FI MU

Faculty of Informatics
Masaryk University

How to Employ Reverse Search in Distributed Single Source Shortest Paths

by

Luboš Brim
Ivana Černá
Pavel Krčál
Radek Pelánek

FI MU Report Series **FIMU-RS-2001-09**

Copyright © 2001, FI MU **November 2001**

How to Employ Reverse Search in Distributed Single Source Shortest Paths*

Luboš Brim, Ivana Černá, Pavel Krčál, and Radek Pelánek

Department of Computer Science, Faculty of Informatics
Masaryk University Brno, Czech Republic
{brim,cerna,xkrcal,xpelanek}@fi.muni.cz

Abstract. A distributed algorithm for the single source shortest path problem for directed graphs with arbitrary edge lengths is proposed. The new algorithm is based on relaxations and uses reverse search for inspecting edges. No additional data structures are required. At the same time the algorithm uses a novel distributed way to recognize a reachable negative-length cycle in the graph which facilitates the scalability of the algorithm.

1 Introduction

The single source shortest paths problem is a key component of many applications and lots of effective sequential algorithms are proposed for its solution (for an excellent survey see [CG99]). However, in many applications graphs are too massive to fit completely inside the computer's internal memory. The resulting input/output communication between fast internal memory and slower external memory (such as disks) can be a major performance bottleneck. Several methods are used to evade this bottleneck. External memory algorithms [Vit98] exploit locality in order to reduce input/output cost. Another successful approach comes out from parallel processing.

In particular, in LTL model checking application (see Section 6) the graph is typically extremely large. In order to optimize the space complexity of the computation the graph is generated *on-the-fly*. Successors of a vertex are determined dynamically and consequently there is no need to store any information about edges permanently. Therefore neither the techniques used in external memory algorithms (we do not know any properties of the examined graph in advance) nor the parallel algorithms based on adjacency matrix graph representation are applicable.

* This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/00/1023.

The approach we have been looking upon is to increase the computational power (especially the amount of randomly accessed memory) by building a powerful parallel computer as a network of cheap workstations with disjoint memory which communicate via message passing.

Futhermore, we require that the distributed algorithm is compatible with other space-saving techniques (e.g. *on-the-fly* technique or *partial order* technique). Our distributed algorithm is based on the relaxation of graph's edges [CLR90]. Distributed relaxation-based algorithms are known only for special settings of single source shortest paths problem. For general digraphs with non-negative edge lengths parallel algorithms are presented in [CMMS98,MS00,RV92]. For special cases of graphs, like planar digraphs [TZ96,KPSZ94], graphs with separator decomposition [Coh96] or graphs with small tree-width [CZ95], more efficient algorithms are known. Yet none of these algorithms is applicable to general digraphs with negative length cycles.

The most notable features of our proposed distributed algorithm are *reverse search* and *walk to root* approaches. The *reverse search* method is known to be an exceedingly space efficient technique [AF96,Nie00]. Data structures of the proposed algorithm can be naturally used by the *reverse search* and it is possible to reduce the memory requirements which would be otherwise induced by structures used for traversing graph (such as a queue or a stack). This could save up to one third of memory which is practically significant.

Walk to root is a strategy how to detect the presence of a negative length cycle in the input graph. The cycle is searched for in the graph of parent pointers maintained by the method. The parent graph cycles, however, can appear and disappear. The aim is to detect a cycle as soon as possible and at the same time not to increase the time complexity of underlying relaxation algorithm significantly. To that end we introduce a solution which allows to amortize the time complexity of cycle detection over the complexity of relaxation.

2 Problem Definition and General Method

Let (G, s, l) be a given triple, where $G = (V, E)$ is a directed graph, $l : E \to R$ is a *length function* mapping edges to real-valued lengths and $s \in V$ is the source vertex. We denote $n =\mid V \mid$ and $m =\mid E \mid$. The *length* $l(p)$ *of the path* p is the sum of the lengths of its constituent edges. We

define the *shortest path length* from s to v by

$$\delta(s,v) = \begin{cases} \min\{l(p) \mid p \text{ is a path from } s \text{ to } v\} & \text{if there is such a path} \\ \infty & \text{otherwise} \end{cases}$$

A *shortest path* from vertex s to vertex v is then defined as any path p with length $l(p) = \delta(s,v)$. If the graph G contains no negative length cycles (*negative cycles*) reachable from the source vertex s, then for all $v \in V$ the shortest path length remains well-defined and the graph is called *feasible*. The single source shortest paths (SSSP) problem is to determine whether the given graph is *feasible* and if so to compute $\delta(s,v)$ for all $v \in V$. For purposes of our algorithm we suppose that some *linear ordering on vertices* is given.

The general method for solving the SSSP problem is the *relaxation* method [CG99,CLR90]. For every vertex v the method maintains its distance label $d(v)$ and parent vertex $p(v)$. The subgraph G_p of G induced by edges $(p(v),v)$ for all v such that $p(v) \neq nil$ is called the *parent graph*. The method starts by setting $d(s) = 0$ and $p(s) = nil$. At every step the method selects an edge (v,u) and *relaxes* it, which means that if $d(u) > d(v) + l(v,u)$ then it sets $d(u)$ to $d(v) + l(v,u)$ and sets $p(u)$ to v.

If no $d(v)$ can be improved by any relaxation then $d(v) = \delta(s,v)$ for all $v \in V$ and G_p determines the shortest paths. Different strategies for selecting an edge to be relaxed next lead to different algorithms. For graphs where negative cycles could exist the relaxation method must be modified to recognize the *unfeasibility* of the graph. As in the case of relaxation various strategies are used to detect negative cycles [CG99]. However, not all of them are suitable for our purposes – they are either uncompetitive (as for example time-out strategy) or they are not suitable for distribution (such as the admissible graph search which uses hardly parallelizable depth-first search or level-based strategy which employs global data structures). For our version of distributed SSSP we have used the *walk to root* strategy.

