Implementation Analysis of Efficient Heuristic Algorithms for the Traveling Salesman Problem[†]

Dorabela Gamboa[a], César Rego[b*], Fred Glover[c]

[a] Escola Superior de Tecnologia e Gestão de Felgueiras, Instituto Politecnico do Porto, Rua do Curral, Casa do Curral, Apt. 205, 4610-156, Felgueiras, Portugal. dgamboa@estgf.ipp.pt

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

[c] Leads School of Business, University of Colorado, Boulder, CO 80309-0419, USA. fred.glover@colorado.edu

Latest Revision: March 2003

Abstract – The state-of-the-art of local search heuristics for the traveling salesman problem (TSP) is chiefly represented by algorithms based on the classical Lin-Kernighan procedure and the Stem-and-Cycle (S&C) ejection chain method. Critical aspects of implementing these algorithms efficiently and effectively derive from considering appropriate candidate lists for the potentially available moves and from taking advantage of special data structures especially for large-scale problems. This paper describes how these elements may affect the performance of the S&C ejection chain method, which also gives insights about how to improve the performance of general local search algorithms for circuit-based permutation problems. Computational findings are reported for TSP instances containing up to 1,000,000 nodes.

Keywords: traveling salesman, local search, data structures, ejection chains

† This research has been supported in part by the Office of Naval Research (ONR) grant N000140110917
* Corresponding author.

1. Introduction

The traveling salesman problem (TSP) has been frequently used as a testbed for the study of new local search techniques developed for general circuit-based permutation problems. An important characteristic of these problems is that tests performed on challenging TSP instances provide a basis for analyzing the performance characteristics of global search metaheuristic techniques.

We focus on the undirected (symmetric) TSP problem in this paper. However, our observations and results also can be extended to the directed (asymmetric) case since the underlying stem-and-cycle reference structure can readily be implemented for directed TSPs. The undirected problem can be stated in graph theory terms as follows. Let $G=(V,A)$ be a graph, where $V=\{v_1,...,v_n\}$ is a vertex (or node) set and $A=\{(v_i,v_j) \mid v_i,v_j \in V, i \neq j\}$ is an edge set, with a non-negative cost (or distance) matrix $C = (c_{ij})$ associated with A. The TSP consists in determining the minimum cost Hamiltonian cycle on the problem graph. This paper considers the symmetric version of the problem ($c_{ij}=c_{ji}$), which satisfies the triangular inequality ($c_{ij} + c_{jk} > c_{ik}$).

To document the current state-of-the-art, and to provide a better understanding of the heuristic components that prove most effective for several classes of heuristic algorithms applied to the TSP, the 8th DIMACS Implementation Challenge on the Traveling Salesman Problem was recently organized by Johnson, McGeogh, Glover and Rego [11]. In this paper we describe findings from this Implementation Challenge as well as our own experience with different algorithm implementations. The outcomes give important insights about how to improve the performance of algorithms for several other circuit-based permutation problems.

Some of the most efficient local search algorithms for the TSP are based on variable depth neighborhood search methods such as the classic Lin-Kernighan procedure [12] and the Stem-and-Cycle ejection chain method [6]. Critical aspects of implementing these algorithms efficiently and effectively derive from considering appropriate candidates for available moves and from taking advantage of specialized data structures, especially for large TSP problems. We describe how these elements affect the performance of a Stem-and-Cycle ejection chain method, and additionally show the impact of several other algorithmic components, including the use of different types of starting solutions and incorporating complementary neighborhood structures. Computational findings are reported for TSP instances containing up to 1,000,000 nodes.

2. Algorithm Description

We consider a local search algorithm based on the stem-and-cycle ejection chain method proposed in Glover [6]. The current implementation of the stem-and-cycle algorithm is a slight variant of the P-SEC algorithm described in Rego [14].
The algorithm can be briefly described as follows. Starting from an initial tour, the algorithm attempts to improve the current solution iteratively by means of a subpath ejection chain method, which generates moves coordinated by a reference structure called a Stem-and-Cycle (S&C). The stem-and-cycle procedure is a specialized variable depth neighborhood approach that generates dynamic alternating paths (as opposed to static alternating paths produced, for example, by the classical Lin-Kernighan approach). The generation of moves throughout the ejection chain process is based on a set of rules and legitimacy restrictions

determining the set of edges allowed to be used in subsequent steps of constructing an ejection chain. Implementation improvements in the basic algorithm strategy (described in Rego [14]) make the current version more efficient and more effective for solving very large scale problems.

Maintaining the fundamental rules of our algorithm unchanged, we have introduced the following modifications as a basis for investigating the effects of alternative candidate list strategies and datat structures. The first introduces the two-level tree data structure described in Fredman et al. [4], and used in some of the most efficient Lin-Kernighan implementations reported in the DIMACS Challenge (e.g. those of Johnson and McGeoch [10]; Appelgate and Cook [1]; and Helsgaun [9]). In our modified algorithm we have adapted the two-level tree data structure to replace the less effective doubly-linked lists previously used to represent both the TSP tours and the S&C reference structures. Another modification is to replace a two-dimensional array-based data structure with an n-vector to control ”legitimacy restrictions” during the ejection chain construction, substantially reducing the storage space required by the earlier version. We have also incorporated in the modified algorithm the ability to directly access external neighbor lists in the format used by the Concorde system, thus providing additional alternatives to the nearest neighbor list used in the P-SEC algorithm.

Our algorithm is implemented as a local search improvement method in the sense that no meta-strategy is used to guide the method beyond local optimality. Also, the method always stops after N iterations if its re-routing strategy fails to improve the best solution found so far. (Re-routing consists of starting an S&C ejection chain from a different route node.) Thus our implementation of the S&C algorithm is simpler than Lin-Kernighan implementations that make use of additional supplementary techniques such as the ”don’t look bits” strategy, caching distances, and other implementation tricks.

2.1. The Stem-and-Cycle Reference Structure

The reference structure used in the subpath ejection chain (SEC) algorithm described in Rego [14] and now enhanced is called Stem-and-Cycle (S&C). The S&C structure has its theoretical foundation in Glover [6] and 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 $CY = (v_r, v_{s_1}, \dots , v_{s_2}, v_r)$. An illustrative diagram of a Stem-and-Cycle structure is shown in Figure 1. The vertex v_r in common to the stem and the cycle is called the root, and consequently the two vertices of the cycle adjacent to v_r (v_{s_1} and v_{s_2}) are called subroots. Vertex v_t is called the *tip* of the stem.

