Available online at www.sciencedirect.com

SCIENCE DIRECT®

European Journal of Operational Research 160 (2005) 154–171

EUROPEAN JOURNAL OF OPERATIONAL RESEARCH

www.elsevier.com/locate/dsw

Data structures and ejection chains for solving large-scale traveling salesman problems [☆]

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, MS 38677, USA*

[c] *Leads School of Business, University of Colorado, Boulder, CO 80309-0419, USA*

Available online 8 June 2004

Abstract

Data structures play a crucial role in the efficient implementation of local search algorithms for problems that require circuit optimization in graphs. The traveling salesman problem (TSP) is the benchmark problem used in this study where two implementations of the stem-and-cycle (S&C) ejection chain algorithm are compared. The first implementation uses an Array data structure organized as a doubly linked list to represent TSP tours as well as the S&C reference structure. The second implementation considers a two-level tree structure. The motivation for this study comes from the fact that the S&C neighborhood structure usually requires subpaths to be reversed in order to preserve a feasible orientation for the resulting tour. The traditional Array structure proves to be inefficient for large-scale problems since to accomplish a path reversal it is necessary to update the predecessor and the successor of each node on the path to be reversed. Computational results performed on a set of benchmark problems up to 316,228 nodes clearly demonstrate the relative efficiency of the two-level tree data structure.

Keywords: Traveling salesman; Local search; Data structures; Ejection chains

1. Introduction

Generically, the traveling salesman problem (TSP) consists in sequentially visiting a set of clients (cities, locations) only once and finally returning to the initial client. The goal is to find the tour of minimum total distance (or other cost measure associated with the performed trajectory).

In graph theory, the problem can be defined as a graph $G = (V, A)$ with n vertices (or nodes) $V = \{v_1, \dots, v_n\}$ and a set of edges $A = \{(v_i, v_j) | v_i, v_j \in V,\ i \neq j\}$ with a non-negative cost (or distance) matrix $C = (c_{ij})$ associated with A. The problem's resolution consists in determining the minimum cost Hamiltonian cycle on the problem graph. In this paper, we consider the symmetric version of the problem $(c_{ij} = c_{ji})$, which satisfies the triangular inequality $(c_{ij} + c_{jk} \geqslant c_{ik})$.

[☆] 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).

The TSP is well known as a NP-hard combinatorial problem; hence, there is no algorithm capable to solve all possible instances in polynomial time. Consequently, it becomes absolutely necessary to use heuristic (or approximate) algorithms to provide solutions that are as good as possible but not necessarily the optimal. The importance of obtaining efficient heuristics to solve large-scale TSPs recently motivated Johnson, McGeogh, Glover, and Rego to organize the "8th DIMACS Implementation Challenge" specific for TSP algorithms [5]. This paper is based on the development of several algorithmic components and data structures with the purpose of improving the efficiency of the stem-and-cycle algorithm described in Rego [9].

A fundamental aspect forming the basis for this study concerns the following. The data structure representation of a symmetric TSP tour requires the specification of an orientation by which the tour can be read (or traversed), even though the cost of crossing in one direction or in the opposite direction is equivalent. The relevance of this orientation becomes more evident with local search algorithms where moves often require the reversal of a subpath in order to preserve an admissible orientation for a TSP tour.

A typical example of the need to reverse paths occurs in the application of classic procedures of the type k-optimal, initially proposed for the TSP [7]. The same phenomenon occurs with the moves generated in some subpath using ejection chain methods, in particular those oriented by a reference structure.

Naturally, the need to reverse paths at each iteration of the algorithm requires a computational effort that becomes particularly relevant when large-scale problems have to be solved.

The choice of the data structure to represent TSP solutions is crucial when applying neighborhood structures that require path reversals, since the algorithm's complexity might drastically be reduced. Fredman et al. [2] show the relevance of that choice on their implementation of the Lin–Kernighan algorithm [8] by comparing the computational times obtained by several implementations using four different data structures: Array, splay-tree, two-level tree and segment tree.

The main goal of this study consists in analyzing and validating the potential of the two-level tree data structure in reducing the running time of the stem-and-cycle algorithm [9] with the aim of improving the algorithm's efficiency in solving large-scale problems.

The motivation for this study comes from the fact that the stem-and-cycle algorithm has proved to be extremely effective and competitive with the best algorithms for the TSP [3,6]. We therefore anticipated that the algorithm's efficiency when solving large scale problems can be improved by the implementation of a data structure specifically designed to reduce the computational complexity associated with the path reversal operations needed at each iteration of the algorithm.

2. Data structures for the S&C procedure

2.1. The stem-and-cycle reference structure

The stem-and-cycle (S&C) reference structure is described in Glover [4] and used in the subpath ejection algorithm described in Rego [9] for the TSP.

In graph theory, the S&C structure is defined by a spanning subgraph of G, consisting of a path $\mathrm{ST} = (v_t, \ldots, v_r)$ called the stem, attached to a cycle $\mathrm{CY} = (v_r, v_{s_1}, \ldots, v_{s_2}, v_r)$. 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. Fig. 1 shows a representation of the stem-and-cycle structure.

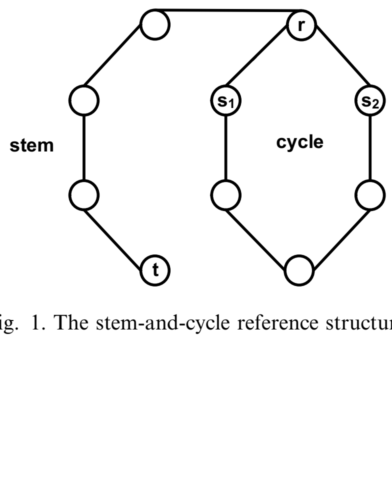


Fig. 1. The stem-and-cycle reference structure.

The 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 immediately defines the root node for the entire ejection chain.

Fig. 2 illustrates one of the possibilities to create the initial S&C structure. In the example, dotted lines represent the edges to be inserted in the new solution and the dashed lines point out possible edges to be removed from the solution. This representation will be adopted in all the figures describing moves through out this paper.

In an ejection chain a reference structure is used to generate ejection moves, which transform a reference structure into another of the same type. Since the resulting structure obtained at each level of the chain (through the application of an ejection move) usually does not represent a feasible tour, a trial move is required to restore the solution feasibility.

We define two types of 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 $(v_t, \ldots, v_p)$ (in the original structure). Vertex v_q becomes the new *tip*.

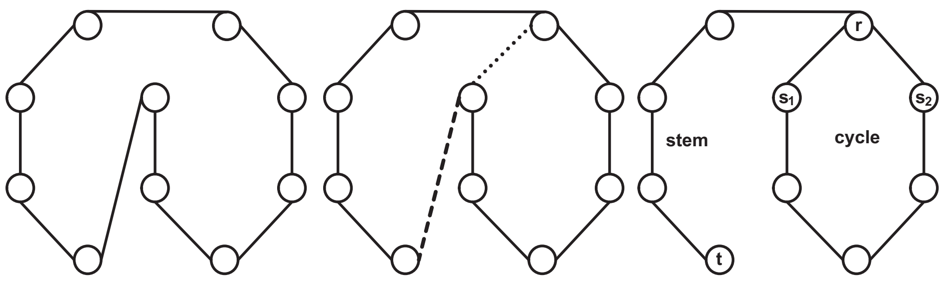


Fig. 2. Creating the initial stem-and-cycle structure.