The sequential walk to root strategy can be described as follows. Suppose the relaxation operation applies to an edge (v,u) (i.e. $d(u) > d(v) + l(v,u)$) and the parent graph G_p is acyclic. The operation creates a cycle in G_p if and only if u is an ancestor of v in the current parent graph (see Fig. 1). This can be detected by following the parent pointers from v to s. If the vertex u lies on this path then there is a negative cycle; otherwise the relaxation operation does not create a cycle.

The walk to root method gives immediate cycle detection and can be easily combined with the relaxation method. However, since the path to the root can be long, it increases the cost of applying the relaxation operation to an edge to $\mathcal{O}(n)$. We can use amortization to pay the cost of checking G_p for cycles. Since the cost of such a search is $\mathcal{O}(n)$, the search is performed only after the underlying shortest paths algorithm performs $\Omega(n)$ work. The running time is thus increased only by a constant factor. To preserve the correctness, the behavior of the walk to root has to be significantly modified. The amortization is used in the distributed algorithm and is described in detail in Section 5.

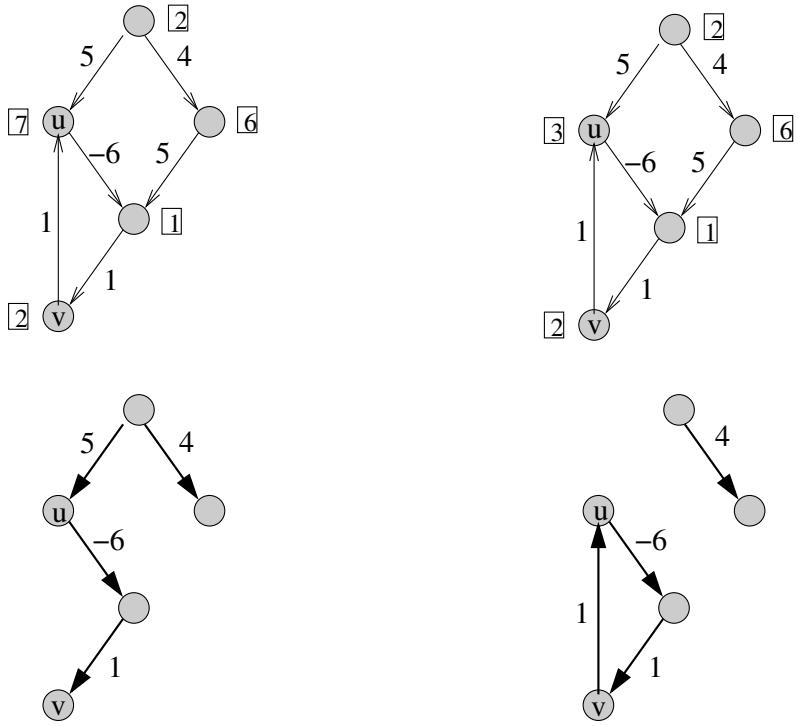

Fig. 1. The part of the graph G and the induced parent graph G_p before and after the relaxation of the edge (v, u) which creates the negative cycle in the parent graph G_p

3 Reverse Search

Reverse search is originally a technique for generating large sets of discrete objects [AF96,Nie00]. Reverse search can be viewed as a depth-first graph traversal that requires neither stack nor node marks to be stored explicitly – all necessary information can be recomputed. Such recom-

putations are naturally time-consuming, but when traversing extremely large graphs, the actual problem is not the time but the memory requirements.

In its basic form the reverse search can be viewed as the traversal of a spanning tree, called the *reverse search tree*. We are given a *local search function* f and an *optimum vertex* v^*. For every vertex v, repeated application of f has to generate a path from v to v^*. The set of these paths defines the *reverse search tree* with the root v^*. A reverse search is initiated at v^* and only edges of the reverse search tree are traversed.

In the context of the SSSP problem we want to traverse the graph G. The parent graph G_p corresponds to the reverse search tree. The optimum vertex v^* corresponds to the source vertex s and the local search function f to the parent function p. The correspondence is not exact since $p(v)$ can change during the computation whereas original search function is fixed. Consequently some vertices can be visited more than once. This is in fact the desired behavior for our application. Moreover, if there is a negative cycle in the graph G then a cycle in G_p will occur and G_p will not be a spanning tree. In such a situation we are not interested in the shortest distances and the way in which the graph is traversed is not important anymore. We just need to detect such a situation and this is delegated to the cycle detection strategy.

```
proc Reverse_search (s)
  p(s) := ⊥;
  v := s;
  while v ≠ ⊥ do
        Do_something (v);
        u := Get_successor (v, NULL);
        while u does not exist do
              last := v; v := p(v);
              u := Get_successor (v, last);
        od
        v := u;
  od
end
```

```
proc Call_recursively (v)
  Do_something (v);
  for each edge (v, w) ∈ E do
      if p(w) = v then
        Call_recursively (w)
      fi
  od
end
```

Fig. 2. Demonstration of the reverse search

Fig. 2 demonstrates the use of the reverse search within our algorithm. Both procedures *Call_recursively(v)* and *Reverse_search(v)* traverse the subtree of v in the same manner and perform operation *do_something(v)*

on its children. *Call_recursively* uses a stack whereas *Reverse_search* uses the parent edges for the traversal. The function *Get_successor(v, w)* returns the first successor u of v which is greater than w with respect to the ordering on the vertices and $p(u) = v$. If no such successor exists an appropriate announcement is returned.

4 Sequential SSSP Algorithm with Reverse Search

We first present the sequential version algorithm (Fig. 3) and prove its correctness and complexity. The algorithm forms the base for the distributed algorithm presented in the subsequent section.

The *Trace* procedure visits vertices in the graph (we say that a vertex is *visited* if it is the value of the variable v in the while cycle in the *Trace* procedure). The procedure terminates either when a negative cycle is detected or when the traversal of the graph is completed.

