On the Performance of Data Structures for the Traveling Salesman Problem

Colin Osterman
United States Navy
8 PUBLICATIONS **115** CITATIONS
SEE PROFILE

Dorabela Gamboa
Polytechnic Institute of Porto
17 PUBLICATIONS **178** CITATIONS
SEE PROFILE

Cesar Rego
University of Mississippi
73 PUBLICATIONS **1,504** CITATIONS
SEE PROFILE

On the Performance of Data Structures for the Traveling Salesman Problem

C.Osterman & C.Rego
Hearin Center for Enterprise Science, School of Business Administration, University of Mississippi, University, MS 38677, USA

D.Gamboa
Escola Superior de Tecnologia e Gestão de Felgueiras, Instituto Politecnico do Porto, Rua do Curral, Casa do Curral, Apt. 205, 4610-156, Felgueiras, Portugal.

ABSTRACT: Recent developments in data structures for the Traveling Salesman Problem (TSP) suggest lower computational costs for tour operations than previously thought possible. Theory suggests that extending current tree representations to consider a variable number of levels reduces operating costs to a minimum. For the symmetric TSP, using a *satellite* design frees the tour from a fixed orientation, resulting in efficient properties that greatly lower either the constant term or the complexity of the highest cost operations. We present preliminary computational results for our benchmark Stem-and-Cycle Ejection Chain algorithm implemented with the linked list, satellite list, 2-level tree, and 3-level satellite tree representations. An analysis of the computation times supports theoretical claims regarding the new data structures.

1 INRODUCTION

The traveling salesman problem (TSP) has been frequently used as a testbed for the study of search techniques developed for general circuit-based permutation problems. In addition, tests performed on TSP instances can provide a basis for analyzing the performance characteristics of data structures useful for representing paths and cycles.

Commonly, the Traveling Salesman Problem (TSP) involves sequentially visiting a set of clients (cities) only once and finally returning to the initial client. The optimal solution is the tour of minimum total distance (or other cost measure associated with client pairs).

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 $(c_{ik}<c_{ij}+c_{jk})$.

The TSP is well known as a NP-hard combinatorial problem; hence, there is no exact algorithm that guarantees an optimal solution for any instance in polynomial time. Consequently, it becomes necessary to use heuristic (or approximate) algorithms to provide solutions that are as good as possible but not necessarily optimal for large problems. The importance of obtaining efficient heuristics to solve large-scale TSPs recently motivated Johnson et al. (2000) to organize the "8th DIMACS Implementation Challenge" specific for TSP algorithms. This paper is based on the development of several data structures proposed to improve the efficiency of a benchmark algorithm from the DIMACS Challenge, the Stem-and-Cycle algorithm described in Rego (1998).

A crucial element of a good search algorithm for the traveling salesman problem (TSP) is the choice of data structure for representing the tour. 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 choice of an appropriate data structure to represent the tour. The problem that must be considered is the computerized TSP, which includes the additional consideration that the tour and the procedure to modify it be represented efficiently in memory.

The most notable comparative study of known structures prior to 2003 is that of Fredman et al. (1995). This study describes the array (linked-list), splay tree, 2-level tree, and segment tree representations and presents computational times for a Lin-Kernighan algorithm (Lin & Kernighan 1973) with each structure using various test problems and machines. The authors note that although the splay tree holds the lowest theoretical complexity, $O(\log n)$, the 2-level tree is shown to be generally superior to the others for instances as large as 10^6, and, consequently, has been the data structure of choice for the most modern algorithms. This paper is intended to offer a preliminary study on the performance of various data structures that include those recently proposed in Osterman & Rego (2003).

This paper is organized to familiarize the reader with TSP data structures, including the linked list, satellite, 2-level tree, and k-level satellite tree representations. Section 3 briefly describes our benchmark algorithm, section 4 offers some preliminary results and comparisons, and section 5 summarizes our conclusions.

2 DATA STRUCTURES FOR THE TSP

Investigation of data structures for use with TSP algorithms has been a primary concern in the implementation of large scale TSP algorithms. Initially used was the doubly-linked list, a generic structure also known as the array representation. With a linked list structure, 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. This is the structure by which more recent structures are benchmarked.

The satellite list, proposed by Osterman & Rego (2003), is designed to be symmetric in order to better capture the true nature of a symmetric TSP tour. A result of the satellite design is that the subpath reversal operation no longer requires a traversal of the subpath. Another result is that tour orientation is not a fixed property of the list, and depends on the direction of the traversal.