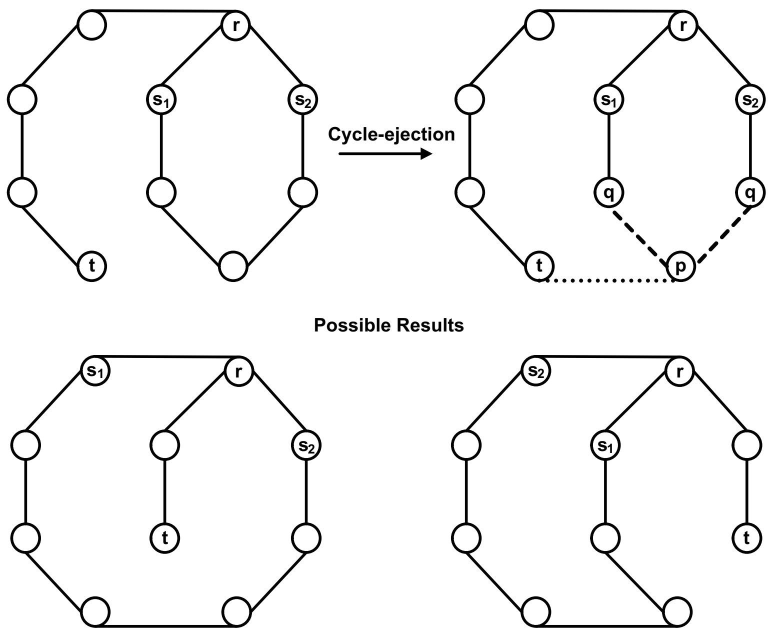


Fig. 3. Cycle-ejection move.

Fig. 3 illustrates an example of the application of a cycle-ejection move. Node t is connected to one of the cycle's nodes, in this case node p, causing the deletion of one of the two edges linking p to its adjacent nodes, in order to preserve a S&C structure. Hence, this type of move provides two possibilities to transform one reference structure into another.

It is important to notice that the operation used to create the initial S&C structure is also a cycle-ejection move that starts with a degenerated S&C structure (an Hamiltonian cycle) where the *root* and the *tip* are the same node.

The application of a stem-ejection move is illustrated in Fig. 4. In this move, node t is connected to node p in the stem; therefore, node q adjacent to p in the path $(p, \dots, t)$ is deleted. This type of move only provides one possibility to transform the S&C structure.

Up to this point, the path reversal issue has not been mentioned for this method. However, from the standpoint of computer implementation an orientation is necessary to make it possible to read the structure. Consequently, the application of any ejection move may create the need to reverse part of the structure to keep a feasible orientation. This issue will be discussed in detail in Section 2.3.

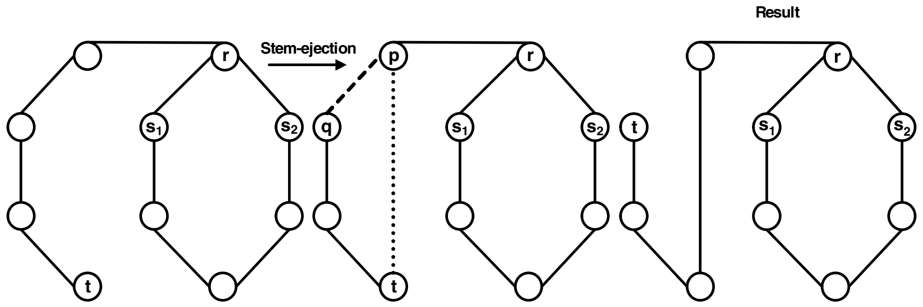


Fig. 4. Stem-ejection move.

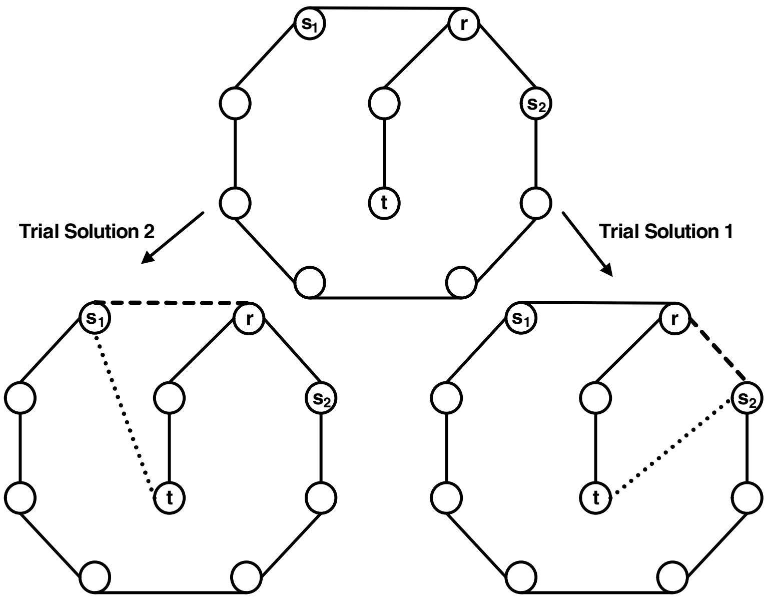


Fig. 5. Trial solutions.

Now, it is necessary to explain how to obtain a TSP solution from a S&C structure by appropriate trial moves.

Fig. 5 shows two possibilities for generating trial solutions that can be obtained from one of the resulting S&C structures in Fig. 3. 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).

2.2. The two-level tree representation

The two-level tree data structure (two-level tree) initially proposed by Chrobak et al. [1] appears as a solution for an efficient implementation of the 2-optimal and 3-optimal procedures as well as their generalization as the Lin–Kernighan procedure [8]. See Fredman et al. [2] for a comparative study on the performance of alternative data structures.

The two-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 subpath (or segment) of the tour. Each segment represents an oriented path, and the right association of all the paths represents a TSP solution. Fig. 6 shows the *Parent* and segment node structures and gives an