The *RGS* function combines the relaxation of an edge as introduced in Section 2 and the *Get_successor* function from Section 3. It finds the next vertex u whose label can be improved. The change of $p(u)$ can create a cycle in G_p and therefore the *WTR* procedure is started to detect this possibility. If the change is "safe" the values $d(u)$ and $p(u)$ are updated and u is returned.

In what follows the correctness of the algorithm is stated.

Lemma 1. *Let G contains no negative cycle reachable from the source vertex s. Then G_p forms a rooted tree with root s and $d(v) \geq \delta(s, v)$ for all $v \in V$ at any time during the computation. Moreover, once $d(v) = \delta(s, v)$ it never changes.*

Proof: The proof is principally the same as for other relaxation methods [CLR90].

Lemma 2. *After every change of the value $d(v)$ the algorithm visits the vertex v.*

Proof: Follows directly from the algorithm.

Lemma 3. *Let G contains no negative cycle reachable from the source vertex s. Every time a vertex w is visited the sequence* S *of the assignments on line 6 of the procedure* Trace *will eventually be executed for this vertex. Until this happens $p(w)$ is not changed.*

Proof: The value $p(w)$ cannot be changed because G has no negative cycle and due to Lemma 1 the parent graph G_p does not have any cycle

```
1  proc Trace (s)
2    p(s) := ⊥; v := s;
3    while v ≠ ⊥ do
4          u := RGS (v, NULL);
5          while u does not exist do
6                last := v; v := p(v);
7                u := RGS (v, last); od
8          v := u; od
9  end
```

```
1  proc RGS (v, last){Relax and Get Successor}
2    u := successor of v greater than last;
3    while d(u) ≤ d(v) + l(u, v) do
4          u := next successor of v; od
5    if u exists then
6                WTR (v, u);
7                d(u) := d(v) + l(u, v); p(u) := v;
8                return u;
9            else return "u does not exist"; fi
10 end
```

```
1  proc WTR (at, looking_for){Walk To Root}
2    while at ≠ s and at ≠ looking_for do at := p(at); od
3    if at = looking_for then negative cycle detected fi
4  end
```

Fig. 3. Pseudo-code of the sequential algorithm

and hence w could not be visited during the traversal of the subtree of w.

Let $h(w)$ denotes the depth of w in G_p. We prove the lemma by backward induction (from n to 0) with respect to $h(w)$. For the basis we have $h(w) = n$, w has no child and therefore *RGS(w,NULL)* returns *"u does not exist"* and the sequence *S* is executed immediately. For the inductive step we assume that the lemma holds for each v such that $h(v) \geq k$ and we will prove it for $h(w) = k - 1$. Let $A = \{a_1, a_2, \ldots, a_r\} = \{u \mid (w, u) \in E\}$ be the set of all successors of w. Since $h(a_i) = k$ for all $i \in \{1, \ldots, r\}$, we can use the induction hypothesis for each a_i, i.e. each a_i will be visited, its subtree will be traversed and the sequence *S* will be executed finishing thus the visit of a_i and starting the visit of a_{i+1} in finite time. Because r is finite too, the *RGS* procedure will return *"u does not exist"* for w after a finite number of steps and the sequence *S* will be executed. ■

Theorem 1 (Correctness of the sequential algorithm). *If G has no negative cycle reachable from the source s then the sequential algorithm terminates with $d(v) = \delta(s, v)$ for all $v \in V$ and G_p forms a shortest-paths tree rooted at s. If G has a negative cycle, its existence is reported.*

Proof: Let us at first suppose that there is no negative cycle. Lemma 3 applied to the source vertex s gives the termination of the algorithm. Let $v \in V$ is any vertex and $< v_0, v_1, \ldots, v_k >, s = v_0, v = v_k$ is a shortest path from s to v. We show that $d(v_i) = \delta(s, v_i)$ for all $i \in \{0, \ldots k\}$ by induction on i and therefore $d(v) = \delta(s, v)$.

For the basis we have $d(v_0) = d(s) = \delta(s, s) = 0$ by Lemma 1. From the induction hypothesis we have $d(v_i) = \delta(s, v_i)$. The value $d(v_i)$ was set to $\delta(s, v_i)$ at some moment during the computation. From Lemma 2 vertex v_i is visited afterward and the edge (v_i, v_{i+1}) is relaxed. Due to Lemma 1, $d(v_{i+1}) \geq \delta(s, v_{i+1}) = \delta(s, v_i) + l(v_i, v_{i+1}) = d(v_i) + l(v_i, v_{i+1})$ is true before the relaxation and therefore $d(v_{i+1}) = d(v_i) + l(v_i, v_{i+1}) = \delta(s, v_i) + l(v_i, v_{i+1}) = \delta(s, v_{i+1})$ holds after the relaxation. By Lemma 1 this equality is maintained afterward.

For all vertices v, u with $v = p(u)$ we have $d(u) = d(v) + l(v, u)$. This follows directly from line 7 of the *RGS* procedure. After the termination $d(v) = \delta(s, v)$ and therefore G_p forms a shortest paths tree.

On the other side, if there is a negative cycle in G, then the relaxation process alone would run forever and would create a cycle in G_p. The cycle is detected because before any change of $p(v)$ *WTR* tests whether this change does not create a cycle in G_p. ■

Let us suppose that edges have integer lengths and let $C = \max\{|$ $l(v,w)$ $| : (v,w) \in E\}$.

Theorem 2. *The worst time complexity of the sequential algorithm is* $\mathcal{O}(Cn^4)$.

