FI MU

Faculty of Informatics
Masaryk University

Distributed Shortest Paths for Directed Graphs with Negative Edge Lengths

by

Luboš Brim
Ivana Černá
Pavel Krčál
Radek Pelánek

FI MU Report Series FIMU-RS-2001-01

May 2001

Distributed Shortest Paths for Directed Graphs with Negative Edge Lengths*

Luboš Brim, Ivana Černá, Pavel Krčál and Radek Pelánek

Faculty of Informatics
Masaryk University Brno
Botanická 68a
Czech Republic

{brim,cerna,xkrcal,xpelanek}@fi.muni.cz

Abstract

A distributed algorithm for single source shortest path problem in an arbitrary directed graph which can contain negative length cycles is presented. The new algorithm is a label-correcting one and uses a novel way for detection of negative length cycles. It works on a network of processors with disjoint memory that communicate via message passing. Correctness of the algorithm is proved. The algorithm is work-efficient as its worst time complexity is $\mathcal{O}(n^2.\frac{n}{P})$, where P is the number of processors. As an application a simple distributed algorithm for LTL model checking is presented.

Keywords: directed graphs, single source shortest paths, negative cycles, distributed algorithms

1 Introduction

The single source shortest path problem (SSSP) is a fundamental problem with many theoretical and practical applications and plenty of effective and well-grounded sequential algorithms. However, in some applications we have to deal with extremely large graphs (a particular application we have

*This work has been partially supported by the Grant Agency of Czech Republic grants No. 201/00/1023 and 201/00/0400.

in mind is briefly discussed bellow). Whenever a graph is too large to fit into memory that is randomly accessed a memory that is sequentially accessed has to be employed. This causes a bottleneck in the performance of a sequential algorithm owing to the significant amount of paging involved during its execution. An usual approach to deal with these practical limitations is to increase the computational power (especially randomly accessed memory) by building a powerful (yet cheap) parallel computer as a network of workstations (NOWs). Individual workstations communicate through message-passing interface such as MPI. From outside, a NOW appears as one single parallel computer with high computing power and huge amount of memory. We stress, that in this structure the available memory is not shared but is distributed among individual processors. Hence, it is an essential goal to find efficient distributed algorithms.

The natural starting point for building a distributed algorithm is to distribute a sequential one. Effective PRAM parallelisations are known for algorithms working with adjacency matrix graph's representation (Floyd's algorithms, see e.g. [Roo00]) but their efficiency hardly depends on assumptions that all processors work synchronously and share common memory. These assumptions are not valid in a distributed environment. Other algorithms (for excellent survey see [CG99]), which are based on relaxation of graph's edges, are inherently sequential and their parallel versions are known only for special settings of the problem. For general digraphs with non-negative edge lengths parallel algorithms are presented in [MS00, RV92, CMMS98] together with studies concerning good decompositions [HTB97]. For special cases of graphs, like planar digraphs [TZ96, DKZ94], graphs with separator decomposition [Coh96] or graphs with small treewidth [CZ95] more efficient algorithms are known. Yet none of these algorithms are applicable on general digraphs with potential negative-length cycles.

This paper presents a distributed algorithm for the SSSP problem on graphs with real edge lengths. The algorithm comes out from the Bellman-Ford algorithm and uses a novel way for detection of negative cycles, allowing thus to employ the full strength of parallelism. We prove the correctness of the algorithm and analyse its worst-case complexity.

Our motivation for this work was to develop a distributed model checking algorithm for linear temporal logic. Despite the development in the last years the so called *state space explosion* still limits practical applications of model checking techniques. We have shown that model checking problem for linear temporal logic can be reduced to the negative cycle detection problem in directed graphs with arbitrary length edge [BČKP01]. In our search for effective distributed algorithm for this problem we wanted to preserve its compatibility with another efficient technique dealing with the state space explosion, namely *on the fly* way of graph generation. Here the graph to be processed is not completely given at the beginning of the computation through its adjacency-list or adjacency-matrix representation. Instead, we are given a source vertex together with a function which for every vertex computes its adjacency-list. As successors of a vertex are determined dynamically there is no need to store any information about edges permanently. Moreover, the technique allows to generate only reachable part of the graph and thus reduces the space requirements for graph representation.

The structure of the paper is the following. In the next section we briefly summarise the necessary background. In Section 3 we present the distributed algorithm and in Section 4 we formally prove its correctness and complexity. Experimental results are discussed in Section 5. Concluding remarks are presented in Section 6.

