Working Paper Series

HEARIN CENTER FOR ENTERPRISE SCIENCE

HCES-06-03

The Satellite List and New Data Structures for Symmetric Traveling Salesman Problems

By
Colin Osterman
César Rego

The Satellite List and New Data Structures for Symmetric Traveling Salesman Problems[†]

Colin Osterman [a], and César Rego [a],

[a] Hearin Center for Enterprise Science, School of Business Administration, University of Mississippi, University, MS 38677, USA. {costerman, crego} @bus.olemiss.edu

Latest Revision: March, 2003.

Abstract — The problem of data representation is fundamental to the efficiency of search algorithms for the traveling salesman problem (TSP). The computational effort required to perform such tour operations as traversal and subpath reversal considerably influence algorithm design and performance. We propose new data structures—the satellite list and *k*-level satellite tree—for representing a TSP tour with a discussion of properties in the framework of general tour operations. Theory suggests that the satellite list data structure is superior in representation to its widely used counterpart, the doubly-linked list, as well as useful in improving and extending the specialized 2-level tree structure for symmetric graph-based optimization problems. The *k*-level satellite tree representation is shown to be far more efficient than its predecessors.

Keywords: Metaheuristics, data structures, combinatorial optimization, graph-based problems, traveling salesman problems.

[†] This research has been supported in part by the Office of Naval Research (ONR) grant N000-14-01-1-0917.

1. Introduction

Possibly the most basic and important building block of a lean search algorithm for the traveling salesman problem (TSP) and possibly the least studied is the primary data structure. To find good solutions for the largest TSP instances, a cleverly designed computer code must be employed, and the quality of the code depends greatly on the solid foundation an appropriate data structure can provide. The problem at hand is no longer the TSP but instead the computerized TSP, which brings with it the additional consideration that the solution and the procedure be represented effectively in memory. The use of an inappropriate data structure can greatly increase the time complexity of the search algorithm even if the theoretical complexity appears acceptable. Therefore this paper is specifically concerned with the computer modeling of paths and cycles and the efficiency to be gained by appropriate tour representation.

The TSP is solved by finding the least cost Hamiltonian cycle visiting n cities or nodes in a graph. In graph theory, the TSP is defined as a graph $G = (V, A)$ with n vertices (or nodes) $V = \{v_1, \cdots, v_n\}$ and a set of edges (or arcs) $A = \{(v_i, v_j) \mid v_i, v_j \in V, i \neq j\}$ with a non-negative cost (or distance) matrix $C = (c_{ij})$ associated with A. The problem's resolution consists of determining the minimum cost Hamiltonian cycle on *G*. In this paper, we consider the symmetric version of the problem $(c_{ij} = c_{ji})$, which satisfies the triangular inequality $(c_{ij} + c_{jk} > c_{ik})$. The term *node* has slightly different uses in the realm of graph theory than it does in the realm of computer science; since the scope of this paper spans both, the term is used alone only when referring to a city or node in the graph. When referring to data constructs, the term is accompanied by specific names to avoid confusion.

An ideal primary data structure for the TSP should be versitile enough to handle a wide range of structures (neighborhood reference structures and infeasible tours other than Hamiltonian cycles). More importantly, an ideal data structure should allow the procedure to perform all needed operations on the tour very quickly. If one structure cannot achieve these goals simultaneously, then it becomes important to study the tradeoffs among the strengths and weaknesses of different data structures, which will vary with different algorithms. To this end, Fredman, Johnson, McGeoch, and Ostheimer [3] provide a good comparison of the array (linked list) representation to the splay tree, 2-level tree, and segment tree representations. We, however, show that there exists a structure that is exceptionally well suited for the symmetric TSP.

This paper is organized to familiarize the reader with issues surrounding TSP data structures and to present our newly discovered structures. Section 2 recognizes prior research and significant contributions in TSP data structures. It also includes brief descriptions of the generic linked list structure and the specialized 2-level tree structure, as well as an overview of the theoretical issues related to the use of these structures in TSP algorithms. Sections 3 and 4 describe two new data structure designs, the satellite list and the *k*-level satellite tree, the latter of which incorporates the former. The new structures are counterparts of the linked list and the 2-level tree and may be thought of as the result of improvements on these structures that resolve the issues presented in Section 2. Section 5 provides a summary and conclusions. Supplementary information on the data structures is given in the appendix.

2. Prior Work In TSP Data Structures

Prior work on data structures for the traveling salesman problem has been limited but productive. A generic structure initially used was the doubly-linked list, also known as the array representation. This is the structure by which more recent structures are benchmarked.

Sleator and Tarjan [11] introduced the splay tree; its representation of a TSP tour is a natural idea and has been independently proposed by several groups of researchers. The splay tree represents a tour as a binary search tree with a city at each vertex. A special reversal bit indicates whether the subtree rooted at a vertex should be traversed in order or in reverse. The splay tree, chosen for its ability to compute a subpath reversal, performs the operation in $O(\log n)$ time, albeit at a high overhead cost associated with accessing adjacent nodes. This reflects the lowest complexity for the subpath reversal operation of any data structure discovered to date. While academically interesting, the splay tree representation has not thus far proved competitive in practice due to its high traversal-related overhead costs. Fredman et al. [3] report that the constant factor is too high to make the structure competitive for n as high as 10^6, yielding asymptotic arguments irrelevant. Therefore, we refer the reader to this reference for a more thorough description of this structure and its properties.

The 2-level tree, proposed by Chrobak, Szymacha and Krawczyk [2], divides the tour into approximately $\sqrt{n}$ segments, each containing as many nodes and grouped under a parent node. This structure improves significantly on the linked list in path traversal and, consequently, speeds up the subpath reversal operation as well. Its effectiveness has been demonstrated by independent implementations due to Fredman et al. [3], Gamboa, Rego, and Glover [4], and numerous participants of the 2001 DIMACS TSP Challenge (Johnson, McGeogh, Glover and Rego [8]). The theory behind the 2-level tree contributes much to our final structure and is examined in detail in Section 2.3.

Fredman et al. [3] present the so-called segment tree representation, crediting the idea to private correspondence from Applegate and Cook. We found that the segment tree was actually an algorithm-specific scheme for doing more efficient look-aheads in the search process rather than a true basic structure. The underlying structure in this case was admitted to be the array representation, although it may be possible to use the scheme in combination with other structures.

2.1 The Generic Linked List structure

The most natural generic list structure for representing a TSP tour, the circularly doubly-linked list, requires little discussion. Each city is stored as a client of a list node structure, which contains references to the nodes of the cities that precede and follow it in the tour in addition to the client. Often, rather than store a value for the client in the structure, the cities are associated with the node structures' indices, so that each list node only contains references to adjacent nodes. The list is considered circularly linked because the tour is a cycle rather than a path, although a doubly-linked list can certainly be used to represent acyclic reference structures.

2.2 Traversal and Subpath Reversal

The set of needed operations for a TSP algorithm generally includes two important classes: traversal and subpath reversal.

Traversal operations are those that require a procedure to follow a pointer or sequence of pointers in order to alter or obtain information about one or more related nodes in the same path. These operations may include finding the next/previous node relative to the current node, determining a path between two nodes, attaching labels to all nodes in a subpath, and others. With a doubly linked list, for example, a *Next*/*Previous* query can easily be performed in constant time—a matter of accessing the "Next"/"Previous" pointer. The cost of traversing a path is clearly proportional to its size.

Subpath reversal is the alteration of a graph such that some subpath in the induced graph is reversed relative to the path that contains it. In the context of a TSP tour, the subpath reversal is often equated to the removal of two arcs *(a,b)* and *(c,d)* and the addition of two others *(a,c)* and *(b,d)* in the path (*a*,*b*,...,*c*,*d*). This is the well-known 2-opt move (Lin [9]). A problem that arises when coding a subpath reversal motivates the first new data structure proposed in this paper. The difficulty is that current data structures make the task of performing a subpath reversal while maintaining a feasible tour representation too computationally taxing.

