ELSEVIER

Available online at www.sciencedirect.com
SCIENCE DIRECT®

Computers & Operations Research 33 (2006) 1154–1172

computers & operations research

www.elsevier.com/locate/cor

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 Politécnico do Porto, Rua do Curral, Casa do Curral, Apt. 205, 4610-156, Felgueiras, Portugal
[b]Hearin Center for Enterprise Science, School of Business Administration, University of Mississippi, University, MS 38677, USA
[c]Leads School of Business, University of Colorado, Boulder, CO 80309-0419, USA

Available online 24 August 2005

Abstract

The state-of-the-art of local search heuristics for the traveling salesman problem (TSP) is chiefly based on algorithms using the classical Lin–Kernighan (LK) procedure and the stem-and-cycle (S&C) ejection chain method. Critical aspects of implementing these algorithms efficiently and effectively rely on taking advantage of special data structures and on maintaining appropriate candidate lists to store and update potentially available moves. We report the outcomes of an extensive series of tests on problems ranging from 1000 to 1,000,000 nodes, showing that by intelligently exploiting elements of data structures and candidate lists routinely included in state-of-the-art TSP solution software, the S&C algorithm clearly outperforms all implementations of the LK procedure. Moreover, these outcomes are achieved without the use of special tuning and implementation tricks that are incorporated into the leading versions of the LK procedure to enhance their computational efficiency.
© 2005 Elsevier Ltd. All rights reserved.

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.
E-mail addresses: dgamboa@estgf.ipp.pt (D. Gamboa), crego@bus.olemiss.edu (C. Rego), fred.glover@colorado.edu (F. Glover).

0305-0548/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cor.2005.06.014

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 meta-heuristic techniques.

We focus on the undirected (symmetric) TSP problem in this paper. However, our observations and results can also be extended to the directed (asymmetric) case since the underlying stem-and-cycle (S&C) 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 $V=\{v_1, \ldots, v_n\}$ 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, where the symmetry implied by the use of undirected edges rather than directed arcs can also be expressed by stipulating that $c_{ij}=c_{ji}$. We focus on the most widely studied form of the symmetric problem in which costs are assumed to satisfy the triangle 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 et al. [1]. 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 (LK) procedure [2] and the (S&C) ejection chain method [3]. 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 S&C ejection chain method, and additionally show the impact of several other algorithmic components, including different types of starting solutions and complementary neighborhood structures.

While many studies have been performed to determine how data structures, candidate lists and starting solutions affect the performance of alternative versions of the LK approach, no comparable study has been performed to determine the effect of these elements on the S&C ejection chain method. We show that by taking advantage of these components, but without relying on any of the additional special tuning and implementation tricks used in the leading versions of the LK procedures, the S&C approach proves superior to all implementations of the LK method. Our findings are documented by extensive computational tests conducted for TSP instances containing from 1000 to 1,000,000 nodes, and suggest several natural ways to obtain further enhancements for future generations of TSP methods.

2. Algorithm description

We consider a local search algorithm based on the S&C ejection chain method proposed in Glover [3]. The current implementation of the S&C algorithm is a slight variant of the P-SEC algorithm described in Rego [4].

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 (SEC) method, which generates moves coordinated by a reference structure called a S&C. The S&C 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 LK approach. A theoretical analysis of the differences between the types of paths generated by ejection chain procedures (including the S&C approach) and the LK approach is given by Funke et al. [5].

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 for S&C ejection chains (described in [4]) make the current version of the S&C approach 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 data structures. The first introduces the two-level tree data structure (2L tree) described in [6], and used in some of the most efficient LK implementations reported in the DIMACS Challenge (e.g. those of Johnson and McGeoch [7]; Applegate, Cook and Rohe [8]; and Helsgaun [9]). In our modified algorithm we have adapted the 2L 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 (NN) 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 and more direct than LK 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 SEC algorithm whose implementation is described in Rego [4] and now enhanced in the present work is called S&C. The S&C structure has its theoretical foundation in Glover [3] and is defined by a spanning subgraph of G, consisting of a path $ST = (v_t, \ldots, v_r)$ called the stem, attached to a cycle $CY = (v_r, v_{s_1}, \ldots, v_{s_2}, v_r)$. An illustrative diagram of a S&C structure is shown in Fig. 1. The vertex v_r is 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.

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:

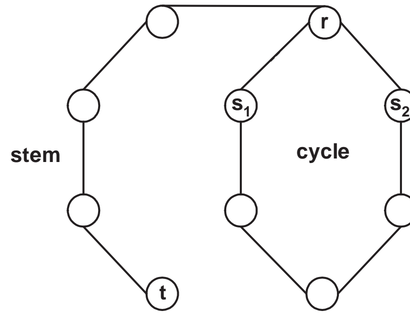

Fig. 1. The stem-and-cycle reference structure.