Proof: Each simple shortest path consists of at most $n-1$ edges and $-C(n-1) \leq \delta(s,v) \leq C(n-1)$ holds for all $v \in V, \delta(s,v) < \infty$. Each vertex v is visited only after $d(v)$ is decreased. Therefore each vertex with finite $\delta(s,v)$ is visited at most $\mathcal{O}(Cn)$ times. If $d(v)$ of some vertex is decresead under $-C(n-1)$ then this vertex could not lie on a simple path from s to v and therefore this vertex lies on a cycle and this cycle is detected by *WTR* procedure. Hence vertices with $\delta(s,v) = -\infty$ are visited at most $\mathcal{O}(Cn)$ times. Each visit consists of updating at most n successors and an update can take $\mathcal{O}(n)$ time (due to the walk to root). Together we have $\mathcal{O}(Cn^3)$ bound for total visiting time of each vertex and $\mathcal{O}(Cn^4)$ bound for the algorithm. ■

We stress that the use of the walk to root in this algorithm is avoidable and the algorithm can be modified to detect a cycle without the walk to root and run in $\mathcal{O}(Cn^3)$ time. The walk to root has been used to make the presentation of the distributed algorithm (where the walk to root is essential) clearer.

5 Distributed Algorithm

In this section we describe a distributed algorithm with amortized walk to root strategy. The algorithm runs on a network of P processors which communicate through a message-passing interface. One processor is distinguished as the *Manager* and is responsible for starting and terminating the distributed computation. All processors perform the same program.

Before introducing our algorithm let us specify the way how a graph is stored in distributed environment. The typical solution is to use an adjacency matrix representation of the graph and divide the matrix among processors. This solution has two major disadvantages. Matrix adjacency representation is not effective for sparse graphs and moreover it is not compatible with on-the-fly construction of the graph, which is essential in the intended application of the algorithm.

Therefore we have to use a different approach. Instead of using an adjacency matrix we generate the adjacency list on-the-fly using the function *successor*. The set of vertices is divided into disjoint subsets and each processor is responsible for the owned subset of vertices and for edges coming out from these vertices. The distribution is determined by

the function *owner* which assigns every vertex v to a processor i. The partition of vertices should be well-balanced and should minimize the number of cross-edges (the edge $(u, v) \in E$ is *cross-edge* iff $owner(u) \neq owner(v)$). Good partition of vertices among processors is important because it has impact on communication complexity and thus on run-time of the program. We do not discuss the balancing of the partition here as it is itself quite a difficult problem and depends on particular application.

The main idea of the distributed algorithm can be summarized as follows (for details see the pseudo-code). The computation is initialized by the processor owning the source vertex by calling *Trace*$(s, \perp)$ and is expanded to other processors as soon as the traversal visits the "border" vertices. Each processor visits and labels vertices basically in the same manner as the sequential algorithm does.

Because the walk to root is expensive operation – it has complexity $\mathcal{O}(n)$ – we amortize it over the relaxation operations. For that reason it is not initiated after every change in the parent graph but only after $\mathcal{O}(n)$ changes. Under such circumstances it can happen that even when the relaxation of an edge (u, v) does not create a cycle in the parent graph G_p, there can be a cycle on the way from v to s which was created meanwhile. Therefore we have to modify the walk to root strategy. In the following we describe the changes that have to be taken into account when running the amortized walk to root in a distributed environment.

The walk to root used in sequential algorithm detects a cycle by returning to the vertex it has been started from. Let us call this vertex the *origin* of the walk. On the contrary the distributed walk to root detects a cycle whenever it returns to *any* vertex it has already passed through. In order to detect such a situation, the distributed walk to root marks every vertex it passes through.

In the distributed environment it is possible to start more than one walk concurrently. It is necessary to distinguish marks set by different walks. Hence every walk marks vertices by its origin. It may happen that a walk reaches a vertex that is already marked by some other walk. To avoid blocking and infinite overwriting of marks set by several walks, we introduce ordering on walks (which is induced by ordering on vertices) and we give "priority" to higher walks (Fig. 4). When walk from the origin x reaches a vertex that is already marked by some other origin y then it is either finished (in case that $x < y$) or it overwrites the previous mark (in case that $x > y$).

After finishing walk its marks have to be removed. Otherwise, the marks could block future runs of walks from lower origins as well as

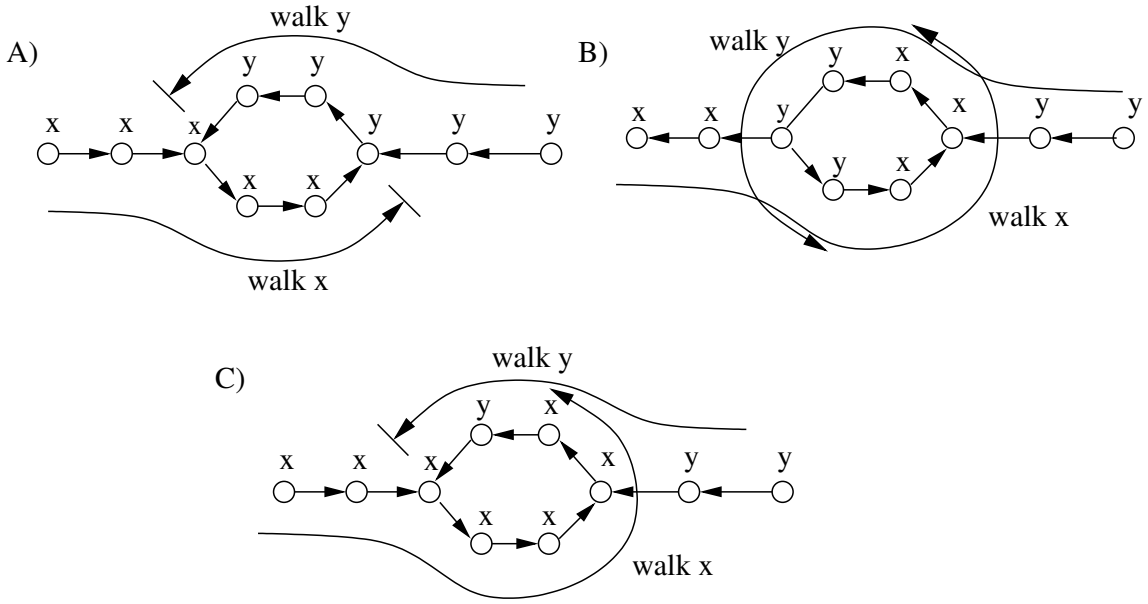

Fig. 4. A) Mutually blocking walks – the cycle is not detected; B) Mutually overwriting walks – walks do not terminate; C) Solution – higher walk has priority, both termination and cycle detection are guaranteed