The 2-level tree, proposed by Chrobak et al. (1990), divides the tour into approximately $n^{1/2}$ 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. (1995), Gamboa et al. (2002), and numerous participants of the DIMACS TSP Challenge (Johnson et al. 2000). The theory behind the 2-level tree contributes much to the latest developments on TSP data structures.

The *k*-level satellite tree, also proposed by Osterman & Rego (2003), expands upon the 2-level tree by allowing intermediate levels to interconnect the parent nodes and client nodes. This structure also incorporates a satellite design. A satellite design for the *k*-level tree is important, not only because of subpath reversal, but also because *Next* and *Previous* queries do not need to access the parent nodes with a satellite design. Theory indicates that an optimally designed *k*-level satellite tree is the most efficient structure proposed to date.

2.1 *Subpath Reversal*

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 1965). A problem that arises when coding a subpath reversal with the linked list structure has motivated the design of multiple structures that can perform the operation more efficiently. The shortcoming of the linked list data structure is that every node in the subpath must be visited when performing a subpath reversal in order to maintain a feasible tour representation. The computational cost of the operation grows prohibitively with problem size ($O(n)$) and contributes significantly to the time complexity of the search, since the operation occurs frequently in classic *k*-opt procedures. The subpath reversal operation is also integral to today's most advanced algorithms: the Lin-Kernighan algorithm, and the Stem-and-Cycle Ejection Chain algorithm.

2.2 *Subpath Reversal with a Satellite List*

The satellite list, proposed by Osterman & Rego (2003), represents a tour without implying a fixed orientation of the path, making it useful for representing symmetric paths or cycles. It can operate in the same capacity as the doubly-linked list and is equally efficient in terms of both memory and computation of the *Next* and *Previous* queries. Because the satellite list avoids a fixed orientation, the subpath reversal operation can be performed in constant time, whereas for the linked list, every pointer associated with nodes in the reversed subpath in a linked list must be changed to reflect the appropriate orientation.

The satellite list achieves a symmetric representation by avoiding direct links between city representations. Each city is orbited by two satellite nodes, each of which is linked with the satellites orbiting the adjacent nodes, thereby linking cities indirectly. In addition to the links to adjacent satellites, there also exists some means to access the complement satellite and the city about which it orbits. The satellites are interchangeable and thus do not necessitate an orientation. Because of the satellite list's symmetric design, the subpath reversal operation is a natural one and is performed easily in constant time.

2.3 *Subpath Reversal with a 2-Level Tree*

The 2-level tree data structure, initially proposed by Chrobak et al. (1990), is a proven structure useful for representing TSP tours. 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. (1995), Johnson & McGeoch (1997), Neto (1999), Applegate & Cook (2000), and Helsgaun (2000). Gamboa et al. (2002) report an impressive set of comparative results obtained from implementing the Stem-and-Cycle Ejection Chain method with this structure.

Each parent node in a 2-level tree structure contains important information about the associated segment such as the total number of clients, pointers to the segment's endpoints, a sequence number, and a "Reverse" bit. The "Reverse" bit makes it possible to reverse the orientation of an entire segment. This is the main feature that reduces the computational time needed by TSP algorithms. If "Reverse" is switched on $(=1)$, the meaning of the "Previous" and "Next" pointers in the segment elements is reversed, and the segment is meant to be traversed in reverse order. "Next" clients are found by following "Previous" pointers of nodes in the reversed segment. However, the meaning of the "Previous" and "Next" pointers in the parent nodes is not reversed. In order to reverse a subpath that embodies only whole segments, it is necessary only to reconstruct the pointers among the nodes at the fractures and alter each parent node in the subpath by flipping the reverse bit and swapping the "Previous" and "Next" parent pointers, while the pointers among the intermediate nodes in the segment remain unchanged. If the subpath to be reversed does not embody whole segments, then cut/merge operations are used to regroup the segments such that they do.

Since the 2-level tree maintains for each level a doubly-linked list, it is a natural idea to use the satellite list to improve the 2-level tree. The result is a 2-level satellite tree, and its generalization to k levels yields the best known structure to date.

