Università di Pisa

Dipartimento di Informatica

Technical Report : TR-99-16

SPT_L shortest path algorithms: review, new proposals and some experimental results

M. Nonato, S. Pallottino, B. Xuewen

July 27, 1999

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

SPT_L shortest path algorithms: review, new proposals and some experimental results[1]

Maddalena Nonato
Department of Electronic Engineering and Information, University of Perugia,
Via G. Duranti 93, 06125 Perugia, Italy

Stefano Pallottino
Department of Computer Science, University of Pisa,
Corso Italia 40, 56125 Pisa, Italy

Bao Xuewen[2]
Department of Mathematics, University of Qinghai,
810003 Xining, China

Abstract

This paper presents in a unified framework the most efficient shortest path algorithms of the *List* class to highlight strategies for handling the candidate node set as well as issues related to the efficient implementation of such strategies. A deep analysis of the strategies which inspired two of the most robust algorithms for the shortest path tree problem, due to Tarjan [27] and to Goldberg and Radzik [18], allows the proposal of some variants. A computational study of such variants is conducted on the same test problem families used in [6], in order to validate "a posteriori" the implementation choices. The experimental data suggest that all of the new variants are robust in practice and, for each of the tested problem families, at least one of the new variants either improves the performance or performs as well as both benchmark algorithms.

---

[1] Under grants MURST-Cofinanziato 1997 and CNR-GNIM.

[2] This work was done while the author was visiting the Department of Computer Science, University of Pisa.

1 Introduction

The shortest path problem is an important combinatorial optimisation problem which has been extensively studied in the literature; see [1, 5, 6, 8, 12, 13, 20, 23] for a survey of classical algorithms, and see also [2, 7, 11, 17, 18] for more recent approaches.

Despite the importance of the theoretical worst case computational complexity, efficient and robust shortest path algorithms are required in a considerable number of applications; besides, in most of such applications, as for instance in communication and transportation problems, the size of the network involved is very large (up to millions of nodes and arcs) and most of the running time is spent in iteratively computing shortest paths.

For this reason, over the last 10 years a significant part of the research in this field has been devoted to improve the practical efficiency of shortest path algorithms by conceiving ad hoc data structures, with the purpose of modifying the order according to which nodes and arcs are visited. Almost all of the proposed algorithms have been described as "new approaches", whereas a deep analysis of the literature allows to classify them in terms of "variants" of previously described algorithms.

The aim of this paper is to focus on the *SPT_L* class of shortest path algorithms, less properly known as the *label correcting class*, and to analyse the peculiarities of each algorithm and their expected effect on practical performance in a unifying framework. A further aim is to show how to design new variants of two of the most efficient and robust algorithms [18, 27], by modifying the algorithm node selection strategy and by building the suitable data structures required to ensure a high level of efficiency for each specific variant. As a validation of the algorithmic choices, the proposed variants have been tested on the set of benchmark test problems introduced in [6].

The *C* language codes are freely available by anonymous ftp (at the ftp-site *ftp.di.unipi.it* in folder *pub.project.orgroup.software.spt_codes*).

2 The shortest path tree problem and an algorithmic scheme

Let $G = (N, A)$ be a directed graph with $n = |N|$ nodes and $m = |A|$ arcs; with each arc $(i,j) \in A$, a *cost* c_{ij} is associated, which may be either positive, zero or negative.

In the following we denote by $FS(u)$ the *forward star* of node u, i.e. the set of outgoing arcs incident to u:

$$FS(u) = \{(i,j) \in A: i = u\}.$$

Let $r \in N$ be the *root*; assume that each node in G is reachable from r and that G contains no directed cycles of negative cost. The *shortest path tree* problem consists of finding a spanning directed tree T^* rooted at r such that, for each node $v \in N$, the unique path from r to v in T^* is a minimum cost path in G.

Almost all shortest path algorithms, at least those within a primal framework, start from a fictitious tree T and iteratively update it until a minimum cost path tree T^* is found. For this purpose, a *label* d_v and a *predecessor function* p_v are associated to each node $v \in N$, thus providing an upper bound to the cost of a minimum cost path from r to v, and an implicit description of T, respectively. Moreover, most of the algorithms handle a *candidate node set* Q and, when a node, say u, is selected and removed from Q, the node is *scanned*, i.e. the whole forward star $FS(u)$ is analysed to check the *Bellman's conditions*:

$$d_u + c_{uv} \geq d_v, \quad \text{for each } (u,v) \in FS(u). \tag{1}$$

Since the *reduced cost* of an arc $(u,v) \in A$, with respect to the label array d, is

$$\bar{c}_{uv} = c_{uv} + d_u - d_v,$$

there is an equivalence between the reduced costs non negativity and Bellman's conditions holding.

In case of violation [$\bar{c}_{uv} < 0$], the label of node v is properly improved [$d_v := d_u + c_{uv}$] and the tree is modified by replacing arc (p_v,v) with arc (u,v) [$p_v := u$]. Let $T_v = (N_v, A_v)$ be the subtree of the current tree T rooted in v; if all nodes in N_v are consequently updated and decreased by $-\bar{c}_{uv}$, as in the primal network simplex, then all arcs in T always verify Bellman's conditions as equalities and dual variables exactly provide the cost of the unique path from r to v on T [19]. In this case labels are said to be *exact*. More efficient approaches do not propagate label updating on the subtrees, but instead keep a record by means of Q of the updated nodes for a later check of Bellman's conditions on their forward star. Here, a node label provides just an upper bound of the path cost from r to the node on T. Since scanning a node is a wasted effort if its label is not exact, most algorithms implement a strategy to reduce the number of wasted scanning.

A scheme of the typical iteration is shown in figure 1, see [13]. In procedure *Qout*(u,Q) a "valid" node is selected and removed from Q; this means that, in general, nodes can be discarded after selection if they do not qualify for inspection, for example if their label is not exact. In procedure *Tupdate*(u,v), the tree T and the node labels are properly updated, and in procedure *Qin*(v,Q) node v is inserted into Q if not already there and Q is consequently adjusted (possibly, by inserting and/or removing other nodes).

```
Qout(u,Q);
foreach (u,v) ∈ FS(u) do
  if c̄uv < 0 then
    begin
      Tupdate(u,v);
      Qin(v,Q)
    end
```

Figure 1 - A general scheme of an iteration

The initial tree T is a fictitious one (all nodes but r have the root as predecessor and a $+\infty$ label value, while $d_r = 0$) and Q contains only r. At each step, the sum of the node labels is non increasing. The algorithm repeats the above described iteration until $Q = \emptyset$; when Bellman's conditions (1) hold for all the arcs, the node labels represent the minimum path costs and the predecessors describe a shortest path tree T^*.

3 A literature review

The algorithms here considered mainly differ on the way of handling Q, i.e. on the selection policy of the next candidate node to be analysed; moreover, in procedure *Tupdate*(u,v), additional node labels can be updated beside the one of the current node, for example by propagating the update over the subtree; furthermore, even the current tree structure may be modified.

Some formal definitions are necessary in order to describe the algorithms' behaviour:

- d_v, the *label* of node v, is an upper bound to the cost of the current path from r to v in T, and it corresponds to the cost of the path from r to v on one of the trees generated so far;
- if d_v is the cost of the path in the current tree T, then it is said to be an *exact label*, otherwise it is said to be *inexact*;
- when $d_v = d_v^*$, i.e. its value is the minimum one, then d_v is said to be *permanent*;

- the tree T described by the predecessor function is said to be the *current tree*, and when $T = T^*$ it is *optimal*;
- a node, say v, never inserted into Q (i.e., $d_v = +\infty$) is said to be *unreached*, while it is said to be *candidate* while it belongs to Q;
- a node selected and removed from Q whose forward star has been examined is said to be *scanned* until it remains out of Q;
- a node selected and removed from Q but not eligible for arc examination is said to be *discarded*.

Let us consider the relation between the current tree T and the label array d. Some considerations are easy to prove:
- if a node $v \in N$ has an exact label, then all the arcs of the path in T from r to v have zero reduced cost; conversely, if the label is inexact, at least one arc of such a path has negative reduced cost;
- as a consequence, if a node v has an inexact label, then at least one of its ancestors in T belongs to Q;
- no arc with positive reduced cost belongs to T;
- scanned nodes with permanent label will not be inserted again into Q;
- all labels which are updated due to the scanning of an exact label node are exact with respect of the current tree; conversely, if the scanned node has inexact label then it yields inexact labels.

In the following, we will denote by *admissible graph* with respect to a given distance label vector d, the partial graph G_A of G containing all the arcs with non positive reduced costs, and by *strictly admissible graph* the partial graph G_{SA} of G_A containing all the arcs with negative reduced costs. A sufficient condition for a node v not to be permanent is the existence of a path from r to v in G_A with at least one arc in G_{SA}.

