is transformed into one operation of the latter, with only a constant amount of additional overhead per operation. The idea is to model the length-M bit vector by a length-$(4M-2)$ string, constructed as follows. Our alphabet Σ will consist of symbols $e_{h,j}, f_{h,j}$ for each of the $2M-1$ non-vacuous update operations $U[h,j]$. The initial string S_0 is constructed hierarchically out of subsegments as follows. There are M level-0 segments $T_{0,j}$, $1 \le j \le M$, where $T_{0,j} = e_{0,j} f_{0,j}$. Inductively, for each h, $1 \le h \le k$, there are 2^{k-h} level-h segments $T_{h,j}$, $1 \le j \le 2^{k-h}$, where $T_{h,j} = e_{h,j} T_{(h-1),(2j-1)} T_{(h-1),(2j)} f_{h,j}$. The initial string $S_0 = T_{k,1}$. Note that this hierarchical structure reflects the nesting property of the strings $\sigma_{U[h,j]}$ that we associated with the update operations in the Bit-Vector Complementation problem.

In our simulation, each time an update operation $U[h,j]$ is to be performed on the current bit-vector, we instead perform the update operation $Reverse(e_{h,j}, f_{h,j})$ on the current string. For the vacuous update operation $U[0]$, we simply perform the vacuous reversal $Reverse(e_{0,1}, e_{0,1})$. These will be the only Reversible String updates we perform. Note that this means that each symbol pair $e_{0,i}, f_{0,i}$ will always remain adjacent, although the relative order of the two symbols may be reversed. In particular, the relative order will be reversed every time we perform an update operation $Reverse(e_{h,j}, f_{h,j})$ where the corresponding $U[h,j]$ complements bit i of the vector, and these are the only updates that will reverse it. Thus the order of the pair in the current string will be $e_{0,i}, f_{0,i}$ if and only if the ith bit in the current bit vector is 1, and we can obtain the answer to the Bit-Vector query $Q[i]$ simply by asking the Reversible String query $Precedes(e_{0,i}, f_{0,i})$.

This completes the description of our simulation. Note that for any sequence of p update and query operations in the Bit-Vector Complementation problem for a vector of length M, our simulation performs precisely the same number of operations in a Reversible String problem for a string of length $4M-2$. The additional work needed per operation to perform the translations will be bounded by a small constant, assuming our computer has word size at least $\Omega(\log M)$. Thus if the Bit-Vector Complementation problem requires amortized time $\Omega(\log M/\log\log M)$ per operation, our Reversible String problem will require amortized time at least $\Omega(\log N/\log\log N)$ per operation, which is what we are trying to prove.

We are now almost ready to state Theorem A and prove our lower bound. First, however, we need two technical definitions.

Definition. *Let $\Delta = (\mathbf{S}, S_0, \mathbf{Q}, \mathbf{U})$ be a dynamic query problem, and let $F_h \subseteq \mathbf{U}^h$ be a collection of update sequences of length h. Let $M = |\mathbf{Q}|$. For any state $S \in \mathbf{S}$ and for each sequence λ in F_h let $\lambda(S)$ be the state obtained by starting with S and sequentially applying the h update operations of λ. For any state S, let $v_S \in \{0,1\}^M$ be the M-dimensional binary vector determined by the responses to the M queries in $\mathbf{Q}$ from the state S. Let δ denote the Hamming distance function between vectors. For positive real numbers γ and ρ we say that F_h is* (γ, ρ)-dispersing *provided that for all $u \in \{0,1\}^M$ and all $S \in \mathbf{S}$, $|\{\lambda \in F_h : \delta(v_{\lambda(S)}, u) \le \gamma M\}| \le |F_h|/2^{\rho h}$.*

Intuitively, this definition asserts (when $\gamma < 1$) that the update sequences in F_h have sufficiently varying impact so as to prevent clustering of the query responses. Note that for the Bit-Vector Complementation problem, v_S simply equals S.