they would disable future walks from the same origin. The removal is done by the *REM* procedure which proceeds in the same way as the *WTR* procedure using parent pointers. This is the reason why the change of $p(v)$ of marked vertices is forbidden (line 6 of the *RGS* procedure). The *REM* procedure finishes after reaching the source or vertex marked by some other walk. It may happen that some of the marks have been overwritten and therefore the *REM* procedure does not remove all marks. These marks will be eventually overwritten and removed by the *REM* procedure started in some other vertex. However, before this happens these marks could spoil the correctness of the cycle detection (see Fig. 5). We stress that scenario sketched in Fig. 5 is possible only in the distributed environment.

To avoid uncorrect cycle detection we introduce *stamps*. Each processor maintains a *counter* of started walks. Vertices are marked both by the origin of the walk and by the current value of the processor counter (stamp) so it is possible to distinguish marks.

Let us summarize the main features of the distributed walk to root strategy (referenced lines are from the *WTR* procedure):

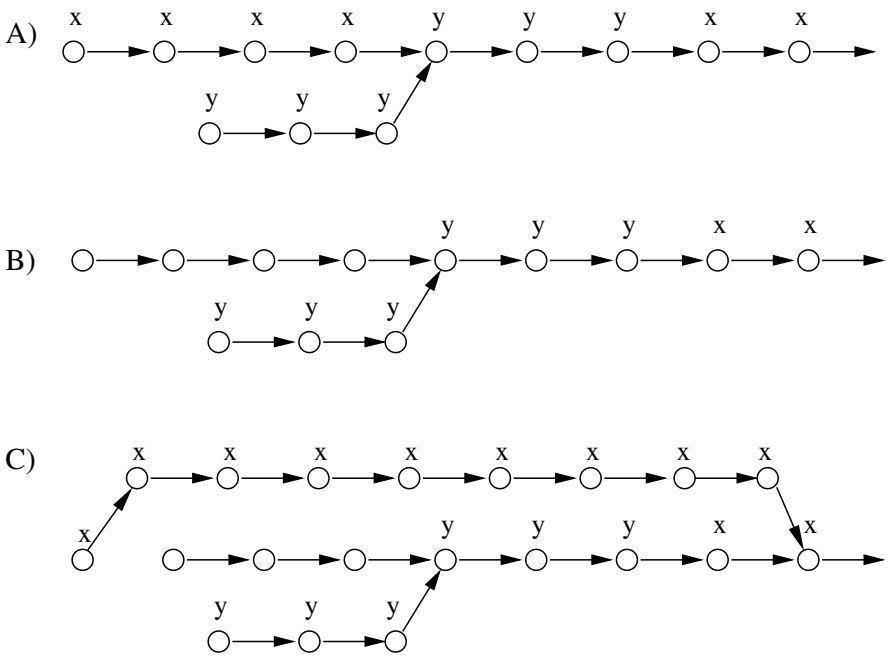

Fig. 5. A) Walk x is partially overwritten by walk y which is still in progress; B) the *REM* procedure does not remove all x marks; C) a new walk from x can detect false cycle

- The walk traverses the parent graph G_p using parent pointers $p(v)$ in the same manner as the sequential one.
- The complexity of walks is amortized over the complexity of the relaxation operations – the condition *WTR_amortization* becomes true every n-th time it is called.
- Vertices through which walk passes are marked by the couple [origin, stamp]. The origin is the vertex where the walk has started and the stamp is the actual value of the counter of started walks on particular processor.
- The walk can reach a vertex already marked by some other walk.
  - If the vertex is marked by the actual origin and stamp then a cycle is detected (line 6). The cycle can be easily reconstructed by following parent edges. If necessary, the path connecting the cycle with the source vertex can be found using a suitable distributed reachability algorithm.
  - If the vertex is marked by lower origin or by the actual origin but lower stamp (line 15) then the walk overwrites the mark by actual mark and continues.
  - If the vertex is marked by higher origin (line 9) then the walk stops and the *REM* procedure is started.
- Marks are removed by the *REM* procedure.

```
1  proc Main
2    while not finished do
3          req := pop(queue);
4          Trace (req.vertex, req.father, req.length);
5    od
6  end
```

```
1  proc Trace (v, father, length)
2    if d(v) ≤ length then exit fi
3    p(v) := father; d(v) := length;
4    while v ≠ father do
5          Handle_messages;
6          u := RGS(v, NULL);
7          while u does not exist do
8                last := v; v := p(v);
9                u := RGS(v, last); od
10         v := u;
11   od
12 end
```

```
1  proc RGS (v, last) {Relax and Get Successor}
2    u := successor of v greater than last;
3    while u exists do
4          if u is local then
5                      if d(u) > d(v) + l(u, v) then
6                        if mark(u) then wait; fi
7                        p(u) := v;
8                        d(u) := d(v) + l(u, v);
9                        if WTR_amortization then WTR([u, stamp], u); inc(stamp); fi
10                       return u;
12                     fi
13                 else send_message(owner(u), "update u, v, d(u) + l(u, v)");
14         fi
15         u := next successor of v;
16   od
17   return u does not exist;
18 end
```

```
1  proc WTR ([origin, stamp], at) {Walk To Root}
2    done := false;
3    while ¬done do
4          if at is local
5            then
6                if mark(at) = [origin, stamp] →
7                  send_message(Manager, "negative cycle found");
8                  terminate
9                □ at = source ∨ mark(at) > [origin, stamp] →
```

```
10                    if origin is local
11                      then REM([origin, stamp], origin)
12                       else send_message(owner(origin),
13                                  "start REM([origin, stamp], origin))" fi
14                    done := true;
15                 □  mark(at) = nil ∨ mark(at) < [origin, stamp] →
16                    mark(at) := [origin, stamp];
17                    at := p(at)
18                 fi
19            else send_message(owner(at), "start WTR([origin, stamp], at)");
20                 done := true
21         fi
22   od
23 end
```

```
1  proc REM([origin, stamp], at) {Remove Marks}
2    done := false;
3    while ¬done do
4          if at is local
5            then if walk(at) = [origin, stamp]
6                    then walk(at) := [nil, nil];
7                         at := p(at)
8                     else
9                         done := true fi
10           else send_message(owner(at), start REM([origin, stamp], at));
11                done := true fi
12   od
13 end
```

```
1  proc Handle_messages
2    if message = req then
3                          if d(req.vertex) > req.length then push_queue(req); fi
4    fi
5    if message = start WTR then WTR; fi
6    if message = start REM then REM; fi
7  end
```