3.1 SPT_S Algorithms

Dijkstra's Theorem [9], in case of non negative arc costs, is based on the selection from Q of the minimum label node at each iteration:

$$u = \text{argmin}\{d_v: v \in Q\}.$$

In such a case, it is easy to prove that every node u, when selected and removed from Q, has a permanent label (i.e., $d_u = d_u^*$) and then will never re-enter Q; consequently, the number of iterations is limited by n.

Algorithms based on Dijkstra's result, called *SPT_S* to emphasise that a “shortest path” node is selected at each iteration, differ on the particular implementation of Q, which is a *priority queue* with distance label as priority key.

Several surveys have been devoted to shortest paths algorithms of the *SPT_S* class; we address the interested reader to [1, 2, 6, 7, 8, 13, 20, 23] while in this paper we will not analyse any further this class of algorithms.

3.2 SPT_L algorithms

The other class of algorithms, denoted by *SPT_L*, is obtained when Q is implemented by means of *lists*, and the only operations allowed are the insertion at the two ends of a list, the removal form the head of a list, and the scanning of a list for internal removals. Other operations can be viewed as a combination of the previous ones (e.g., moving one or more nodes from one list to another can be realised by scanning the first list and by iteratively moving nodes from this one to the other). In case of multiple lists, a

priority is given for the purpose of node selection, so that selection is always performed at the head of the preferred list *regardless of the value of the node's label*. It is always possible to redefine which list is the currently preferred one.

Research on this family of algorithms has been devoted to conceive a variety of strategies which realise a node selection order, with the purpose of discarding or postponing the exam of nodes whose scanning is considered not useful (or not performing), of avoiding to scan nodes with inexact label, and of increasing the probability of selecting permanent label nodes. The performance of all these algorithms is closely related both to the effectiveness of the order perturbation strategies and to the computational cost of the operations involved, so that what is gained by the reordering is not lost in efficiency. In fact, any such policy involves additional computation and requires specific data structures, so that a trade-off between efficacy and efficiency must be sought in order to obtain a good overall performance.

Hereafter follows a list of the main algorithm families within the *SPT_L* class, highlighting their rationales and how their target is achieved by way of particular data structures.

A single list implemented as a queue

The review starts from the classical Bellman's, Ford's and Moore's algorithm [3, 10, 24], called S*PT_L_queue*, where Q is implemented as a *queue* so that insertion occurs at the tail of Q while removal occurs at the head of Q, by realising a FIFO policy on the selection order.

The whole sequence of scanning operations performed during the algorithm can be partitioned into a sequence of passes, where the first pass consists of scanning the root r, while pass k consists of processing the nodes which have been added to Q during pass $k-1$. Therefore, a node can be scanned at most once for each pass, and at most n passes are necessary; so, a single pass costs $O(m)$ and the overall complexity of *SPT_L_queue* is $O(mn)$. After 40 years, this algorithm still provides the best complexity for shortest path problems on graphs with arbitrary cost arcs.

Two lists connected in series

Algorithms in this family try to anticipate, when compared to a plain FIFO policy, the updating of inexact labels of the nodes currently in Q before they are actually scanned. Lists with two insertion points have been proposed for this purpose in [22, 25, 26]. In these algorithms, Q can be viewed as two lists Q' and Q'' where selection is made from the head of Q' (from the head of Q'' whether $Q' = \emptyset$) while insertion can be made either in Q' or in Q''. Since inexact labels may arise only once a node has been inserted into Q more than once, in algorithm S*PT_L_deque* (proposed by Pape [26] and independently by Levit and Livshits [22]) candidate nodes are inserted the first time at the tail of Q'', while later insertions take place at the head of Q'. Therefore, in S*PT_L_deque*, Q' is used as a *stack* while Q'' is used as a *queue* and contains each node at most once. The stack nature of Q' is the cause of the exponential time complexity, $O(n2^n)$ [21]; in fact, it is possible that a node is selected and scanned 2^{n-2} times in the same pass.

The variant proposed in [25] is based on the implementation of Q' as a *queue*, where nodes are re-inserted at the tail of Q'. The resulting algorithm, called S*PT_L_2queue,* results to be polynomial, $O(mn^2)$, since a node can be selected and scanned no more than n times in the same pass.

A double insertion policy, that is a *deque* as a data structure for Q, is adopted also in [4], but it is based on a different rationale; higher selection priority is given to nodes with lower label for the purpose of increasing the probability of selecting permanent label nodes. There, a node is inserted into Q'' if its label is higher than a given threshold value σ, while it is inserted at the head of Q' if not. In the following

we will refer to this algorithm family as *SPT_L_deque_on_values* (*SPT_L_dov*). A variant of this algorithm has been proposed, in which, during the selection, a node whose label is greater than σ is moved to the tail of Q'' to postpone the scanning of its forward star. The value of σ depends on the values of the labels of the nodes currently in Q; two different threshold values have been proposed in the same paper: the average label, and the label of the first node of Q.

Two lists used as dynamic buckets

Other algorithms based on a threshold value, which have been proposed before [4], are described in [14, 15, 16] with the same aim of increasing the probability of selecting permanent label nodes. In this family, called *SPT_L_threshold*, Q' contains all the nodes whose label is less than or equal to a threshold σ and node selection takes place at the head of Q'. A node belonging to Q'', whose label is decreased to a value not greater than σ, is virtually moved from Q'' to Q'; practically, the node is inserted into Q' and its removal from Q'' is *postponed* in time. When $Q' = \emptyset$, σ is properly increased (at least to the minimum label value of the nodes in Q'') and Q'' is scanned to perform the postponed removals and to move to Q' the nodes whose label does not exceed the new threshold. Note that, in case of non negative arc costs, if

$$\sigma \leq \min\{d_u: u \in Q\} + \min\{c_{ij}: (i,j) \in A\},$$

the algorithm reduces to a *SPT_S* algorithm; on the contrary, a sufficiently large initial value of σ (i.e., $\sigma > (n\text{-}1)\max\{c_{ij}: (i,j) \in A\}$) forces the insertion of all the nodes directly to Q' and, if implemented as a *queue*, the algorithm reduces to *SPT_L_queue* algorithm.

Single queue with removals

In the algorithm proposed by Tarjan [27], called in the following *SPT_L_disassembling_queue* (*SPT_L_dq*), only nodes with exact label are scanned. This condition is enforced by discarding from Q all candidate nodes with inexact label as soon as they arise; these are collected by traversing the subtree T_v right after d_v has been updated, according to the so-called *subtree disassembly* strategy. Such strategy is implemented by describing the tree structure, beside the usual predecessor function, by means of the *first son* and *adjacent brothers* functions (next and previous); this allows the traversal of a subtree T_v with a complexity linear in its size, and the tree modification in constant time. At each iteration, when a node u is selected and $FS(u)$ is scanned, if $\bar{c}_{uv} < 0$, v is updated and all its descendant labels become inexact; then T_v is traversed, every node but v is marked as *discarded* and its removal from Q is postponed in time. At the end of the process, v is inserted at the tail of Q, which is realised as a *queue*, if not already in Q. Ideally, the subtree T_v is removed from T and v is inserted in T as a leaf. Therefore, the current tree T contains only nodes with exact label.

As for *SPT_L_queue*, in *SPT_L_dq* a node can be scanned at most once for each pass and at most n passes are necessary; so, the overall complexity of *SPT_L_dq* is $O(mn)$ (see [27] and [5]).

```
Procedure Qout(u,Q);
  begin
    while Q ≠ ∅ and u = first(Q) is marked discarded do remove u from Q
  end;
```

Figure 2. Procedure *Qout*(u,Q) in algorithm *SPT_L_dq*

```
Procedure Qin(v,Q);
  begin
    traverse T_v and mark discarded all nodes ≠ v;
    if v ∉ Q then insert v at the tail of Q
  end;
```

Figure 3. Procedure *Qin*(v,Q) in algorithm *SPT_L_dq*

Lists for a topological scanning

A different condition which also remarkably reduces the number of wasted scanning, is implemented by a strategy based on the *topological ordering* of candidate nodes, which has been proposed by Goldberg and Radzik in [18] and widely tested in [5, 6].

Let us introduce the following definition: a node v *improvable* by a node w if v is connected to w in the admissible graph G_A by a path containing at least one arc of G_{SA}. We say that, given the current set of labels, a node order is an *improvable* or a *topological order* if every node always follows all other nodes by which is improvable. Note that the label of an improvable node is necessarily not permanent.

The strategy consists of altering the order of candidate nodes to realise a topological order, by implicitly visiting the admissible graph G_A; note that G_A is acyclic regardless of possible *return* arcs along zero reduced cost cycles. For efficiency purposes, in [18] it is proposed to reorder the nodes belonging to Q only at the beginning of each pass.