2 Basic Notations and Definitions

We are given a triple (G, s, l), where $G = (V, E)$ is a directed graph with n vertices and m edges, $l : E \to R$ is a *length function* mapping edges to real-valued lengths, and $s \in V$ is the *source* vertex. The *length of the path* $\rho = < v_0, v_1, \ldots, v_k >$ is the sum of the lengths of its constituent edges, $l(\rho) = \sum_{i=1}^{k} l(v_{i-1}, v_i)$. We define the *shortest path length* from s to v by

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

A *shortest path* from vertex s to vertex v is then defined as any path ρ with length $l(\rho) = \delta(s, v)$. If the graph G contains no cycle with negative length (*negative cycle*) that is 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*. If there is a negative cycle reachable from s, shortest paths are not well-defined as no path from s to a vertex on the cycle can be a shortest path. If there is a negative cycle on some path from s to v, we define $\delta(s, v) = -\infty$.

The SSSP problem is to decide whether, for a given triple (G, s, l), the graph G is feasible and if it is then to compute the shortest paths from the source vertex s to all vertices $v \in V$. The negative cycle problem is to decide whether G is feasible.

The general sequential method for solving the SSSP problem is the *scanning* method. For every vertex v, the method maintains its distance label $d(v)$, parent vertex $p(v)$ and status $S(v) \in \{unreached, labelled, scanned\}$. 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*. Initially for every vertex v, $d(v) = \infty, p(v) = nil$ and $S(v) = unreached$. The method starts by setting $d(s) = 0$, $p(s) = nil$ and $S(s) = labelled$. At every step, the method selects a *labelled* vertex v and applies on it the *SCAN* operation (see Figure 1). During scanning a vertex v the edges out-coming from v are *relaxed* which means that if $d(u) > d(v) + l(v, u)$ then $d(u)$ is set to $d(v) + l(v, u)$ and $p(u)$ is set to v. The status of v is changed to *scanned* while the status of u is changed to *labelled*. If all vertices are either *scanned* or *unreached* then d gives the shortest path lengths and G_p is the graph of shortest paths.

```
proc SCAN(v)
  for each edge (v,u) ∈ E do {relax(v,u)}
     if d(u) > d(v) + l(v,u) then d(u) := d(v) + l(v,u); p(u) := v;
                                  S(u) := labelled fi od;
  S(v) := scanned
end
```

Figure 1: Scanning of vertex v

Different strategies for selecting a labelled vertex to be scanned next lead to different algorithms. The optimal strategy is to scan vertices with exact distance label, i.e. vertices for which $d(v) = \delta(s, v)$. However, efficient implementations of this idea are known only for special problem settings as graphs with non-negative length edges (Dijkstra's algorithm [Dij59]) or directed acyclic graphs (Lawler's algorithm [Law76]). In comparison with these special, so called *label-setting* algorithms, algorithms for the general problem are called *label-correcting*.

Label-correcting SSSP algorithms may select arbitrary labelled vertex to be scanned next, hence generally to re-insert the vertices for scanning until they are finally settled ($d(v) = \delta(s, v)$). The well-known Bellman–Ford algorithm [Bel58, For56] uses FIFO strategy to select vertices. The next vertex to be scanned is removed from the head of the queue; a vertex that becomes labelled is added to the tail of the queue if it is not already in the queue. The algorithm runs in $\mathcal{O}(mn)$ time in the worst case.

For graphs where negative cycles could exist the scanning method must be modified in such a way, that in the presence of a negative cycle the method detects such a cycle and terminates. As in the case of scanning, various strategies are used to detect negative cycles. For our distributed algorithm we have used the *walk to root* cycle detection strategy. This strategy is based on the following fact (see e.g. [CG99]).

Fact 1 *Any cycle in the parent graph G_p has negative length. If G_p is acyclic, then its edges form a tree rooted at s.*

The *walk to root* method tests whether G_p is acyclic. Suppose the scanning operation applies to an edge (v, u) (i.e. $d(u) > d(v) + l(v, u)$) and the parent graph G_p is acyclic. This operation will create a cycle in G_p if and only if u is an ancestor of v in the current tree. Before applying the operation, we follow the parent pointers from v until we reach u or s. If we stop at u we have found a negative cycle; otherwise, the scanning operation does not create a cycle.