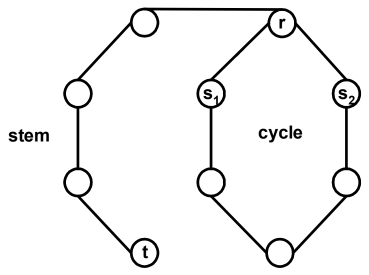

Figure 1 – The Stem-and-Cycle reference structure.

In each step of the ejection chain process, a subpath is ejected in the form of a stem. The method starts by creating the initial S&C reference structure from a TSP tour. Its creation is accomplished by linking two nodes of the tour and removing one of the edges adjacent to one of those nodes. The possible transformations for the S&C structure at each level of the chain are defined by two distinct ejection moves 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*.

Stem-ejection move: Insert an edge (v_t, v_p), where v_p belongs to the stem. Identify the edge (v_p, v_q) so that v_q is a vertex on the subpath $(v_t, \dots, v_p)$. Vertex v_q becomes the new *tip*.

Figure 2 illustrates an example of the application of each of these moves to the S&C structure of Figure 1. In the example, grey lines represent the edges to be inserted in the new structure, and the dotted lines point out possible edges to be removed from the current structure.

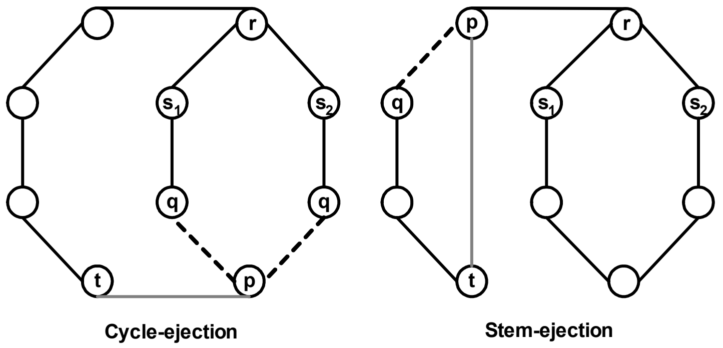

Figure 2 – Ejection moves.

The structure obtained through the application of an ejection move usually does not represent a feasible tour (unless $v_t = v_r$); thus, a trial move is required to generate a feasible TSP solution. Trial solutions are obtained by inserting an edge (v_t, v_s), where v_s is one of the subroots, and removing edge (v_r, v_s). Figure 3 shows the two possible trial solutions that can be obtained from the S&C structure in Figure 1.

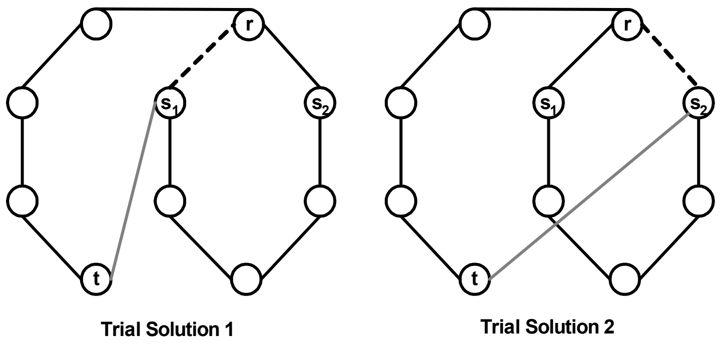

Figure 3 – Trial solutions.

2.2. Implementation Issues

One of the primary modifications in our algorithm was the replacement of the Array data structure (implemented as a doubly-linked list) by the Two-Level Tree data structure (2-level tree). This structure was initially proposed by Chrobak, Szymacha and Krawczyk [3] and has been used in efficient implementations of the 2-Opt and 3-Opt procedures as well as their generalization, the Lin-Kernighan procedure [12]. Fredman *et al.* [4] show the improvement in performance over the array data structure obtained in their implementation of the Lin-Kernighan algorithm by using the 2-level tree - they also report results for two other data structures.

The main advantage of the 2-level tree is its ability to reverse an entire subpath of the structure in $O(\sqrt{n})$ time, thus drastically reducing the computational times for large-scale problems. The path reversal issue arises because in computer implementation, an orientation is assumed for the structure in order to make it possible to read it. Hence, it is likely that an application of an ejection move would cause the need to reverse a subpath of the structure in order to preserve a feasible orientation.

In Lin-Kernighan implementations, the 2-level tree is used to represent a TSP tour; however, in our algorithm it also represents the S&C structure, and hence requires some modifications from the way the tree is implemented for procedures based on the Lin-Kernighan approach.

Our 2-level tree structure consists of two interconnected doubly-linked lists forming two levels of a tree. The first list defines a set of *Parent* nodes, each one associated with a segment of the S&C structure (or tour). Each segment represents an oriented path, and the correct association of all the paths represents a S&C structure (or tour). Figure 4 shows the *Parent* and the segment node structures and gives an example of the 2-level tree representation of the S&C structure with *CY=(5, 0, 7, 1, 4, 2, 5)* and *ST=(6, 8, 9, 3, 5)*.

Each member of a segment contains a pointer to the associated *Parent* node, *Next* and *Previous* pointers, the index of the client it represents (Client), and a sequence number (I.D.). The numbering within one segment is required to be consecutive (but it does not need to start at 1), since each I.D. indicates the relative position of that element within the segment.

A *Parent* node contains relevant information about the associated segment: the total number of clients (Size), pointers to the segment's endpoints, a sequence number (I.D.), a reverse bit (Reverse), and a presence bit (Presence). The reverse bit is used to indicate whether a segment should be read from left to right (if set to 0) or in the opposite direction (if set to 1). Likewise, the presence bit indicates whether a segment belongs to the stem (if set to 0) or to the cycle (if set to 1).

Segments are organized to place the root and the tip nodes as endpoints of a cycle and a stem segment, respectively. A null pointer is set to the tip node to indicate the end of the stem. The numbering of the *Parent* nodes always starts at the *tip*'s *Parent* that also fails to define one of its links.

Also, clients are organized in an array structure allowing for random access to any client node in the 2-level tree.