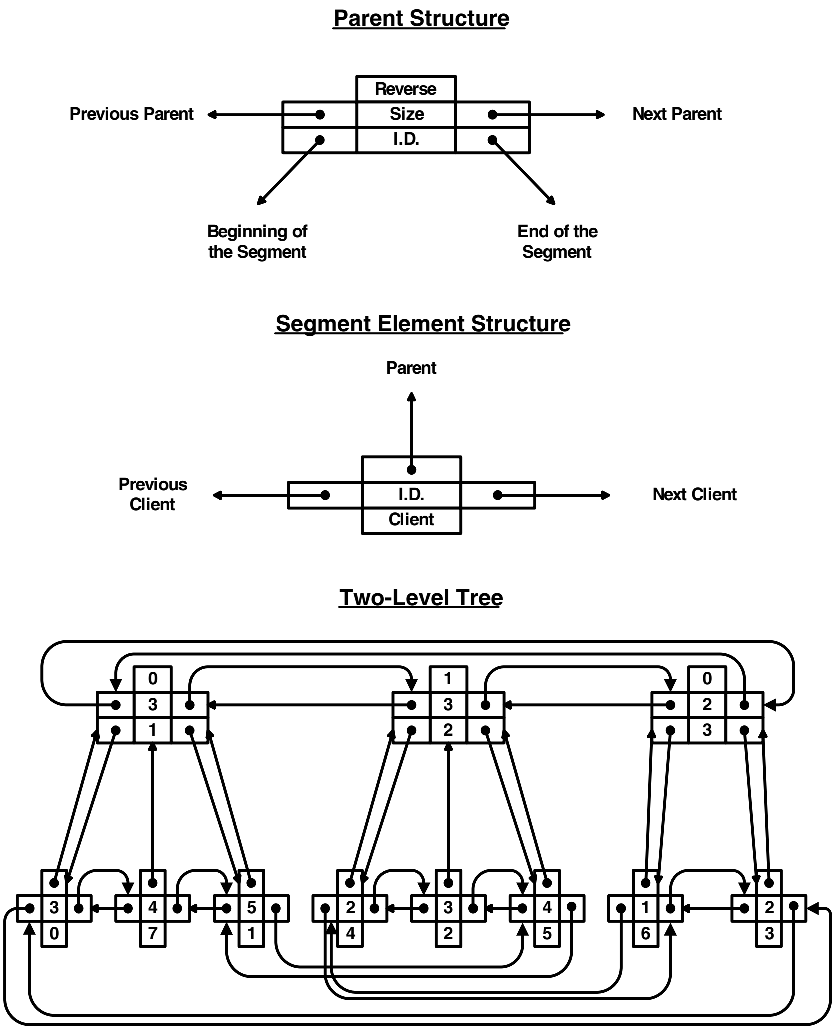


Fig. 6. The two-level tree representation of a TSP tour.

example for the representation of the tour (0, 7, 1, 5, 2, 4, 6, 3, 0) in a two-level tree structure.

Besides *Next* and *Previous* pointers, each member of a segment contains a pointer to the associated *Parent* node, 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 in the segment.

The *Parent* node construction contains important information about the associated segment such as the total number of clients (Size), pointers to the segment's endpoints, a sequence number (ID), and a reverse bit (Reverse) that indicates whether the segment should be read from left to right (if it is set to 0) or in the opposite direction (if it is 1). This makes possible to reverse the orientation of an entire segment just by flipping the reverse bit of its *Parent* node. This is the main feature that may drastically reduce the computation time of neighborhood search algorithms dealing with the optimization of circuits in graphs or networks.

Also, it is important to notice that client nodes are organized in an array structure allowing random access to any client node in the two-level tree.

2.3. Path reversal in S&C

In this section we specify when the need to reverse subpaths arises during the application of the S&C moves described in Section 2.1. Considering an orientation for both the stem and the cycle in the examples illustrated, it seems quite obvious which subpath of the structure must be reversed in order to preserve a feasible S&C structure. Likewise, after selecting the orientations in the S&C structure of Fig. 5 we can easily find out that at least for trial solution 2, it is imperative to reverse the orientation of a subpath of the structure, for example the one containing the edges belonging to the stem, so that the resulting structure represents a valid TSP solution (cf. Fig. 7).

Likewise, the application of any of the two types of ejection moves may require reversing a subpath of the reference structure. Considering the orientation chosen in Fig. 8 for the S&C structure of Fig. 3, the application for the cycle-ejection move causes the path $(r, \ldots, t)$ to be reversed (denoted by the thick arrows).

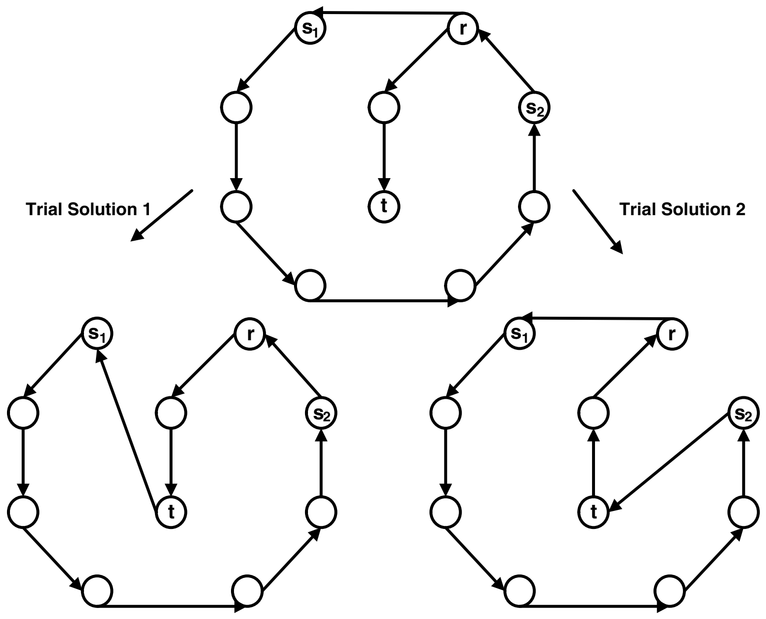


Fig. 7. Path reversal: trial solutions.

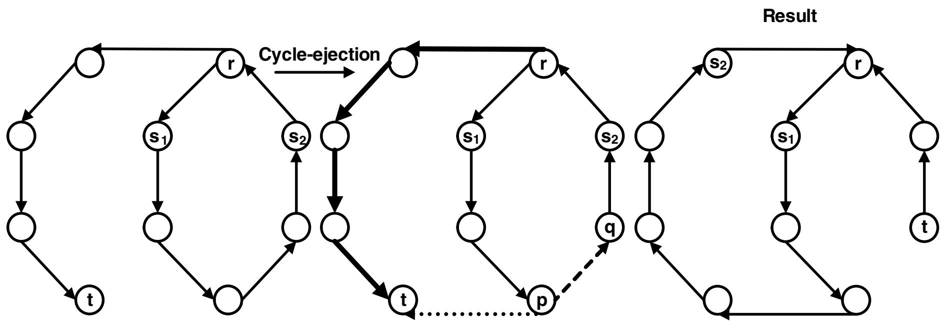


Fig. 8. Path reversal: cycle-ejection move.

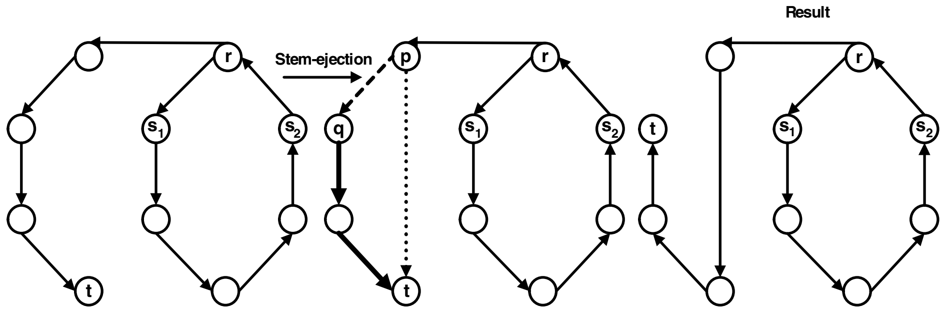


Fig. 9. Path reversal: stem-ejection move.

Fig. 9 shows the application of a stem-ejection move where the path $(q, \dots, t)$ has been reversed in order to obtain a valid S&C structure.

It is important to point out that we can always choose one of two possible paths to reverse. In fact, in the cycle-ejection move represented in Fig. 8, we could reverse path $(p, \dots, r)$ instead of path $(t, \dots, r)$. For the stem-ejection move of Fig. 9, the choice could be the path $(p, \dots, r)$. Also, for the trial move in Fig. 7, we could reverse path $(s_2, \dots, r)$, and generate a valid S&C structure.