2.4 *Subpath Reversal with a k-Level Tree*

The k-level satellite tree, proposed by Osterman & Rego (2003), makes use of both "satellite" and "2-level tree" design concepts to achieve superior performance. Each of the substructures, including the top and bottom levels and the individual segments of the intermediate levels, is structured as a satellite list rather than a linked list. The design of the structure expands the 2-level tree design to allow k levels instead of two. When k is chosen optimally, a path between two client nodes in the tree can be traversed with a complexity $O(\log n)$ rather than $O(n^{1/2})$. See the reference for a detailed description of the k-level tree and its properties.

The mechanics of performing a subpath reversal in a k-level tree are similar to those of the performance with the 2-level tree. If the entire subpath embodies a parent segment, then the links between the clients must be updated at the fractures, but there is no need to flip a "Reverse" bit due to the satellite design. If the subpath embodies multiple parent segments, then the link updates at the client level are the same, but links at the fractures must also be updated at the parent level. In this case, the satellite design nullifies the need to traverse the parent nodes. If either of the subpath endpoints lie in the middle of a parent segment, then the tree is rearranged with cut and merge operations before the subpath is reversed, as is true of the 2-level tree as well. Cut and merge operations for the 2-level tree are generalized for the k-level tree by iteratively regrouping segment elements under intermediate parents at each level of the tree.

In addition to its attractive properties for subpath reversal, a major benefit of implementing the k-level tree with a satellite design is that *Next* and *Previous* queries do not require access to the parent nodes, since the current orientation depends on the satellites rather than the parent nodes. The implications of this benefit are tremendous, considering the frequency of the need for *Next* and *Previous* and the fact that the cost of accessing a parent node varies with the problem size when the data structure is designed optimally (Osterman & Rego 2003).

3 EXPERIMENTAL ANALYSIS

We have chosen the Stem-and-Cycle Ejection Chain algorithm (S&C), described in Rego (1998), as the benchmark algorithm for comparing different data representations of the TSP tour and specific Stem-and-Cycle reference structure. Gamboa et al. (2003) show that this algorithm achieves results competitive with the best algorithms available to date.

The Stem-and-Cycle (S&C) reference structure is described in Glover (1992) and implemented with the 2-level tree representation in the subpath ejection algorithm described in Gamboa et al. (2002) for the TSP. This reference supplies specific details on implementing the subpath reversal operation with the 2-level tree.

3.1 *Subpath Reversal Considerations for the Stem-and-Cycle Algorithm*

In graph theory, the S&C structure is defined by a spanning subgraph of G, consisting of a path $ST=(v_t,\dots,v_r)$ called the stem, attached to a cycle $CY=(v_r,v_{s_1},\dots,v_{s_2},v_r)$. Vertex v_r in common to the stem and the cycle is called the root and, consequently, its adjacent vertices v_{s_1} and v_{s_2} are called subroots. Finally vertex v_t is called the *tip* of the stem.

The Stem-and-Cycle Ejection Chain method starts by creating the S&C structure from an initial tour. This is done by linking two nodes of the tour and removing one of the edges adjacent to one of those nodes. The selection of the edge to be removed defines the root node, common to the stem and the cycle. In each step of the ejection chain process, a subpath is ejected in the form of a stem. The possible transformations for the S&C structure at each level of the chain are defined by two distinct ejection moves, which transform a reference structure into another of the same type. Since the resulting structure obtained at each level of the chain usually does not represent a feasible tour, a trial move is required to restore the solution feasibility. Considering an orientation for both the stem and the cycle, the application of any of these moves may require reversing a subpath of the reference structure.

The ejection moves and the trial move required to restore a feasible tour are described as follows:

- *Cycle-Ejection move:* Insert an edge (v_t, v_p), where v_p belongs to the cycle. Choose an edge of the cycle (v_p, v_q) to be removed, where v_q is one of the two adjacent vertices of v_p. Vertex v_q becomes the new tip. Depending on the orientations of the current structure, the path between v_t and v_r may be reversed.
- *Stem-Ejection move*: Insert an edge (v_t, v_p), where v_p belongs to the stem. Identify and remove the edge (v_p, v_q) so that v_q is a vertex on the subpath $(v_t, \ldots, v_p)$. Vertex v_q becomes the new tip. The path between v_t and v_q is reversed.
- *Trial move:* Insert an edge (v_t, v_s) and remove edge (v_r, v_s), where v_s is the chosen as the most desirable of the two subroots. Depending on the orientations in the current structure, the path between v_t and v_r may not need to be reversed.