This 2-level tree structure is a special adaptation of the one described in Fredman et al. [4]. This adaptation involves a substantial modification of the operations described in [4] due to the significant differences between the Stem-and-Cycle and the Lin-Kernighan neighborhood structures. Since the S&C structure usually does not represent a Hamiltonian cycle and different rules for ejection moves apply to the stem and the cycle, a presence bit has been added to the *Parent* node structure to indicate whether one segment belongs to the stem or to the cycle. Another difference in our structure is the existence of null pointers in the *tip* node and associated *Parent* structures. We also introduce specialized 2-level tree update operations to ensure that each entire segment is either part of the stem or the cycle. The basic scheme is outlined in the following diagrams.

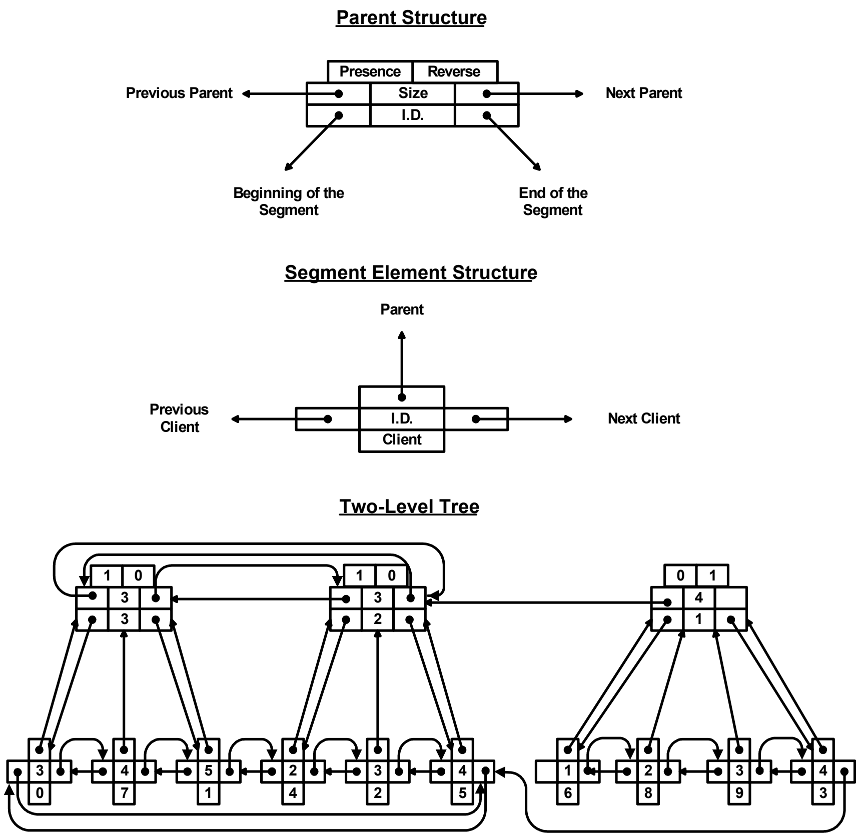

Figure 4 – The Two-Level Tree representation of the S&C structure

Five basic operations are needed to manage the 2-level tree data structure. Two operations deal with structure's orientation and are used to traverse the structure. Three other operations are designed to implement each type of move considered in the S&C ejection chain method. These operations can be defined as follows:

Path Traversal Operations

Next(a), returns *a*'s successor in the current structure. First, it finds node *a* in the segment list and follows the pointer to its *Parent* node. If the reverse bit is set to zero, the return value is obtained by following *a*'s *Next* pointer or following *a*'s *Previous* pointer, otherwise.

Previous(a), returns *a*'s predecessor in the current structure.

Move Operations

CycleEjection(r, t, p, q), updates the reference structure by removing edge *(p, q)* and inserting edge *(t, p)*. Depending on the orientations of the paths within the current structure, the path between *t* and *r* may have to be reversed.

StemEjection(r, t, p, q), updates the reference structure removing edge *(p, q)* and inserting edge *(t, p)*. The path between *t* and *q* is reversed.

Trial(r, t, s), updates the reference structure by removing edge *(s, r)* and inserting edge *(t, s)*. Depending on the orientations of the current structure, the path between *t* and *r* may have to be reversed.

In the move operations, any time the edge to be deleted is in the same segment, the operation involves splitting the segment between the edge's nodes and merging one of the resulting partitions with a neighbor segment. Besides these cut and merge procedures, other actions are needed to update the structure through the execution of any of the operations, such as updating pointers and renumbering sequence values for the segment nodes and *Parent* nodes involved as well as other *Parent* information such as segment size and presence bits. The values of the reverse bits change every time the associated segment is part of a path to be reversed. After performing the necessary cut and merges, path reversal is performed by flipping the reverse bits of the *Parent* nodes of all the segments in the path.

Our cut and merge operations are designed to maintain a 2-level tree structure with the following characteristics. Each segment is restricted such that all its nodes either belong to the cycle (cycle-segment) or to the stem (stem-segment). In addition, the root is set to be an endpoint of a cycle-segment, and the tip is set to be an endpoint of the rightmost segment of the stem. These specifications complicate the merge options and sometimes necessitate additional cuts and merges. Special cases also cause extra cuts and merges, such as the case that the cycle occupies a single segment. For a comprehensive description and detailed explanation of all these operations and special cases we refer the reader to Gamboa, Rego and Glover [5].

In order to clarify the effects of the execution of an operation, an example of a cycle-ejection move on the 2-level tree of Figure 4 is illustrated in Figure 5. In this move, edge (4, 2) is deleted and edge (4, 6) is added, which generates the S&C structure with *CY=(5, 0, 7, 1, 4, 6, 8, 9, 3, 5)* and *ST=(2, 5)*. The execution of *CycleEjection(5, 6, 4, 2)* involves the following operations. Number the new part of the cycle (6, 8, 9, 3, 5) by setting to 1 the presence bit of *Parent* 1 (original I.D.). Also, because nodes 4 and 2 are on the same segment, split it up between those nodes and merge node 4 to the initial segment 3. As the root (node 5) is now the new stem-segment, merge it into segment 3. Set up the links between nodes 4 and 6 and between the associated *Parent* nodes. Reverse the path 6 and 5 by flipping the reverse bit of *Parent* 1. Set up links between the root and its new subroot (node 3) and between the associated *Parent* nodes. Flip the presence bit of *Parent* 2 and number the new stem. Finally, reorder the I.D. numbers of the *Parent* nodes starting at the new *tip*'s (node 2) *Parent* node.