A simple example can best describe how the problem occurs. Suppose a local search (maintaining a feasible TSP tour while considering changes to improve it) is performed on a graph using 2-opt moves. The search chooses two arcs *(a,b)* and *(c,d)* to be deleted in favor of another two arcs *(a,c)* and *(b,d)*. If the resulting cycle is traversed from any point in either direction, exactly one of the subpaths between the new arcs (e.g. the *b*-*c* path) must be traversed in reverse order—a subpath reversal. Let us assume a linked list representation is used (an example of the C code for this operation is provided in the appendix). The obvious part of the move is updating the arcs among *a*,*b*,*c*, and *d* by assigning *c* to follow *a*, *a* to precede *c*, *d* to follow *b*, and *b* to precede *d*. Now, however, the "Next" and "Previous" pointers of a node chosen from the path between *b* and *c* no longer reference the correct nodes. For example, *c* now follows *a*, but the node that now follows *c* is the node that previously preceded *c*; so the pointer must be updated. Completing the move and maintaining a readable tour involves swapping the "Next" and "Previous" pointers in each node of the reversed subpath.

With a linked list structure, the expense of the subpath reversal grows embarrassingly with problem size ($O(n)$) and contributes significantly to the time complexity of the search, since the operation is clearly material in the total running time of the 2-opt procedure. It turns out that this operation is also common and material to today's most advanced algorithms: the Lin-Kernighan algorithm and the Stem-and-Cycle Ejection Chain algorithm. Hence the challenge is to find a data structure that allows a given algorithm to make the move and restore a readable structure efficiently.

Indeed, the desire to perform both traversal and subpath reversal operations efficiently motivate the design of the second new structure; it is a result of the separate but simultaneous resolution of these two issues.

2.3 The 2-Level Tree

To date, the most acclaimed data structure adopted for use in TSP algorithms is the 2-level tree, originally proposed by Chrobak, Szymacha and Krawczyk [2]. Lin-Kernighan algorithms with effective implementations of the 2-level tree data structure have shown dramatic gains in efficiency over linked list implementations. Some of these are Fredman et al. [3], Johnson and McGeoch [7], Neto [10], Appelgate and Cook [1], and Helsgaun [6]. Gamboa, Rego, and Glover [4] demonstrate the striking computational outcomes obtained from implementing the Stem-and-Cycle Ejection Chain method with this structure.

The key to the 2-level tree's practical effectiveness is its ability to provide a framework for quickly traversing a path between two nodes. The structure is also useful for storing information that is likely to be duplicated among nodes in a series. It manifests these properties by dividing a given path organized as a doubly-linked list into segments and assigning a shepherding "parent node" to each segment. The parents are all linked with a doubly linked list, so that a path among them can be traversed in $O(\sqrt{n})$ time. A diagram of a 2-level tree structure is shown in Figure 2.1.

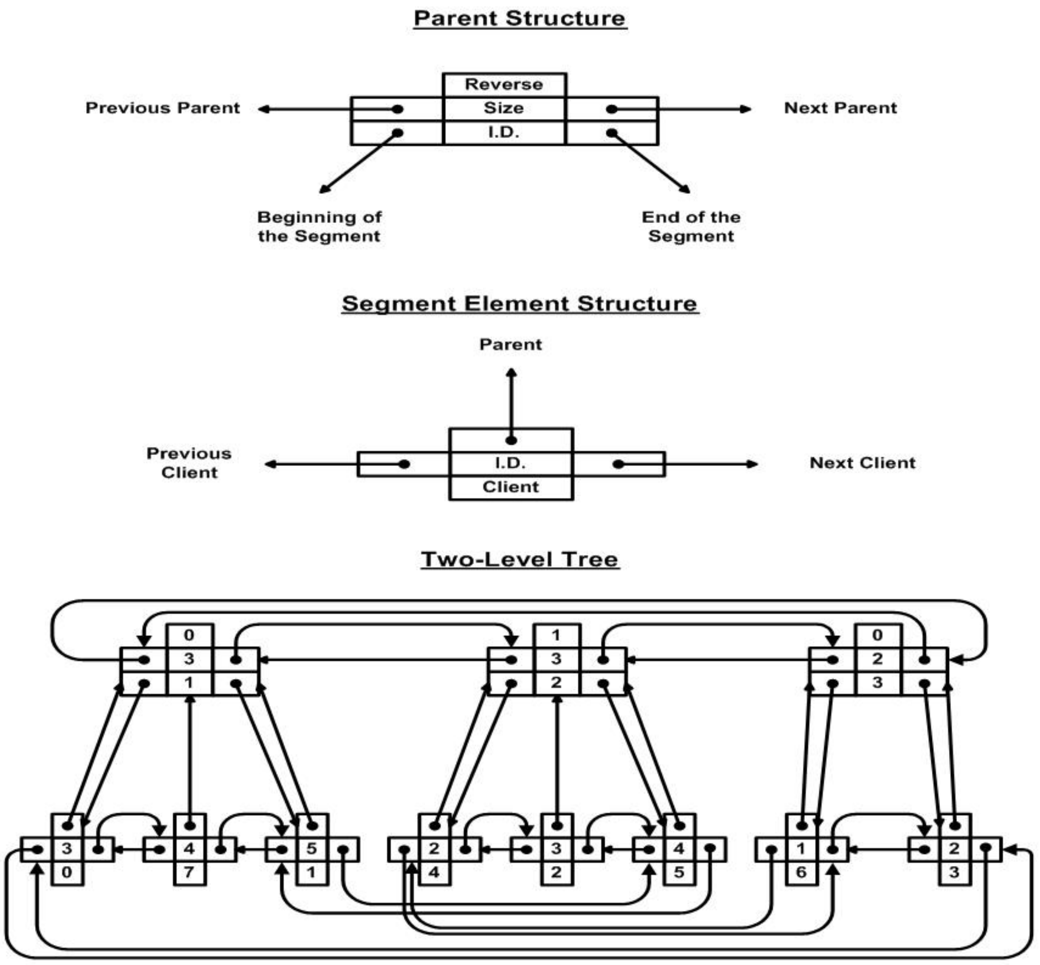


Figure 2.1. The 2-Level Tree Structure

Each element of a segment contains a pointer to the associated parent node, the index of the client it represents ("Client"), and a sequence number ("I.D."), in addition to the "Next" and "Previous" pointers. The "Client" data member represents the index of the city represented by the element. The "I.D." data member identifies the relative position of a client node within a segment. Although we find that the structure may operate without this data member, it facilitates the *Between* operation (as defined by Fredman et al. [3]) and can be used to quickly calculate sizes of potential segments if such information is beneficial for maintaining tree balance and computational efficiency.

Each parent contains pointers to the next and previous parents as well as to the clients that begin and end the segment. The "Reverse" data member switches the meaning of the "Next" and "Previous" pointers in the client nodes within the segment. "Size" gives the number of elements in the segment, and "I.D." is a sequence number. Again, we find certain data members to be possibly useful for balancing but not essential to the core structure; in the case of the parent structure, these data members are the pointers to the beginning and end of the segment, "Size," and "I.D." Although not essential, "Size," the segment endpoint pointers, and the associated tree balancing techniques are required to guarantee the worst-case time boundaries shown for various operations in Fredman et al. [3]. "I.D." serves a similar role in the parent structure as it does in the client structure in facilitating the *Between* operation.

If the "Reverse" switch is on, the meaning of the "Next" and "Previous" pointers in the segment elements is reversed, and the segment is meant to be traversed in reverse order. However, the meaning of the "Next" and "Previous" pointers in the parent nodes is not reversed. In the 2-level tree in Figure 2.1, the middle segment is switched on, indicating that "Next" clients be found by following "Previous" pointers of nodes in this segment. Notice that in order to reverse the subpath (5,2,4), it is necessary only to reconstruct the pointers among nodes 1, 4, 5, and 6, and flip the reverse bit in the parent node, while the pointers among the intermediate nodes in the segment remain unchanged.