To realise this strategy, Q is implemented by means of two lists Q' and Q'' with selection at the head of Q' and insertion in Q''; when $Q' = \emptyset$, Q'' is scanned to topologically reorder the nodes and move them to Q'. To do that, for each selected node v from Q'' not already moved to Q', a depth-first search (*dfs*) on G_A is performed to collect all the nodes connected to v in G_A and not inserted in Q', and to order them according to the order of the *dfs* visit. To maintain the whole topological order, such sequence of nodes, which is itself topologically ordered, is inserted at the head of Q'. Therefore, Q' is a *sublist-stack* since the stack order holds among the various inserted sublists, while the inner order of each sublist is the *dfs* ordering of the visit of G_A. Visiting G_A implies the scan of the whole forward star of each visited node in order to retrieve the arcs with non positive reduce cost.

At the beginning, $Q' = \emptyset$ and $Q'' = \{r\}$ to perform the first topological ordering.

Note that the topological order may not hold any longer during the computation in the pass due to labels updating, and, as a consequence, nodes with inexact label can be selected and scanned. For example, consider four nodes u, v, w and z in that order in Q' such that, when reordering takes place, arcs (u,v) and $(u,w) \in G_{SA}$ while arcs (w,v) and $(v,z) \notin G_A$. Now suppose that the scanning of u updates the labels of v and w in such a way that, with respect of the new labels, arcs (w,v) and $(v,z) \in G_{SA}$; the next node selected, i.e. v, improves the label of z which becomes its son. Then, the scanning of w updates the label of v, which is inserted in Q'' without any updating on z which is then selected and scanned even if its label is inexact.

In [18] two algorithms are proposed:

- *SPT_L_reorder* (*SPT_L_r*), in which *dfs* is performed by examining the whole forward star of the visited nodes to collect all improvable nodes from each candidate node, and order them in topological order. After the complete reordering of Q', the nodes are selected and removed from the head of Q' and scanned in order to actually update the labels (see Figures 4 and 5 for a more detailed description).
- *SPT_L_reorder&scan* (*SPT_L_r&s*), in which, during the *dfs*, labels are immediately updated to enlarge the portion of the visited graph, since G_A and G_{SA} vary during the visit according to the current d, and the improvable node set varies accordingly. Thus, the visit is performed on the *current* strictly admissible graph G_{SA}, that is the one yielded by each label update. Figure 6 sketches the scheme of this operation.

  The label updating can be viewed as an anticipation of the node scanning when the nodes are actually selected from the head of Q'; not to waste time, a *mark* indicates whether a node label has been further

improved after the reordering phase. Then, nodes selected at the head of Q' will be scanned only if marked as improved (see Figure 7).

```
Procedure Qout(u,Q);
    begin
      if Q' ≠ ∅ then u := first(Q')
        else while Q'' ≠ Ø do
          begin
            select&remove v = first(Q'');
            if v ∉ Q' then
              begin
                first := i := v; insert v into Q' before first(Q');        {start the topological visit}
                repeat
                  foreach (i,j) ∈ FS(i) do
                    if d_i + c_ij ≤ d_j and j ∉ Q' then insert j into Q' after i;      {depth-first search}
                  i := next(i,Q')
                until i = first(Q');                                       {the topological visit is completed}
                first(Q') := first
              end
          end
    end;
```

Figure 4. Procedure *Qout*(u,Q) in algorithm *SPT_L_r*

```
Procedure Qin(v,Q);
    begin
      if v ∉ Q'' then insert v at the head of Q''
    end;
```

Figure 5. Procedure *Qin*(v,Q) in algorithm *SPT_L_r*

```
Procedure Qout(u,Q);
    begin
      while Q' ≠ ∅ and u = first(Q') is not marked improved do remove u from Q';
      if Q' = ∅ then while Q'' ≠ Ø do
        begin
          select&remove v = first(Q'');
          if v ∉ Q' then
            begin
              first := i := v; insert v into Q' before first(Q');          {start the topological visit}
              repeat
                foreach (i,j) ∈ FS(i) do
                  if d_i + c_ij < d_j then
                    begin
                      d_j := d_i + c_ij; p_j := i;
                      if j ∈ Q'
                        then mark j as improved
                        else insert j into Q' after i                      {depth-first search}
                    end;
                i := next(i,Q')
              until i = first(Q');                                         {the topological visit is completed}
              first(Q') := first
            end
        end
    end;
```

Figure 6. Procedure *Qout*(u,Q) in algorithm *SPT_L_r&s*

```
Procedure Qin(v,Q);
    begin
      if v ∉ Q'' then if v ∉ Q' then insert v at the head of Q''
                                else mark v as improved
    end;
```

Figure 7. Procedure *Qin*(*v*,*Q*) in algorithm *SPT_L_r&s*

Note that, in algorithm *SPT_L_r&s*, because of the label updating as discussed before, the candidate node set right after the reordering and scanning phase may not be topologically ordered with respect to the current G_A. Moreover, such order does not even hold on the subset of the improved nodes reached in *dfs* starting from the same node. Consider the following example: let us suppose that, starting the *dfs* visit of the current G_{SA} from node u, the adjacent nodes v, w and z are visited, updated, scanned in that order and inserted in Q'. Since the two arcs (z,v) and (z,w) have negative reduced cost with respect to current labels, the labels of both v and w are properly updated, without further scanning since both have been already inserted in the sequence of visited nodes. Let us suppose that there exist arc (v,w) whose reduced cost, once positive when scanning v, has become strictly negative relative to the updated labels: the inner order is (u, z, w, v); then, w and v are marked improved and they are not in topological order with respect to the current label values.

Both algorithms are proved of time complexity $O(mn)$ [6, 18]; it is also proved that if G is acyclic, the topological ordering performed by *SPT_L_r&s* in the first iteration is a topological order also for G and, so, the time complexity is linear on the number of arcs (see [28]).

Other *SPT_L* algorithms can be found in the literature, and almost any one can be classified as a member of one of the above mentioned families. For sake of brevity we prefer to omit them, due to the minor impact from the practical point of view as well as for their computational complexity (such as *SPT_L_stack*).

4 New algorithmic variants

In this section, *SPT_L_disassembling* and *SPT_L_reorder* families are further investigated with the aim of producing new variants which overcome some drawbacks of the originally proposed algorithm. We show how to design new variants of *SPT_L* algorithms by introducing “ad hoc” strategies in order to achieve particular targets in the node selection, and how to conceive tailored data structures for the efficient implementation of Q.

4.1 A “reassembling” algorithm

Algorithm *SPT_L_dq* in practice performs less scan operations than other *SPT_L* algorithms, although the updating of the current tree is time-consuming. In fact, a node may be inserted and removed from Q more than once in between successive scanning operations, since it may turn from discarded into candidate and vice versa several times. Such “disassembling and reassembling” operations cost some cpu time. Moreover, disassembling subtrees may mean discarding some useful information about the tree structure that the algorithm has collected. These observations lead to the following variants of the algorithm.

The rationale behind the proposed variant, which we call *SPT_L_reassembling_queue* (*SPT_L_rq*), is to exploit the same tree structure used in *SPT_L_dq* to propagate, as in the simplex approach, the label updating of a node over its descendants in order to keep their labels exact but, at the same time, to limit the number of updating that can occur to a node during the same pass.

For this purpose, Q is implemented by a *swapping list* (Q', Q''), that is, Q' is the list from where removals are performed, insertions are made only in Q'', and when $Q' = \emptyset$, by simply *swapping* the list names in $O(1)$ time, Q'' becomes Q', and vice versa. Initially, $Q' = \{r\}$ and $Q'' = \emptyset$. Since a pass consists of the entire scanning of Q, thus every node can be selected at most twice within a single pass.

Let us consider now the iteration in which, after the selection of node u from Q', Bellman's condition (1) does not hold for arc (u,v), i.e. $\bar{c}_{uv} < 0$; if v belongs to Q, its label is properly improved and no other operation is performed on its (possible) subtree. Conversely, if $v \notin Q$, the tree data structure is used to partially traverse the subtree T_v and to properly update the node labels. The visit of T_v is only *partial* since, when a descendant of v, say i, belonging to Q'' is visited, its label is properly updated but the visit on its subtree in not further pursued; on the contrary, when $i \notin Q''$, the visit will continue on its sons, when present. Note that, in the latter case, it may happen that $i \in Q'$; in such a case it is ideally removed from Q' *(postponed removal)*. At the end of the partial visit of T_v, the sequence of the "reassembled" nodes, i.e. v and all its descendants not belonging to Q'', is inserted at the beginning of the sublist-stack Q'' in the *tree order*, i.e. any node follows in the order its father, to maintain the tree order with the descendants of v already belonging to Q''.