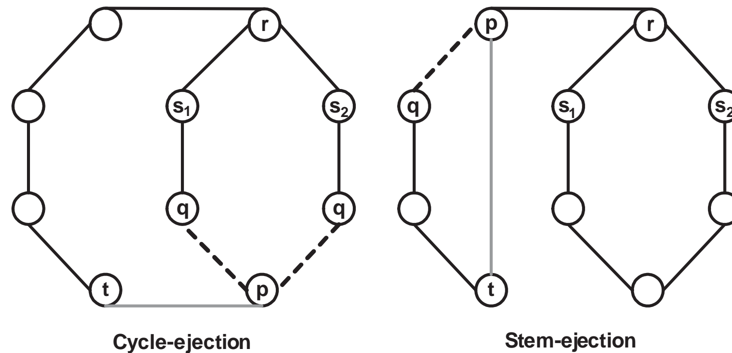

Fig. 2. Ejection moves.

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 $(v_\mathrm{t}, \ldots, v_p)$. Vertex v_q becomes the new tip.

Fig. 2 illustrates an example of the application of each of these moves to the S&C structure of Fig. 1. In the example, gray 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.

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

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 2L tree. This structure was initially proposed by Chrobak et al. [10] and has been used in efficient implementations of the 2- and 3-Opt procedures as well as their

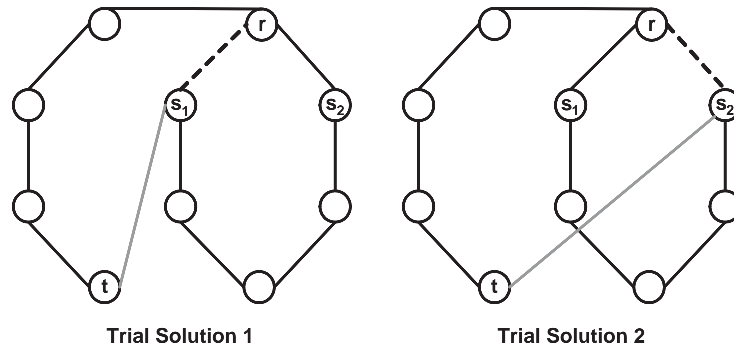

Fig. 3. Trial solutions.

generalization of the LK procedure [2]. Fredman et al. [6] show the improvement in performance over the array data structure obtained in their implementation of the LK algorithm by using the 2L tree—they also report results for two other data structures.

The main advantage of the 2L tree is its ability to reverse an entire subpath of the structure in constant time, thus significantly reducing the computational times for large-scale problems. The path reversal issue arises because in computer implementation, an orientation is assumed in order to make the structure readable. 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 an orientation that corresponds to the tour produced.

In LK implementations, the 2L 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 LK approach.

Our 2L 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). Fig. 4(B) shows the *Parent* and the segment node structures and gives an example of the 2L tree representation of the S&C structure, shown in Fig. 4(A), 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 (ID). The numbering within one segment is required to be consecutive (but it does not need to start at 1), since each ID 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 (ID), 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.