Definition. *Let* $\Delta = (\mathbf{S}, S_0, \mathbf{Q}, \mathbf{U})$ *and let* $M = |\mathbf{Q}|$. *Let* $F \subseteq \mathbf{U}^m$ *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, γ, ρ)-dispersed *if the following holds. For all pairs of integers* i, h *satisfying* $i \geq 1$, $t \leq h \leq \sqrt{M}$, *and* $i + h - 1 \leq m$, *the set* $F[i, i+h-1] = F_i F_{i+1} \cdots F_{i+h-1}$ *is* (γ, ρ)-*dispersing.*

Theorem A. *Let* $\Delta = (\mathbf{S}, S_0, \mathbf{Q}, \mathbf{U})$ *be a dynamic query problem, let* $M = |\mathbf{Q}|$, *and let F be a set of length-m update sequences for* Δ *that is* (t, γ, ρ)-*dispersed, where* $m \geq 2\sqrt{M}$, $t \leq M^{1/4}$ *and* $\rho \geq (\log M)^{-\kappa}$. *Let* $\mathbf{H}$ *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 the update subsequence* $(U_1\ \cdots\ U_m) \in F$. *Let A be an on-line algorithm for* Δ *in the cell probe model of computation with b-bit word size, where* $b \leq (\log M)^{\kappa}$. *Then for at least one sequence* $\alpha \in \mathbf{H}$, *A performs* $\Omega(\gamma \cdot m \cdot \log M / (\kappa \log\log M))$ *memory accesses when executing the operations in* α.

In applying Theorem A to the Bit-Vector Complementation problem, we shall restrict attention to sequences of operations having the form

$$u_1\ u_2\ \cdots\ u_m\ ,\ m \geq \sqrt{M} \tag{4.1}$$

where each sequence is derivable from a special sequence

$$\Gamma = U_1\ U_2\ \cdots\ U_m \tag{4.2}$$

in that for each i, $1 \leq i \leq m$, u_i must either equal U_i or be the vacuous update $U[0]$. In other words, the set of sequences in which we will be interested is the concatenation $F = F_1\ F_2\ \cdots\ F_m$, where $F_i = \{U_i, U[0]\}$, $1 \leq i \leq m$.

The sequence Γ is structured as follows. We partition Γ into k (non-contiguous) subsequences Γ_h, $1 \leq h \leq k$, where subsequence Γ_h consists of those U_i with index $i \equiv h (mod\ k)$. Focusing on subsequence Γ_h, the operations comprising this subsequence systematically cycle through the updates $U[h, j]$, $1 \leq j \leq 2^{k-h}$. If we define the *weight* of an update $U[h, j]$ to be the number of 1's in the corresponding vector $\sigma_{U[h,j]}$, this means that the updates in Γ_h all have weight 2^h.

The following two lemmas will make it possible to apply Theorem A. Given a binary vector σ, the *1-components* of σ refers to the set of components for which σ has the value 1.

Lemma 4.1. *Suppose $k = \log M \geq 2$. Given a sequence L of consecutive update operations in the sequence Γ, $k \leq |L| \leq M$, there exists a subsequence L' with $|L'| > |L|/(2k)$ such that all update operations in L' have the same weight ω,*

$$\frac{Mk}{|L|} \leq \omega < \frac{2Mk}{|L|}, \tag{4.3}$$

and such that the 1-components of the vectors associated with the operations in L' are all disjoint.