The *walk to root* method gives immediate cycle detection and can be easily combined with various scanning heuristics. However, since the path to the root can be long, the cost of applying the scanning operation to an edge becomes $\mathcal{O}(n)$ instead of $\mathcal{O}(1)$. In order to optimise the overall computational complexity we propose to use amortisation to pay the cost of checking G_p for cycles. More precisely, the parent graph G_p is tested only after the underlying shortest paths algorithm performs $\Omega(n)$ work. The running time is thus increased only by a constant factor.

However, a cycle in G_p can appear after a relaxation of some edge and disappear after some later relaxation (see Figure 2). The correctness of the amortised strategy is based on the following fact (see e.g. [CG99]).

Fact 2 *If G contains a negative cycle reachable from s, then after a finite number of scanning operation G_p always has a cycle.*

Another drawback of the amortised strategy is the fact that even if the relaxation of an edge (v, u) does not create a cycle in G_p, there can be a cycle on the way from v to s. The particular modification of the *walk to root* strategy allowing to cope with this situation is presented in the next section.

We stress that the presented sequential approaches to the SSSP problem and especially to the negative cycle detection problem are not the only ones - for an excellent survey see [CG99]. Those presented above are in our opinion the best starting points for developing a distributed algorithm.

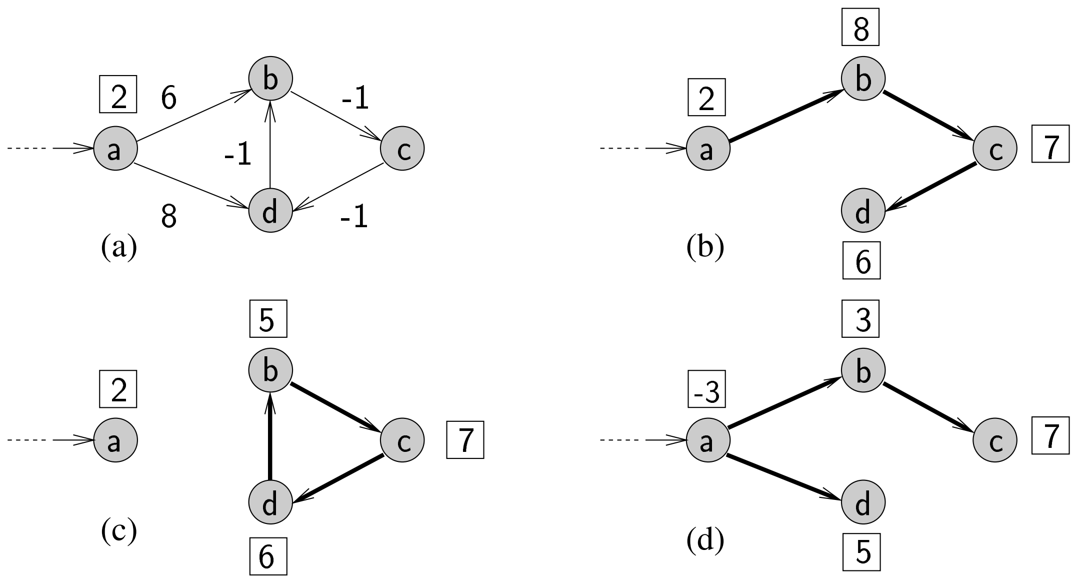

Figure 2: Disappearing cycles in the parent graph. (a) Part of the input graph. Vertex a has distance label 2. (b) G_p after scanning vertices a, b, c. (c) Scanning of the vertex d. (d) The distance label of the vertex a is decreased to -3 and the vertex a is scanned once more.

3 Distributed SSSP Algorithm

In this section we describe a new algorithm that works in a distributed environment and solves the SSSP problem for arbitrary directed graphs that can contain negative cycles. The algorithm runs on a network of P processors which communicate through some message-passing interface. One of the processors is distinguished as the *Manager* processor and is responsible for starting and terminating the computation. All processors perform the same program. The set of vertices is supposed to be *a priori* divided into several disjoint subsets. The number of subsets corresponds to the number of processors involved in the computation. The partition of vertices is given by the function *owner* which assigns a vertex to a processor. Each processor α is responsible for its own part of the graph G determined by the owned subset of vertices. Good partition of vertices among processor can evidently affect the overall computational complexity but its construction is not a subject of our interest in this paper. The function *successor* gives for a vertex its adjacency list. It allows to construct the graph *on the fly* (see Section 1). Each processor knows the functions *owner* and *successor*. Moreover, we suppose that some total linear order on vertices is given (induced e.g.

from the numerical representation of vertices).