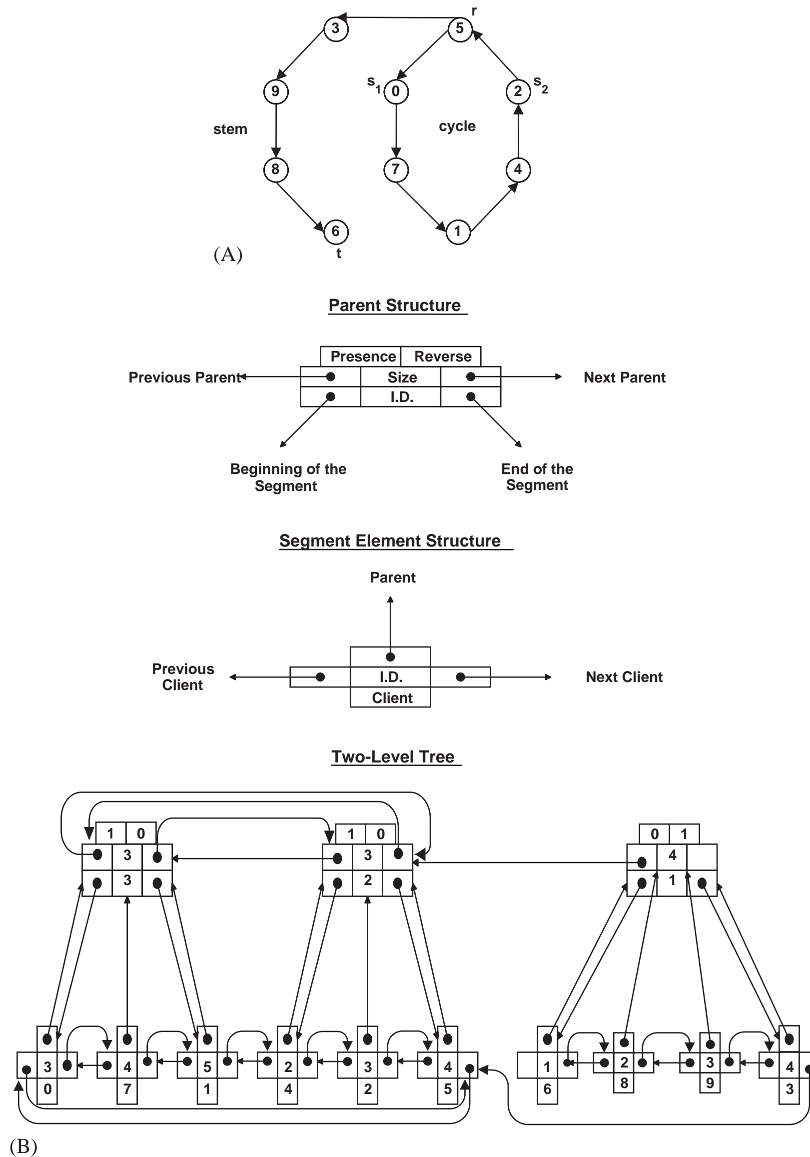

Fig. 4. (B) A two-level tree representation of the S&C structure shown in graph (A).

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

This 2L tree structure is a special adaptation of the one described in Fredman et al. [6]. This adaptation involves a substantial modification of the operations described in [6] due to the significant differences between the S&C and the LK 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 2L 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.

Five basic operations are needed to manage the 2L 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:

2.2.1. 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.

2.2.2. 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 2L 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 et al. [11].

In order to clarify the effects of the execution of an operation, an example of a cycle-ejection move on the graph of Fig. 4(A) and the associated 2L tree (Fig. 4(B)) is illustrated in Fig. 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 ID). 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

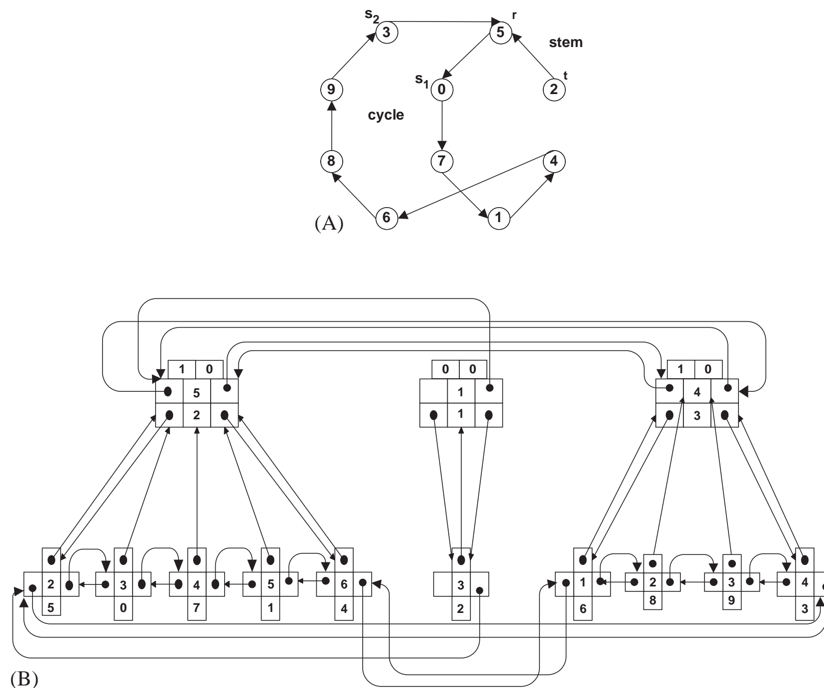

Fig. 5. Cycle-ejection move: resulting graph and 2L tree structure.

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 ID numbers of the *Parent* nodes starting at the new *tip*'s (node 2) *Parent* node.

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" [1] from classes E (uniformly distributed clients) and C (clients organized in clusters) as well as a set of instances from the TSPLIB library [12] 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 2L 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 [4] that only differ in the data structures used to represent the S&C structure

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

Problem	n	Time/n		Difference (1) − (2)	Times (1) slower than (2)
		Array(1)	2L Tree(2)		
E1k.0	1000	0.013	0.008	0.005	0.6
E3k.0	3162	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	1000	0.011	0.008	0.003	0.4
C3k.0	3162	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.tsp	7397	0.091	0.015	0.076	5.1
rl11849.tsp	11,849	0.145	0.020	0.125	6.3
usa13509.tsp	13,509	0.129	0.019	0.110	5.8
d18512.tsp	18,512	0.267	0.025	0.242	9.7
pla33810.tsp	33,810	0.758	0.060	0.698	11.6
pla85900.tsp	85,900	0.701	0.048	0.653	13.6

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 2L tree version.