Note that, the set of nodes improvable by v in the current G_A includes the set of reassembled nodes, since such nodes are connected to v in the current tree T, which is indeed a subgraph of G_A. The efficiency of subtree traversing allows to reset the tree order after each label update of a node not belonging to Q; moreover, the sublist-stack structure of Q'' ensures that the tree order holds at any iteration.

It is easy to see that the tree ordered node insertion policy guarantees that nodes, when selected and scanned, have exact labels. It is also easy to prove that the partial traversal of subtrees ensures that the number of labels updating for every node cannot exceed the number of node examinations per pass, i.e. $2n$, and the number of passes cannot exceed n. Then, the time complexity of *SPT_L_rq* is $O(mn)$ since the number of arc examination per pass is bounded by $2m$ and the reassembling operations in the same pass cost globally $O(n)$ since every node can be reassembled at most once.

Let us provide some details about an efficient implementation of *SPT_L_rq*, as described in Figures 8 and 9. T is implemented by the same data structure as in *SPT_L_dq*, that is the predecessor, first-son, next and previous brother functions, which allow the efficient subtree traversal. The partial subtree traversal of T_v is performed as a breadth-first search (*bfs*) and the tree order sequence is built-up directly into Q''; at the end of the visit, the last entered node will point to the first node of Q'' and v will be the new first node of Q'' (*concatenation operation*). For sake of efficiency, it is computationally profitable to record the cumulative label improvement of a node which has not been already propagated to its sons, in order to update the son labels when the node is dragged from Q' to Q'' to pursue the visit of a subtree.

```
Procedure Qout(u,Q);
    begin
      while Q' ≠ ∅ and u = first(Q') ∈ Q'' do remove u from Q';
      if Q' = ∅ then begin Q' ↔ Q''; Qout(u,Q) end
    end;
```

Figure 8. Procedure *Qout*(u,Q) in algorithm *SPT_L_rq*

```
Procedure Qin(v,Q);
    begin
      if v ∉ Q then
        begin
          first := last := i := v; insert v into Q″;          {start of the partial tree visit}
          repeat
            select j next of i in the partial tree visit;          {breadth-first search}
            update d_j so that it is exact;
            if j ∉ Q″ then begin insert j into Q″ after last; last := j end;
            i := next(i,Q″)
          until i = nil;                                  {the partial tree visit is completed}
          next(last) := first(Q″); first(Q″) := first      {concatenate first(Q″) as following of last}
        end
    end;
```

Figure 9. Procedure *Qin*(*v*,*Q*) in algorithm *SPT_L_rq*

4.2 An "alternating_reassembling" variant

A possible improvement of *SPT_L_rq* comes from a trade-off between efficiency and efficacy. In *SPT_L_rq*, it is possible that a node is moved from Q' to Q'' twice in the same pass. In fact, a first reassembling operation can force, in the first half of the pass, that a node $j \in Q'$ is traversed and inserted into Q''; in the second half, i.e. after the swapping between Q' and Q'', it may happen that a new reassembling operation force again j to move to Q'' without any scan of its forward star in the whole pass.

To avoid that, the two subpasses can be implemented in an asymmetric form, in such a way that a node which has been moved from Q' to Q'' in the first half, in the second half will not be traversed during the reassembling operation and left into Q' (as it occurs to nodes in Q''). The so obtained variant, called *SPT_L_alternating_reassembling_queue* (*SPT_L_arq*), reduces drastically the number of possible removals from Q' with the consequence that the examined nodes can have an inexact label. In the section devoted to experimentation, we show that, for a large variety of types of graphs, the above proposed partial label correction and partial reassembling can be effective.

4.3 A dynamic topological ordering variant

The main feature of *SPT_L_r* is to implicitly scan G_A starting from the nodes in Q'' at the beginning of each pass, with the purpose of topologically ordering the improvable node set with respect to the labels given at the beginning of the reordering phase. However, after any following label update, such order may not hold any longer. Indeed, the label updating performed in *SPT_L_r&s* during the *dfs* visit, as previously described, may produce a sequence of nodes not necessarily ordered with respect to the current labels. However, in both algorithms, once Q'' has been emptied, no further use of the current G_A is done to re-enforce the ordering until the end of the pass, i.e., when $Q' = \emptyset$.

Our proposal is that, once an improvable node has been discovered, it could be effective to scan the involved portion of the *current* G_{SA} right away, instead of postponing such scan at the end of the pass. In the variant that we propose here, that we call *SPT_L_dynamic_reorder&scan* (*SPT_L_dr&s*), every time the label of a node $v \notin Q$ is improved, a partial visit of the current G_{SA} is performed by updating the node labels along the visit, as adopted in *SPT_L_r&s*. As shown in Figure 11, and like what is done in *SPT_L_rq* and in *SPT_L_arq*, the visit is only *partial* since it is not pursued through nodes already belonging to Q. At the end of the current strictly admissible graph search, the sequence of nodes improvable by v and not belonging to Q is inserted at the beginning of the sublist-stack Q''.

```
Procedure Qout(u,Q);
  begin
    while Q′ ≠ ∅ and u = first(Q′) is not marked improved do remove u from Q′;
    if Q′ = ∅ then begin Q′ ↔ Q″; Qout(u,Q) end
  end;
```

Figure 10. Procedure *Qout*(*u*,*Q*) in algorithm *SPT_L_dr&s*

```
Procedure Qin(v,Q);
  begin
    if v ∈ Q then mark v as improved
      else
        begin
          first := i := v; insert v into Q″ before first(Q″);        {start the topological visit}
          repeat
            foreach (i,j) ∈ FS(i) do
              if d_i + c_ij < d_j then
                begin
                  d_j := d_i + c_ij; p_j := i;
                  if j ∈ Q then mark j as improved
                          else insert j into Q″ after i              {depth-first search}
                end;
            i := next(i,Q″)
          until i = first(Q″);                                       {the topological visit is completed}
          first(Q″) := first
        end
  end;
```

Figure 11. Procedure *Qin*(*v*,*Q*) in algorithm *SPT_L_dr&s*

Due to the partial visit, the nodes belonging to Q' which have been visited during the search, are not moved to Q'', and their labels may be inexact at selection time. In fact, consider a node j in Q' whose label is updated during a partial visit of G_{SA} starting from an improvable node, say v; it is possible that a selected node, say u, improves the label of v but, since v has been inserted into Q'', this improvement is not propagated to j creating an inexact label node in Q'.

The scanning of G_{SA} and the use of the sublist-stack Q'' are similar to what described for *SPT_L_r&s*. Here we use the swapping list $Q = (Q', Q'')$ and, in the selection at the head of Q', nodes not marked as improved are discarded similarly to what described for *SPT_L_r&s*.

It is easy to prove that the partial visit of the current G_{SA} ensures that the computational complexity is still O(*mn*), as stated in the following Theorem.

Theorem 1: *SPT_L_dr&s runs in O(mn) time, and in O*(*m*) *time on acyclic graphs.*

Proof. SPT_L_dr&s ends in at most n passes since at least one node per pass has a permanent label. In each pass, a node is selected and scanned at most twice; moreover, the partial visits of the current strictly admissible graphs in the same pass may involve the same node, and its forward star, at most twice, so the complexity of one pass is $O(m)$ time. Then, the total time for *SPT_L_dr&s* is $O(mn)$.

For acyclic graphs, after the root r is scanned for the first time, all nodes of G except r have been added to Q'' in a topological order. The algorithm ends in the first pass. Therefore each node is scanned at most twice. The proof is completed.

4.4 A single list dynamic topological ordering variant

In the second variant here proposed, namely *SPT_L_list_dynamic_reorder&scan* (S*PT_L_ldr&s*), the dynamic reordering and scanning approach is maintained while, to ensure a “global” topological

ordering, any ordered sequence of the improvable nodes not belonging to Q are inserted at the head of Q itself. As a consequence, only one list, more precisely a sublist-stack, is required to implement Q.

The description of this algorithm is quite simple: every time a label of a node, say v, not belonging to Q is improved, a *dfs* of the current G_{SA} is performed starting from v to update the labels of its currently improvable nodes,, and sort them, before analysing any other candidate node in Q. The label updating is applied to the nodes belonging to Q without further continuing the *dfs* from them (*postponed scanning* by way of the mark improved).

As far as the exact labelling is concerned, also for S*PT_L_ldr&s* it might happen that a selected node has inexact label. In fact, consider two nodes j and i with j preceding i in Q and $\overline{c}_{ij} > 0$; consider that a third selected node improves their labels in such a way that the new reduced cost of arc (i,j) be negative. j will have inexact label at selection time.

No further detail is necessary to implement S*PT_L_ldr&s*.

Regarding the computational complexity, we argue that the sublist-stack structure of Q may allow pathological effects as the double-ended-queue and the stack (see [21]); so, we guess that its complexity is exponential.

On the contrary, from Theorem 1 follows that, when the graph is acyclic, S*PT_L_ldr&s* runs in $O(m)$ time.