2.4. Implementation details

Because the S&C structure usually does not represent a Hamiltonian cycle and different rules for ejections moves apply to the stem and the cycle, a presence bit has been added to the *Parent* node structure to indicate whether one node belongs to the stem or to the cycle (cf. Fig. 10). (Note that when the array is the data structure used to represent the S&C structure this information is

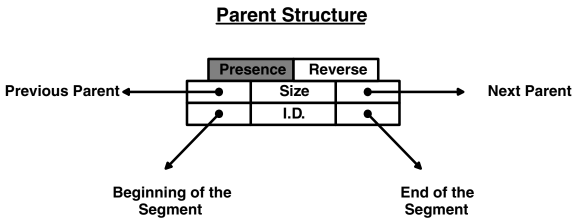


Fig. 10. Parent node structure: stem-and-cycle.

stored for every node of the structure.) Since each *Parent* node represents a segment in the two-level tree, specialized two-level tree update operations are necessary to ensure that the whole segment is either part of the stem or the cycle.

A possible two-level tree representation of a S&C structure is shown in Fig. 11. In order to simplify the illustration we restricted *Parent* node information to the presence bit, the reverse bit, and the sequence number. We also restricted node information to the index of the client it represents. The bidirectional arrows represent links in both

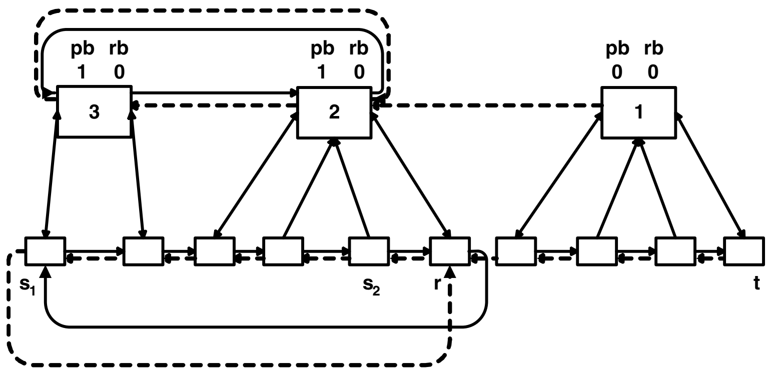


Fig. 11. A two-level tree representation of a stem-and-cycle structure.

directions and only appear in connections between segment endpoints and the associated *Parent* nodes, representing a pointer to the beginning or to the end of the segment and a pointer to the associated *Parent* node. For the remaining segment elements we represent the link to its *Parent* node. In both lists the dashed arrows represent *Previous* pointers and the other arrows *Next* pointers.

In the figure, the presence bit (pb) can be checked for each *Parent* node to verify that segments 2 and 3 store the cycle and that the stem occupies segment 1. For convenience, the root node is set to be in one of the edges of the cycle segment and the *tip* node to be the last node of one of the stem segments with one null pointer (either the previous or the next link depending on the orientation). According to this representation, the numbering of the *Parent* nodes will always start at the segment containing node t. As with node t, the associated *Parent* node also fails to define one of its links.

Besides the compulsory operations regarding the structure's orientation, $Next(a)$ and $Previous(a)$ three other operations are necessary, with one operation for each type of move considered in the stem and cycle ejection chain method. These operations can be defined as follows:

$Next(a)$, returns a's successor in the current structure. First, the operation finds node a 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 and in the opposite case following a's *Previous* pointer.

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

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

$StemEjection(r,t,p,q)$, updates the reference structure by removing edge (p,q) and inserting edge (t,p). This operation reverses the path between t and q.

$Trial(r,t,s)$, 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 be reversed.

In ejection chain moves, every time the edge to be deleted is in the same segment, the operation involves splitting (or partitioning) the segment between the edge's nodes and merging one of the resulting partitions with a neighbor segment.

Since these operations are part of the algorithm, they appear in both implementations (array and two-level tree).

Now, we describe in detail how the ejection chain operations are implemented in order to achieve the desired algorithm efficiency. For convenience we define a node a as being the *Parent* of a node b if the latter is in the segment that has a as its *Parent* node.

$CycleEjection(r,t,p,q)$ procedure:

Step 1. Renumbering of the new cycle subpath.
Flip (to 1) the presence bit of the *Parent* nodes associated with segments of the new cycle subpath (between t and r).

Step 2. Reorganizing the structure.
If p and q are in the same segment, organize the structure for splitting the segment between those nodes as follows (otherwise go to Step 3). If the root is in the same segment as p and q go to 2(a) otherwise go to 2(b).

(a) If the whole cycle is in the same segment go to (i) otherwise go to (ii).

(i) Update one of the root links to make possible the segment partitioning—link the root to the new subroot that is in the stem of the original structure. Merge the segment partition that contains the root node and then split this latter again to make the root node a segment endpoint. The merge must be with the neighbor segment different from the one that originally contained nodes p and q. These operations require updating links between the new root's *Parent* and one of the adjacent *Parent* nodes—the one associated with the subroot that keeps belonging to the cycle. Go to Step 3.

(ii) Merge the partition that does not contain the root node. Go to Step 3.

(b) Perform a legitimate merge based on the presence bit of the *Parent* nodes of the adjacent segments preventing a merge with a segment that contains the root as the endpoint to be linked to the first node of the partition.

Step 3. Setting up links between p and t.
Set up the appropriate *Next* and *Previous* pointers in order to link nodes p and t.

Step 4. Setting up links between the Parent nodes associated with p and t.
Set up the appropriate *Next* and *Previous* pointers in order to link the *Parent* nodes of p and t.

Step 5. Reversing the path between t and r.
If reversing is not necessary go to Step 6. Reverse *Next* and *Previous* pointers for each *Parent* of inner segments in the stem and flip the reverse bit for each *Parent* of reversed segments.

Step 6. Inserting the root in a cycle-segment.
If the root is in a stem-segment move it to a cycle-segment.

Step 7. Setting up the link between r and the new subroot.
If step (i) was not performed, update one of the *Next* and *Previous* pointers of node r in order to connect the root to the new subroot.

Step 8. Setting up links between r and the new subroot Parent nodes.
If step (i) was not performed, set up the appropriate *Previous* and *Next* pointers in order to link the root and the new subroot *Parent* nodes.

Step 9. Numbering the new stem (between q and r).
Set to 0 the presence bit for *Parent* nodes of segments containing the new stem.

Step 10. Renumbering the sequence numbers of the Parent nodes.
Starting from the *tip*'s *Parent* node, renumber the ID numbers of *Parent* nodes.

Step 2 needs additional clarification. If the cycle only takes one segment (e.g. case (i)) it will not be possible to split the segment since there are no adjacent segments available for completing the associated merge (cf. Fig. 12).

This problem can be solved by updating the link between the root node and the new subroot before splitting the segment. After the merge another problem arises as the root appears in the middle of the segment. The solution to this setback consists of splitting the segment and merging one of the resulting partitions with the sole possible adjacent segment, making the root as an endpoint of one of those segments (cf. Fig. 13).