The results show that the efficiency of the 2L 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 2L 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

Table 2
Results for different initial solutions

Problem	Candidate lists							
	20QN				50NN			
	% above the optimal solution or above the Held and Karp lower bound							
	Boruvka	*Greedy*	*Nearest*	*Random*	*Boruvka*	*Greedy*	*Nearest*	*Random*
E100k.0	1.709	1.762	**1.694**	1.707	1.738	**1.662**	1.815	1.666
C100k.0	3.684	3.496	**3.453**	3.859	6.839	**6.529**	8.276	38.612
pla7397.tsp	**0.846**	1.168	0.958	0.943	**0.756**	1.964	1.098	0.918
rl11849.tsp	1.654	**0.931**	1.390	1.537	1.249	**0.998**	1.064	1.507
usa13509.tsp	1.100	1.109	1.186	**1.080**	1.158	1.257	0.960	**0.934**
d18512.tsp	**0.754**	0.877	0.958	0.947	0.915	0.887	**0.822**	0.873
pla33810.tsp	1.115	1.481	**1.089**	1.201	1.518	1.143	1.214	**0.993**
pla85900.tsp	1.358	1.337	**1.113**	1.840	1.115	**1.020**	1.235	1.034
Average	1.528	1.520	**1.480**	1.639	**1.911**	1.933	2.061	5.817
Number of times best	2	1	4	1	1	4	1	2

(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). We should point out that the purpose of using random solutions is not to provide a significant statistical assessment on the performance or robustness of the algorithm when starting from random initial solutions, but rather it is to verify how the algorithm performs starting from significantly unstructured solutions comparatively to well structured solutions provided by various constructive methods. To this end, we have generated four random solutions for each instance based on the systematic generator in Rego [4], with $k = 2, 5, 8$ and 10 to provide starting solutions for four runs—results for the "Random" columns are the average of the four runs. The Concorde TSP Library [13] was used to generate the remaining initial solutions and the candidate lists. All runs were performed with a fixed set of algorithm parameters.

The aforementioned candidate lists are created by finding 20 quadrant neighbors—5 nearest neighbours from each of the four quadrants of the Euclidian plane—(20QN) or 50 nearest neighbors (50NN) for each node in the problem. The numbers 20 and 50 were chosen experimentally. Other candidate numbers were tested but it turned out that these numbers provide a good trade-off between solution quality and running times. In fact, for superior values the solution quality does not improve, only the running times increase.

The choice of the constructive algorithms was also decided experimentally. In both cases our conclusions led to the use of candidate lists and constructive algorithms similar to the ones used by the other authors reporting results in the "8th DIMACS Implementation Challenge" [1].

The quality of the solutions obtained is measured by the percentage above the optimal solution (when known) or the Held and Karp lower bound [14,15].

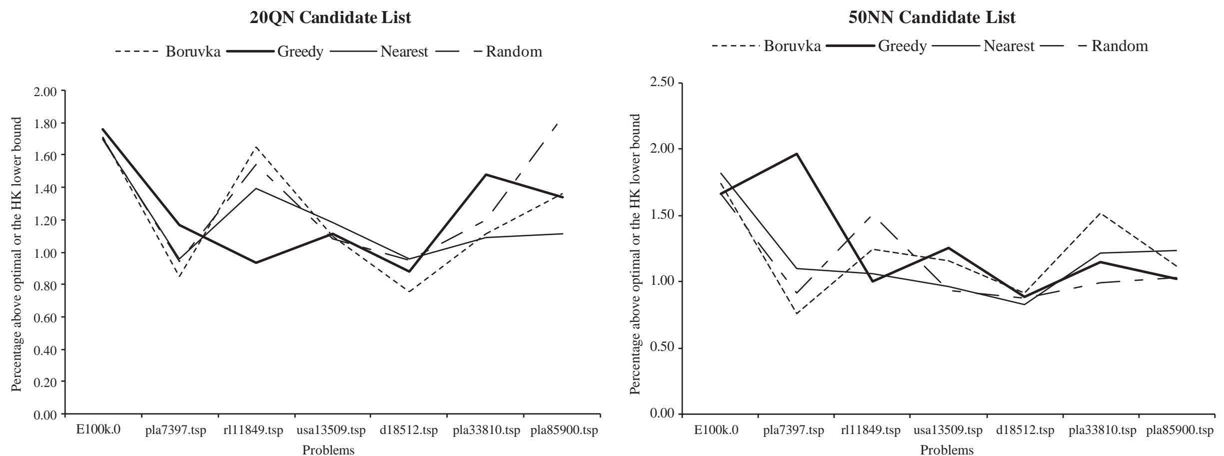

Fig. 6. Comparing the effect of the initial solutions.

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 (the values in bold point out these best results).

Graphs in Fig. 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 for each type of candidate list (20QN or 50NN), the algorithm finds solutions of similar quality (on average) regardless of the type of constructive algorithm used to generate the starting solution (Boruvka, Greedy, or Nearest).

However, by contrasting both sides of the table, we can see that the average quality of the solutions produced using the 20QN candidate list is superior to the ones achieved with the 50NN candidate list. Especially, for clustered problems the candidate list can have a great impact on the quality of the final solution, as indicated by the terrible result obtained for the C100k.0 instance using the 50NN candidate list. Specifically, these results indicate 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. At this point, it is clear that the candidate list is a key factor for the performance of a local search algorithm. A more extensive examination of the possible effect of the candidate design is presented in the next section.