The application of the cycle-ejection move causes the stem to become a part of the new cycle. Whether a subpath reversal is required in this case depends on the choice of v_q. If v_q is chosen from the side of the candidate whose orientation does not match the stem, either the old stem or subpath $(v_q, \ldots, v_r)$ must be reversed. The application of a stem-ejection move requires either the path $(v_q, \ldots, v_t)$ or the path $(v_p, \ldots, v_r)$ to be reversed. Likewise, for the trial move, it is imperative to reverse the orientation of either the stem or the cycle if the choice of the subroot for the trial solution causes the orientation of the stem and cycle to be mismatched.

3.2 *Data Structure Considerations for the Stem-and-Cycle Algorithm*

Although the satellite list reduces the time complexity of the subpath reversal operation to a constant, the S&C algorithm requires that nodes be labeled as being in the cycle or the stem. Updating these labels when a cycle-ejection move is performed requires a traversal of the relevant nodes. For this reason, the cost of operating the satellite list in the S&C context is still bounded by the cost of the traversal operation, $O(n)$.

The *k*-level tree defers the cost of traversals to multiple levels, reducing the complexity of traversals to $O(n^{1/k})$, since the cost is proportionate to the number of parent nodes. If the value of *k* is fixed for all *n,* then the cost of managing additional levels remains constant as *n* increases. However, the higher *k* is chosen, the larger the constant term associated with the operating cost, regardless on *n*. Osterman & Rego 2003 show that when *k* is chosen optimally, the time complexity of operating the structure is $O(\log n)$.

To summarize, we expect the computational cost of operating these structures in the context of the S&C Ejection Chain algorithm to be limited by the following functions of problem size:

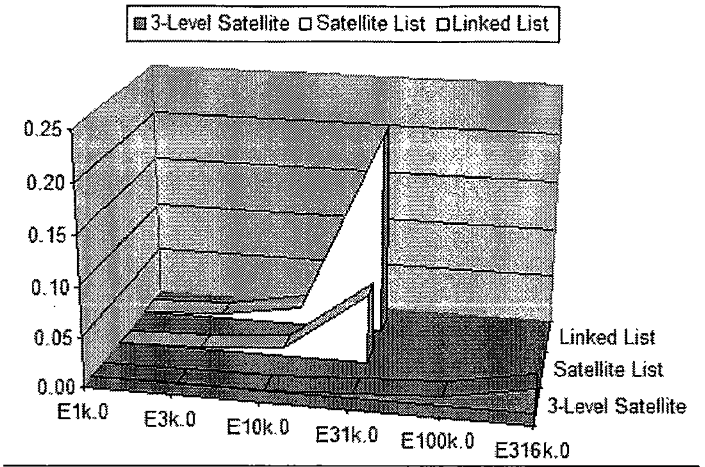
- linked list—$O(n)$, due to traversal operations,
- satellite list—$O(n)$, due to traversal operations,
- 2-level tree—$O(n^{1/2})$, due to traversal and cut/merge operations
- *k*-level satellite tree (*k* fixed)—$O(n^{1/k})$, due to traversal and cut/merge operations
- *k*-level satellite tree (*k* chosen optimally)—$O(\log n)$, due to traversal operations due to parent access and cut/merge operations

The satellite design is expected to reduce costs by a constant factor for two reasons. In the case of the satellite list, the need to traverse a reversed subpath is negated. In the case of the *k*-level tree, there is no need to access the parent node to perform *Next* or *Previous* operations. The cost of these operations is constant; thus, the coefficient of $\log n$ in the total operating cost will be much lower than for the splay tree, which performs *Next* and *Previous* in $O(\log n)$.

4 COMPUTATIONAL RESULTS

We have implemented the Stem-and-Cycle Ejection Chain algorithm (S&C) described in Rego (1998) with four data structures: the doubly-linked list, the satellite list, the 2-level tree, and the *k*-level satellite tree with *k* fixed to 3. Research is ongoing, and an implementation with the *k*-level tree in which *k* is chosen optimally is presently in development. We present computational times for running each of these codes on a Dell Powerstation (1.4 GHz processor with 1GB RAM). The testbed of TSP problem instances is taken from the set of uniformly random Euclidean instances used in the DIMACS TSP Challenge (Johnson et al. 2000).