5 Experimental results

5.1 Practical performance of SPT_L algorithms

A significant amount of experimental testing has been produced in the last 15 years on *SPT_L* algorithms, see for instance [4, 5, 6, 13, 14, 16, 20, 23], so that a certain number of facts are well known. Here we briefly sketch the main results.

All *SPT_L* algorithms behave quite well on sparse graphs, especially when the arc costs have a close relation to the topology of the graph, i.e. when they represent lengths, travel times, or travel costs of physical networks (communication, distribution, transport, hydraulic, power networks, etc.).

The basic algorithm *SPT_L_queue* results to be the worst on average since it performs quite a large number of inexact label scanning operations; algorithm *SPT_L_dq*, seldom tested, appears to be of poor performance on some classes of graphs, while performs very well on others. The reason is due to the high cost of subtrees disassembling, which is not always compensated by the gain of discarding inexact label nodes. In [5], this algorithm shows a very good behaviour in detecting negative cycles. For this reason we argue that it is possible to improve its efficiency by studying and producing new variants.

A better behaviour is achieved by *SPT_L_deque*, *SPT_L_2queue* and *SPT_L_dov* whose performance is quite close; for some classes of graphs, *SPT_L_dov* appears of low performance, since its postponed selection policy does not exclude the scan of inexact labels.

The best performance on average is obtained by the *SPT_L_threshold* and *SPT_L_reorder* families; they appear to be quite robust on the whole set of classes of graphs even if for some instances they are dominated by *SPT_L_deque* and *SPT_L_2queue*. The major drawback of *SPT_L_threshold* is the tuning of the threshold updating formula; in fact, a general purpose formula does not guarantee a good performance for all classes of graphs, while it is possible to devise particular formulas for given classes of graphs, which allow to achieve the best performance among all *SPT_L* algorithms.

5.2 The experimental framework

Such a well established knowledge, concerning the relative efficiency of *SPT_L* algorithms on different classes of graphs, has been gained by the various researchers by way of extensive computational experiments on different classes of test graphs (we refer the interested reader to [5, 6, 23] for the most recent and complete computational studies). However, most papers devoted to the experimental tests mainly comment on the relative score of the tested algorithms, and not much about the different behaviour of the same algorithm on different classes of graphs. Moreover, only a subset of papers link the analysis of the graph characteristics with their influence on the performance of the algorithms.

This paper does not intend to be an experimental work but rather to discuss the "design" of new variants of shortest path algorithms. Therefore, each variant is first validated "a priori" by a theoretical argument, in which the benchmark algorithm node selection strategy and the related data structures are analysed, together with the experimental behaviour, in order to highlight any possible pitfall. Then alternative strategies and the corresponding data structures are proposed and discussed, with the aim of mending such pitfalls. Moreover, an "a posteriori" experimental validation of the new variants is conducted in order to confirm, by way of numerical evidence, the conjectures previously derived.

Therefore, this section plays the role of such "a posteriori" validation. For this purpose, we use *SPT_L_dq*, *SPT_L_r* and *SPT_L_r&s* as a benchmark, and test the variants only against such algorithms. The codes of the benchmark algorithms have been downloaded from Goldberg's web site (the authors are grateful for this opportunity), while the other codes have been implemented according to a uniform style, and so that they all fit into the testing environment established in [6].

Even if the classes of graphs used in [6] are not completely convincing (as an example the grid family is rather a family of one way layered networks) we have adopted them in the testing; therefore, we provide the least details on this issue and address to [6] for a broader description.

The experiments were conducted on a Pentium II machine with a 233 MHZ processor running LINUX (Red Hat 5.2, kernel version 2.0.34) with 128 MB memory. All codes are written in *C* language and compiled with the *gcc* compiler with the O3 option (version 2.7.2.3).

Each table provides the running time in cpu seconds (in bold), excluding input and output, and the average number of scan operations per node (below). Each entry is the average of five runs of the code on problem instances produced with the same generator parameters, except for the pseudo random generator seed.

5.2.1 Results on SPGRID problems

A so called grid network is a sequence of x rings, each of them being a double connected cycle of y nodes and $2y$ arcs. The connection between rings is given by y disjoint paths, made of x nodes and x-1 arcs each, oriented from the first to the last ring. A dummy node, used as root, precedes the first ring and has y outgoing arcs, each one entering one node of the first ring. Then a grid network is formed by xy+1 nodes and $3xy$ arcs.

Square grids with x=y spanning from 2^7 to 2^{10} have been used. Wide grids are obtained for a small amount of rings (x=16 and y spanning from 2^{10} to 2^{15}) while long grids are formed by several small size rings (x spanning from 2^{10} to 2^{15} and y=16). The arc costs are generated at random uniformly in the interval $[0, 10^4]$.

A special class of square grids (square_S) is obtained by adding an additional node which is used as the root instead of the dummy node, and is connected by an arc to every other node, included the dummy one. The cost of the arc from the root to the dummy node is set to 0 while all the other

additional arcs have a very large cost. Note that, as discussed in the following, such topology forces the insertion of all the nodes in Q after the first iteration.

The Grid-Phard family and the Grid-Nhard family are harder problems with a more complex layer structure and special values of arc costs.

Regarding the results obtained on square grids (see Table 1), as expected, the reorder&scan policy results to be unsuitable, since the layered structure of the network is practically forgotten by the initial insertion of all the nodes in Q. It is worth noting that the dynamic variants smooth the pathological effect, especially *SPT_L_ldr&s* whose performance is comparable to *SPT_L_r* concerning both time and number of scans. This is due to the fact that the sublist-stack structure of Q tends to postpone the scan of the nodes belonging to furthest layers.

Reassembling appears slightly effective than disassembling, since it reduces the number of scans. Globally, *SPT_L_rq* achieves the same time as *SPT_L_r* even though with a lower number of scans, since the reassembling strategy is quite effective but the tree data structure requires some additional computing time.

A similar behaviour is obtained on long grids, see Table 2, with a less emphasis on the gap between reassembling and disassembling. It is worth noting that the average number of scans increases with the number of layers for *SPT_L_r&s* and *SPT_L_dr&s*, while it remains constant for *SPT_L_ldr&s*, as for *SPT_L_r*.

Table 1. Grid-Ssquare family data

nodes/arcs	*SPT_L_dq*	*SPT_L_rq*	*SPT_L_arq*	*SPT_L_r*	*SPT_L_r&s*	*SPT_L_dr&s*	*SPT_L_ldr&s*
16385	**0.03**	**0.02**	**0.03**	**0.02**	**0.07**	**0.04**	**0.03**
49152	1.43	1.31	1.31	2.29	5.25	3.22	2.24
65537	**0.13**	**0.11**	**0.12**	**0.11**	**0.42**	**0.23**	**0.12**
196608	1.48	1.32	1.33	2.28	7.41	4.17	2.24
262145	**0.59**	**0.47**	**0.49**	**0.47**	**2.06**	**1.25**	**0.52**
786432	1.52	1.34	1.35	2.29	8.11	4.99	2.25
1048577	**2.67**	**1.95**	**2.02**	**1.92**	**11.80**	**6.36**	**2.18**
3145728	1.57	1.34	1.35	2.30	11.25	6.10	2.25

Table 2. Grid-Slong family data

nodes/arcs	*SPT_L_dq*	*SPT_L_rq*	*SPT_L_arq*	*SPT_L_r*	*SPT_L_r&s*	*SPT_L_dr&s*	*SPT_L_ldr&s*
16385	**0.02**	**0.03**	**0.03**	**0.02**	**0.10**	**0.07**	**0.02**
49152	1.39	1.29	1.31	2.27	9.09	6.38	2.24
32769	**0.05**	**0.05**	**0.05**	**0.05**	**0.25**	**0.15**	**0.06**
98304	1.37	1.30	1.31	2.27	9.88	5.85	2.24
65537	**0.12**	**0.10**	**0.11**	**0.10**	**0.54**	**0.39**	**0.13**
196608	1.39	1.30	1.32	2.27	10.57	7.46	2.24
131073	**0.21**	**0.20**	**0.22**	**0.20**	**1.24**	**0.89**	**0.27**
393216	1.38	1.30	1.31	2.27	11.65	8.31	2.24
262145	**0.43**	**0.41**	**0.43**	**0.39**	**2.66**	**1.90**	**0.54**
786432	1.39	1.30	1.31	2.27	12.42	8.70	2.24
524289	**0.87**	**0.80**	**0.86**	**0.78**	**5.27**	**4.89**	**1.07**
1572864	1.39	1.30	1.31	2.27	12.24	10.90	2.24

The above mentioned pathological effect of the insertion of all nodes in Q from the beginning in the *r&s* variants, is confirmed by comparing the results in Table 1 with those in Table 3, which are

relative to special grids having all nodes adjacent to the root. In fact, all *r&s* variants practically perform the same, while for *SPT_L_r* cpu time increases 2.5 times with twice the number of scans.