As far as the combination of the initial solution and candidate list is concerned the 20QN/nearest leads to the highest average solution quality and allows the algorithm to find four best solutions (if restricting the analysis to the 20QN candidate list). However, if the 50NN candidate list is used, the Greedy procedure seems to be a better choice as the algorithm finds four best solutions (if restricting the analysis to the 50NN candidate list) with an average solution quality similar to the one obtained by the best combination (50NN/Boruvka) in this set. These results indicate that there is no significant evidence of domination of one initial solution procedure over all the others. In fact, the graphics in Fig. 6 clearly show that the choice of any of these initial solutions does not greatly influence the quality of the final solutions.

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.

The motivation for the chosen candidate lists results from the fact that NN is a classical candidate (or neighbor) list but has important limitations, such as its fixed size, not exploiting the geometric structure of the problem and not suitable for problems where vertices on a Euclidian plane occur in separate clusters. Quadrant neighbor (QN) is a more appropriate neighbor list as it considers nearest neighbors in four different quadrants. As shown in Reinelt [16] a candidate list based on the computation of the Delaunay graph generates edges that are useful for many TSP instances. Moreover, this type of neighbor list is usually different from those produced by NN or QN lists.

Fig. 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.

Table 3 shows that the use of the 12QN+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

Table 3
Results for different candidate lists

Problem	Initial solutions							
	Greedy				Random			
	% above the optimal solution or above the Held and Karp lower bound							
	12QN	12QN+D	50NN	50NN+D	12QN	12QN+D	50NN	50NN+D
E100k.0	1.757	1.653	1.662	**1.632**	1.760	1.694	**1.666**	1.819
C100k.0	**3.558**	3.881	6.529	4.051	4.439	**3.929**	38.612	5.482
pla7397.tsp	0.998	**0.931**	1.964	1.305	2.021	0.819	0.918	**0.815**
rl11849.tsp	1.642	1.200	**0.998**	1.165	1.842	1.615	1.507	**1.257**
usa13509.tsp	0.868	1.284	1.257	**0.800**	**0.905**	0.994	0.934	1.093
d18512.tsp	1.237	**0.795**	0.887	0.949	0.961	0.889	**0.873**	0.964
pla33810.tsp	1.551	**1.0165**	1.143	1.0171	2.457	1.599	**0.993**	1.378
pla85900.tsp	1.223	1.168	**1.020**	1.131	2.949	1.797	1.034	**1.027**
Average	1.604	**1.491**	1.933	1.506	2.167	**1.667**	5.817	1.729
Number of times best	1	3	2	2	1	1	3	3

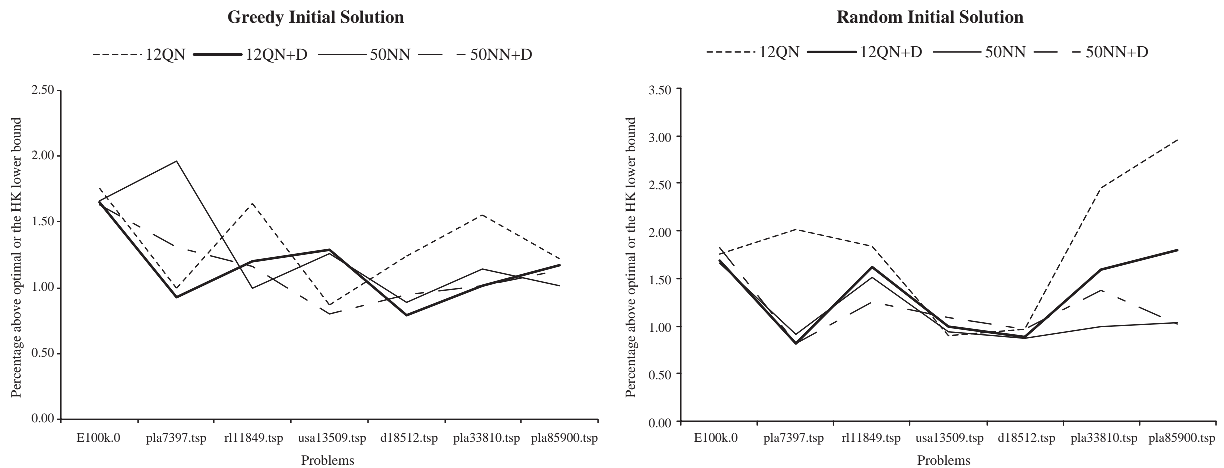

Fig. 7. Comparing candidate lists.

and relative advantage of this candidate list is illustrated in Fig. 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 NN list enriched with Delaunay 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 NN list (50NN), regardless of the initial solution.

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” [1] 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 [1].

The complete testbed consists of instances of class E (sizes between 1000 and 10,000,000 nodes), C (sizes between 1000 and 316,228 clients), and those from the TSP Library [12] with at least 1000 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 3000 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.

Table 4 and Figs. 8 and 9 provide comparative results on the performance of the following algorithms:

SC: The S&C algorithm, 12QN+D candidate list, *Boruvka* (SC-B) and *Greedy* (SC-G) 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 LK algorithm by Johnson and McGeoch [7], *Greedy* initial solutions, 20QN candidate list, “don’t look bits” strategy, two-level tree structure.