Concerning case (ii) only the segment's partition that does not contain the root can be merged since there is not a legitimate segment adjacent to the root. If the root is not in the same segment as p and q (case 2(b)), the merge is decided by taking into consideration the values of the presence bit of both neighbor segments and the position of the root in those segments. The crucial decision concerns the root as its presence in one of the possible segments immediately rules out the merge with that segment because it would place the root in the

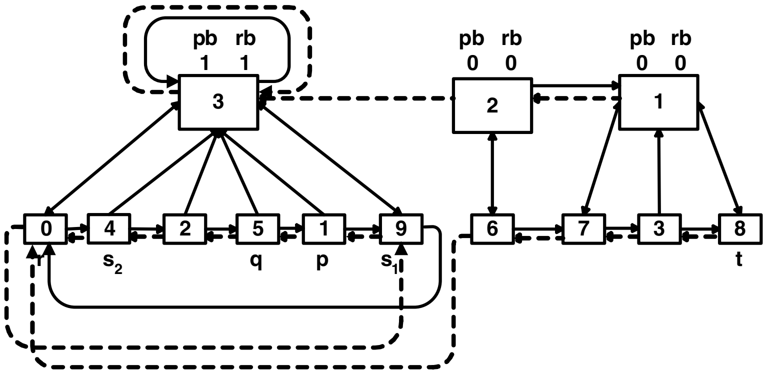


Fig. 12. Cycle-ejection: cycle occupies only one segment.

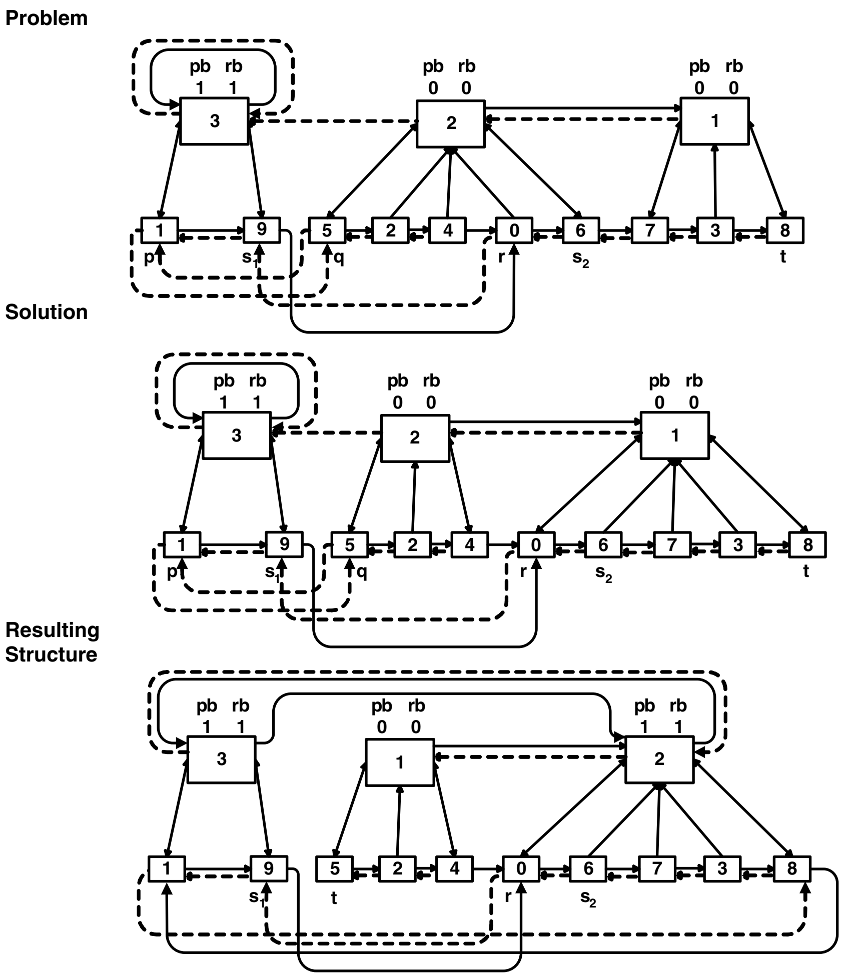


Fig. 13. Cycle-ejection: root in the middle of a segment.

middle of the segment. However, if the root causes no problem then the choice is taken realizing that the segment's partition containing q will become part of the stem and therefore should be merged into a stem segment. In the same way the segment's partition containing p should be merged into a cycle segment.

Fig. 14 exemplifies the execution of this operation based on the move illustrated in Fig. 8. The first tree structure represents a possible node distribution for the initial reference structure throughout the three tree segments. The second structure is the resulting tree right after performing the appropriate steps of the *CycleEjection* prodecure.

Initially, the cycle occupies segments 2 and 3 while the stem resides in segment 1. The operation involves numbering the new part of the cycle, which is attained by setting to 1 the presence bit of *Parent* 1. Since p and q are in the same segment, it must be split between those nodes, and node p must be merged to the initial segment 3. This is because although the root is in the split segment, the cycle occupies more than one segment. With this merge the root is placed in a stem-segment hence it is necessary to merge it into segment 3. Set up the links between p and t and between the associated *Parent* nodes. Reverse the path between t and r which is accomplished by flipping the reverse bit (rb) of *Parent* 1. Set up links between the root and its new subroot (node 6) and between the associated *Parent* nodes. Flip the presence bit of *Parent* 2 so as to number the new stem. Finally, reorder the ID numbers of the *Parent* nodes starting at the new *tip*'s *Parent* node.

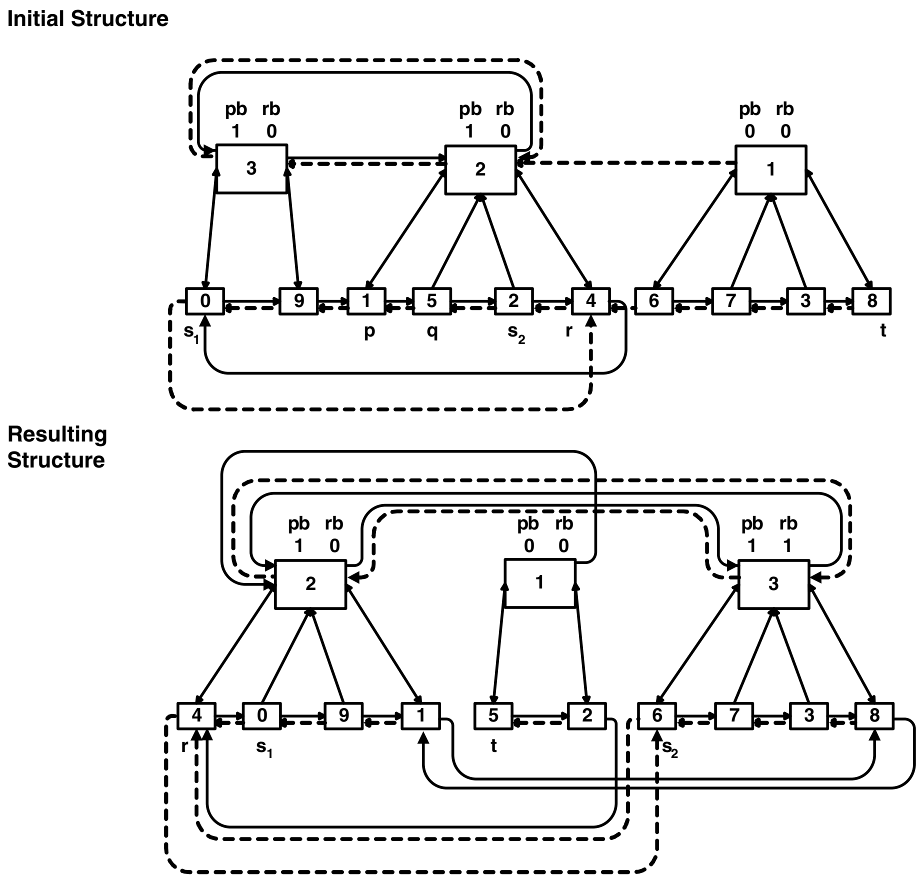


Fig. 14. Two-level tree: cycle-ejection.