Proof. Given that the upper and lower bounds in (4.3) differ by a factor of 2, there is a unique weight $\omega = 2^h$ that satisfies those bounds. The sequence Γ will contain update operations of this weight so long as $1 \leq h \leq k$. We have $h \leq k$ by (4.3) since $|L| \geq k$ by hypothesis and $k = \log M$ by definition. We have $h \geq 1$ since $|L| \leq M$ and $k \geq 2$ by hypothesis. Let L' be a set of M/ω consecutive members of Γ_h contained in L. (Note that M/ω is an integer since $M = 2^k$.) We know that there are enough members of Γ_h in L for this selection to be made since $M/\omega \leq |L|/k$ by the first inequality in (4.3). The second inequality in (4.3) then tells us that $|L'| = M/\omega > |L|/(2k)$, as desired. Finally, taking into consideration the cyclic construction of the Γ_h and the fact that $|L'| = M/\omega$, it is easy to see that the 1-components of the vectors associated with the operations in L' are all disjoint, our final requirement. □

Lemma 4.2. *Let $\sigma_1, \cdots, \sigma_s$, $s \geq 1$, be M-dimensional binary vectors having common Hamming weight $\omega \geq M/(2s)$, and assume that the 1-components of these vectors are all disjoint. Let u be any fixed M-dimensional binary vector. Then at most $2^{.95s}$ of the 2^s vectors spanned by the σ_j (sums taken $(mod\ 2)$) fall within Hamming distance $M/12$ of u.*

Proof. Let Λ denote the space spanned by the σ_j. Let $\hat{u} = \Sigma_i\, \alpha_i \sigma_i$ denote the vector in Λ closest to u in terms of Hamming distance, and let $x = \Sigma_i\, \beta_i \sigma_i$ be an arbitrary vector in Λ other than $\hat{u}$. Let δ be the Hamming distance between the respective coefficient vectors, $(\alpha_1, \cdots, \alpha_s)$ and $(\beta_1, \cdots, \beta_s)$. Note that the Hamming distance between $\hat{u}$ and x is consequently $\delta\omega$. Since u is closer to $\hat{u}$ than to x, the triangle inequality implies that the Hamming distance between x and u is at least half that between x and $\hat{u}$ and hence is at least $\delta\omega/2 \geq \delta M/(4s)$. Thus, in order for x and u to be within distance $M/12$, δ cannot exceed $s/3$. Hence, the number of x in Λ within distance $M/12$ of u does not exceed the number of binary s-dimensional vectors of Hamming weight at most $s/3$. This number is easily estimated using Stirling's formula and does not exceed $2^{.95s}$, completing the proof. □

We are now ready to state and prove our lower bound result for the Bit-Vector Complementation problem.

Theorem 4.1. *Suppose $M \geq 100{,}000$ and let A be any on-line algorithm that solves the Bit-Vector Complementation problem in the cell probe model of computation with word size no larger than some fixed power of* $\log M$. *Given any $m > 2\sqrt{M}$, there exists a sequence of m operations for which A performs $\Omega(m \cdot \log M / \log\log M)$ memory accesses during the course of executing these operations.*

Proof. Let κ be such that the word size is bounded by $(\log M)^{\kappa}$. We may assume without loss of generality that $\kappa \geq 2.5$. Let F be the set of length-m update sequences patterned by (4.1). We shall show that F is (t,γ,ρ)-dispersed, with $t = k$, $\gamma = 1/12$ and $\rho = .025/k$. The other hypotheses of Theorem A will also apply: $m > 2\sqrt{M}$ by definition, $t \leq M^{1/4}$ if $M \geq 100{,}000$ since $t = k = \log M$, and $\rho = .025/k$ $=$ $.025/(\log M) > (\log M)^{-\kappa}$ for all $\kappa \geq 2.5$ when $M \geq 100{,}000$. Consequently, Theorem A will apply and imply the claimed result.