Table 4
Comparing TSP algorithms

Problem	Optimal or HK lower bound	SC-B	SC-G	LK-LM	LK-N	LK-ABCC	LK-ACR
		Percentage above optimal or HK lower bound					
E1k.0	23,360,648	**0.856**	0.939	1.422	1.055	1.525	1.632
E1k.4	22,698,717	0.983	**0.683**	1.360	1.135	1.435	1.851
E1k.8	23,025,754	**0.512**	0.759	0.899	1.031	2.116	1.143
E3k.0	40,634,081	1.056	**0.825**	1.125	1.444	1.560	2.080
E3k.2	40,303,394	0.858	**0.743**	1.382	1.262	1.519	2.883
E3k.4	40,757,209	**0.866**	1.014	1.176	1.208	1.934	2.094
E10k.0	71,362,276	1.706	**1.555**	1.958	2.045	2.603	2.591
E10k.1	71,565,485	1.718	**1.570**	2.093	1.949	2.572	3.124
E10k.2	71,351,795	**1.665**	1.921	2.000	1.963	2.629	2.435
E31k.0	126,474,847	**1.456**	1.566	2.057	1.868	2.542	2.945
E31k.1	126,647,285	**1.616**	1.656	1.985	1.899	2.425	2.497
E100k.0	224,330,692	1.751	**1.653**	2.022	1.954	2.551	2.624
E100k.1	224,241,789	1.744	**1.641**	1.924	1.954	2.528	2.862
E316k.0	398,582,616	1.876	**1.864**	1.963	1.966	2.668	2.748
E1M.0	708,703,513	2.022	**1.907**	1.956	1.925	2.684	2.770
	Average	1.379	**1.353**	1.688	1.644	2.219	2.419
C1k.0	11,387,430	1.189	1.355	**0.758**	1.361	2.233	3.416
C1k.4	11,499,958	1.727	**1.334**	2.313	2.324	2.231	2.701
C1k.8	11,605,723	**1.105**	1.187	1.297	1.329	3.019	6.439
C3k.0	19,198,258	**1.619**	1.851	1.812	2.225	6.201	5.993
C3k.2	19,547,551	1.960	2.362	**1.732**	4.008	5.139	4.830
C3k.4	18,864,046	**1.484**	1.839	2.335	3.294	6.453	7.327
C10k.0	32,782,155	3.709	2.884	**2.600**	4.534	5.436	5.796
C10k.1	32,958,946	**2.882**	3.344	4.654	5.014	6.198	6.378
C10k.2	32,926,889	3.253	3.494	**2.985**	4.737	5.472	5.599
C31k.0	59,169,193	3.367	**3.035**	3.824	4.540	5.677	5.720
C31k.1	58,840,096	3.601	**3.278**	3.610	4.298	7.080	6.835
C100k.0	103,916,254	3.528	3.881	**3.409**	4.702	5.507	5.480
C100k.1	104,663,040	3.710	**3.501**	3.856	4.864	5.106	5.613
C316k.0	185,576,667	4.097	3.990	**3.667**	–	5.452	5.542
	Average	**2.659**	2.667	2.775	3.633	5.086	5.548
pcb3038	137,694	0.755	**0.696**	1.166	1.339	2.160	2.914
fl3795	28,772	0.848	**0.834**	2.252	3.211	2.259	6.433
fnl4461	182,566	0.752	**0.555**	1.229	1.131	1.756	1.811
rl5915	565,530	1.025	**0.970**	1.325	1.072	3.321	3.080
rl5934	556,045	**1.179**	1.451	1.648	1.723	2.618	2.607
pla7397	23,260,728	**0.804**	0.931	1.308	1.408	1.938	2.804
rl11849	923,288	1.312	**1.200**	1.496	1.395	2.469	2.800
usa13509	19,982,859	**1.109**	1.284	1.263	1.255	2.521	2.639
brd14051	469,375	0.910	**0.882**	1.453	1.265	1.773	3.540
d15112	1,573,040	**0.885**	0.898	1.107	1.145	1.821	1.879
d18512	645,230	0.823	**0.795**	1.135	1.167	1.552	1.769
pla33810	66,033,000	**1.003**	1.017	1.227	1.491	1.654	2.400
pla85900	142,360,000	1.190	**1.168**	1.213	—	1.208	2.003
	Average	**0.969**	0.975	1.371	1.467	2.081	2.821

Fig. 8. Comparing TSP algorithms.

LK-N : Implementation of the LK algorithm by Neto [17], 20QN+20NN candidate list, especial routines for "Clusters" compensation, "don't look bits" strategy, two-level tree structure.

LK-ABCC: Implementation of the LK algorithm (in the Concorde library) by Applegate et al. [18,13], 12QN candidate list, *Q-Boruvka* initial solutions, "don't look bits" strategy, two-level tree structure.

LK-ACR: Implementation of the LK algorithm by Applegate et al. [8], 12QN candidate list, "don't look bits" strategy, two-level tree structure.