Since the 2-level tree groups client nodes together into segments under parent nodes, information that is common to a number of clients in a series can be stored solely in the parent node, instead of duplicated in every client node. This is not only true for the "Reverse" bit, but can be applied in general, as exemplified by the "presence bit" proposed by Gamboa, Rego and Glover [4].

For large problems the 2-level tree greatly reduces the effort required to reverse a subpath (compared to the linked list), but it is not the best. To find the reasons, let us examine more closely what happens when a subpath is reversed with a 2-level tree.

We employ the previous conventions and notation regarding subpath reversal, except that the tour is represented with a 2-level tree. Suppose the subpath that we intend to reverse (*b,…,c*) contains many segments and that the end-segments are whole (since we can insure this with cut and merge operations if necessary). The obvious necessity is to update the arcs at the fractures, replacing arcs (*a,b*) and (*c,d*) with (*a,c*) and (*b,d*). The mechanics of these updates are slightly more interesting with the 2-level tree. First, the pointers in both the parent list and the client list must be updated. Second, instead of assigning *a* to precede *b*, as is done with the linked list, *a* is assigned to follow *c*, while *c* is still assigned to follow *a*. Similarly, *d* is assigned to precede *b*. This is so that the pointers in *b* and *c* remain consistent with their respective segments. Since (*b,…,c*) contains multiple segments, the pointers between parents are updated as usual, with *Parent*(*i*) taking the place of node *i*. To restore a readable structure, the parent nodes between *Parent*(*b*) and *Parent*(*c*) are traversed, and 1) the reverse bit is flipped to switch the meaning of the "Next" and "Previous" pointers within the segment, and 2) the parent's "Next" and "Previous" pointers are swapped to reflect the appropriate orientation. Since there are $\sqrt{n}$ parent nodes, the worst case time complexity is limited to $O(\sqrt{n})$.

The reversal operation must visit and alter the parent nodes rather than the client nodes themselves, effectively shortening the reversal operation. On the other hand, the subpath must still be traversed and altered in order to regain a feasible structure. Although its efficiency at reversing a subpath has been touted, clearly the strength of the 2-level tree lies in its enhanced traversal abilities, and not in any property specific to subpath reversal. These abilities could be further improved if the tree could be augmented to further reduce the relative size of a parent traversal efficiently and effectively. Additionally, the subpath reversal operation could be significantly simplified if it was not necessary to perform the traversal step at all. In Section 4, we show that our data structure extends and generalizes the 2-level tree data structure to solve both the traversal and the subpath reversal problems.

Making changes to the structure often requires that segments be regrouped. Regrouping is accomplished with cut/merge operations, which consist of splitting a segment between two elements and combining two segments under one parent, respectively. The important thing to note about these operations is their necessity and contribution to time complexity. Most moves that add and delete arcs between clients that are not segment endpoints require that a cut/merge operation be performed. The operation boils down to reassigning the "Parent" pointer of each segment element; thus the worst case cost turns out to be $O(\sqrt{n})$ when tree balance is maintained. Therefore, these operations make at least as significant a contribution to the total computational cost of a move as subpath reversal.

Finally, the advantages of the 2-level tree over the linked list do not come without cost. Traversal operations with this structure require knowledge of the correct orientation; thus, the structure cannot be read without accessing the parent nodes. Before using the "Next" or "Previous" pointers of a client node, the parent node must be accessed and the "Reverse" bit checked. This additional computation increases the cost of traversing client nodes by a constant factor. In addition, the memory requirement is higher than it is for the linked list, though acceptable. Even supposing that the balancing components are omitted, there is still an additional pointer to the parent for each client node, and also some requirement for the parent nodes, though the proportion of this requirement grows small as the problem size grows large.

3. New Data Structures for the TSP: The Satellite List

There are two fundamental differences between the data structures we propose and their predecessors. The first difference stems from a contradiction in beliefs regarding what is known about tour representation. The second difference involves extending and leveraging the theoretical advantage of the 2-level and is discussed in Section 4.

It is a commonplace observation that imposing additional constraints on a problem can only reduce its feasible region. We find that this principle applies meaningfully to the problem faced in seeking an ideal data structure, where requiring that the tour be oriented unnecessarily constrains our design choices. Fredman et al. [3] observe that, "tour representation can be simplified by requiring that the tour be oriented." However, we respectfully disagree with the assumption that "simpler is better" and present an analysis of the alternative. The new data structure designs we propose are symmetric and overcome the inherent weakness present in the doubly-linked list. A notorious consequence of this structure's asymmetric nature is its inability to efficiently reverse the order of the nodes in a given subpath. We show that our new list design achieves the desired flexibility without giving up any time or memory efficiency relative to the doubly-linked list.

Since the 2-level tree is defined as having linked lists as components (as discussed), it inherits a similar fixed orientation problem. However, we show that the 2-level tree may adopt our new list design in place of its linked list components. We also show that the new list design facilitates the generalization of the 2-level tree to additional levels and increases their potential benefit.

3.1 The Satellite List

Our new data structure, the *satellite list*, provides a basis for representing any symmetric path or cycle that may have been previously encoded using a doubly-linked list. The satellite list can operate in the same capacity as a doubly-linked list with the following key difference: the satellite list represents a tour without implying a fixed orientation of the path, which is why we qualify our claim for symmetric tours. The "next" node in the path depends on the current orientation, information that is naturally preserved in traversing the satellite list but otherwise neglected by the doubly-linked list. A primary consequence of avoiding a fixed orientation is that the subpath reversal operation is performed easily and in constant time. In addition, the satellite list retains the same efficiency as the doubly-linked list in terms of the memory it occupies and the commands it requires to access adjacent nodes.

To obtain a suitable structure that lacks the structural weakness of the linked list but retains its simplicity, its strict representation is disassembled. A linked list node contains two data members: a pointer to the previous client node and a pointer to the next client node. Pointers should operate in a symmetric fashion, suggesting that they point toward "adjacent" nodes, rather than "next" or "previous" ones. To accomplish this, the pointers are first removed from the structure and given their own structures, called *satellites*, whose sole purpose is handling the links among clients in the list. Each satellite points not to an adjacent list node, as in the linked list representation, but rather to the list node's satellite. To read the tour starting from a client, one of its satellites is chosen to begin the traversal. It is arbitrary which satellite is chosen, just as it should be, since it makes no difference in which direction a symmetric tour is traversed. The traversal operation then follows the satellite's pointer to the next satellite, the client is noted, and the process continues. Therefore, reading the tour involves traversing one of two distinct singularly linked lists of satellites and noting the associated cities.

3.2 Logical Representation

Figure 3.1 depicts a city represented first as a doubly-linked list node and then as a satellite list node. The distinguishing characteristic of the satellite list node is its avoidance of direct links with adjacent structures. Instead, its satellites link the cities indirectly. The dashed lines in the figure emphasize the parts of the structure that indicate relationships among components of the list node. From a satellite, there exists some means to immediately access the *complement satellite*. A similar relationship associates a satellite with its city. It is possible for these relationships to be constructed (e.g. storing a pointer or an index value of the desired component), but it is more efficient to take advantage of the implementation in which these relationships are implied, as is illustrated in more detail in the next section and the C code in the appendix.