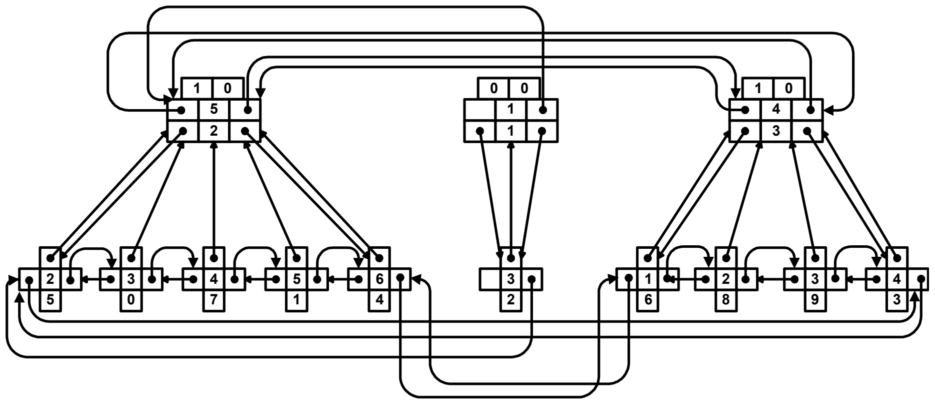

Figure 5 – Cycle-ejection move: resulting 2-level tree structure.

3. Experimental Results

This section studies the effects of a number of algorithm features that are usually critical for the performance of local search algorithms. The testbed consists of instances used in the "8th DIMACS Implementation Challenge" [10] from classes E (uniformly distributed clients) and C (clients organized in clusters) as well as a set of instances from the TSPLIB library [15] with different characteristics and sizes.

Runs were performed on a Sun Enterprise with two 400 MHz Ultra Sparc processors and 1 GB of memory.

3.1. Efficiency Analysis

To measure the relative efficiency of the 2-level tree implementation, several computational tests were carried out on three classes of problems. Table 1 reports the running times for two implementations of the same S&C algorithm that only differ in the data structures used to represent the S&C structure and the TSP tour. Besides the designation and size of each instance, the table shows the normalized (i.e. divided by n) computational times, the difference between the running times obtained by the two implementations, and the number of times the array version is slower than the 2-level tree version.

		Time/n		Difference	Times (1)
Problem	N	Array(1)	2L Tree(2)	(1)-(2)	slower than (2)
E1k.0	1,000	0.013	0.008	0.005	0.6
E3k.0	3,162	0.041	0.012	0.029	2.4
E10k.0	10,000	0.124	0.018	0.106	5.9
E31k.0	31,623	0.438	0.044	0.394	9.0
E100k.0	100,000	0.926	0.055	0.871	15.8
E316k.0	316,228	4.231	0.202	4.029	20.0
C1k.0	1,000	0.011	0.008	0.003	0.4
C3k.0	3,162	0.027	0.010	0.017	1.7
C10k.0	10,000	0.087	0.020	0.067	3.4
C31k.0	31,623	0.360	0.059	0.301	5.1
C100k.0	100,000	1.032	0.109	0.923	8.5
pla7397	7,397	0.091	0.015	0.076	5.1
rl11849	11,849	0.145	0.020	0.125	6.3
usa13509	13,509	0.129	0.019	0.110	5.8
d18512	18,512	0.267	0.025	0.242	9.7
pla33810	33,810	0.758	0.060	0.698	11.6
pla85900	85,900	0.701	0.048	0.653	13.6

Table 1 – Running times (seconds) for three classes of problems.

The results show that the efficiency of the 2-level tree implementation over the array implementation grows significantly with the problem size. In fact, if we consider the real times (not normalized), in order to solve the largest instance, the array implementation takes about 15 days to obtain a solution identical to the one provided by the 2-level tree implementation in 17 hours.

3.2. Analysis on the Effect of the Initial Solution

Typically, constructive algorithms are used to rapidly create a feasible solution for an optimization problem; however, the quality of the solutions provided by these algorithms is usually far below from the one that can be obtained by effective local search procedures. Because local search algorithms are based on the definition of a neighborhood structure designed to locally explore the solution space of the current solution at each iteration of the method, different solutions may be obtained depending on the initial solution from which the method starts.

This section analyzes the possible effect of the starting solution on the quality of the solutions provided by the S&C algorithm. Table 2 reports results for the algorithm starting from three constructive algorithms (Boruvka, Greedy and Nearest Neighbor) as well as randomly generated solutions (Random). For each type of initial solution, the results were obtained by running the algorithm using two different candidate lists: 20 quadrant neighbors (20QN) and 50 nearest neighbors (50NN). The Concord TSP Library was used to generate both the initial solutions and the candidate lists, except for the random solutions, where the systematic generator in Rego [14] was used with k=2 and k=5 to provide starting solutions for two runs (results for the "Random" columns are the average of the two runs). All runs were performed with a fixed set of algorithm parameters.

	Candidate Lists							
	20QN				50NN			
	% above the optimal solution or above the Held and Karp lower bound							
Problem	*Boruvka*	*Greedy*	*Nearest*	*Random*	*Boruvka*	*Greedy*	*Nearest*	*Random*
E100k.0	2.071	2.052	2.017	2.079	2.207	2.006	1.990	1.995
C100k.0	3.813	4.197	3.897	4.022	7.634	7.322	9.399	38.556
pla7397.tsp	0.964	1.055	0.700	0.732	1.032	1.026	0.923	0.913
rl11849.tsp	1.491	1.467	1.870	1.533	1.399	1.438	1.368	1.490
usa13509.tsp	1.153	1.390	1.233	1.215	1.210	1.153	1.221	1.248
d18512.tsp	1.471	1.160	1.321	1.249	1.234	1.205	1.278	1.068
pla33810.tsp	1.531	1.139	1.455	1.275	1.497	1.384	1.331	1.571
pla85900.tsp	1.287	1.727	1.672	1.617	1.252	1.086	1.115	1.147
Average	1.723	1.773	1.771	1.715	2.183	2.078	2.328	5.999
Number of times best	3	3	2	0	0	2	4	2

Table 2 – Results for different initial solutions.

The quality of the solutions obtained is measured by the percentage above the optimal solution (when known) or the Held and Karp lower bound [7, 8].

The last two lines in the table show the average solution quality for each column and the number of times the algorithm found the best solution starting from the initial solution provided by the associated constructive algorithm.