Table 4 summarizes the quality of solutions obtained by the five algorithms on the testbed described, grouped by class of problems, presenting average values and pointing out the best results (values in bold). The graphics in Fig. 8 only show the S&C results using the *Greedy* initial solution.

Table 4 and the graphics in Fig. 8 show that the S&C algorithm is very robust and clearly outperforms all implementations of the LK procedure.

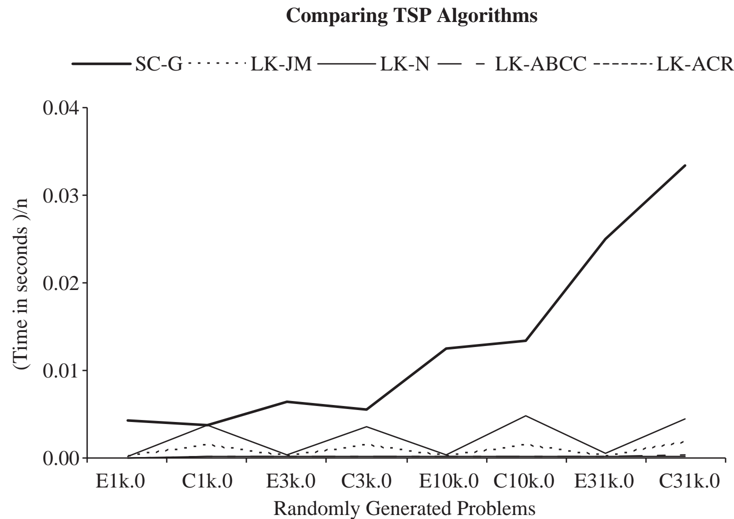

Fig. 9. Comparing running times.

Fig. 9 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-G algorithm suddenly deteriorate for sizes larger than 10,000 nodes, making the algorithm less efficient than the LK implementations for such problem sizes. However, it is important to note that state-of-the-art LK implementations (e.g. LK–JM) contain several tuning and implementation tricks (not described in this paper) that explain its relative efficiency. In contrast, the S&C implementation has no additional components or tuning beyond the basic approach already indicated.

Graphics in Fig. 10 show the normalized running times of the S&C algorithm. For the problems of groups E and C the running times significantly increase for problems over 100,000 nodes. As shown in Fig. 10 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 S&C 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.

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.

Efficiency can also be improved by avoiding repeated computations of the same objective value as in the Johnson and McGeogh LK implementation [7] where the run-time performance is optimized by using a caching technique.