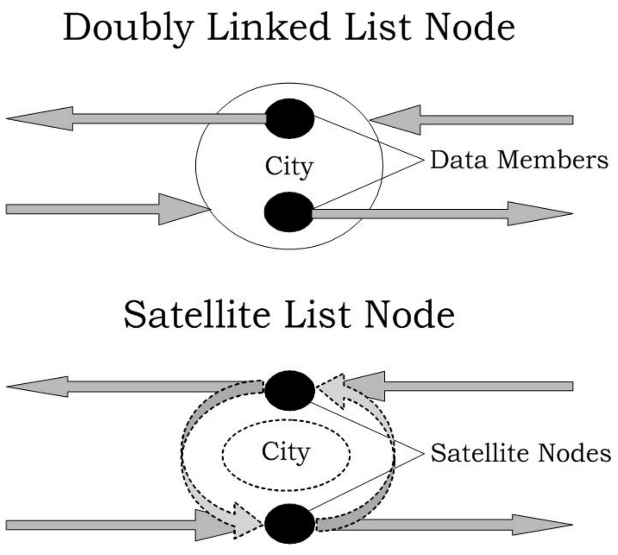


Figure 3.1. The Doubly-Linked List vs. the Satellite List

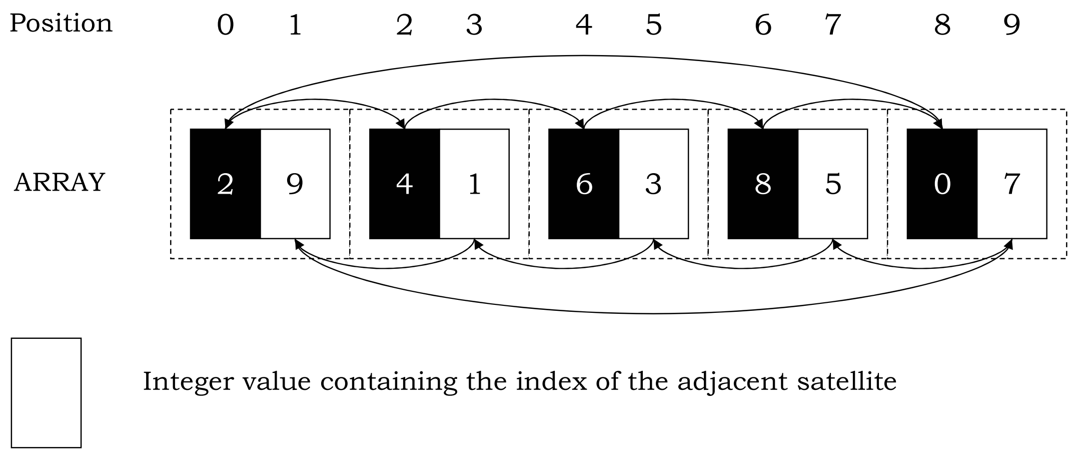
Figure 3.1 discloses the functional differences between the two types of nodes. The city represented by the doubly-linked list node is constructed in memory in such a way that its data members (typically "Next" and "Previous") cannot be referenced directly[†]. The data members are accessed by name, and are thus not interchangeable and necessitate an orientation. Also, since the data members store references to entire list nodes, the node from which a reference was obtained is not remembered. For instance, it is not correct to assume that a node has been obtained from the "Next" pointer of the preceding node because it is equally likely to have been obtained from the "Previous" pointer of the following node, and storing the information explicitly costs time and memory. On the other hand, this information remains evident when using the satellite list because each satellite is reached from a distinct source. Furthermore, two satellites sharing a list node operate as independent nodes in separate singly-linked lists, lending directional independence to the list node as a whole. Its unique symmetry allows the "direction" of the node to depend on how the node was obtained without incorporating any costly, explicit decision making into the code.

3.3 Efficient Physical Representation

A satellite list may be constructed and used without specifically declaring each satellite's links (pointers or indices) to its associated complement satellite or its client (city). This property reduces the number of pointers needed to represent a list node from four to two, cutting the memory requirement by half.

The tour is maintained in a single one-dimensional array of length twice n (where n is the number of cities). Each element in the array is a satellite node and contains the value of (or pointer to) its adjacent satellite. For each city, there exists two physical array positions (satellites), which, together, are considered a logical position representing the client. The city itself needs no physical position since it can be uniquely identified by the indices of its satellites. Figure 3.2 diagrams an example satellite list stored in an integer array representing the circuit (0,1,2,3,4,0).

[†] In pointer-supporting languages, it is, in fact, possible to obtain a direct reference for these members, but the reference can only be used to indirectly retrieve values that can also be stored directly, and so the reference is not meaningful.


Position
0 1 2 3 4 5 6 7 8 9
ARRAY
2 9 4 1 6 3 8 5 0 7
Integer value containing the index of the adjacent satellite

	In General	Bitwise Ops in C
Index of the Satellite:	i	i
ID number of the node (city):	$i/2$	$i >> 1$
Index of the complement satellite:	$i+1-2(i\%2)$	i^1

(the dashed line indicates that the boundary is logical)

Figure 3.2. Example of Efficient Satellite List Implementation

A key factor that makes the use of the efficient implementation desirable is the ease with which the implied relationships mentioned in the previous section can be found. Given some satellite, the queries are to determine its complement satellite and its client (city). The general formulas to compute these queries are given in the figure along with the equivalent bitwise operations in the C language. A satellite's client (city) can be computed by integer-dividing the index by 2 (no remainder). An identical result is achieved with the bitwise shift operator, ">>". To find a satellite's complement, the index should be incremented if it is even and decremented if it is odd. The "%" gives the remainder of division. A bitwise operator is also available for computing a satellite's complement satellite—the exclusive or, "^". These and other bitwise operators are extremely fast because they don't request any arithmetical or logical computation from the processor; thus, the measure of overhead they contribute to routine operations such as *Previous*() is insignificant. For languages that do not support bitwise operations, using the general formulas may be too costly. In this case, the computational overhead can be avoided by storing "client" and "complement" explicitly, although doing so doubles the memory requirement for the structure. Example C code for both full and efficient implementations and for queries using the bitwise operators is provided in the appendix.

Although Figure 3.2 shows all odd satellites pointing to odd satellites, the list is perfectly functional when some odd satellites point to even satellites, or vice versa. Indeed, this becomes the case for the endpoints of a reversed subpath. Interestingly, when a satellite list is organized so that the entire list can be read from just the even or odd satellites, the satellite list array can be unioned with a doubly-linked list structure, and the tour can be read correctly from either.

3.4 Memory and Time Efficiency

Unlike other data structures proposed for the TSP, the satellite list does not impose additional memory requirements for representing the tour. The memory required to store a tour represented by the satellite list is exactly equal to that required by the doubly-linked list—two pointer slots per city. This is not to say that additional fields cannot be added to a list node for use with specialized algorithms.

Also, the satellite list does not impose additional computational effort for the traversal procedures. Enumerating across several nodes is simply a matter of following a singly-linked list of satellite nodes; thus, the computational cost is the same as for the array representation. There is no need to check a “Reverse” bit (2-level tree) or to splay to the root of a tree (splay tree).

3.5 Subpath Reversal

We return to the subpath reversal issue, discussed in Section 2.2. This problem has been of great interest to scholars studying TSP data structures due to its occurrence in both classic k-opt and state-of-the-art search methods. In fact, this issue motivates the satellite list design.

The tree-based data structures mentioned in Section 2 were proposed with the aim of lowering the computational complexity of the subpath reversal operation. The splay tree claims to handle the operation in $O(\log n)$ time, while the 2-level tree claims $O(\sqrt{n})$ time, with better results for problems as large as $n = 10^6$ due to lower overhead costs. Clearly, however, because of the satellite list’s symmetric design, the subpath reversal operation is a natural one and is performed easily in constant time, $O(1)$. The ease of the operation is illustrated in Figure 3.3.