Normalized CPU Running Times (seconds/N)



		Code		
Problem	Size (N)	Linked List	Satellite	3-Level Satellite
E1k.0	1,000	0.0015	0.0009	0.0007
E3k.0	3,162	0.0052	0.0025	0.0009
E10k.0	10,000	0.0193	0.0088	0.0013
E31k.0	31,623	0.2057	0.0706	0.0054
E100k.0	100,000	1.1362	0.6691	0.0088
E316k.0	316,228	-	-	0.0244

Figure 1. Data Structure Comparison using Identical Algorithm and Machine.

Figure 1 compares normalized running times for the linked list, satellite list, and 3-level satellite tree implementations of our benchmark algorithm. The times displayed in the chart for the linked list and satellite list codes only include instances as large as 31,623 clients. Times for the E100k.0 instance is omitted to keep the chart visually informative. The E316k.0 instance took too long to compute for these codes.

The values confirm the theoretical implications of the data structures. Due to required traversal operations, times for the linked list and satellite list codes vary similarly with problem size, although a significant constant factor (one half) is achieved by the satellite code. The 3-level satellite times vary with $n^{1/3}$ in a similar fashion to the way the linked list times vary with n, admittedly with a higher constant factor associated with managing the more complex structure[†].

Data for the 2-level tree appear to be missing from the chart and table above. The 2-level tree code currently gives a different iteration results from the other codes, so it would not be accurate to include it in the comparison above. Instead, we offer two means of comparing this data structure to the 3-level satellite tree. A version of the 2-level tree code whose results match exactly with our other three codes is in development.

Table 1. Relative Measure Comparison

	Code		Difference	Relative Measure (3LST)	Relative Measure (2-Level Tree)
Problem	Linked List (LL)	3-Level Satellite Tree (3LST)	(LL – 3LST)	Difference / 3LST	Reported[‡]
E1k.0	0.0015	0.0007	0.0008	1.208	0.625
E3k.0	0.0052	0.0009	0.0043	4.674	2.417
E10k.0	0.0193	0.0013	0.0179	13.423	5.889
E31k.0	0.2057	0.0054	0.2002	36.790	8.955
E100k.0	1.1362	0.0088	1.1274	128.114	15.836

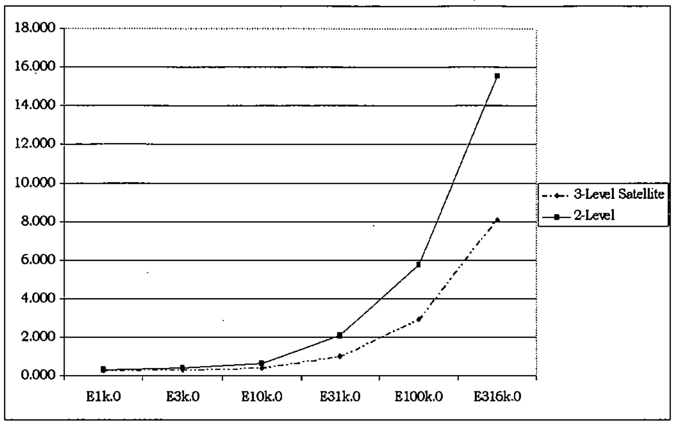
In producing the Table 1, the 2-level tree code and 3-level satellite code are each compared to linked list versions that produce identical solutions at each iteration. By comparing separate relative measures, this approach to comparison avoids the problem faced in the difference in solutions produced by the two codes. The relative measure in Table 1 can be interpreted as "how many times faster" the 2-level code executed than the linked list code. It appears that the implementation using the 3-level satellite tree holds a huge speed advantage over the 2-level tree implementation. The drawback in this comparison is that the values for the 2-level tree code were not computed on the same machine as the values for the 3-level satellite code. A "times faster" measure varies with the speed of the test machine and makes such a measure somewhat unreliable, although this fact has been often ignored in the literature.

A second approach involves dividing the running time by a legitimate basis for comparison. Figure 2 presents a comparison of the results given by the current versions of the 2-level tree and 3-level satellite tree implementations. "Milliseconds per ejection chain" represents a good measure for comparison of the two algorithms, since the length and computational difficulty of a typical ejection chain move is expected to be the same for both implementations. This is clear because the codes differ only in the choice of the trial solution. No statistically significant differences were detected in chain length or subpath size measurements taken from the sets of iteration results for the two codes.

[†] The fact that the times do not vary exactly with their respective functions can be explained by the hardware limitations of the cache memory structure. The result is that on average, the time required to access a single array element increases as the size of the array increases. Therefore, we look for consistency in the variation using the linked list times as a basis.