To show that F is (t,γ,ρ)-dispersed, we first recall that it has the correct format as a product $F_1\, F_2\, \cdots\, F_m$, where the F_i are based on the sequence Γ of (4.2), and each contains two elements. Consider a sequence L of consecutive update operations from Γ, where $t \leq |L| \leq \sqrt{M}$, say the one starting at position i and extending to position $i+|L|-1$. Let $F[i,i+|L|-1]$ be the product of the sets of updates corresponding to the positions in Γ of the elements of L. We must show that $F[i,i+|L|-1]$ is (γ,ρ)-dispersing. So let u be an arbitrary vector in $\{0,1\}^M$, and let S be an arbitrary state in **S**. We must show that the number of update sequences λ in $F[i,i+|L|-1]$ such that $v_{\lambda(S)}$ is within Hamming distance $|M|/12$ of u is at most $|F[i,i+|L|-1]|/2^{.025|L|/k}$.

By Lemma 4.1, there is a subset $L' \subseteq L$ of size $s > |L|/(2k)$ such that all update operations in L' have the same weight ω, $Mk/|L| \leq \omega < 2Mk/|L|$, and such that the 1-components of the vectors associated with these updates are all disjoint. Let ψ denote the set of positions in Γ corresponding to the operations in L', and let ϕ denote the positions in Γ corresponding to operations in $L - L'$. Note that $|\psi| = s$. We now partition $F[i,i+|L|-1]$ into $2^{|\phi|}$ subsets, one for each way of choosing one member each from the sets F_i, $i \in \phi$. The subsets are constructed as follows. We identify each way of choosing the members with a function $f : \phi \to \{0,1\}$, and the subset corresponding to f consists of the 2^s sequences of the form $u_i\, u_{i+1}\, \cdots\, u_{i+|L|-1}$ in $F[i,i+|L|-1]$, where

$$u_j = \begin{cases} U[0] & j \in \phi \text{ and } f(j) = 0 \\ U_j & j \in \phi \text{ and } f(j) = 1 \\ U[0] \text{ or } U_j & j \in \psi \end{cases}$$

Let G_f be one such subset, and let us extend the correspondence between updates and strings to arbitrary sequences of updates in the natural way. That is, if $\lambda = u_1, u_2, \ldots, u_t$, then σ_λ is the vector sum of σ_{u_1} through σ_{u_t}. As with individual updates, the result $\lambda(S)$ obtained by applying the operations of λ to a vector S is simply the vector sum $(mod\ 2)$ of S and σ_λ. Since individual update operations commute, we can also speak of σ_X for a *set* X of operations, where σ_X is the vector sum of the vectors σ_u for all $u \in X$. For each λ in G_f, the corresponding vector σ_λ can be viewed as the vector sum $\sigma_{\lambda_\psi} + \sigma_{\lambda_\phi} (mod\ 2)$, where λ_ψ and λ_ϕ are the sets of updates in ψ and ϕ respectively. Now σ_{λ_ϕ} is fixed for all members of G_f by definition, whereas the vector σ_{λ_ψ} is not fixed, there being a different one for each member of G_f. Let us denote this set of vectors σ_{λ_ψ} by $V(\psi)$. We now apply Lemma 4.2, with the role of $\sigma_1, \ldots, \sigma_s$ played by the vectors σ_{U_j}, $j \in \psi$, and with the set of 2^s vectors spanned by them being simply $V(\psi)$. Observe that the vectors in $V(\psi)$ actually comprise an affine subspace. The hypotheses of Lemma 4.2 are satisfied since we have $s > |L|/(2k)$ and the common Hamming weight of the s basis vectors (which have disjoint 1-components) is at least $Mk/|L| > M/(2s)$.