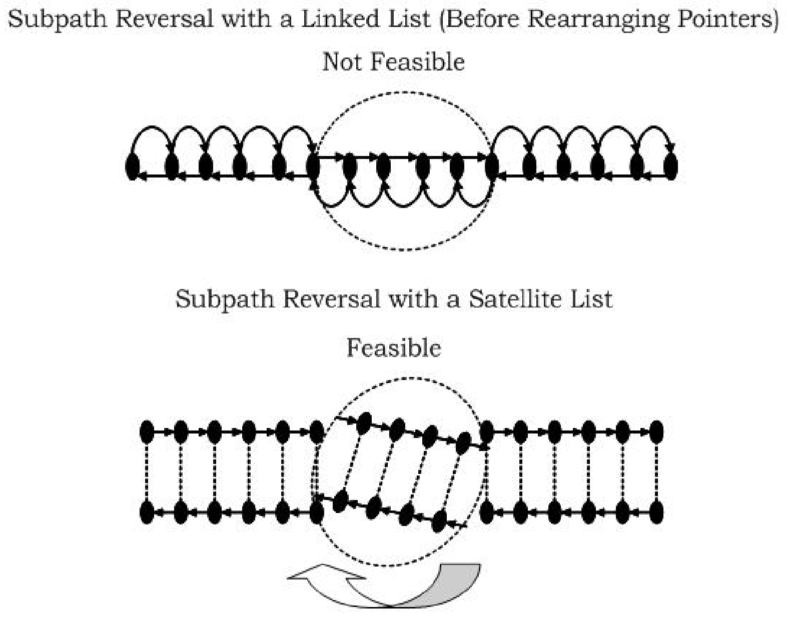

Subpath Reversal with a Linked List (Before Rearranging Pointers)
Not Feasible
Subpath Reversal with a Satellite List
Feasible

Figure 3.3. Subpath Reversal with a Linked List vs. a Satellite List

Figure 3.3 depicts isolated subpaths for a doubly-linked list (top) and a satellite list (bottom), where straight arcs point to the following nodes in the list, and the curved arcs point to preceding nodes (only applicable to the doubly-linked list). An analogy would be to view the linked list as a one-way street and the satellite list as a two-lane road. As is easily seen, the subpath in the satellite list reforms the original structure when rotated 180 degrees—only pointers associated with nodes in the broken edges need to be changed. The subpath in the doubly-linked list, however, does not match correctly when rotated—every pointer associated with nodes in the subpath must be changed. The "street" analogy would have traffic flowing smoothly only on the two-lane road if a section of the road were reversed. In the linked list, additional computational time must be taken to correct the intermediate arcs in the reversed subpath. Although the 2-level tree reduces the time complexity of this correction, the satellite data structure eliminates the problem.

The data structure currently thought to be the state-of-the-art, the 2-level tree, maintains for each level a doubly-linked list. Therefore, it is a natural idea to use the satellite list to improve the 2-level tree for any algorithm that is thought to be most efficiently implemented with this structure (i.e. *k*-opt procedures, their generalization the Lin-Kernighan procedure, and the Stem-and-Cycle Ejection Chain method).

4. New Data Structures for the TSP: The *k*-Level Satellite Tree

The *k*-level satellite tree makes use of both "satellite" and "2-level tree" design concepts to achieve superior performance. Each component of the tree is structured as a satellite list rather than a linked list. Also, the idea behind the 2-level tree has been extended to increase its leverage on traversal speed. The leverage that the 2-level tree allows over the linked list is derived from its ability to be traversed more quickly. The linked list structure still exists within the 2-level tree, but for traversal activity, we may traverse parent nodes rather than client nodes, reducing the complexity of the cost of traversal from $O(n)$ to $O(\sqrt{n})$. So, to further perfect the design of our structure, we have expanded on this idea by allowing *k* levels to the tree instead of just two. This enables the tree to be traversed with a complexity $O(\ln n)$ rather than $O(\sqrt{n})$.

4.1 The k-Level Satellite Tree

The *k*-level satellite tree is plainly stated as follows. The lowest and largest level (level 1) consists of a satellite list, containing all client nodes or cities in the traveling salesman problem. Level *k* is also a satellite list containing approximately $n^{1/k}$ parent nodes, analogous to the second level in the 2-level tree. Levels 2 through $k - 1$ contain intermediate parent nodes whose function is to vertically link sub-segments into larger segments, ultimately grouping a large number of client nodes under a few parent nodes. Each satellite list is symmetric, and therefore it might be expected that the orientation of each is independent. However, for the *k*-level satellite tree, directional consistency is maintained throughout all levels by following a consistent practice in the assignment of vertical pointers. Vertical pointers are those from client nodes to intermediate parent nodes, intermediate parent nodes to higher intermediate parent nodes, and pointers from intermediate parent nodes in level $k - 1$ to parent nodes. Figure 4.1 gives an illustration of a 3-level satellite tree that holds the TSP tour represented in Figure 2.1.

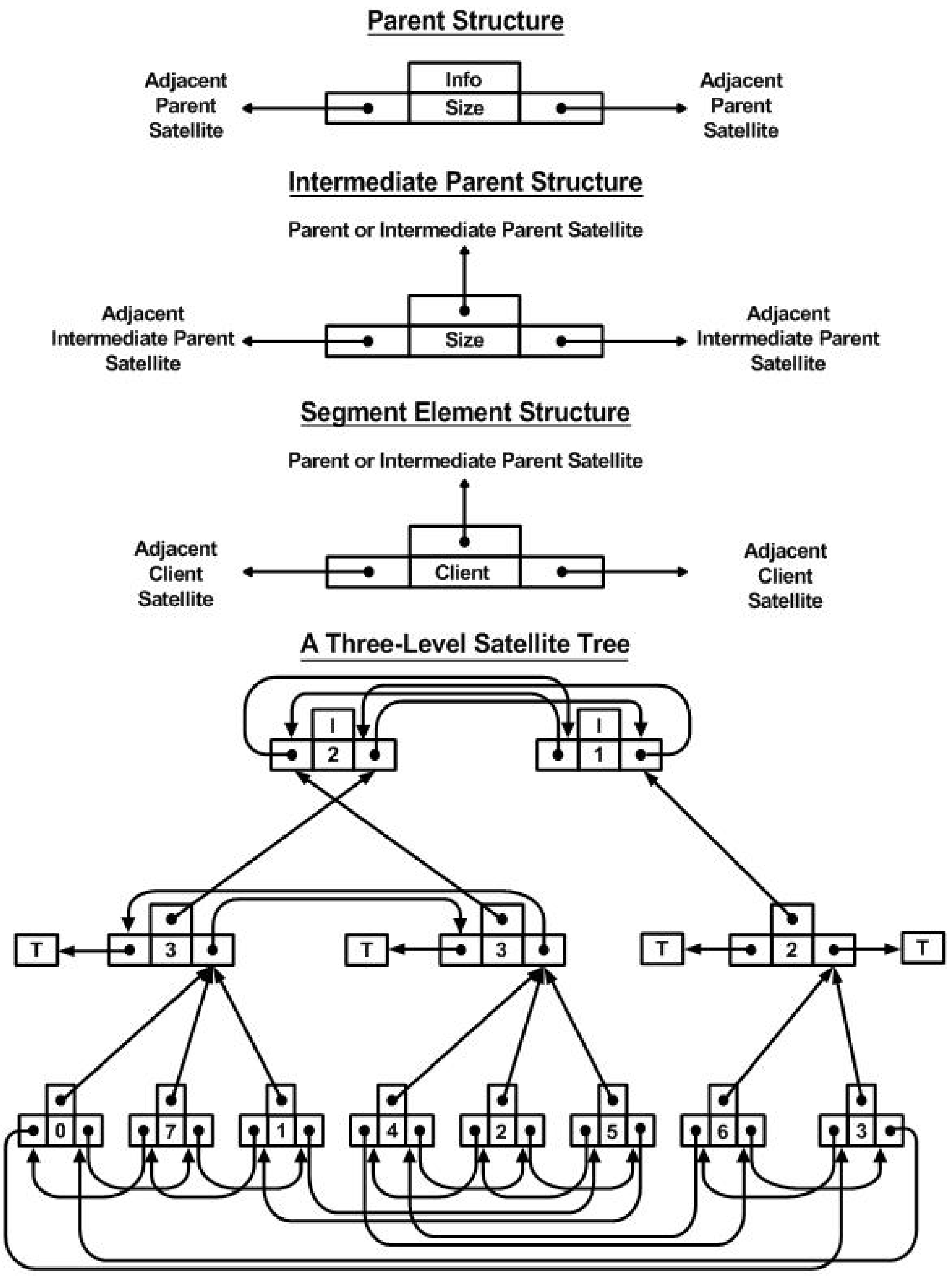

Parent Structure
Adjacent Parent Satellite
Info
Size
Adjacent Parent Satellite
Intermediate Parent Structure
Parent or Intermediate Parent Satellite
Adjacent Intermediate Parent Satellite
Size
Adjacent Intermediate Parent Satellite
Segment Element Structure
Parent or Intermediate Parent Satellite
Adjacent Client Satellite
Client
Adjacent Client Satellite
A Three-Level Satellite Tree
I
2
I
1
T
3
T
3
T
2
T
0
7
1
4
2
5
6
3