Whenever a processor is about to process a vertex (during traversing or walk to root) it checks whether the vertex belongs to its own subgraph. If the vertex is local, the processor continues locally otherwise a message is sent to the owner of the vertex. The algorithm periodically checks incoming messages (line 5 of *Trace*). When a request to update parameters of a vertex u arrives, the processor compares the current value $d(u)$ with the received one. If the received value is lower than the current one then the request is placed into the local *queue*. Whenever the traversal ends the next request from the *queue* is popped and a new traversal is started. Another type of message is a request to continue in the walk to root (resp. in removing marks), which is immediately satisfied by executing the *WTR* (resp. the *REM*) procedure.

The distributed algorithm terminates when all local *queues* of all processors are empty and there are no pending messages or when a negative cycle is detected. The *Manager* process is used to detect the termination and to finish the algorithm by sending a *termination* signal to all processors.

In what follows we prove the correctness of the algorithm. The proof is similar to the sequential one but is more technically involved.

Lemma 4. *Let G be a given graph. Then $d(v) \geq \delta(s, v)$ for all $v \in V$ and this invariant is maintained over any sequence of relaxations performed on the graph G. Moreover, once $d(v)$ achieves its lower bound $\delta(s, v)$, it never changes.*

Lemma 5. *Let G contains no reachable negative cycle. Then the parent graph G_p forms a rooted tree with root s, and any sequence of relaxations on G maintains this property invariantly true.*

Lemma 6. *Let G contains no negative cycle reachable from the source vertex s. Every time a vertex w is visited the sequence* S *of the assignments on line 8 of the procedure* Trace *will eventually be executed for this vertex. Until this happens $p(w)$ is not changed.*

Proofs of Lemmas 4-6 are analogical to the proofs of their sequential counterparts.

Lemma 7. *The algorithm visits a vertex u if and only if the value $d(u)$ has been decreased.*

Proof: "$\Rightarrow$" Let us suppose that vertex u is visited, i.e. it is the value of the variable v in the while cycle of the *Trace* procedure. This may happen in two ways. At first, the procedure *Trace* has been called with the parameter u. Due to the test at the beginning of the procedure, value $d(u)$ has been decreased by the assignment on line 3. Secondly, the variable v may be set to u during the previous iteration of the while cycle. This may happen only if the *RGS* procedure has returned u and consequently $d(u)$ has been decreased on line 8 of the *RGS* procedure.

"$\Leftarrow$" Let us suppose that $d(u)$ has been decreased. This may happen only on line 3 of the *Trace* procedure or on line 8 of the *RGS* procedure. In both cases, vertex u is visited immediately afterwards.

Note that the algorithm can not visit vertex with the same $d(v)$ twice.

■

Lemma 8. *Every walk takes at most $O(n)$ time.*

Proof: A walk can visit each vertex only once – the linear ordering on walks prevents multiple visits of the vertex by the same walk. Since there are n vertices, walk can take at most $O(n)$ time. ■

Lemma 9. *If G contains no reachable negative cycle then the distributed algorithm terminates with $d(v) = \delta(s, v)$ for all $v \in V$ and G_p forms a shortest-paths tree rooted at s.*

Proof: In the absence of negative cycles algorithm terminates when all local queues are empty and there are no pending messages. Every vertex is placed into a queue only after the value $d(v)$ has been decreased. This can happen at most $2 \cdot C \cdot (n-1)$ times (see the proof of Theorem 2). After removing the vertex v from a queue the procedure $Trace(v)$ is executed. Lemma 6 applied to the vertex v gives the termination of the *Trace* procedure. Therefore, all queues will become empty in finite time and the algorithm will terminate.

The arguments, that $d(v) = \delta(s, v)$ and that G_p forms shortest path tree are the same as in the sequential case (see the proof of Theorem 1). ■

Lemma 10. *If the procedure* WTR *detects a cycle, then there is a negative cycle in G.*

Proof: The walk initiated in a vertex u with *stamp* i detects a cycle by finding a vertex marked with the same *origin* u and *stamp* i. The equality of stamps ensures that this vertex has already been visited during the current walk. From Lemma 5 it follows that any cycle in G_p is a negative cycle in G. ■

Lemma 11. *If G contains a reachable negative cycle then after finite time either a cycle is detected or for every visited vertex v holds that v lies on cycle or on a path to cycle in G_p.*

Proof: Let us suppose that cycle was not detected. According to lemma 7 each visit of vertex v is consequence of lowering of $d(v)$. Hence after finite time, all $d(v)$'s are either settled and v is not visited any more or $d(v)$ is lower than $C(n-1)$. Because there could not exist simple path in G_p from s to v if $d(v) < C(n-1)$, each such vertex has to lie either on a cycle or on a path to a cycle. ■