[‡] Results in this column reported in Gamboa, Rego, and Glover [4].

Normalized Average Milliseconds Per Ejection Chain



	Number of Chains				Solution Value Reached	
Problem	Chains	M/C	Chains	M/C		
E1k.0	2,315	0.29	2,408	0.34	23,682,343	23,955,085
E3k.0	9,236	0.31	6,960	0.41	41,455,191	42,018,455
E10k.0	32,618	0.41	28,017	0.65	73,326,803	74,096,043
E31k.0	169,618	1.02	117,510	2.09	129,424,475	131,344,421
E100k.0	300,516	2.91	283,272	5.77	230,333,945	233,355,871
E316k.0	952,164	8.09	471,075	15.52	410,139,302	417,024,423

Figure 2. Results Given by 2-Level and 3-Level Satellite Implementations.

For the problems tested (1000 nodes and larger), the computational time required by the 3-level satellite code is always less than that required by the 2-level code, and the gap grows larger as a percentage with problem size due to the higher constant. The values given by "millisecs/chain" confirm that the cost of performing the ejection chain for the 2-level and 3-level satellite codes are similar in the way they vary with $n^{1/2}$ and $n^{1/3}$, respectively. The reason for the discrepancy in the solution value reached between the two codes is the result of a bug that was found and fixed in the version with the new data structure (3-level satellite tree). Research is ongoing, and we present these results as preliminary confirmatory evidence of efficiency gained with the new data structure.

5 CONCLUSION

These preliminary experiments provide convincing evidence of the efficiency to be gained from utilizing a satellite design and from grouping nodes into segments in multiple levels. The computation times obtained using the more advanced data structures are lower in a manner that is consistent with theory. Despite the fact that the moves actually performed and the solution reached by the two tree-based codes differ, these preliminary results are persuasive. Assuming some problem size, any material difference between the average computation time needed to perform one ejection chain is expected to be the result of the different data structures. The characteristics for each structure underlying the efficiency gains indicate that similar results could be achieved for *k*-opt and Lin-Kernighan algorithms using these data structures. The consistency with which these experiments match expectations indicates that even better results can be expected when the *k*-level satellite tree is implemented to choose *k* optimally in consideration of the problem size.

6 REFERENCES

Applegate, D. & Cook, W. 2000. Chained Lin-Kernighan for Large Traveling Salesman Problems. *Technical report,* Rice University.

Chrobak, M., Szymacha, T. & Krawczyk, A. 1990. A Data Structure Useful for Finding Hamiltonian Cycles. *Theoretical Computer Science* 71: 419-424.

Fredman, M., Johnson, D., McGeoch, L. & Ostheimer, G. 1995. Data Structures for Traveling Salesmen. *Journal of Algorithms* 18: 423-479.

Gamboa, D., Rego, C. & Glover, F. 2002. 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.

Gamboa, D., Rego, C. & Glover, F. 2003. Implementation Analysis of Efficient Heuristic Algorithms for the Traveling Salesman Problem. *Hearin Center for Enterprise Science Research Report* HCES-05-03, University of Mississippi.

Glover, F. 1992. New Ejection Chain and Alternating Path Methods for Traveling Salesman Problems. *Computer Science and Operations Research,* 449-509.

Helsgaun, K. 2000. An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic. *European Journal of Operational Research* 1: 106–130.

Johnson, D. & McGeoch, L. 1997. Local Search in Combinatorial Optimization. *The Traveling Salesman Problem: A Case Study in Local Optimization.* John Wiley and Sons, Ltd., 215–310.

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

Lin, S. 1965. Computer Solutions of the Traveling Salesman Problem. *Bell System Computer Journal* 44: 2245-2269.

Lin, S. & Kernighan, B. 1973. An Effective Heuristic Algorithm for the Traveling Salesman Problem. *Operations Research* 21: 498-516.

Neto, D. 1999. Efficient Cluster Compensation for Lin-Kernighan Heuristics. Department of Computer Science, University of Toronto.

Osterman, C. & Rego, C. 2003. The Satellite List and New Data Structures for Symmetric Traveling Salesman Problems, *Hearin Center for Enterprise Science Research Report* HCES-06-03, University of Mississippi.

Rego, C. 1998. Relaxed Tours and Path Ejections for the Traveling Salesman Problem. *European Journal of Operational Research* 106: 522-538.