Figure 4.1. The *k*-Level Tree

In comparing the diagrams in Figure 4.1 to those in Figure 2.1, it is important to recall that the pointers shown in a satellite tree point to satellites, not whole clients or parents. The lowest and highest levels consist of complete satellite lists containing client nodes and parent nodes, respectively. Each level l of the (k – 2) intermediate levels has approximately $n^{(k-l+1)/k}$ intermediate parents grouped into approximately

$n^{(k-l)/k}$ segments, each containing approximately $n^{1/k}$ elements[†]. Each segment in the intermediate level(s) is maintained in a separate satellite list rather than linked with other segments in the same level. There appears to be a cost (but no benefit) to linking intermediate parent segments; thus, end-nodes of these segments are neatly terminated[‡] (denoted "T" in Figure 4.1). The great benefit, of course, to linking all the parent nodes in the kth level is the creation of a much shorter path between client nodes in different parent segments (size = $cn^{1/k} + 2(k-2)$) as opposed to the obvious path given by the satellite list in the first level (size = $c_0 n$), where c and c_0 are constants.

Parents can also serve to contain information common to all client nodes in the entire segment (denoted "Info" in Figure 4.1). The "Info" data member is for general use and, in contrast to the "Reverse" bit in the 2-level tree, is not necessary to read the structure! The tour can be read from the satellite list in the first level. Storing information in the top level of the tree when possible as opposed to the bottom level can achieve tremendous efficiency gains if that information is updated periodically for entire segments. "Info" is thus specified generally, and the type of information it contains is algorithm specific.

The "Size" data member plays the same role here as it does in the 2-level tree, as discussed thoroughly in Fredman et al. [3]. The worst case guarantees (to be discussed) for cuts, merges and traversals can be met as long as the segment sizes are kept between $\frac{1}{2}n^{1/k}$ and $2n^{1/k}$. As with the 2-level tree, we find that this extra data member and the associated tree balancing techniques are unnecessary in practice.

Not included in our structure is the sequence number ("I.D." in Figure 2.1), which is included in the parent nodes of the 2-level tree primarily to facilitate an efficient implementation of the *Between*(*a,b,c*) operation, as defined in Fredman et al. [3]. Traversing a path between the parent nodes of *a* and *c* can be avoided by comparing sequence numbers to determine if *b* lies in the path from *a* to *c*. In the case of the 2-level tree, the result is a decrease in the worst-case time bound from $O(\sqrt{n})$ to $O(1)$. Although it is possible to include a sequence number in parents of the *k*-level tree, it is not theoretically relevant, since (as we will show) the cost of traversing the parent nodes becomes constant when *k* is chosen optimally.

In the spirit and fashion of the satellite list, vertical pointers point not to whole parent nodes but to satellites of those nodes. A traversal can rely on the vertical pointers to jump to higher levels of the tree and continue in the same direction. For example, suppose we arrive at a satellite of a client node (1st level) during some traversal. Then we are "moving" in some current direction, and that information is reflected by the satellite on which we arrive. In order to access the client's intermediate parent node, we follow the intermediate parent pointer to a satellite of the intermediate parent, recognizing that we would follow the same pointer if we were to arrive at the client on the complement satellite (i.e. from the opposite direction). To obtain the satellite of the intermediate parent that is consistent with our current direction, we compute a bitwise operation involving the satellite of the client and the satellite of the intermediate parent obtained from the pointer. This arrangement helps to keep our memory requirement low, since we need only one vertical pointer per list node (rather than one per satellite)

[†] The tree in the diagram is not balanced in exactly this way in order to retain similarity with its counterpart in Figure 2.1.

[‡] Termination may be accomplished by pointing to NULL or -1.

to interconnect the levels. The same principal applies throughout the tree so that directional consistency is maintained regardless of the level negotiated.

4.2 Choosing the Optimal *k* and Consequences Thereof

The best value of *k* depends not only on *n*, but also the nature of the algorithm. Factors unique to a particular implementation, such as the size of the average traversal and the expected number of traversals required relative to merging and cutting activities, determine trade-offs between the costs of managing additional levels and the benefits they provide. Thus, we must recommend some experimentation in determining the best *k*. However, some guidelines given by theory simplify and otherwise reduce necessary experiments to a matter of estimating the relevant constants associated with different types of operations.

Whether the algorithm is a *k*-opt, LK, or S&C, the commands associated with operating on the *k*-level tree belong to one of three types:

1) those that execute for each of several nodes in a segment at each level but one (e.g. cut/merge operations),
2) those that execute for each of several nodes in a single segment (e.g. parent traversal operations), and
3) those that execute once for each level but one (e.g. parent access operations).

Given these types, the worst-case cost of operating the structure can be written:

$$C(n,k,c_1,c_2,c_3) = c_1(k-1)n^{1/k} + c_2 n^{1/k} + c_3(k-1)\,. \qquad (1)$$

Clearly, this expression implies the time complexity of operating the structure is at most $O(n^{1/k})$ if *k* is assumed to be fixed for all *n*. Here, it is assumed that the tree-balancing techniques are employed to guarantee a size of $\frac{1}{2}n^{1/k} \le s \le 2n^{1/k}$ for any segment s, justifying the assertion that the cost of operating on several nodes in a segment is proportional to $n^{1/k}$.

The optimal number of levels for the tree minimizes its operating cost. Since constants do not affect the optimization, *C* is normalized to obtain one fewer parameter:

$$\text{MIN}_k\, C'(n,k,c_2',c_3') = (k-1)n^{1/k} + c_2' n^{1/k} + c_3'(k-1)\,, \qquad (2)$$

where $c_2' = c_2 / c_1$ and $c_3' = c_3 / c_1$ are the relative costs of type 2 and 3 operations to type 1 operations, respectively.

Unfortunately, there exists no closed form solution for the general problem of choosing k to minimize cost given the other parameters. This being so, suppose that a simplifying assumption can be made:

$$c_1 = c_2 \qquad (3)$$

It may not be unreasonable that these two parameters are approximately equal; in our experience, a given move often has the same number and type of commands in each of operation types 1 and 2. Then the cost expression in equation (1) becomes

$$C(n,k,c_1,c_3) = c_1 k n^{1/k} + c_3(k-1)\,. \qquad (4)$$

The problem of choosing k to minimize C becomes equivalent to

$$\text{MIN}_k\, C'(n,k,c_3') = kn^{1/k} + c_3' k\,, \qquad (5)$$

the first order condition of which gives

$$\frac{\partial Z}{\partial k} = c_3' + n^{1/k^*} - \frac{n^{1/k^*}\ln n}{k^*} = 0\,,$$

$$k^* = \frac{\ln n}{1 + PL(c_3'/e)}, \tag{6}$$

where *PL*(z) is the product log of z, which gives the principal solution for y in $z = ye^y$, satisfying the differential equation $\frac{\partial y}{\partial z} = \frac{y}{z(1+y)}$. Equation (6) provides the theoretical basis for $O(k^*)$ = $O(\ln n)$, since c_3' is a constant.