With regard to the disassembling and reassembling variants, cpu time roughly doubles, and the average number of scans increases by one unit. As expected, *SPT_L_ldr&s* achieves the best performance; it is also worth noting that the reassembling variants have a better score than all the benchmark algorithms.

Table 3. Grid-Ssquare-S family data

nodes/arcs	*SPT_L_dq*	*SPT_L_rq*	*SPT_L_arq*	*SPT_L_r*	*SPT_L_r&s*	*SPT_L_dr&s*	*SPT_L_ldr&s*
16386	**0.05**	**0.05**	**0.05**	**0.08**	**0.07**	**0.04**	**0.03**
65537	2.48	2.29	2.33	4.57	5.19	3.23	2.24
65538	**0.27**	**0.23**	**0.24**	**0.29**	**0.40**	**0.25**	**0.15**
262145	2.52	2.30	2.34	4.59	6.48	4.06	2.24
262146	**1.13**	**0.98**	**1.01**	**1.28**	**2.01**	**1.28**	**0.63**
1048577	2.56	2.32	2.38	4.62	7.53	4.73	2.25
1048578	**5.06**	**3.97**	**4.14**	**5.14**	**10.32**	**6.30**	**2.54**
4194305	2.62	2.33	2.38	4.62	9.46	5.75	2.25

Wide grids have only 16 layers; the dynamic *r&s* policy, as shown in Table 4, results to be particularly adapt in the updating operations on very long and sparse networks, as the rings are. *SPT_L_ldr&s* achieves the best score. No significant difference is shown by disassembling and reassembling policy.

Table 4. Grid-Swide family data

nodes/arcs	*SPT_L_dq*	*SPT_L_rq*	*SPT_L_arq*	*SPT_L_r*	*SPT_L_r&s*	*SPT_L_dr&s*	*SPT_L_ldr&s*
16385	**0.03**	**0.03**	**0.03**	**0.03**	**0.04**	**0.03**	**0.02**
49152	1.26	1.26	1.26	2.23	3.02	2.34	2.22
32769	**0.07**	**0.07**	**0.07**	**0.06**	**0.09**	**0.07**	**0.06**
98304	1.26	1.25	1.25	2.22	3.05	2.32	2.21
65537	**0.17**	**0.14**	**0.15**	**0.15**	**0.21**	**0.14**	**0.14**
196608	1.27	1.26	1.25	2.22	3.01	2.32	2.21
131073	**0.37**	**0.35**	**0.36**	**0.34**	**0.45**	**0.32**	**0.31**
393216	1.26	1.26	1.26	2.23	3.03	2.32	2.22
262145	**0.80**	**0.77**	**0.79**	**0.73**	**0.96**	**0.71**	**0.67**
786432	1.27	1.28	1.28	2.25	2.97	2.37	2.23
524289	**1.68**	**1.63**	**1.68**	**1.57**	**2.01**	**1.48**	**1.34**
1572864	1.27	1.28	1.28	2.25	3.04	2.37	2.23

In the so called hard grid problems, as pointed out in Section 4.1 for *SPT_L_rq*, the “perverse” effect of sequences of disassembling and reassembling operations without updating the tree structure, is evidenced in Tables 5 and 6, where the results of the *alternate* variant show how it is advantageous to impeach it: with the same number of average scans, the cpu time reduces to just one third.

Similar considerations, based on different reasons, can be made on the dynamic *r&s* variants; the swapping list structure in *SPT_L_dr&s* causes the pathological behaviour of an enormous number of scans per node; the sublist-stack structure in *SPT_L_ldr&s* prevents that pathology by updating node labels “as soon as possible” in a favourable order; cpu time and number of scans decrease by at least one order of magnitude. The small number of scans performed by *SPT_L_ldr&s* is the reason why it results by far the best of the tested variants.

Table 5. Grid-Phard family data

nodes/arcs	*SPT_L_dq*	*SPT_L_rq*	*SPT_L_arq*	*SPT_L_r*	*SPT_L_r&s*	*SPT_L_dr&s*	*SPT_L_ldr&s*
8193	**0.19**	**0.40**	**0.14**	**0.14**	**0.10**	**0.12**	**0.04**
63808	12.44	7.99	9.06	16.90	11.66	15.63	5.36
16385	**0.40**	**0.90**	**0.30**	**0.33**	**0.24**	**0.33**	**0.10**
129344	12.53	7.97	9.31	17.98	12.52	18.72	5.42
32769	**0.81**	**1.94**	**0.61**	**0.71**	**0.53**	**0.94**	**0.19**
260416	12.75	8.15	9.07	17.87	13.07	24.14	5.42
65537	**1.64**	**4.04**	**1.26**	**1.47**	**1.20**	**2.41**	**0.40**
522560	12.83	8.17	9.31	17.84	14.01	29.60	5.45
131073	**3.30**	**8.06**	**2.53**	**2.99**	**2.57**	**7.85**	**0.80**
1046848	12.93	8.13	9.36	17.98	14.73	47.05	5.45
262145	**6.58**	**16.30**	**5.06**	**5.96**	**5.37**	**23.44**	**1.61**
2095424	12.87	8.14	9.36	17.82	15.12	68.73	5.43

Table 6. Grid-Nhard family data

nodes/arcs	*SPT_L_dq*	*SPT_L_rq*	*SPT_L_arq*	*SPT_L_r*	*SPT_L_r&s*	*SPT_L_dr&s*	*SPT_L_ldr&s*
8193	**0.50**	**0.44**	**0.17**	**0.15**	**0.09**	**0.12**	**0.04**
63808	27.01	10.08	9.78	17.88	11.66	15.63	5.36
16385	**1.13**	**1.03**	**0.38**	**0.38**	**0.23**	**0.33**	**0.08**
129344	29.11	10.14	10.02	19.89	12.52	18.89	5.42
32769	**2.42**	**2.35**	**0.76**	**0.86**	**0.54**	**0.96**	**0.18**
260416	30.65	10.20	9.95	20.66	13.07	24.95	5.42
65537	**4.93**	**4.80**	**1.53**	**1.85**	**1.19**	**2.62**	**0.38**
522560	30.87	10.29	9.98	21.30	14.01	32.90	5.45
131073	**10.00**	**9.85**	**3.09**	**4.00**	**2.56**	**8.03**	**0.77**
1046848	31.32	10.35	10.10	22.58	14.73	48.28	5.45
262145	**20.05**	**19.63**	**6.20**	**8.41**	**5.37**	**22.95**	**1.56**
2095424	31.31	10.35	10.04	23.47	15.12	66.88	5.43

In conclusion, on this set of grid families we can argue that: *i*) reassembling yields more robust algorithm than disassembling, and *ii*) the sublist-stack dynamic *r&s* appears to be a challenging policy.

5.2.2 Results on SPRAND problems

The SPRAND generator yields graphs made by one hamiltonian cycle plus some randomly selected arcs. The cost of the arcs of the hamiltonian cycle is 1 and all other arcs have costs picked uniformly at random in the interval $[0, 10^4]$. In the Rand-4 family the density is 4, i.e. $m = 4n$, while in the Rand-1:4 family it is $m = n^2/4$, so Rand-4 graphs are sparse while Rand1:4 graphs are quite dense. The Rand-Len family is designed to test the dependency of algorithms on arc costs. These problems only differ in the range $[L, U]$ in which additional arc costs are selected. Problems in the Rand-P family have the same structure, except that the costs are modified by assigning to each node v a weight w_v chosen uniformly at random from the interval $[0, P]$ and by replacing c_{uv} with $c_{uv} + w_u - w_v$.

Looking at the sparse random graphs of Rand-4 family, see Table 7, one may notice that the reassembling variants ensure less scans than disassembling algorithm (this is quite a constant effect on all types of networks), and the cpu time drops drastically since sequences of discarding-reinserting nodes are avoided. On the contrary, dynamic reordering does not appear useful since the number of scans increases, especially for *SPT_L_ldr&s*. Globally, *SPT_L_rq*, *SPT_L_arq* and *SPT_L_r&s* behave similarly.