The pseudo-code of the distributed algorithm is presented below. Its main idea can be summarised as follows. Each processor performs repeatedly the basic scanning operation on all owned vertices with labelled status (the *MAIN* procedure). Such vertices are maintained in the processor's local *queue* Q^α. To relax a *cross-edge* (v, u), where v and u belong to different processors, a message is sent to the owner of u. In each iteration the processor at first handles messages received from other processors.

Pseudo-Code of the Distributed Algorithm *DSP*

```
proc MAIN() {running on each processor α}
  stamp := 0;
  if α = Manager then Q^α = {s}; d(s) := 0; p(s) := nil else Q^α := ∅ fi
  while not finished do process_messages; v := pop(Q^α); SCAN(v) od
end

proc SCAN(v)
  foreach (v, u) ∈ E do
    if owner(u) = α
      then UPDATE(u, v, d(v) + l(v, u))
      else send_message(owner(u), "start UPDATE(u, v, d(v) + l(v, u))") fi od
end

proc UPDATE(u, v, t)
  if d(u) > t then if walk(u) ≠ nil
                     then if owner(v) = α
                            then push(Q^α, v)
                            else send_message(owner(v), "do push(Q, v)") fi
                     else d(u) := t; p(u) := v;
                          if WTR_amortization then WTR([u, stamp], u);
                                                   stamp++ fi;
                          if u ∉ Q^α then push(Q^α, u) fi fi fi
end


proc WTR([origin, stamp], at) {Walk To Root}
  done := false;
  while ¬done do
        if owner(at) = α
          then
              if walk(at) = [origin, stamp] →
                 send_message(Manager, "negative cycle found");
                 terminate
              ▯ (at = source) ∨ (walk(at) > [origin, stamp]) →
```

```
                    if owner(origin) = α
                      then REM([origin, stamp], origin)
                      else send_message(owner(origin),
                                  "start REM([origin, stamp], origin))" fi
                    done := true;
                  ▯ (walk(at) = [nil, nil]) ∨ (walk(at) < [origin, stamp]) →
                    walk(at) := [origin, stamp];
                    at := p(at)
                  fi
            else
                  send_message(owner(at), "start WTR([origin, stamp], at)");
                  done := true
        fi
  od
end
```

```
proc REM([origin, stamp], at) {Remove Marks}
  done := false;
  while ¬done do
        if owner(at) = α
          then if walk(at) = [origin, stamp]
                  then walk(at) := [nil, nil];
                       at := p(at)
                  else
                       done := true fi
          else send_message(owner(at), start REM([origin, stamp], at));
                  done := true fi
  od
end
```

For the detection of a negative cycle in the graph G the procedure *WTR* is used in an amortised manner. The processor starts the detection only after it has relaxed n edges. *WTR* tries to identify a cycle in the parent graph by following the parent pointers from a current vertex.

As the relaxation of edges is performed in parallel and the cycle detection is not initiated after every change in the parent graph, it can happen that even if the relaxation of an edge (v, u) does not create a cycle in the parent graph G_p there can be a cycle in G_p on the way from v to the source s (see Figure 3). In order to recognise such a cycle we introduce for each vertex x a new variable *walk(x)*. Its initial value is nil. Once the *WTR* procedure started in a vertex (let us call this vertex *origin*) passes through a vertex v, the value of the variable *walk(v)* is set to *origin* (we say that the vertex v has been *marked*). Reaching a vertex already marked with the value *origin* clearly indicates a cycle in the parent graph. However, it can happen that more than one *WTR* procedure is active at a time. Consequently *WTR* initiated in *origin* can get to a vertex marked with a value different from