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

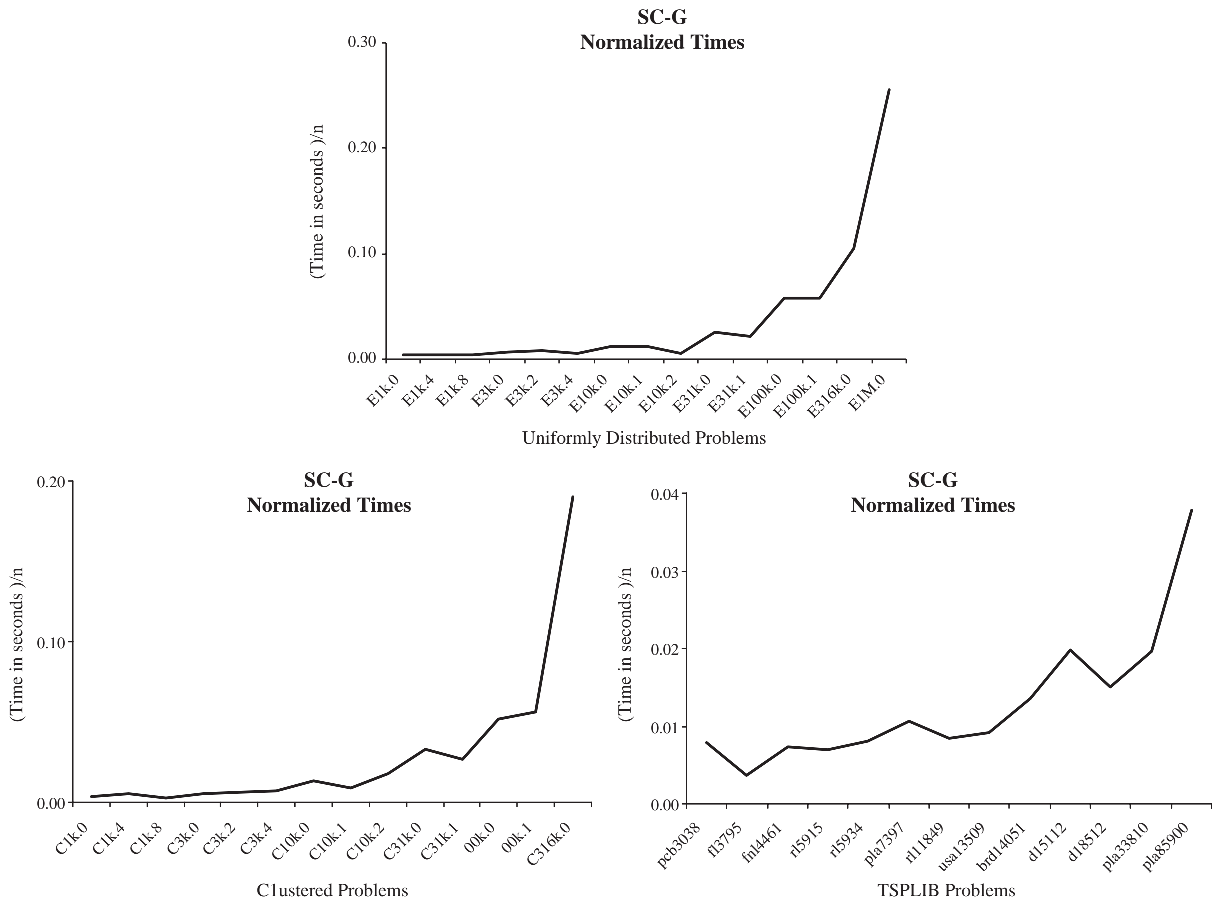

Fig. 10. Stem-and-cycle running times.

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 S&C algorithm.

These results suggest that the S&C neighborhood structure provides additional advantages over the LK structure.

Although the possibilities for disconnected moves exist as an integral component of a generalization of the S&C ejection chain method (the doubly rooted reference structure of Glover [3]), 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 S&C approach to outperform the leading LK methods, without recourse to the usual array of supplementary strategies and auxiliary neighborhoods used to make these alternative

methods competitive, suggests that a significant opportunity exists for additional enhancement of the S&C approach. 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 S&C approach remain unexplored, including those arising from more general doubly routed reference structures. The incorporation of multilevel strategies as in the interesting study of Walshaw [19] also provides an area whose investigation may hold promise.

Apart from the quality of the solutions that can be obtained by the S&C approach, the computation time remains an important factor to consider, especially when very large instances have to be solved. 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 more effective data structures than the current widely used 2L 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 are currently under examination and will be reported in future work.

References

[1] Johnson DS, McGeogh L, Glover F, Rego C. 8th DIMACS Implementation Challenge: The Traveling Salesman Problem. Technical report, AT&T Labs, 2000 ⟨http://www.research.att.com/∼dsj/chtsp/⟩

[2] Lin S, Kernighan B. An effective heuristic algorithm for the traveling salesman problem. Operations Research 1973;21: 498–516.

[3] Glover F. New ejection chain and alternating path methods for traveling salesman problems. Computer Science and Operations Research (1992) 449–509.

[4] Rego C. Relaxed tours and path ejections for the traveling salesman problem. European Journal of Operational Research 1998;106:522–38.

[5] Funke B, Grünert T, Irnich S. A note on single alternating cycle neighborhoods for the TSP, Lehr- und Forschungsgebiet Operations Research und Logistic Management, Rheinisch-Westfälische Hochschule (RWTH) Aachen, Germany, 2003.

[6] Fredman ML, Johnson DS, McGeoch LA, Ostheimer G. Data structures for traveling salesman. Journal of Algorithms 1995;18:432–79.

[7] Johnson DS, McGeoch LA. The traveling salesman problem: a case study in local optimization. In: Aarts EHL, Lenstra JK, editors. Local search in combinatorial optimization. New York: Wiley; 1997. p. 215–310.

[8] Applegate D, Cook W, Rohe A. Chained Lin–Kernighan for large traveling salesman problems. INFORMS Journal on Computing 2003;15:82–92.

[9] Helsgaun K. An effective implementation of the Lin–Kernighan traveling salesman heuristic. European Journal of Operational Research 2000;1:106–30.

[10] Chrobak M, Szymacha T, Krawczyk A. A data structure useful for finding Hamiltonian cycles. Theoretical Computer Science 1990;71:419–24.

[11] Gamboa D, Rego C, Glover F. Data structures and ejection chains for solving large scale traveling salesman problems. European Journal of Operational Research, 2005;160(1):154–71.

[12] Reinelt G. TSPLIB—a traveling salesman problem library. ORSA Journal on Computing 1991;3:376–84.

[13] Chvatal V, Applegate D, Bixby R, Cook W. Concorde: a code for solving traveling salesman problems. 1999 ⟨http://www.math.princeton.edu/tsp/concorde.html⟩

[14] Held H, Karp RM. The traveling salesman problem and minimum spanning trees. Operations Research 1970;18:1138–62.

[15] Held H, Karp RM. the traveling salesman problem and minimum spanning trees: Part II. Mathematical Programming 1971;1:6–25.

[16] Reinelt G. Fast heuristics for large geometric traveling salesman problems. ORSA Journal on Computing 1992;4:206–17.

[17] Neto D. Efficient cluster compensation for Lin–Kernighan heuristics, Department of Computer Science, University of Toronto, 1999.

[18] Applegat D, Bixby R, Chvatal V, Cook W. Finding Tours in the TSP, Research Institute for Discrete Mathemetics, Universitat Bonn, 1999.

[19] Walshaw C. A multilevel approach to the travelling salesman problem. Operations Research 2002;50(5):862–77.