StemEjection(r, t, p, q) *procedure:*

Step 1. Reorganizing the structure.
If nodes p and q are in the same segment, organize the structure for splitting the segment between those nodes as follows (otherwise go to Step 2). If node t is in the same segment as p and q, go to 1(a), otherwise go to 1(b).

(a) Merge the partition that does not contain t. If the merge is to be made with a cycle segment, split this latter to create a stem-segment and a cycle-segment. Perform the merge with the cycle-segment just created and set to 0 the presence bit of the new stem-segment. Appropriately update the links between the *Parent* node of the new segment containing the root node and the adjacent *Parent* node containing the subroot that was not in the same segment of r before the merge. Go to Step 2.

(b) Perform a legitimate merge based on the presence bit of the *Parent* nodes of the adjacent segments. The merge must be with a stem-segment. If both neighbors are stem-segments, perform the merge with the shortest segment.

Step 2. Setting up links between p and t.
Set up the appropriate *Next* and *Previous* pointers in order to link nodes p and t.

Step 3. Setting up links between the Parent nodes associated with p and t.
Set up the appropriate *Next* and *Previous* pointers in order to link the *Parent* nodes of p and t.

Step 4. Reversing the path between t and q.
Reverse *Next* and *Previous* pointers for each *Parent* of inner stem-segments in the subpath between t and q. Flip the reverse bit for *Parent* nodes of segments to be reversed.

Step 5. Renumbering the sequence numbers for parent nodes in the subpath.
Set up a sequence number ID for *Parent* nodes in the linked list starting from *tip*'s *Parent* node up to the last *Parent* before the root's *Parent* node.

If node t appears in the segment to be split in Step 1 (case 1(a)), merging the partition containing t is no longer an option. In fact, as explained earlier, node t does not have a link to an adjacent segment to be considered for the merge operation. On the other hand, the obligation to use the remaining partition may create a problem as the partition may be merged into a cycle-segment. This means that after the merge the root is placed in the middle of a segment and the result is a segment containing cycle and stem nodes. To overcome this difficulty, the new segment is split up in a way that each resulting partition exclusively contains one type of nodes (cycle or stem). After this operation the partition belonging to the cycle is merged into its neighbor cycle-segment leaving the original segment with the stem nodes. Thus, it is necessary to set the presence bit of the associated *Parent* node to 0—the stem now occupies one more segment. Furthermore, given that this is the only case involving cycle segments when applying a stem-ejection move, it is necessary to change the links between the *Parent* node of the cycle segment that now contains the root and the *Parent* node of the adjacent segment containing the subroot that was not on the same segment as the root before the last merge. This special case occurs in the example showed in Fig. 15.

On the other hand, if node t is not in the same segment as p and q (case 1(b)), the merge is decided based on the presence bit of the *Parent* nodes of the adjacent segments. The first choice is always the neighbor that belongs to the stem. However, if both neighbors are stem-segments, the merge involving the smaller partition is chosen.

Fig. 15 shows the transformations undergone by the initial tree after the application of the move illustrated in Fig. 9. Segment 1 stores the whole stem and segments 2 and 3 hold the cycle. Therefore, the execution of the *StemEjection* procedure involves: splitting segment 1 between p and q and merging p into segment 2, which is the only possible adjacent segment as t is also in segment 1. As the root appears in the middle of segment 2, this segment must be split, and the partition containing the root node must be merged to segment 3. Because the stem also occupies segment 2, the presence bit of its *Parent* node must be set to 0. The final structure is obtained by changing the following node information: establishing links between the

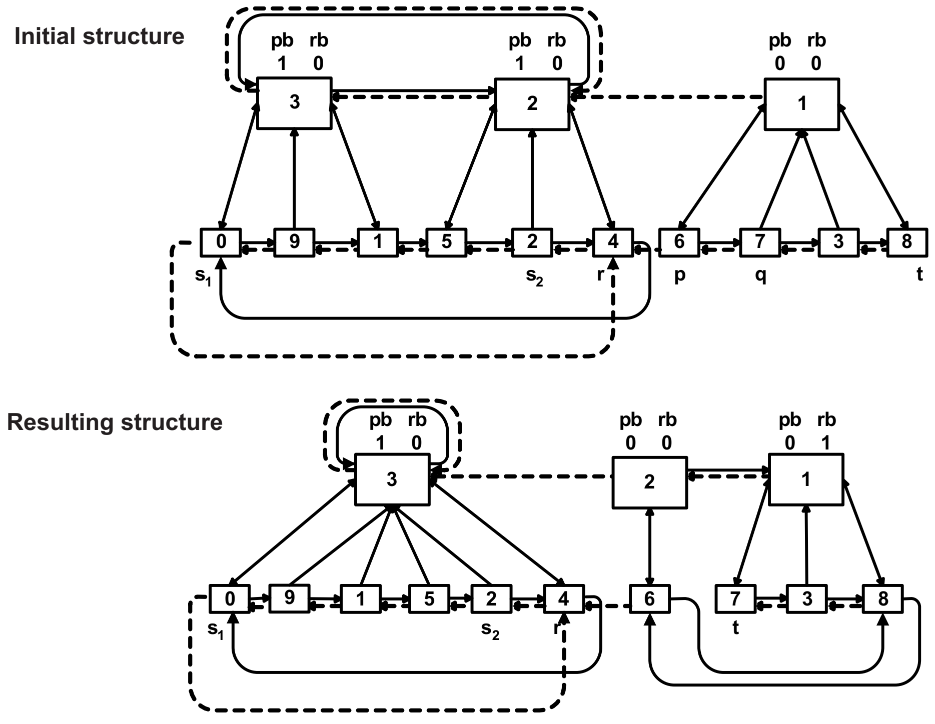


Fig. 15. Two-level tree: stem-ejection.

Parent node of the new segment containing the root (segment 3) and the *Parent* of its neighbor segment (in the case, the segment 3 itself); establishing links between p and t and between the associated *Parent* nodes; and reversing the path between t and q just by flipping the value of the reverse bit of *Parent* 1. Finally, renumber *Parent* node's ID, if necessary to keep a valid sequence.

Trial(r, t, s) *procedure:*

Step 1. Renumbering of the new cycle subpath.
Flip (from 0 to 1) the presence bit for *Parent* nodes associated with segments in the path between t and r.

Step 2. Reversing the path between t and r.
If subpath reversal is not necessary go to Step 3.
Reverse the *Previous* and *Next* pointers for each *Parent* node of inner segments in the path. Flip the reverse bit for *Parent* nodes of segments to be reversed.

Step 3. Setting up links between the root and the new subroot.
Set up either the *Previous* or *Next* pointer of node r to establish the link between the root and the new subroot.