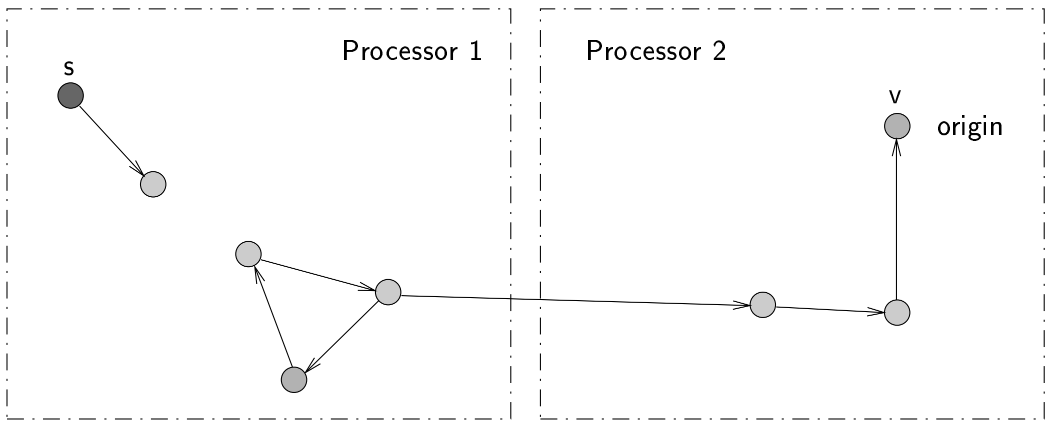

Figure 3: Cycle on the path to the source

origin indicating that some other *WTR* is active. Such a collision is resolved in favour of the procedure which has started in the higher vertex (with respect to the linear ordering on vertices). *WTR* started in the lower vertex is terminated. Another situation that could happen is that *WTR* started in *origin* gets to a vertex already marked with *origin* but this mark has been introduced by some previous *WTR* initiated from *origin*. This would lead to a detection of a false cycle (see Figure 4). To guarantee correctness even in this special situation we in fact mark every vertex with two values: *origin* and *stamp* (the initial value of *walk(x)* beeing $[nil, nil]$). The value *stamp* is equal to the number of *WTR* procedures initiated by the particular processor. This value allows to distinguish among vertices marked by current and some previous *WTR* initiated from the same vertex.

Let us summarise the four possible situations which can happen in cycle detection.

- the procedure *WTR* reaches the source vertex s (line 9 in *WTR*). A negative cycle has not been detected and the *REM* procedure is started.

- the procedure *WTR* reaches a vertex marked with the same *origin* and the same *stamp* (line 6). This indicates that a negative cycle has been recognised. The cycle can be easily reconstructed by following the parent edges. If necessary, the path connecting the cycle with the source vertex can be found using a suitable distributed reachability algorithm.

- the procedure *WTR* reaches a non-marked vertex, a vertex already marked with lower *origin* or a vertex marked with the same *origin* but

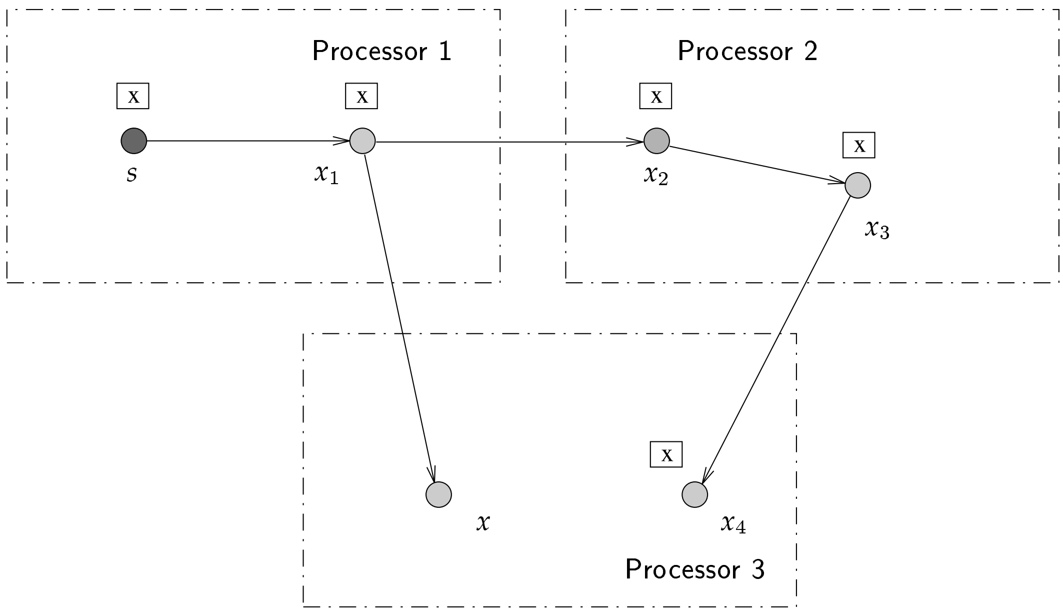

Figure 4: Detection of a false cycle. Vertices s, x_1, x_2, x_3, x_4 have been marked by the value x. After that the parent pointer of x has been changed to x_1 and a new *WTR* is started from the vertex x.

lower *stamp* (line 15). The vertex is marked with [*origin*, *stamp*] and the walk follows the parent edge.