Lemma 12. *If G contains a reachable negative cycle then some procedure* WTR *detects the cycle.*

Proof: According to Lemma 11 after finite time the cycle is either already detected or for every visited vertex v holds that v lies on a cycle or on a path to cycle in G_p. In the second case, after at most n steps a walk will be started. This walk will either detect cycle or reach a vertex that is marked by some higher walk. However, in this case the cycle in the parent graph is the same for the second walk. We now have a similar situation as before for the walk started at higher origin. As the number of possible origins is finite, there has to be a walk that will not be terminated by another one and that will detect the cycle. ■

Theorem 3 (Correctness of the distributed algorithm). *If G contains no reachable negative cycle then the distributed algorithm terminates with $d(v) = \delta(s, v)$ for all $v \in V$ and G_p forms a shortest-paths tree rooted at s. If G has a negative cycle, its existence is reported.*

Proof: Lemma 9 gives the correctness for the case that G contains no negative cycle. Lemmas 10 and 12 give the correctness of the other case.

Theorem 4 (Complexity of the distributed algorithm). *The worst time complexity of the algorithm is $\mathcal{O}(Cn^3)$.*

Proof: Complexity is $\mathcal{O}(Cn^3)$ due to the amortization of the walk to root, other arguments are the same as for the sequential case (see proof of Theorem 2). ■

6 Experiments

We have implemented the distributed algorithm (*DSP-R*) in C++ and the experiments have been performed on a cluster of eight 366 MHz Pentium PC Linux workstations with 128 Mbytes of RAM each interconnected with a fast 100Mbps Ethernet and using Message Passing Interface (MPI) library.

Our objective was to compare the performance of our algorithm with other algorithms on particular types of graphs that represent the *LTL model checking problem*. Automata based approach to model-checking of linear temporal logic (LTL) formulas is a very elegant method developed by Vardi and Wolper [VW86]. The basic idea is to associate with each LTL formula a Büchi automaton that accepts exactly all the computations that satisfy the formula. At the same time, states of modelled finite-state system are indentified with states of a Büchi automaton. This enables the reduction of the model-checking problem to the non-emptiness problem

for Büchi automata. It can be shown that non-emptiness problem for Büchi automata is equivalent to the problem of finding a cycle that is reachable from the initial state and contains an accepting state.

The connection between the negative cycle problem and the Büchi automaton emptiness problem is following. A Büchi automaton corresponds to a directed graph. Let us assign lengths to its edges in such a way that all edges out-coming from vertices corresponding to accepting states have length -1 and all others have length 0. With this length assignment, negative cycles simply coincide with accepting cycles and the problem of Büchi automaton emptiness reduces to the negative cycle problem.

The algorithm used in LTL model-checkers is very effective nested depth first search (*NDFS*) algorithm [HPY96]. For instance, SPIN verification tool [Hol97] uses this algorithm. In its distributed version the graph is divided over processors like in the *DSP* algorithm. Only one processor, namely the one owning the actual vertex in the *NDFS* search, is executing the nested search at a time. The network is in fact running the sequential algorithm with extended memory.

At the same time we wanted to compare our algorithm with similar algorithm [BCKP01] which does not employ reverse search (*DSP-Q*), but which traverses graph using the queue (classical Bellman-Ford approach [For56,Bel58]). The distribution of *DSP-Q* algorithm is the very same as the distribution of *DSP-R* algorithm. Our aim was to document, that avoiding the additional data structure (such as the queue in this case) does not impact the running time.

In the implementation of the *DSP-R* and the *DSP-Q* algorithms we have employed the following optimization scheme. For more efficient communication between processors we do not send separate messages. The messages are sent in packets of pre-specified size. The optimal size of a packet depends on the network connection and the underlying communication structure. In our case we have achieved the best results for packets of size around 100 single messages.

Our experiments were performed on three kinds of systems given by random graphs, product graphs and generated graphs. Graphs were generated using a simple specification language and an LTL formula. In all the cases we tested graphs with and without cycles to model faulty and correct behavior of systems. As our real example we tested the parametrised Dining Philosophers problem. Each instance is characterized by the number of vertices and the number of cross-edges. The number of cross-

edges significantly influences the overall performance of distributed algorithms.

For each experiment we report the average time in minutes as the main metric. Table 1 summarizes the achieved results.

Vertices	Cross–edges	NDFS	DSP-Q	DSP-R
Generated, without cycle				
40398	34854	1:01	0:11	0:09
71040	1301094	31:13	0:48	0:42
696932	1044739	27:02	1:31	1:36
736400	5331790	126:46	5:17	4:25
777488	870204	21:36	2:02	2:05
1859160	1879786	49:04	6:00	4:39
Generated, with cycle				
18699	22449	0:06	0:05	0:04
33400	2073288	0:37	0:24	0:22
46956	83110	0:05	0:09	0:09
Product, without cycle				
20360	783170	18:16	0:10	0:13
35300	2634667	63:43	2:05	0:35
71040	1301094	31:13	0:48	0:42
736400	5331790	126:46	5:17	4:25
Product, with cycle				
33400	2073288	0:37	0:24	0:22
Random, without cycle				
4000	353247	14:03	0:17	0:06
5000	839679	31:48	0:32	0:21
80000	522327	30:11	1:39	0:40
60000	1111411	57:19	4:08	1:07
Random, with cycle				
18000	1169438	1:20	0:09	0:09
Philosophers				
(12)	94578	42154	2:06	0:13	0:15
(14)	608185	269923	16:11	1:40	1:35

Table 1. Summary of experimental results

The experiments lead basically to the following conclusions:

- *DSP-R* algorithm is comparable with the *NDFS* one on all graphs.
- *DSP-R* algorithm is significantly better on graphs without negative cycles.
- *DSP-R* algorithm is practically the same as the *DSP-Q* one on all graphs.