Graphs in Figure 6 depict the effect of the different starting solutions using the results in Table 2. The results for problem C100k.0 are not shown on the graphics because their high values (especially for the Random initial solution and 50NN candidate list) would not allow a clear reading of the graphics.

Table 2 shows that the combination 20QN/Random gives the best results (on average), but it did not find any best solution. The 50NN/Random combination reports the worst average; however, it finds two best solutions. These results indicate that no initial solution dominates all the others. In fact, the graphics in Figure 6 clearly show that the choice of any of these initial solutions does not greatly influence the quality of the final solutions. The fact that the algorithm fails for the C100k.0/50NN/Random indicates that the candidate list plays an important role in the effectiveness of a local search algorithm. Specifically, this result indicates that in cases where the starting solution does not contain a sufficient number of arcs in common with the optimal or high-quality solutions and the missing arcs are not in the candidate list either, it is very unlikely that the algorithm could find a good solution.

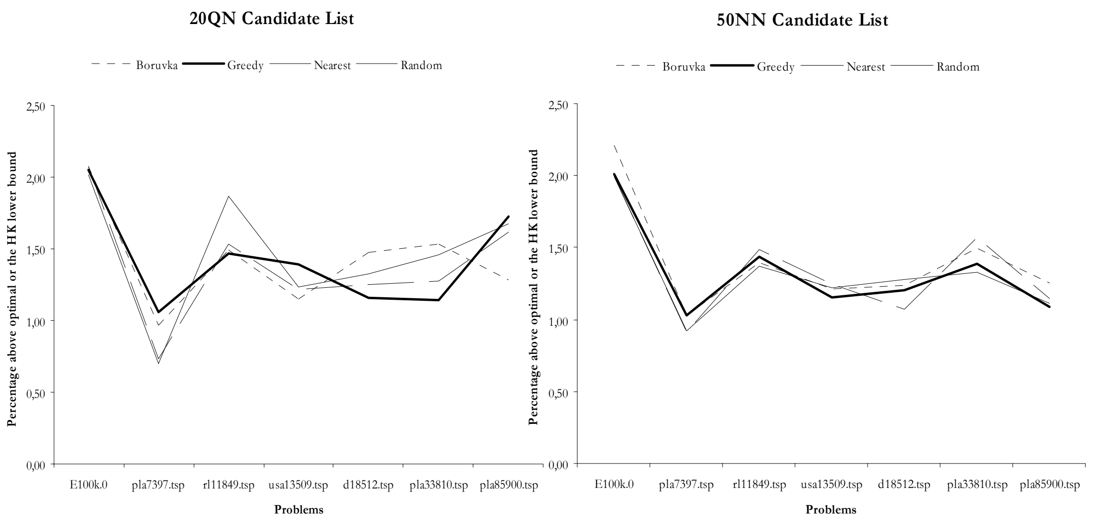

Figure 6 – Comparing the effect of the initial solutions.

Another indication of the relevance of an effective candidate list is that the algorithm achieves significantly better solutions (on average) using a 20-node candidate list (20QN) than the 50-node candidate list (50NN), regardless of the initial solution. A more extensive examination of the possible effect of the candidate design study is presented in the next section.

3.3. Analysis on the effect of the candidate list

The application of neighborhood search procedures to large-scale TSP problems requires the utilization of candidate lists in order to restrict the size of the neighborhood to search (at each iteration). The candidate list should contain all the components (or move attributes) necessary for an effective application of the neighborhood structure. Therefore, different candidate list designs may lead to different levels of performance for the same algorithm.

Table 3 reports comparative results for four different candidate lists: 12 quadrant neighbors (12QN), 50 nearest neighbors (50NN) and two other lists obtained by concatenating the first two with the list generated by the construction of Delaunay triangulations. We denote the latter by 12QN+D and 50NN+D, respectively.

Figure 7 provides illustrative graphics for the results presented in Table 3. In order to expose possible dependencies of the candidate lists on the structure of the initial solution, two different solutions (Greedy and Random) were used with each candidate list.

	Initial Solutions							
	Greedy				*Random*			
	% above the optimal solution or above the Held and Karp lower bound							
Problem	12QN	12QN+D	50NN	50NN+D	12QN	12QN+D	50NN	50NN+D
E100k.0	2.118	2.060	2.006	1.997	2.053	2.081	1.995	2.025
C100k.0	4.309	3.958	7.322	4.848	4.684	3.988	38.556	4.688
pla7397.tsp	0.786	1.050	1.026	1.000	1.077	1.761	0.913	1.120
rl11849.tsp	1.948	1.365	1.438	1.315	1.514	1.816	1.490	1.245
usa13509.tsp	1.246	1.299	1.153	1.286	1.355	1.376	1.248	1.197
d18512.tsp	1.242	1.195	1.205	1.128	1.211	1.212	1.068	1.217
pla33810.tsp	1.258	1.406	1.384	0.974	1.834	1.258	1.571	1.275
pla85900.tsp	2.262	1.608	1.086	1.074	2.368	1.483	1.147	1.222
Average	1.897	1.743	2.078	1.703	2.012	1.872	5.999	1.749
Number of times best	1	1	2	4	0	2	4	2

Table 3 – Results for different candidate lists.

Table 3 shows that the use of the 50NN+D candidate list with a Greedy initial solution provides better final solutions (on average) and usually finds the best solutions more often. The overall performance and relative advantage of this candidate list is illustrated in Figure 7. On the other hand, it appears that quadrant-based candidate lists result in better solutions (as expected) for clustered problems, as shown by the results obtained for the C100k.0 problem. (Even a 12-node quadrant list can do better than a 50-node nearest neighbor list enriched with delauny triangulations for this clustering problem). This conclusion is reinforced by the results in Table 2 where a 20-node quadrant list (20QN) significantly outperforms (on average) a 50-node nearest neighbor list (50NN), regardless of the initial solution.