As a consequence of the Lemma, for each vector w, the number of vectors in $V(\psi)$ that are within distance $M/12$ of $u = w + S + \sigma_{\lambda_\phi} (\ mod\ 2)$ is at most $2^{.95s}$. This inequality will continue to hold if we translate all the vectors (including u) by a fixed vector, in particular if we add $S + \sigma_{\lambda_\phi}$ to each $(mod\ 2)$. Adding $S + \sigma_{\lambda_\phi}$ to the vectors in $V(\psi)$ yields $\{\lambda(S) : \lambda \in G_f\} = \{v_{\lambda(S)} : \lambda \in G_f\}$, since recall that for the Bit-Vector Complementation problem, $v_S = S$ for all states S. Adding $S + \sigma_{\lambda_\phi}$ to u yields $w + 2\sigma_S + 2\sigma_{\lambda_\phi} = u (mod\ 2)$. Thus the number of strings $\lambda \in G_f$ such that $v_{\lambda(S)}$ is within $M/12$ of u is at most $2^{.95s}$. Consequently, the proportion of these vectors that are that close is at most $2^{.95s}/2^s = 2^{-.05s} < 2^{-.025|L|/k}$. Since this holds true for all the sets G_f, the desired inequality holds. Hence $F[i, i+|L|-1]$ is (γ,ρ)-dispersing and F is (t,γ,ρ)-dispersed, as claimed, yielding the theorem. □

As an immediate corollary of Theorem 1, our simulation of the Bit-Vector Complementation problem by the Reversible String problem, and the method explained above for solving the Reversible String problem in terms of the *Tour* datatype, we have our desired lower bound result for the latter. Filling in the precise values of the derived constants, the lower bound result for *Tour* operations can be stated as follows.

Theorem 4.2. *Suppose A is any on-line algorithm that implements the* Tour *datatype in the cell probe model of computation with word size no larger than some fixed power of* log *N, where N is the number of cities. Then for all* $N > 400{,}000$ *and any* $m > 4\sqrt{N}$, *there exists a sequence* λ *of* m Tour *operations such that* A *performs* $\Omega(m \log N / \log\log N)$ *memory accesses during the course of executing these operations.*

5. Conclusions and Directions for Further Research

In this paper we identified the choice of *Tour* representation as a critical decision in the implementation of local search heuristics for the traveling salesman problem. When N is large, this choice can have a dramatic effect on the running time of algorithms such as Lin-Kernighan. We suggested three alternatives to the traditional array-based representation of the tour, and analyzed each from both theoretical and experimental perspectives. We also showed that asymptotically the best of these is the Splay Tree representation, which has a worst-case $O(\log N)$ amortized cost per operation. This is near-optimal in the sense that any *Tour* representation must have worst-case amortized cost $\Omega(\log N / \log\log N)$ per operation, assuming the general cell probe model of computation.

The major remaining theoretical open problem is to close the gap between these upper and lower bounds. It is quite possible that a new representation with worst-case amortized cost $O(\log N / \log\log N)$ per operation might exist, although we do not currently have any ideas about how to construct one. Note, however, that for N less than four billion, $\log\log N$ is no more than 5, so the potential savings over Splay Trees on real-world instances would not be observable in practice unless the new representation had little additional overhead. Even if it had significantly less overhead than Splay Trees, the new representation could still lose out to them in practice, unless like them it had a strong ability to take advantage of locality of reference in sequences of operations. Moreover, even if the new representation actually beat Splay Trees in practice, it is still not clear that it would be useful, since in our experiments on instances with as many as a million cities, Splay Trees never turned out to be the *Tour* representation of choice. Within this range of instance sizes, the real competition was always between our Two-Level Tree and Segment Tree representations, with the former always outperforming Splay Trees and the latter doing so for all but our million-city instance.

It should be noted, however, that all three representations significantly outperformed the original Array representation, and so in this sense all can be judged successes. Moreover, one conclusion we might draw from our 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 $N < 1{,}000$, Two-Level Trees and Segment Trees each win by significant amounts on certain types of instances, and Splay Trees seem likely to come into their own should we ever want to tackle a two- or three-million city instance. For someone designing a new implementation of Lin-Kernighan from scratch, however, and not wanting to go to the effort of implementing more than one *Tour* representation, we would at this point recommend Two-Level Trees as the fastest and most robust on instances that might arise in practice.