A quick check of the second order condition,

$$\frac{\partial^2 Z}{\partial^2 k} = \frac{n^{1/k^*}(\ln n)^2}{k^{*3}} > 0, \tag{7}$$

confirms that the point does indeed represent a minimum for all $n, k^* > 0$.

Since the average segment size is $n^{1/k}$, the optimal average size of a segment s* is:

$$\begin{aligned} n^{1/k^*} &= n^{(1+PL(c_3'/e))/\ln n} \\ &= e^{1+PL(c_3'/e)} \end{aligned} \tag{8}$$

So, the optimal average size of a segment does not depend on *n*! Instead, it depends on the constant term, c_3'. For example, if $c_3' = 0$ (there are no type 3 operations), the optimal segment size is just *e*; for larger values of c_3', *s** is larger.

Substituting the optimal segment size *s** for $n^{1/k}$ and *k** for *k* simplifies the modified cost given in equation (4) dramatically:

$$\begin{aligned} C(n,k^*,c_1,c_3') &= c_1 k^* e^{1+PL(c_3'/e)} + c_1 c_3'(k^* - 1) \\ &= c_1 \frac{\ln n}{1+PL(c_3'/e)} e^{1+PL(c_3'/e)} + c_1 c_3' (\frac{\ln n}{1+PL(c_3'/e)} - 1) \\ &= a \ln n - c_3, \end{aligned} \tag{9}$$

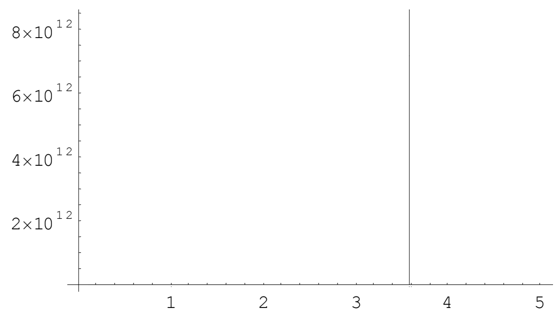
where $a = \frac{c_1}{1+PL(c_3'/e)}(e^{1+PL(c_3'/e)} + c_3')$

Hence the time complexity of operating the *k*-level satellite tree is $O(\ln n)$ when *k* is chosen optimally as opposed to $O(n^{1/k})$ when *k* is fixed.

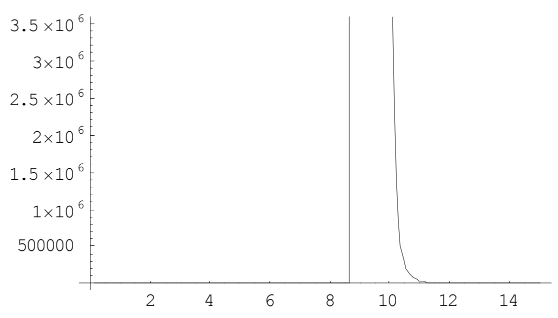
If the assumption in (3) does not hold, then the optimal segment size can still be considered constant for practical purposes. As a consequence, *k** can still be considered proportional to the natural logarithm of *n*. As previously noted, *k** cannot be written as a function of *n* in the general situation, and neither can the segment size, *s** = n^{1/k^*}. However, the relationship

$$n = e^{\frac{(c_2'-1)s^*(\ln(s^*))^2}{c_3' + s^* - s^* \ln s^*}} \tag{10}$$

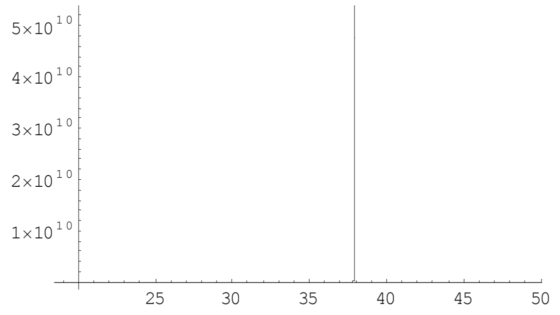
follows from the first order condition, and a numerical examination of this equation reveals that *n=f(s*)* has a singularity for some value of *s**, dependent on the constants.



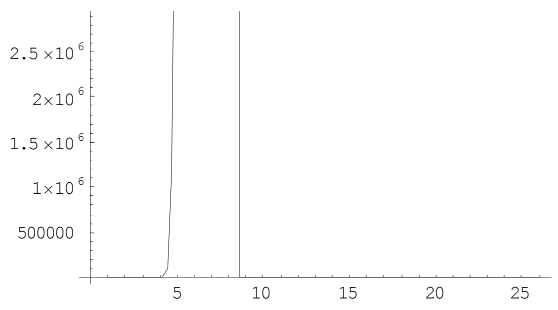
A. $c_2' = 1.01 \quad c_3' = 1.0$



B. $c_2' = 0.10 \quad c_3' = 10.0$



C. $c_2' = 1.01 \quad c_3' = 100.0$



D. $c_2' = 10.0 \quad c_3' = 10.0$

Figure 4.2. *n* as a Function of s^*

Figure 4.2 displays plots for equation (10) for varied values of c_2' and c_3', with *n* on the vertical axis and *s** on the horizontal. The plots provide confirmation that *s** can be considered constant for values of *n* of any significant size. Even in the extreme cases depicted in panels B and D, the variation in *s** that exists when c_2' is much different from 1 is small. As *n* grows large, *s** is bounded asymptotically by the value for *s** when $c_2' = 1$.

The plots also confirm the intuition regarding c_2' and c_3'. If the relative cost of managing a segment is low, then it is better to have larger segments and thus fewer levels (panels B and D). A similar result applies if the relative cost of managing levels is high (panel C). The values for the constants in Figure 4.2 are chosen to span a reasonable range. Likely values for c_3 are quite a bit higher than for c_1 or c_2. This is because one of the type 3 operations, accessing a parent node, is usually required in the evaluation of possible moves, a task that is performed more often than the actual moves.

In general, increasing *k* decreases the cost of traversing a segment but increases the number of levels to manage. Since type 1 operations involve traversing $k - 1$ segments, one cost component increases while the other decreases for this type. These components are separate in types 2 and 3. It is clear from the theory that as *n* increases, it is typically best to increase *k*, but keep the segment size the same. In other words, to manage additional clients, it is less costly to add levels to the tree than to inflate the segments. Therefore, a segment size that is sufficiently close to the optimal can be computed merely by determining good estimates for the relative values of the constants.

5. Summary and Conclusions

The satellite list representation allows a TSP tour to be represented without the encumbrance of orientation. Desirable properties that result include the performance of the subpath reversal operation in constant time, memory and time efficiency comparable to the linked list in general, and simpler, shorter code.

Situations may exist where abandoning the notion of direction in a TSP tour may not be helpful. First, as previously noted, the solution to an asymmetric problem cannot be represented without a defined orientation; thus, a satellite design doesn't make sense for the tour representation. Second, if it is simply more efficient for a particular search method to maintain a fixed orientation within the data structure, a satellite design will be of little use. No present TSP algorithms we know of gain efficiency with a fixed orientation, but if one were to be devised our data structure would offer fewer advantages than it provides for the types of algorithms currently employed. In any event, the satellite list can do no worse than the linked list, because orientation can always be enforced in a satellite list by imposing a restriction on the satellites. The need to impose such a restriction does not exist in general, and the satellite structure simplifies and shortens the code for symmetric TSP algorithms even in situations where efficiency gain is not as appreciable as in the algorithmic designs favored at present.

The k-level satellite tree structure expands on the 2-level tree by distributing the workload of traversals and cut/merges to several levels. As a result, the complexity of operating on the structure is reduced from $O(\sqrt{n})$ to $O(\ln n)$ when k is chosen optimally. A satellite design is utilized rather than the standard linked list design used in the 2-level tree, and the "Reverse" bit is discarded. In addition to its desirable properties for subpath reversal, the improved design allows the tour to be read from the structure without the need to access parent nodes, which reduces the cost of operations and shortens and simplifies the coding.