Step 4. Setting up the link between the root and the new subroot Parent nodes.
Set up one of the *Previous* and *Next* pointers for the root's *Parent* to connect the root and the new subroot *Parent* nodes.

Step 5. Reorganizing the structure.
If r and s are in the same segment, organize the structure for splitting this segment between those nodes, if necessary (otherwise go to Step 6). This decision is made based on the following conditions: if r and s are the endpoints of the segment (and the size of the segment is greater than 2), go to 5(a) otherwise go to 5(b).

(a) It is not necessary to partition the segment. Go to Step 6.

(b) Split the segment merging the root with the segment containing the new subroot and update the link between r and s_1 Parent nodes.

Step 6. Setting up links between s and t.
Set up the appropriate *Next* and *Previous* pointers in order to link nodes s and t.

Step 7. Setting up links between s and t Parent nodes.
Set up the appropriate *Next* and *Previous* pointers in order to link the *Parent* nodes of s and t.

If the first possibility occurs in Step 5, it will not be necessary to split the segment, meaning that it will be sufficient to establish the new connections between r and the new subroot and between their *Parent* nodes. In the opposite case, the segment is split and the root is merged into the segment containing the new subroot using the link created in Step 3.

An example of this operation based on the move represented in Fig. 7, is shown in Fig. 16. Once again the stem occupies segment 1 and the cycle is distributed throughout the two other segments. Completing the *Trial* procedure involves: converting the stem-segments into cycle-segments by flipping to 1 the presence bit of *Parent* 1; reversing the path between t and r by setting to 1 the reverse bit of *Parent* 1; linking the root to its new adjacent node (the new s_2, in this case) and also setting up links between the associated *Parent* nodes (since r and s_2 are in the same segment but not endpoints it is necessary to split the segment, merging the root into the segment of the new s_2 (node 0) and set up the links between r and s_1 *Parent* nodes); setting up links between s_2 and t and between their *Parent* nodes.

2.5. Experimental results

In order to assess the relative efficiency of the new stem-and-cycle algorithm implementation (with the two-level tree data structure) compared to the original implementation (using the array data structure), several computational tests were carried out on three classes of problems. The testbed consisted of instances used in the "8th

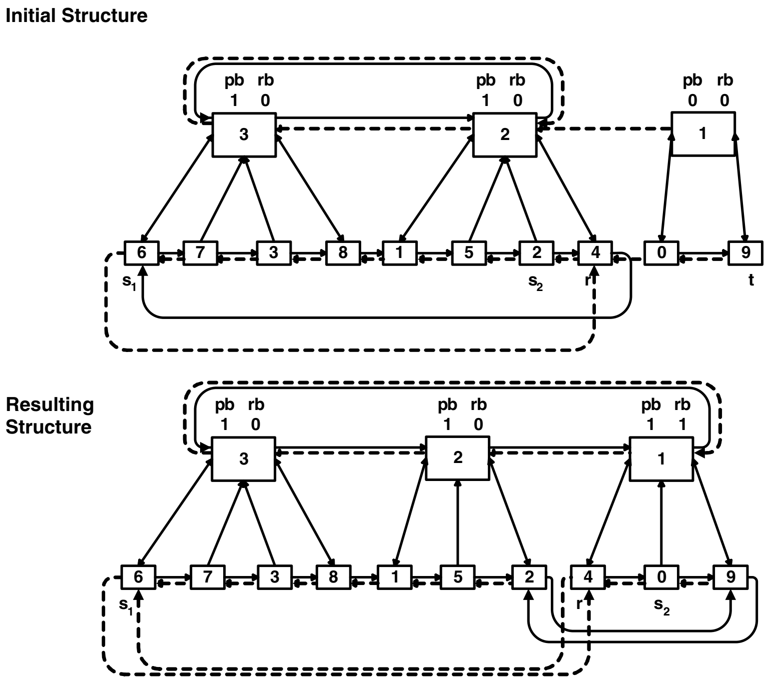


Fig. 16. Two-level tree: trial Solutions.

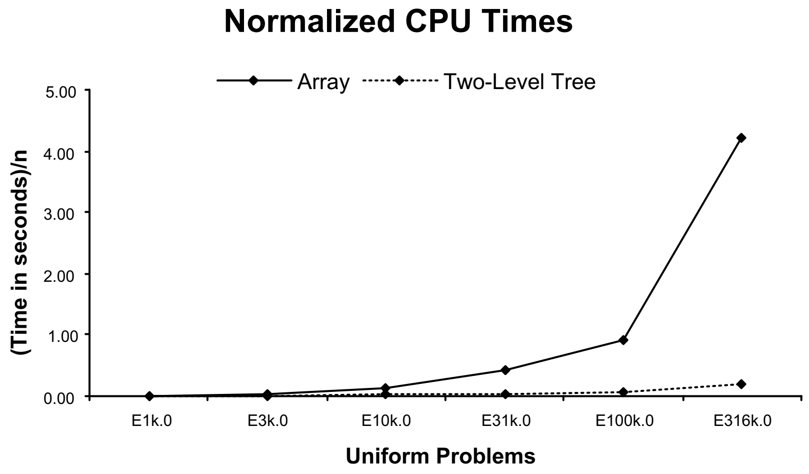
DIMACS Implementation Challenge'' [5] from classes E (uniformly distributed clients) and C (clients organized in clusters) as well as a set of instances from the TSPLIB library [10] with different characteristics and sizes. The tables/graphics in Figs. 17–19 summarize the obtained computational results that are relevant for the present study. A more extensive list of results can be found in [5]. Besides the designation and size of each instance, the tables show the total number and average length of the paths to be reversed during the algorithm's execution, 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 two-level tree version. For the largest problem some of the values are not reported since it would take to much time to rerun the array implementation for that instance.

Runs were performed on a Sun Enterprise with two 400 MHz Ultra Sparc processors and 1 GB of memory. Although this is a multiprocessor machine, the stem-and-cycle algorithm is implemented as a serial algorithm and no compiler directives for implicit parallelism are used.

In a general analysis, we can see that the efficiency of the two-level tree implementation over the array implementation grows with problem size. Considering the real values (not normalized) we can verify that in Fig. 17 the array implementation takes about 15 days to obtain an identical solution to the one provided by the two-level tree implementation in less than one day (specifically, 17 hours). According to the results reported in Fig. 18 one can expect the same gain in efficiency for the clustered type of problems.

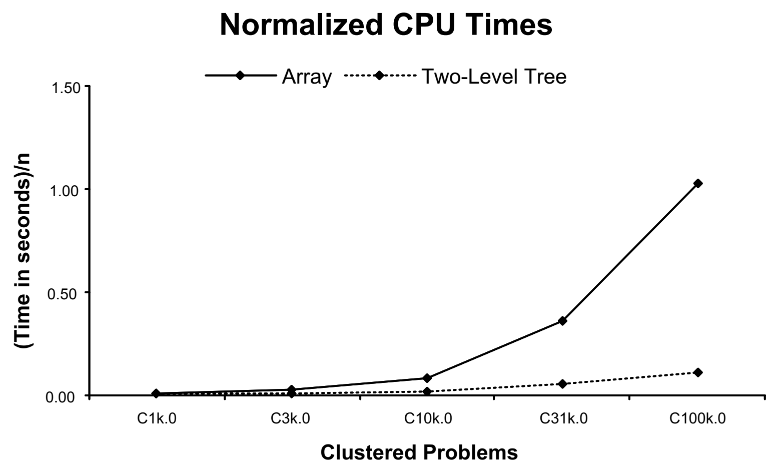
The number of paths to be reversed is usually larger for clustered problems (Fig. 18) than for uniformly distributed problems (Fig. 17). Nevertheless, the average length of the paths is generally superior for uniformly distributed problems. We attribute this fact to the problems structure.