We should remind the reader, however, that since the running times in our experiments for all the new representations tended to remain within a factor of two of each other, they are well within the range where additional code-tuning and minor algorithmic modifications might well have a major impact on the winners for particular problem instances, and on the various crossover points between approaches. Consequently, the results we have presented should for now be taken only as preliminary guides. There is much further experimentation that could be done. Here are some questions that might be addressed.

First, can the Two-Level Tree representation be improved by explicit rebalancing, either by implementing the scheme we proposed in Section 2 or by some more-efficient alternative? Based on our experiments and intuition, we expect not, but we may be wrong. The only work so far along this line has been done by Chrobak et al. [7], who report on limited experiments with a version of Two-Level trees that does have explicit rebalancing, although it does not implement the *Between* query and so is not faced with the problem of keeping that operation efficient without increasing the rebalancing cost. Unfortunately, it is difficult to make comparisons based on their data since they only measure performance on a sequence of random flips, rather than on the operation of the representation during actual runs of an algorithm. Furthermore, the maximum value of N they considered was 1,500.

Second, can the Segment Tree representation be improved by a better handling of the *MakePermanent* operation? Applegate and Cook, in their implementation of this approach [2], allow themselves to omit some of the *MakePermanent* operations, using an out-of-date permanent tour as their reference array until the number of permanent flips required to bring that reference tour up-to-date exceeds some threshold. This is especially important in their version of Segment Trees, since they always implement *MakePermanent* using the $\Theta(N)$ process of rebuilding the permanent tour completely, which as we have seen is not always the most efficient way to proceed. Moreover, in order to allow the omission of *MakePermanent* operations, they need frequently to undo entire sequences of tentative *Flips* in the tentative tour, being careful to merge segments back together whenever possible. These operations significantly increase the total number of *Flips* performed, while slightly increasing the cost per *Flip*. We suspect that the result of these compromises is a slower overall running time than for our version, but given that the Applegate-Cook version of Segment Trees is embedded in a different implementation of the Lin-Kernighan algorithm, we have not yet had a chance to do a direct comparison. Moreover, it is possible that some combination of the two versions, using features derived from both, might be preferable to either. This would be an interesting question to explore.

Third, is there a better way than Splay Trees to maintain the *Tour* at a cost of $O(\log N)$ per operation? There are at least two alternative methods in the literature for obtaining this asymptotic guarantee (although neither seems likely to be competitive with Splay Trees in practice). Chrobak et al. [7] describe how to do this with AVL trees. (See [1] for an introduction to AVL trees.) This AVL-tree approach has the theoretical advantage that it yields $O(\log N)$ time per operation in a pure worst-case sense, rather than an amortized worst-case sense. In the context of a full run of an algorithm like Lin-Kernighan, however, such a distinction is meaningless, and the AVL-tree approach has two key drawbacks. First, it is much more difficult to program than Splay Trees (although not impossible, since Chrobak et al. implemented AVL-Trees and tested them along with Two-Level Trees in [7]). Second, it does not have the capacity to take advantage of locality of reference. Thus it is unlikely that this variant would prove competitive in practice. Similar remarks can be made about a representation proposed by Margot [24]. This representation uses a nested hierarchy of linked cycles that, from another point of view, looks much like a binary tree with the cities located at leaves and with the vertices at each level connected by doubly-linked cyclic lists. (The doubly-linked list at the bottom level gives the tour.) Margot provides a method for performing *Flips* in this structure while keeping it roughly in balance, thus guaranteeing $O(\log N)$ time per operation. (The *Between* operation is not considered in [24], but can also be implemented to run within that time bound for this representation.) Margot's representation has the same defects as the AVL-tree approach: programming complexity and no natural capacity for taking advantage of locality of reference. Thus it is unlikely that an implementation of either of these two approaches would prove competitive, although it would be interesting to see just how well they do perform.