Due to the cut-and-merge requirements of modifying the k-level satellite tree, this structure can only perform a subpath reversal in constant time if the subpath is composed of whole segments. Otherwise the subpath reversal must incorporate a cut/merge operation, which is $O(\ln n)$ when k is chosen optimally. In this sense, the satellite list, which performs subpath reversals in $O(1)$ time, outperforms the tree, but it cannot be competitive overall due to its time complexity of $O(n)$ for traversals, whereas the k-level satellite tree boasts $O(\ln n)$ for traversals as well. A similar apparent discrepancy in complexity exists in comparing the k-level satellite tree to the 2-level tree. Accessing parent nodes, a routine operation, is $O(1)$ for the 2-level tree but $O(\ln n)$ for the k-level tree. This is because the number of levels is fixed at 2 in the former, whereas k varies with n in the latter. However, arguments to the effect that a structure fixed with just one level (satellite list) or two levels (2-level tree) can outperform a k-level tree are doomed to be fallacious. If true, such an argument would imply that choosing values for k greater than 1 or 2 is not optimal, contradicting our findings that k^* always increases with n. For this reason, and for reasons of theoretical complexity already indicated, we anticipate that the k-level satellite tree representation will prove useful for enhancing the computational performance of a broad range of TSP methods, including those based on algorithmic designs yet to be investigated[†].

[†] The more advanced reference structures introduced in Glover [5] are an example of methods in this category.

References

[1] D. Applegate and W. Cook, “Chained Lin-Kernighan for Large Traveling Salesman Problems,” *Technical report, Rice University*, 2000, http://www.isye.gatech.edu/~wcook/papers/chained_lk.ps.

[2] M. Chrobak, T. Szymacha, and A. Krawczyk, “A Data Structure Useful for Finding Hamiltonian Cycles,” *Theoretical Computer Science*, 71 (1990), 419-424.

[3] M. Fredman, D. Johnson, L. McGeoch, and G. Ostheimer, “Data Structures for Traveling Salesmen,” *Journal of Algorithms* 18 (1995), 423-479.

[4] D. Gamboa, C. Rego, and F. Glover, “Data Structures and Ejection Chains for Solving Large-Scale Traveling Salesman Problems,” *Hearin Center for Enterprise Science, Research Report* HCES-05-02, University of Mississippi, 2002.

[5] F. Glover, “New Ejection Chain and Alternating Path Methods for Traveling Salesman Problems,” *Computer Science and Operations Research* (1992), 449-509.

[6] K. Helsgaun, “An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic,” *European Journal of Operational Research* 1 (2000), 106–130.

[7] D. Johnson and L. McGeoch, “Local Search in Combinatorial Optimization,” chapter, *The Traveling Salesman Problem: A Case Study in Local Optimization*, pages 215–310, John Wiley and Sons, Ltd., 1997.

[8] D. Johnson, L. McGeogh, F. Glover and C. Rego, “8th DIMACS Implementation Challenge: The Traveling Salesman Problem,” *Technical report, AT&T Labs*, 2000, http://www.research.att.com/~dsj/chtsp/.

[9] S. Lin, “Computer Solutions of the Traveling Salesman Problem,” *Bell System Computer Journal* 44 (1965), 2245-2269.

[10] D. Neto, “Efficient Cluster Compensation for Lin-Kernighan Heuristics,” Department of Computer Science, University of Toronto, 1999.

[11] D. Sleator and R. Tarjan, “Self-adjusting Binary Search Trees,” *J. Assoc. Comput. Mach.* 32 (1985), 652-686.

Appendix—Implementation Details

Given here is C code for linked list and satellite list implementations. The purpose is to demonstrate some of their key differences.

1. Full linked list node structure

```
struct LinkedListNode
{
      struct LinkedListNode*  next;
      struct LinkedListNode*  previous;
      long                    client;
}
```

2. Full satellite list node structure

```
struct SatelliteNode
{
      struct SatelliteNode*   next;
      struct SatelliteNode*   complement;
      long                    client;
}
struct SatelliteListNode
{
      struct SatelliteNode satellite_0;
      struct SatelliteNode satellite_1;
}
```

3. Efficient linked list implementation

```
/* Eliminate "long client" declaration from LinkedListNode structure by
assuming the index "i" of the CLIENT array identifies the city. */
      struct LinkedListNode
      {
            long  next;
            long  previous;
      }
      struct LinkedListNode* CLIENTS;
      void init()
      {
            long N = NUMBER_OF_CITIES;
            CLIENTS = (struct LinkedListNode*)malloc(N*sizeof(struct
                  LinkedListNode));
            CLIENTS[0].previous = N-1;
            for(long i = 0; i < N; i++)
            {
                  if (i!=N-1) CLIENTS[i].next = i+1;
                  if (i!=0) CLIENTS[i].previous = i-1;
            }
            CLIENTS[N-1].next = 0;
      }
/* Returns the "next" city in the tour. */
      long next(long city_index)
      {
            return CLIENTS[city_index].next;
      }
/* Returns the "previous" city. */
      long previous(long city_index)
      {
            return CLIENTS[city_index].previous;
      }
```

```
/* Returns the city indexed (in this case, return = parameter). */
      long city(long city_index)
      {
            return city_index;
      }
/* Replaces arcs ab and cd with arcs ac and bd.  Implies the bc-path is
reversed. */
      void Reverse_Subpath(   long city_index_a,
                              long city_index_b,
                              long city_index_c,
                              long city_index_d )
      {
            long k = CLIENTS[city_index_b].next;
            while (k != city_index_c)
            {
                  temp_1 = CLIENTS[k].previous;
                  CLIENTS[k].previous = CLIENTS[k].next;
                  temp_2 = CLIENTS[k].next;
                  CLIENTS[k].next = temp_1;
                  k = temp_2;
            }
            CLIENTS[city_index_b].previous = CLIENTS[city_index_b].next;
            CLIENTS[city_index_c].next = CLIENTS[city_index_c].previous;

            CLIENTS[city_index_d].previous = city_index_b;
            CLIENTS[city_index_b].next = city_index_d;
            CLIENTS[city_index_c].previous = city_index_a;
            CLIENTS[city_index_a].next = city_index_c;
      }
```

4. Efficient satellite list implementation

```
/* Eliminate structure; replace with long array twice the size of N;
assume each city gets two satellites, for example, city "0" is assigned
satellite "0" and satellite "1". */
      void init()
      {
            const long DN = NUMBER_OF_CITIES<<1;
            SATELLITES = (long*)malloc(DN*sizeof(long));
            SATELLITES[1] = DN-1;
            for(long i = 0; i < DN; i+=2)
            {
                  if (i!=DN-2) SATELLITES[i] = i+2;
                  if (i!=0) SATELLITES [i+1] = i-1;
            }
            SATELLITES[DN-2] = 0;
      }
/* Returns satellite of the "next" city in the tour. */
      long next(long satellite_index)
      {
            return SATELLITES[satellite_index];
      }
/* Returns satellite of the "previous" city. */
      long previous(long satellite_index)
      {
            return SATELLITES[satellite_index ^ 1] ^ 1;
      }
/* Returns the city indexed (in this case, return != parameter). */
      long city(long satellite_index)
      {
            return satellite_index >> 1;
      }
/* N/A to LL implementation. */
      long complement(long satellite_index)
      {
            return satellite_index ^ 1;
      }
/* Replaces arcs ab and cd with arcs ac and bd.  Implies the bc-path is
reversed. */
      void Reverse_Subpath(   long satellite_index_a,
                              long satellite_index_b,
                              long satellite_index_c,
                              long satellite_index_d  )
      {
            SATELLITES[satellite_index_a] = satellite_index_c^1;
            SATELLITES [satellite_index_c] = satellite_index_a^1;

            SATELLITES [satellite_index_d^1] = satellite_index_b;
            SATELLITES [satellite_index_b^1] = satellite_index_d;

            // There are no intermediate pointers to flip
      }
```