Experiments show that in spite of worse theoretical worst time complexity of *DSP-R* algorithm its behavior in practice can outperform the theoretically better *NDFS* one. This is due to the number of communications which has essential impact on the resulting time. In *DSP-R* algorithm the messages can be grouped into packets and sent together. It is a general experience that the time needed for delivering t single messages is much higher than the time needed for delivering those messages grouped into one packet. On the other hand, *NDFS* algorithm does not admit such a grouping. Another disadvantage of *NDFS* is that during the passing of messages all the processors are idle, while in *DSP-R* algorithm the computation can continue immediately after sending a message. Last but not least, in *NDFS* all but one processor are idle whereas in *DSP-R* all can compute concurrently. We notice that all mentioned advantages of *DSP-R* algorithm demonstrate themselves especially for systems without cycles where the whole graph has to be searched. This is in fact the desired property of our algorithm as the state explosion demonstrates itself just in these cases. Both algorithms perform equally well on graphs with cycles. When comparing the results of the *DSP-R* algorithm and the *DSP-Q* one we can conclude that avoiding the additional data structure for the traversal of the graph does not impact the running time. More significant differences can be explained by different traversal order - the *DSP-R* algorithm performs quasi-depth-first traversal and the *DSP-Q* algorithm performs quasi-breath-first traversal. Practical running time of these two approaches can differ, but it depends on the particular graphs.

We have accomplished yet another set of tests (see Table 2) in order to validate the scalability of the *DSP-R* algorithm. The table shows how the number of computers influences the computation time. Time is given in minutes, 'M' means that the computation failed due to low memory. The tests confirm that it scales well, i.e. the overall time needed for treating a graph is decreasing as the number of involved processors is heightened.

	Computers						
Vertices	1	2	3	4	5	6	7
94578	0:38	0:35	0:26	0:21	0:18	0:17	0:15
608185	5:13	4:19	3:04	2:26	2:03	1:49	1:35
777488	M	6:50	4:09	3:12	2:45	2:37	2:05
736400	M	M	M	6:19	4:52	4:39	4:25

Table 2. Scalability over the number of the computers

7 Conclusions

We have proposed a distributed algorithm for the single source shortest paths problem for arbitrary directed graphs which can contain negative length cycles. The algorithm employs reverse search and uses one data structure for two purposes — computing the shortest paths and traversing the graph. A novel distributed variant of the walk to root negative cycle detection strategy is engaged. The algorithm is thus space-efficient and scalable.

Because of the wide variety of relaxation and cycle detection strategies there is plenty of space for future research. Although not all strategies are suitable for distributed solution, there are surely other possibilities besides the one proposed in this paper.

References

[AF96] D. Avis and K. Fukuda. Reverse search for enumeration. *Discrete Appl. Math.*, 65:21–46, 1996.

[BCKP01] L. Brim, I. Cerná, P. Krčál, and R. Pelánek. Distributed shortest path for directed graphs with negative edge lengths. Technical Report FIMU-RS-2001-01, Faculty of Informatics, Masaryk University Brno, 2001.

[Bel58] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[CG99] B. V. Cherkassky and A. V. Goldberg. Negative-cycle detection algorithms. *Mathematical Programming, Springer-Verlag*, 85:277–311, 1999.

[CLR90] T. H. Cormen, Ch. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT, 1990.

[CMMS98] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In *Proc. 23rd MFCS'98, Lecture Notes in Computer Science*, volume 1450, pages 722–731. Springer-Verlag, 1998.

[Coh96] E. Cohen. Efficient parallel shortest-paths in digraphs with a separator decomposition. *Journal of Algorithms*, 21(2):331–357, 1996.

[CZ95] S. Chaudhuri and C. D. Zaroliagis. Shortest path queries in digraphs of small treewidth. In *Automata, Languages and Programming*, pages 244–255, 1995.

[For56] L.R. Ford. Network flow theory. Rand Corp., Santa Monica, Cal., 1956.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special Issue: Formal Methods in Software Practice.

[HPY96] G.J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *The Spin Verification System*, pages 23–32. American Mathematical Society, 1996. Proc. of the Second Spin Workshop.

[KPSZ94] D. Kavvadias, G. Pantziou, P. Spirakis, and C. Zaroliagis. Efficient sequential and parallel algorithms for the negative cycle problem. In *Proc. 5th ISAAC'94, LNCS*, volume 834, pages 270–278. Springer, 1994.

[MS00] U. Meyer and P. Sanders. Parallel shortest path for arbitrary graphs. In *6th International EURO-PAR Conference*. LNCS, 2000.

[Nie00] J. Nievergelt. Exhaustive search, combinatorial optimization and enumeration: Exploring the potential of raw computing power. In *SOFSEM 2000*, number 1963 in LNCS, pages 18–35. Springer, 2000.

[RV92] K. Ramarao and S. Venkatesan. On finding and updating shortest paths distributively. *Journal of Algorithms*, 13:235–257, 1992.

[TZ96] J. Traff and C.D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. In *Parallel algorithms for irregularly structured problems*, volume 1117 of *LNCS*, pages 183–194. Springer, 1996.

[Vit98] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. In *Proc. of 6th European Symposium on Algorithms (ESA '98)*, pages 1–25, 1998.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *1st Symp. on Logic in Computer Science, LICS'86*, pages 332–344. Computer Society Press, 1986.

**Copyright © 2001, Faculty of Informatics, Masaryk University.
All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

Publications in the FI MU Report Series are in general accessible via WWW and anonymous FTP:

`http://www.fi.muni.cz/informatics/reports/`
`ftp ftp.fi.muni.cz (cd pub/reports)`

Copies may be also obtained by contacting:

**Faculty of Informatics
Masaryk University
Botanická 68a
602 00 Brno
Czech Republic**