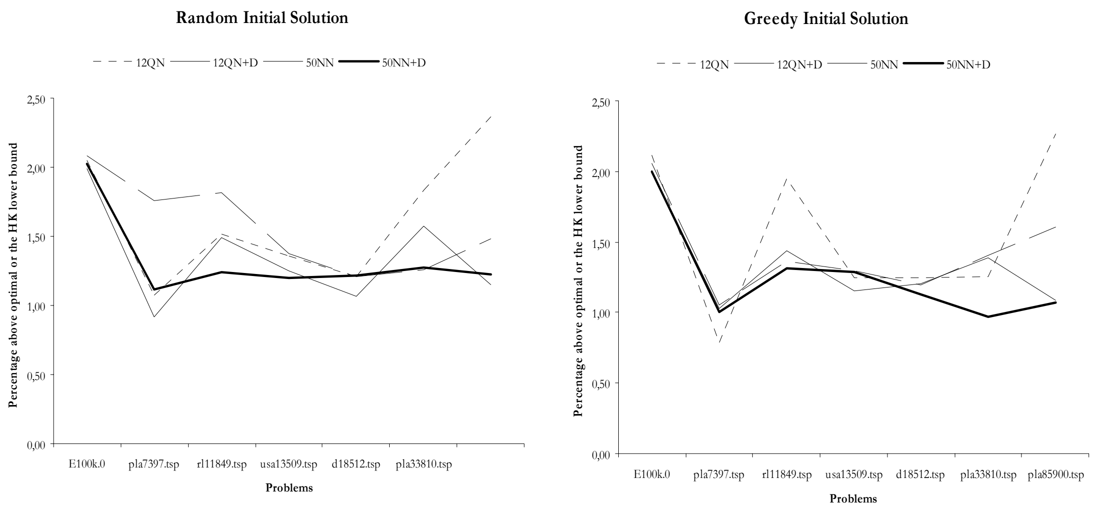

Figure 7 – Comparing candidate lists.

3.4. Comparative Analysis of Alternative Algorithms

We now analyze the performance of several highly effective heuristic algorithms using the results submitted to the "8th DIMACS Implementation Challenge" [10] for a comparative analysis. For the purpose of this study we restrict our attention to the information that is relevant to the analysis under consideration. For a complete list of results and details on generating the testbed instances, running times, scale factors for different computer systems, and other settings used in the challenge, we refer the reader to the Challenge web site [10].

The complete testbed consists of instances of class E (sizes between 1,000 and 10,000,000 nodes), C (sizes between 1,000 and 316,228 clients), and those from the TSP Library [15] with at least 1,000 nodes. However, for the current study we limited the number of problems to instances up to 1,000,000 (for classes E and C) and to problems larger than 3,000 nodes for the TSP instances.

In the attempt to render an accurate comparison of running times, a benchmark code was provided for Challenge participants to run in the same machines as the competing algorithms were run.

Figures 8, 9 and 10 provide comparative results on the performance of the following algorithms:

SC: The Stem-and-Cycle algorithm, 12QN+D candidate list, *Boruvka* initial solutions, two-level tree structure. All runs were performed with a fixed set of algorithm parameters. The algorithm considers ejection chains of 50 levels (component steps or "depth").

LK-JM: Implementation of the Lin-Kernighan algorithm by Johnson and McGeoch [10], *Greedy* initial solutions, 20QN candidate list, "don't look bits" strategy, two-level tree structure.

LK-N: Implementation of the Lin-Kernighan algorithm by Neto [13], 20QN+20NN candidate list, especial routines for "Clusters" compensation, "don't look bits" strategy, two-level tree structure.

LK-ABCC: Implementation of the Lin-Kernighan algorithm (in the Concord library) by Applegate, Bixby, Chvátal and Cook [2], 12QN candidate list, Q-Boruvka initial solutions, "don't look bits" strategy, two-level tree structure.

LK-ACR: Implementation of the Lin-Kernighan algorithm by Applegate, Cook and Rohe [1], 12QN candidate list, "don't look bits" strategy, two-level tree structure.

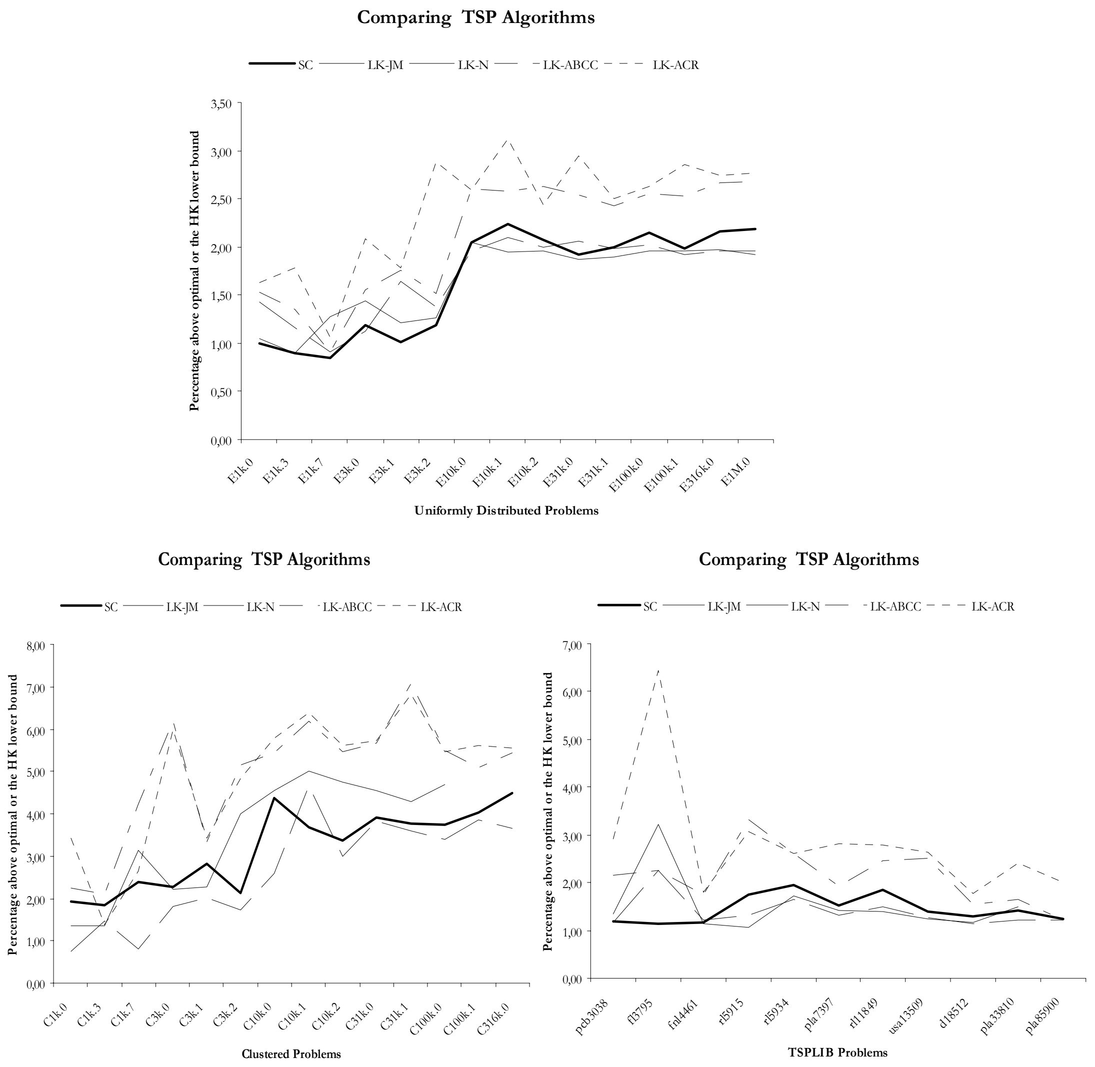

Figure 8 – Comparing TSP algorithms.