An interesting phenomenon occurs with the last two problems in Fig. 19, where a much larger problem requires a smaller number of paths to be



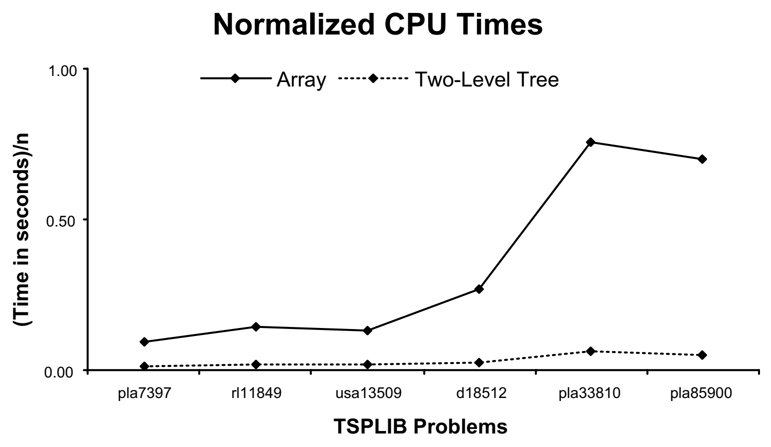
		Paths to reverse		Time/n		Difference	Times(1)
Problem	n	Number	Length	Array(1)	2L Tree(2)	(1)−(2)s	slower than(2)
E1k.0	1,000	197,680	188.86	0.013	0.008	0.005	0.6
E3k.0	3,162	720,348	669.66	0.041	0.012	0.029	2.4
E10k.0	10,000	2,272,103	2141.00	0.124	0.018	0.106	5.9
E31k.0	31,623	8,272,730	6810.98	0.438	0.044	0.394	9.0
E100k.0	100,000	10,993,795	21690.35	0.926	0.055	0.871	15.8
E316k.0	316,228			4.231	0.202	4.029	20.0

Fig. 17. Running times (seconds): uniformly distributed problems.



		Paths to reverse		Time/*n*		Difference	Times (1)
Problem	*n*	Number	Length	Array(1)	2L Tree(2)	(1)−(2)	slower than (2)
C1k.0	1,000	209,703	89.09	0.011	0.008	0.003	0.4
C3k.0	3,162	584,131	394.58	0.027	0.010	0.017	1.7
C10k.0	10,000	2,233,323	1197.17	0.087	0.020	0.067	3.4
C31k.0	31,623	9,224,851	3894.47	0.360	0.059	0.301	5.1
C100k.0	100,000	17,170,440	12393.95	1.032	0.109	0.923	8.5

Fig. 18. Running times (seconds): clustered problems.



		Paths to reverse		Time/*n*		Difference	Times (1)
Problem	*n*	Number	Length	Array(1)	2L Tree(2)	(1) – (2)	slower than (2)
pla7397	7,397	1,919,503	1316.99	0.091	0.015	0.076	5.1
rl11849	11,849	3,298,213	1984.60	0.145	0.020	0.125	6.3
usa13509	13,509	2,308,352	2895.24	0.129	0.019	0.110	5.8
d18512	18,512	4,151,195	4633.68	0.267	0.025	0.242	9.7
pla33810	33,810	14,220,359	7470.94	0.758	0.060	0.698	11.6
pla85900	85,900	13,075,760	19239.09	0.701	0.048	0.653	13.6

Fig. 19. Running times (seconds): TSPLIB problems.

Table 1
Running times (seconds) for Lin–Kernighan: *pla* instances

Problem	Size	Time/*n*		Difference (1) − (2)	Times (1) slower than (2)
		Array (1)	2L Tree(2)		
pla7397	7,397	0.063	0.034	0.029	0.9
pla33810	33,810	0.220	0.052	0.168	3.2
pla85900	85,900	0.352	0.040	0.312	7.8

Table 2
Running times (seconds) for Lin–Kernighan: random problems

Size	Time/*n*		Difference (1) − (2)	Times (1) slower than (2)
	Array (1)	2L Tree(2)		
10^3	2.2	1.9	0.3	0.2
10^4	6.5	2.8	3.7	1.3
10^5	57.3	3.6	53.7	14.9

reversed. As a result we obtain lower normalized computational times, which means that the number of paths to reverse has a dramatic influence in the running times comparing to average paths' length.

An interesting result stems from the observation that the gain in efficiency of the two-level tree implementation over the array implementation for the stem-and-cycle algorithm is significantly superior to the one reported by Fredman et al. [2] for the Lin–Kernighan algorithm, specially for *pla* instances (cf. Tables 1 and 2—since the instances characteristics are very similar, we can use the values reported for the S&C algorithm in Fig. 17, to compare with the results presented in Table 2). This result clearly demonstrates the effectiveness of the implementation reported in this study.

3. Conclusions

The main purpose of this study is the design and development of new data structures seeking to improve the efficiency of the stem-and-cycle algorithm described in Rego [9]. To achieve this goal, the study focused on reducing the number of operations needed to execute path reversals performed during the application of the stem-and-cycle neighborhood structure. This was done by creating a special adaptation of the two-level tree data structure described in Fredman et al. [2] and successfully used in the implementation of the well-known Lin–Kernighan algorithm. The significant differences between the stem-and-cycle and the Lin–Kernighan neighborhood structures required substantial modifications of the operations described in [2], in order to handle the transformations that are critical for applying the stem-and-cycle neighborhood structure.

The results obtained with this new implementation [5] on a testbed provided for the "8th DIMACS Implementation Challenge" [5] clearly demonstrate the efficiency of the new implementation over the original version and, more importantly, establish the stem-and-cycle algorithm as one of the most efficient methods currently available for the TSP [3,6].

References

[1] M. Chrobak, T. Szymacha, A. Krawczyk, A data structure useful for finding Hamiltonian cycles, Theoretical Computer Science 71 (1990) 419–424.

[2] M.L. Fredman, D.S. Johnson, L.A. McGeoch, G. Ostheimer, Data structures for traveling salesman, Journal of Algorithms 18 (1995) 432–479.

[3] D. Gamboa, C. Rego, F. Glover, Implementation analysis of efficient heuristic algorithms for the traveling salesman problem, Computers and Operations Research, in press.

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

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

[6] D.S. Johnson, L.A. McGeoch, Experimental analysis of heuristics for the STSP, in: G. Gutin, A. Punnen (Eds.), The Traveling Salesman Problem and Its Variations, Kluwer Academic Publishers, Dordrecht, 2002, pp. 369–443.

[7] S. Lin, Computer solutions of the traveling salesman problem, Bell System Computer Journal 44 (1965) 2245–2269.

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

[9] C. Rego, Relaxed tours and path ejections for the traveling salesman problem, European Journal of Operational Research 106 (1998) 522–538.

[10] G. Reinelt, TSPLIB—a traveling salesman problem library, ORSA Journal on Computing 3 (1991) 376–384.