Table 7. Rand-4 family data

nodes/arcs	*SPT_L_dq*	*SPT_L_rq*	*SPT_L_arq*	*SPT_L_r*	*SPT_L_r&s*	*SPT_L_dr&s*	*SPT_L_ldr&s*
8192	**0.09**	**0.07**	**0.07**	**0.08**	**0.08**	**0.08**	**0.17**
32768	7.75	6.95	7.02	14.58	10.92	11.83	23.90
16348	**0.29**	**0.22**	**0.23**	**0.26**	**0.22**	**0.24**	**0.51**
65536	8.61	7.72	7.65	16.19	11.72	12.82	29.06
32768	**0.74**	**0.55**	**0.55**	**0.65**	**0.56**	**0.62**	**1.25**
131072	8.78	7.90	7.89	16.47	12.00	13.38	27.86
65536	**1.86**	**1.34**	**1.28**	**1.61**	**1.35**	**1.52**	**3.19**
262144	9.80	8.92	8.43	18.21	13.29	14.42	31.43
131072	**4.37**	**2.99**	**2.98**	**3.69**	**3.04**	**3.56**	**8.29**
524288	10.48	9.24	9.19	19.48	14.15	15.75	37.88
262144	**9.78**	**6.49**	**6.40**	**8.27**	**6.59**	**7.77**	**17.43**
1048576	11.07	9.60	9.39	20.76	14.62	16.46	37.42
524288	**20.84**	**13.89**	**13.54**	**17.51**	**14.10**	**16.60**	**37.03**
2097152	11.38	10.00	9.68	21.02	15.02	16.83	37.57
1048576	**45.38**	**29.52**	**29.34**	**38.29**	**28.34**	**32.77**	**78.46**
4194304	12.00	10.37	10.35	22.40	15.18	16.70	38.57

Vice versa, in the case of dense networks, see Table 8, disassembling significantly dominates all the reordering variants; moreover, reassembling appears more convenient than disassembling, essentially for the reduction of the number of scans.

Table 8. Rand-1:4 family data

nodes/arcs	*SPT_L_dq*	*SPT_L_rq*	*SPT_L_arq*	*SPT_L_r*	*SPT_L_r&s*	*SPT_L_dr&s*	*SPT_L_ldr&s*
512	**0.03**	**0.02**	**0.02**	**0.03**	**0.03**	**0.03**	**0.04**
65536	4.40	4.20	4.23	7.91	6.22	6.92	10.90
1024	**0.14**	**0.13**	**0.13**	**0.21**	**0.19**	**0.19**	**0.27**
262144	4.29	4.24	4.24	7.91	6.21	6.79	11.27
2048	**0.54**	**0.51**	**0.53**	**0.91**	**0.82**	**0.89**	**1.40**
1048576	4.01	3.85	3.97	7.23	6.01	6.79	11.41
4096	**2.31**	**2.10**	**2.18**	**3.90**	**3.36**	**3.67**	**5.77**
4194304	4.10	3.84	3.95	7.24	5.87	6.56	10.72

As far as the Rand-Len and the Rand-P families are concerned, see Tables 9 and 10, the relative performances among the variants are very close to those obtained for sparse graphs: reassembling behaves better than disassembling; dynamic reordering is worse than one reordering per pass.

Table 9. Rand-Len family data. All problems have 131072 nodes and 524288 arcs

[*L*,*U*]	*SPT_L_dq*	*SPT_L_rq*	*SPT_L_arq*	*SPT_L_r*	*SPT_L_r&s*	*SPT_L_dr&s*	*SPT_L_ldr&s*
[1,1]	**0.35**	**0.35**	**0.35**	**0.44**	**1.25**	**1.30**	**1.68**
	1.00	1.00	1.00	1.61	4.48	4.78	6.33
[0,10]	**1.10**	**0.96**	**0.98**	**1.02**	**1.56**	**1.66**	**2.61**
	2.36	2.40	2.42	4.36	5.94	6.39	10.39
[0,100]	**2.21**	**1.68**	**1.72**	**1.91**	**1.99**	**2.24**	**4.24**
	4.75	4.50	4.53	8.93	8.21	9.01	17.88
[0,10000]	**4.38**	**2.99**	**2.99**	**3.70**	**3.06**	**3.54**	**8.29**
	10.48	9.24	9.19	19.48	14.15	15.75	37.88
[0,1000000]	**5.84**	**4.00**	**4.08**	**5.10**	**2.33**	**2.51**	**5.22**
	15.29	13.18	13.46	27.96	12.26	12.59	23.65

In the Rand-Len family, *SPT_L_r&s* dominates the others for large cost values ($U = 10^6$) while it is dominated by the two reordering variants for smaller values ($U \leq 10^4$). Such behaviour is essentially due to the average number of scans.

In the Rand-P family, all the variants are "stable", i.e. the parameter P has no effect on the algorithms behaviour. The very small number of scans is the reason why *SPT_L_rq* and *SPT_L_arq* behave better than the others, even though the differences among the variants are virtually unnoticeable, except for *SPT_L_ldr&s*.

Table 10. Rand-P family data. All problems have 131072 nodes and 524288 arcs

P	*SPT_L_dq*	*SPT_L_rq*	*SPT_L_arq*	*SPT_L_r*	*SPT_L_r&s*	*SPT_L_dr&s*	*SPT_L_ldr&s*
0	**4.38**	**3.00**	**2.98**	**3.70**	**3.05**	**3.54**	**8.29**
	10.48	9.24	9.19	19.48	14.15	15.75	37.88
1000	**4.37**	**3.00**	**2.98**	**3.57**	**3.05**	**3.56**	**8.30**
	10.48	9.24	9.19	19.48	14.15	15.75	37.88
10000	**4.37**	**3.01**	**2.93**	**3.62**	**3.06**	**3.55**	**8.29**
	10.48	9.24	9.19	19.48	14.15	15.75	37.88
100000	**4.37**	**3.00**	**3.01**	**3.66**	**3.06**	**3.56**	**8.30**
	10.48	9.24	9.19	19.48	14.15	15.75	37.88
1000000	**4.38**	**3.01**	**3.02**	**3.76**	**3.06**	**3.54**	**8.30**
	10.48	9.24	9.19	19.48	14.15	15.75	37.88
10000000	**4.37**	**3.00**	**2.99**	**3.77**	**3.06**	**3.55**	**8.29**
	10.48	9.24	9.19	19.48	14.15	15.75	37.88

Globally, on the random families we can conclude that the most robust variants appear to be *SPT_L_rq*, *SPT_L_arq* and *SPT_L_r&s*. It is noticeable that the stack structure of Q is the cause of the poor performance of *SPT_L_ldr&s*.

5.2.3 Results on SPACYC problems

Acyclic graphs are yielded by the SPACYC generator, with nonnegative arc costs in the Acyc-Pos family, with non positive arc costs in the Acyc-Neg, and with both in the Acyc-P2N family. In the first two, the algorithms are tested with respect of the network size, while in the last family, the size is fixed but the interval $[L, U]$ varies, from which all arc costs are selected uniformly at random, so that the values L and U determine the expected fraction f of negative cost arcs. See [6] for more details.

On acyclic networks, as pointed out in Theorem 1, the reorder&scanning performed at the beginning from root r realises the topological order. As a consequence, the number of scans is, for all *r&s* variants, 2 for each node. This particular feature makes unfair any further comparison with the disassembling and reassembling variants.

Looking at Tables 12 and 13, note that the pathological behaviour of *SPT_L_dq* for Acyc-Neg and Acyc-P2N families is due to the enormous overhead of scanning exact labels in an almost reverse order respect to the topological one; the perverse effect is sensibly reduced by the reassembling variants, especially by *SPT_L_arq*. No effect is given by the dynamic reordering since the right order is ensured following the first visit of G_{SA}. Note that *SPT_L_r* behaves quite poorly for both Acyc-Pos and Acyc-P2N families, while the negativity of the arc costs in Acyc-Neg family ensures implicitly a quite good node order for *SPT_L_r* too. The results obtained for the Acyc-P2N family, see Table 13, confirm the above diagnosis; in fact by increasing the percentage of negative arcs (until 60%), the number of scans

performed by *SPT_L_r* increases, to then slightly decrease (until 90%), finally reaching the average value of 2 once all the arcs have a negative cost.