Graphics in Figure 8 show that the Stem-and-Cycle algorithm outperforms all implementations (for uniformly distributed and clustered problems) of the Lin-Kernighan procedure, except for the one presented by Johnson and McGeoch [10], which appears more effective for class C instances. However, as pointed out by the authors, it is important to note that the LK-JM implementation contains several tuning and implementation tricks (not described in the paper) that explain its relative performance. In contrast, the Stem-and-Cycle implementation has no additional components or tuning beyond the basic approach already indicated. These results suggest that the Stem-and-Cycle neighborhood structure provides additional advantages over the Lin-Kernighan structure.

Likewise, for the TSPLIB instances, the results presented in the third graphic of Figure 8 clearly show that the Stem-and-Cycle algorithm is very robust and outperforms several Lin-Kernighan implementations except LK-JM and LK-Neto. However, it is interesting to point out that if, in the TSPLIB graphic, we replace the SC results using Boruvka's initial solution by the ones shown in Table 3 for the Greedy/50NN+D combination, the Stem-and-Cycle algorithm outcomes are superior to the other results (cf. Figure 9).

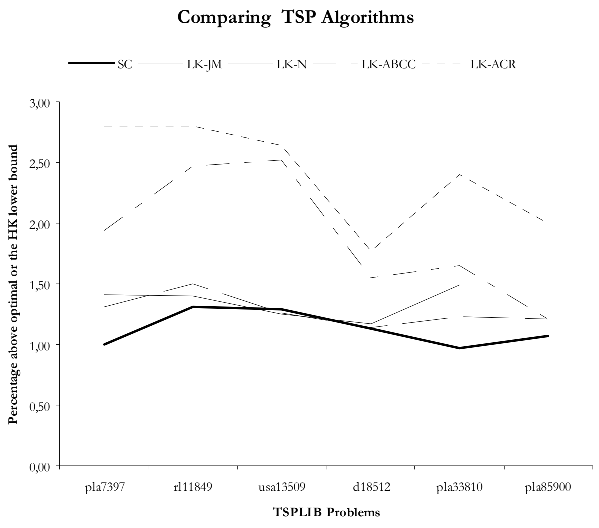

Figure 9 – The effect of using Greedy and 50NN+D.

Figure 10 depicts the running time variation as the problem size increases (for both groups E and C). We can see that the computation times for the SC-B algorithm suddenly deteriorate for sizes larger than 10,000 nodes, making the algorithm less efficient than the LK implementations for such problem sizes.

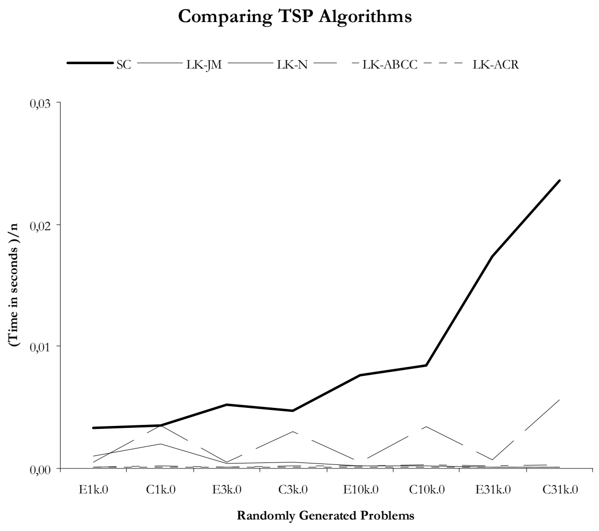

Figure 10 – Comparing running times.

Graphics in Figure 11 show the normalized running times of the Stem-and-Cycle algorithm. For the problems of groups E and C the running times significantly increase for problems over 100,000 nodes. As shown in Figure 11 in the graphic for the TSPLIB problems, problem size is not the only factor affecting the computational times. The structure of the problem also appears to be important. This result suggests that the increase in running times for very large scale instances is due to the fact that the current implementation of the stem-and-cycle algorithm does not take advantage of any specialized mechanism to reduce the size of the neighborhood operating on the initial candidate list. Under this assumption, the "don't look bits" strategy used in the LK implementations is particularly critical for the relative performance of these algorithms when solving large scale instances.