Finally, given that our experiments have already shown that asymptotics may not be relevant even for N as large as 10^6, are there any completely new sorts of *Tour* representations that are worth considering for N bounded by this limit? This may well be the most interesting question to pursue.

References

1. A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
2. D. APPLEGATE AND W. COOK, private communication (1990).
3. J. L. BENTLEY, ‘‘Fast algorithms for geometric traveling salesman problems,’’ *ORSA J. Comput.* **4** (1992), 387-411.
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. CHROBAK, T. SZYMACHA, AND A. KRAWCZYK, ‘‘A data structure useful for finding Hamiltonian cycles,’’ *Theor. Comput. Sci.* **71** (1990), 419-424.
8. 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.
9. 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.
10. A. M. FRIEZE, ‘‘Worst-case analysis of algorithms for travelling salesman problems,’’ *Methods of Operations Research* **32** (1979), 97-112.
11. F. GLOVER, ‘‘Tabu search – Part I,’’ *ORSA J. Comput.* **1** (1989), 190-206.
12. M. HELD AND R. M. KARP, ‘‘The traveling-salesman problem and minimum spanning trees,’’ *Operations Res.* **18** (1970), 1138-1162.
13. M. HELD AND R. M. KARP, ‘‘The traveling-salesman problem and minimum spanning trees: Part II,’’ *Math. Programming* **1** (1971), 6-25.
14. 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.
15. D. S. JOHNSON, D. L. APPLEGATE, AND E. E. ROTHBERG, ‘‘Asymptotic experimental analysis of the Held-Karp lower bound for the traveling salesman problem,’’ in preparation.
16. D. S. JOHNSON, J. L. BENTLEY, L. A. MCGEOCH, AND E. E. ROTHBERG, ‘‘Near-optimal solutions to very large traveling salesman problems,’’ in preparation.
17. S. KIRKPATRICK, ‘‘Optimization by simulated annealing: Quantitative studies,’’ *J. Stat. Physics* **34** (1984), 976-986.
18. B. KORTE, ‘‘Applications of combinatorial optimization,’’ talk at the 13th International Mathematical Programming Symposium, Tokyo, 1988.
19. J. LAM AND J.-M. DELOSME, ‘‘An efficient simulated annealing schedule: imple-

mentation and evaluation,'' manuscript (1988).
20. F. T. LEIGHTON, private communication (1989).
21. E. L. LAWLER, J. K. LENSTRA, A. H. G. RINNOOY KAN, AND D. B. SHMOYS, *The Traveling Salesman Problem*, John Wiley & Sons, Chichester, 1985.
22. S. LIN, ''Computer solutions of the traveling salesman problem,'' *Bell Syst. Tech. J.* **44** (1965), 2245-2269.
23. S. LIN AND B. W. KERNIGHAN, ''An Effective Heuristic Algorithm for the Traveling-Salesman Problem,'' *Operations Res.* **21** (1973), 498-516.
24. F. MARGOT, ''Quick updates for p-opt TSP heuristics,'' *Operations Res. Lett.* **11** (1992), 45-46.
25. O. MARTIN, S. W. OTTO, AND E. W. FELTEN, ''Large-step Markov chains for the traveling salesman problem,'' *Complex Systems* **5** (1991), 299-326.
26. H. MÜHLENBEIN, M. GORGES-SCHLEUTER, AND O. KRÄMER, ''Evolution algorithms in combinatorial optimization,'' *Parallel Comput.* **7** (1988), 65-85.
27. G. REINELT, ''TSPLIB—A traveling salesman problem library,'' *ORSA. J. Comput.* **3** (1991), 376-384.
28. D. D. SLEATOR AND R. E. TARJAN, ''Self-adjusting binary search trees,'' *J. Assoc. Comput. Mach.* **32** (1985), 652-686.
29. R. E. TARJAN, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.
30. A. C. YAO, ''Should tables be sorted?,'' *J. Assoc. Comput. Mach.* **28** (1981), 615-628.