Table 11. Acyc-Pos family data

nodes/arcs	*SPT_L_dq*	*SPT_L_rq*	*SPT_L_arq*	*SPT_L_r*	*SPT_L_r&s*	*SPT_L_dr&s*	*SPT_L_ldr&s*
8192	**0.18**	**0.16**	**0.17**	**0.21**	**0.05**	**0.05**	**0.05**
131072	6.59	6.50	6.60	12.51	2.00	2.00	2.00
16384	**0.49**	**0.46**	**0.48**	**0.58**	**0.10**	**0.10**	**0.10**
262144	7.14	7.58	7.76	14.52	2.00	2.00	2.00
32768	**1.25**	**1.12**	**1.15**	**1.41**	**0.22**	**0.24**	**0.25**
524288	7.46	7.55	7.69	14.55	2.00	2.00	2.00
65536	**3.12**	**2.85**	**2.97**	**3.77**	**0.51**	**0.55**	**0.55**
1048576	8.35	8.78	9.10	17.23	2.00	2.00	2.00
1331072	**6.72**	**5.81**	**6.06**	**7.79**	**1.08**	**1.19**	**1.20**
2097152	8.43	8.58	8.93	16.78	2.00	2.00	2.00

Table 12. Acyc-Neg family data

nodes/arcs	*SPT_L_dq*	*SPT_L_rq*	*SPT_L_arq*	*SPT_L_r*	*SPT_L_r&s*	*SPT_L_dr&s*	*SPT_L_ldr&s*
8192	**1.31**	**0.59**	**0.43**	**0.05**	**0.05**	**0.05**	**0.05**
131072	45.21	14.00	13.88	2.00	2.00	2.00	2.00
16384	**4.73**	**2.08**	**1.23**	**0.11**	**0.12**	**0.12**	**0.12**
262144	64.83	16.72	17.52	2.00	2.00	2.00	2.00
32768	**16.49**	**6.96**	**3.51**	**0.26**	**0.27**	**0.28**	**0.28**
524288	92.76	18.44	21.32	2.00	2.00	2.00	2.00
65536	**53.95**	**21.47**	**9.46**	**0.62**	**0.60**	**0.67**	**0.67**
1048576	130.20	22.44	26.40	2.00	2.00	2.00	2.00
1331072	**168.26**	**64.05**	**25.86**	**1.36**	**1.35**	**1.50**	**1.50**
2097152	185.73	25.48	34.55	2.00	2.00	2.00	2.00

Table 13. Acyc-P2N family data. All problems have 16384 nodes and 262144 arcs

f(%)	*SPT_L_dq*	*SPT_L_rq*	*SPT_L_arq*	*SPT_L_r*	*SPT_L_r&s*	*SPT_L_dr&s*	*SPT_L_ldr&s*
10	**0.15**	**0.14**	**0.14**	**0.14**	**0.10**	**0.11**	**0.10**
	2.15	2.07	2.08	3.01	2.00	2.00	2.00
20	**0.33**	**0.26**	**0.27**	**0.23**	**0.11**	**0.10**	**0.12**
	5.39	4.15	4.20	6.29	2.00	2.00	2.00
30	**0.88**	**0.51**	**0.50**	**0.44**	**0.11**	**0.12**	**0.12**
	14.43	7.70	7.90	12.49	2.00	2.00	2.00
40	**2.81**	**1.10**	**0.87**	**0.84**	**0.11**	**0.11**	**0.12**
	43.65	13.67	13.23	22.81	2.00	2.00	2.00
50	**11.01**	**2.83**	**1.34**	**1.18**	**0.12**	**0.12**	**0.12**
	153.54	26.66	18.65	30.81	2.00	2.00	2.00
60	**27.74**	**5.34**	**1.57**	**1.35**	**0.12**	**0.12**	**0.12**
	371.18	45.70	21.34	33.52	2.00	2.00	2.00
70	**42.43**	**6.68**	**1.69**	**1.29**	**0.11**	**0.12**	**0.12**
	533.43	57.24	21.99	31.96	2.00	2.00	2.00
80	**51.40**	**7.31**	**1.68**	**1.15**	**0.12**	**0.12**	**0.12**
	626.06	61.48	21.67	29.30	2.00	2.00	2.00
90	**57.73**	**7.36**	**1.68**	**1.01**	**0.12**	**0.12**	**0.12**
	681.89	60.60	21.93	25.24	2.00	2.00	2.00
100	**62.62**	**7.54**	**1.65**	**0.12**	**0.12**	**0.12**	**0.12**
	723.44	62.62	20.51	2.00	2.00	2.00	2.00

5.3 A global evaluation

The performed tests allow us to draw the following considerations:

- in general, it is better to reassemble than to disassemble due to a smaller number of node scans;
- dynamic reordering is helpful in structured graphs but the cases in which the number of layers is enormous;
- the reorder&scan strategy ensures a $\Theta(m)$ behaviour of the corresponding variants on acyclic graphs.

Finally, among the variants proposed, there is always at least one which appears to perform better than, or at least as well as, the best benchmark algorithm. Thus, we can conclude that the tests performed provide an "a posteriori confirmation" of what has been "a priori" conjectured during the design of the variants.

6 Conclusions

In this paper, we introduce two variants of Tarjan's disassembling algorithm [27] and two variants of the topological ordering algorithms by Goldberg and Radzik [18]. The variants have been conceived on the basis of the analysis of the experimental results obtained by the three benchmark algorithms in previous studies [6].

The first two variants exploit the tree data structure to postpone the inspection of inexact label nodes by "reassembling" tree portions of the current tree instead of "disassembling".

The Goldberg and Radzik's variants investigate the effects of applying the topological reordering and scanning at each step instead of at the beginning of each pass.

The algorithm design work is completed by a computational validation through a set of experimental tests. The results obtained suggest that all the four new variants are rather robust in practice and are sometimes comparable or even better than the "parents" algorithms.

References

[1] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network flows: theory, algorithms and applications*, Prentice Hall, Englewood Cliffs, NJ (1993).

[2] R. K. Ahuja, K. Mehlhorn, J. B. Orlin and R. E. Tarjan, Faster algorithms for the shortest path problem, *Journal of the A.C.M.* 37 (1990) 213-223.

[3] R. E. Bellman, On a routing problem, *Quarterly Applied Mathematics* 16 (1958) 87-90.

[4] D. P. Bertsekas, A simple and fast label correcting algorithm for shortest paths, *Networks* 23 (1993) 703-709.

[5] B. V. Cherkassky and A. V. Goldberg, Negative-cycle detection algorithms, *Mathematical Programming, series A* 85 (1999) 277-311.

[6] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, Shortest paths algorithms: theory and experimental evaluation, *Mathematical Programming, series A* 73 (1996) 129-174.

[7] B. V. Cherkassky, A. V. Goldberg, and C. Silverstein, Buckets, heaps, lists, and monotone priority queues, *SIAM Journal on Computing* 28 (1999) 1326-1346.

[8] N. Deo and C. Pang, Shortest path algorithms: taxonomy and annotation, *Networks* 14 (1984) 275-323.

[9] E. W. Dijkstra, A note on two problems in connexion with graphs, *Numerische Matematik* 1 (1959) 269-271.

[10] L. R. Ford Jr., Network flow theory, *Rand Corporation Report* No. P-293 (1956).

[11] M. L. Fredman and D. E. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, *Journal of Computers and Systems Science* 48 (1994) 533-551.

[12] H. N. Gabow and R. E. Tarjan, Faster scaling algorithms for network problems, *SIAM Journal on Computing* (1989) 1013-1036.

[13] G. Gallo and S. Pallottino, Shortest paths algorithms, in (B. Simeone et al., eds.) *Annals of Operations Research*, 13 (1988) 3-79.

[14] F. Glover, D. Glover and D. Klingman, Computational study of an improved shortest path algorithm, *Networks* 14 (1984) 25-37.

[15] F. Glover, D. Klingman and N. Phillips, A new polynomially bounded shortest path algorithm, *Operations Research* 33 (1985) 65-73.

[16] F. Glover, D. Klingman, N. Phillips and R. F. Schneider, New polynomial shortest path algorithms and their computational attributes, *Management Science* 31 (1985) 1106-1128.

[17] A. V. Goldberg, Scaling algorithms for the shortest paths problem, *SIAM Journal on Computing* 24 (1995) 494-504.

[18] A. V. Goldberg and T. Radzik, A heuristic improvement of the Bellman-Ford algorithm, *Applied Mathematics Letters* 6 (1993) 3-6.

[19] D. Goldfarb, J. Hao and S.-R. Kai, Efficient shortest path simplex algorithms, *Operations Research* 38 (1990) 624-628.

[20] M. S. Hung and J. J. Divoky, A computational study of efficient shortest path algorithms, *Computers and Operations Research* 15 (1988) 567-576.

[21] A. Kershenbaum, A note on finding shortest path trees, *Networks* 11 (1981) 399-400.

[22] B. Ju. Levit and B. N. Livshits, Neleneinye setevye transportnye zadachi, *Transport*, Moscow (1972) in Russian.

[23] J.-F. Mondou, T. G. Crainic and S. Nguyen, Shortest path algorithms: a computational study with the C programming language, *Computers and Operations Research* 18 (1991) 767-786.

[24] E. F. Moore, The shortest path through a maze, in *Proceedings of the International Symposium on the Theory of Switching*, Harvard University Press (1959) 285-292.

[25] S. Pallottino, Shortest-path methods: complexity, interrelations and new propositions, *Networks* 14 (1984) 257-267.

[26] U. Pape, Implementation and efficiency of Moore algorithms for the shortest root problem, *Mathematical Programming* 7 (1974) 212-222.

[27] R. E. Tarjan, Shortest paths, *Technical report*, AT&T Bell Laboratories, Murray Hill, NJ (1981).

[28] R. E. Tarjan, *Data structures and network algorithms*, SIAM, Philadelphia, PA (1983).