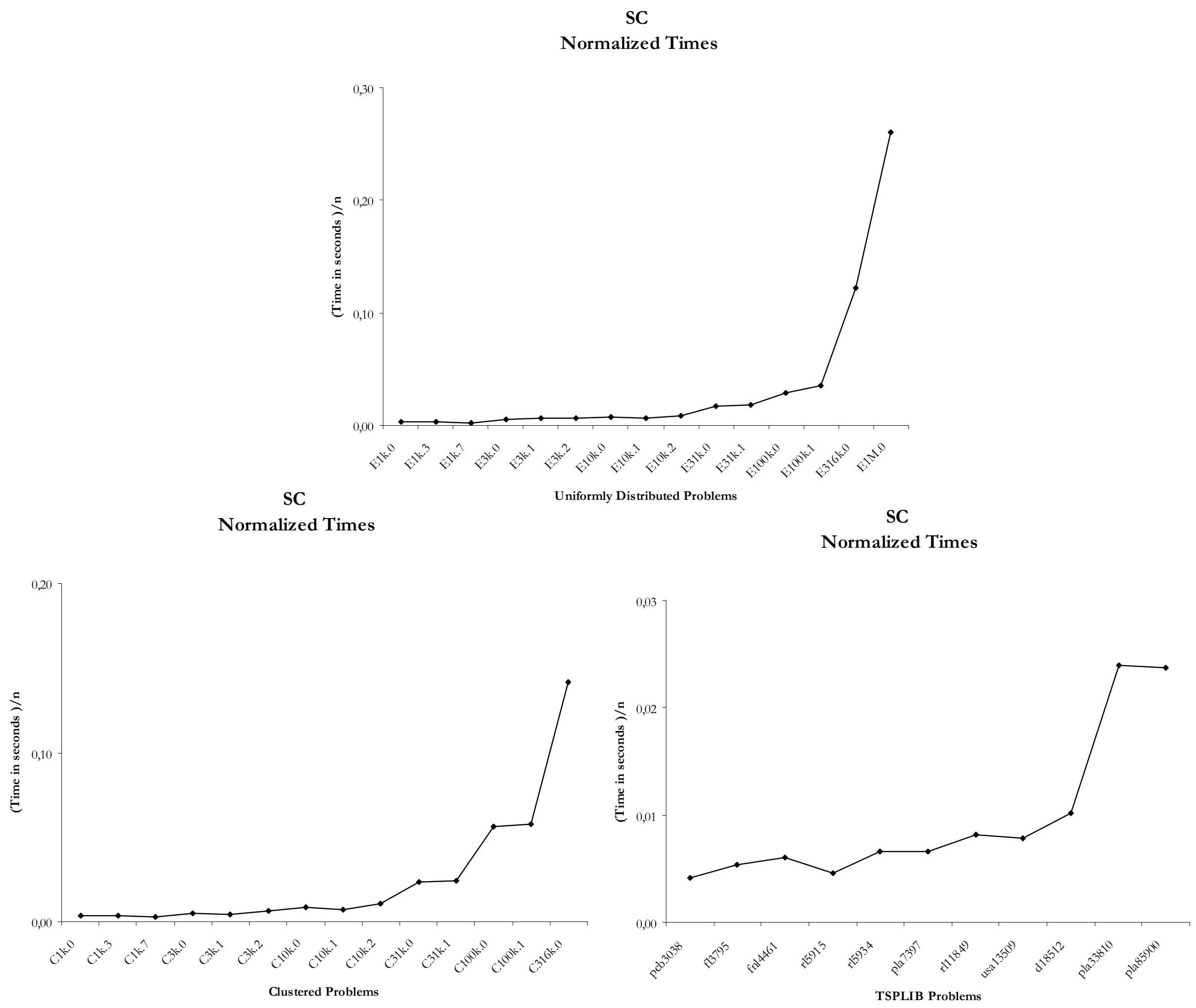

Figure 11 – Stem-and-Cycle running times.

Another important factor worth noting as a basis for further improvement is the use of more efficient candidate lists to restrict the neighborhood size while keeping the "right" edges to be considered for a move. In fact, it is clearly shown in Helsgaun [9] that the choice of an appropriate candidate list has great influence on both the efficiency and effectiveness of a LK implementation and thus a local search algorithm.

Another key feature that has proved crucial in the most efficient LK-based implementations (including the ones discussed in this study) is the use of supplementary neighborhoods called “double-bridges” that generate disconnected moves that cannot be achieved with the basic LK neighborhood. Again, we should point out that no alternative neighborhood is used in our stem-and-cycle algorithm. Although the possibilities for disconnected moves exist as an integral component of a generalization of the S&C ejection chain method (called doubly-rooted), this feature remains unexplored in our current implementations.

Finally, other possible improvements may consist in using adaptive memory programming (as proposed in tabu search contexts) to implement intensification and diversification strategies for effective exploration of the solution space.

4. Conclusions

The ability of the basic Stem-and-Cycle approach to obtain high quality outcomes without recourse to the usual array of supplementary strategies and auxiliary neighborhoods used to make other methods competitive suggests the opportunity for additional enhancement. A natural change in this direction is simply to utilize more effectively designed candidate lists. In addition, several types of choice and trial solution options in the stem-and-cycle approach remain unexplored, including those arising from more general doubly-routed reference structures.

Apart from the quality of the solutions that can be obtained, the computation time remains an important factor to consider, especially when very large instances have to be solved. Therefore, and because finding good solutions for large scale problems necessarily requires a significant number of iterations of a local search algorithm, there are two natural ways to reduce the time complexity for the next generation of TSP algorithms. One way is to develop even more effective data structures than the current widely-used 2-level tree to maintain and update a TSP tour (and possible reference structures). Another obvious way is to call upon parallel processing, which not only allows for reducing computation times but also provides an opportunity to design new neighborhood structures that may be effectively implemented in parallel. These possibilities for improvements afford interesting avenues for future investigation.

References

[1] D. Appelgate and W. Cook, Chained Lin-Kernighan for large traveling salesman problems, Technical report, Rice University, 2000, http://www.caam.rice.edu/˜keck/reports/chained lk.ps.

[2] V. Chvatal D. Applegate, R. Bixby and W. Cook, Concorde: A code for solving traveling salesman problems, Technical report, Rice University, 2001.

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

[4] M. L. Fredman, D. S. Johnson, L. A. McGeoch and G. Ostheimer, Data Structures for Traveling Salesman, J. Algorithms 18 (1995) 432-479.

[5] D. Gamboa, C. Rego and F. Glover, Data Structures and Ejection Chains for Solving Large Scale Traveling Salesman Problems, in preparation.

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

[7] Held, H. and Karp, R. M., The Traveling Salesman Problem and Minimum Spanning Trees, Operations Research, 18 (1970) 1138-1162.

[8] Held, H. and Karp, R. M., The Traveling Salesman Problem and Minimum Spanning Trees: Part II, Math. Programming, 1 (1971) 6-25.

[9] K. Helsgaun, An effective implementation of the Lin-Kernighan traveling salesman heuristic, European Journal of Operational Research 1 (2000) 106–130.

[10] D. S. Johnson and L. A. 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.

[11] D. S. 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/.

[12] S. Lin and B. Kernighan, An Effective Heuristic Algorithm for the Traveling Salesman Problem, Operations Research 21 (1973) 498-516.

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

[14] C. Rego, Relaxed Tours and Path Ejections for the Traveling Salesman Problem, European Journal of Operational Research 106 (1998) 522-538.

[15] G. Reinelt, TSPLIB - A Traveling Salesman Problem Library, European Journal of Operational Research 17 (1991) 125.