- the procedure *WTR* reaches a vertex already marked with higher *origin* (line 9). The walk is stopped and the *REM* procedure is started.

Whenever *WTR* has to continue in a non-local vertex a request to the vertex owner is sent and the local walk is finished.

Once the *WTR* procedure finishes without detecting a cycle, the marks introduced by it must be removed (i.e. the value of the variable *walk* is reset to [*nil*, *nil*]). A non-removed mark y can prevent detection of a cycle by *WTR* which has started in a vertex x where $x < y$.

The *REM* procedure is responsible for removing marks. Marks to be removed are found with the help of parent edges. This is why the updating of a marked vertex is postponed (line 2 of the procedure *UPDATE*). The *REM* procedure follows the path in the parent graph in a similar way as *WTR* does. *REM* starts from the *origin* and terminates after reaching a source vertex or a vertex with different mark.

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

4 Correctness and Complexity

The formal proof of the correctness of the distributed algorithm comes out from several lemmas. Some of the arguments are similar to that for the sequential algorithms (see e.g.[CLR90]). The main difference in the distributed case is that vertices are scanned in parallel, hence the edges can be relaxed concurrently. For the correctness argumentation we will suppose a *fixed but arbitrary linear ordering* of concurrently performed steps.

Lemma 1 *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.*

Proof: The invariant $d(v) \geq \delta(s,v)$ is true after the initialisation performed on each processor. We shall use the proof by contradiction to show that the invariant is maintained over any linearly ordered sequence of relaxation steps performed on G. Suppose that $v \in V$ is the first vertex (in the sense of linear ordering of relaxation steps) for which $d(v) < \delta(s,v)$. Then there is a vertex $u \in V, u \neq v$ (in fact $u = p(v)$) such that

$$d(u) + l(u,v) = d(v) < \delta(s,v) \leq \delta(s,u) + l(u,v)$$

which implies that $d(u) < \delta(s,u)$. Then we have a contradiction, because the relaxation of the edge (u,v) does not change the value $d(u)$ and v was the first vertex violating the invariant.

To see that the value of $d(v)$ never changes once $d(v) = \delta(s,v)$ note, that relaxation steps do not increase d values. ■

Lemma 2 *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.*

Proof: Initially, the only vertex in G_p is the source vertex s, and the lemma is trivially true.

We first prove that G_p forms a tree. Consider a parent graph G_p that arises after an arbitrary sequence of relaxations and suppose that G_p has a cycle. Let the cycle be $c =< v_0, v_1, \ldots, v_k >$, where $v_k = v_0$. Then $p(v_i) = v_{i-1}$ for $i = 1, 2, \ldots, k$ and, without loss of generality, we can assume that it was the relaxation of the edge (v_{k-1}, v_k) that created the cycle in G_p. Each vertex on the cycle c is reachable from s. We show that c is a negative cycle, thereby contradicting the assumption that G contains no negative cycles reachable from s. Before relaxing the edge (v_{k-1}, v_k) we had $d(v_i) \geq d(v_{i-1}) + l(v_{i-1}, v_i)$ for all $i = 1, 2, \ldots, k-1$. Because $p(v_k)$ has changed, immediately beforehand we also have the strict inequality $d(v_k) > d(v_{k-1}) + l(v_{k-1}, v_k)$ from which we obtain

$$\sum_{i=1}^{k} d(v_i) > \sum_{i=1}^{k} (d(v_{i-1}) + l(v_{i-1}, v_i)) = \sum_{i=1}^{k} d(v_{i-1}) + \sum_{i=1}^{k} l(v_{i-1}, v_i).$$

But $\sum_{i=1}^{k} d(v_i) = \sum_{i=1}^{k} d(v_{i-1})$ since each vertex in the cycle c appears exactly once in each summation. This implies $0 > \sum_{i=1}^{k} l(v_{i-1}, v_i)$. Thus the sum of lengths around the cycle c is negative, thereby providing the desired contradiction.

To show that G_p forms a rooted tree with root s, it suffices to prove that for each vertex $v \in V_p$, there is a unique simple path from s to v in G_p. This follows directly from the definition of the parent graph and the way it is constructed in the *UPDATE* procedure. ■

Lemma 3 *Let $s \rightsquigarrow u \rightarrow v$ be a shortest path in G for some vertices $u, v \in V$. Suppose a sequence of relaxations that include the edge (u, v) has been executed on G. If $d(u) = \delta(s, u)$ at any point prior this execution, then $d(v) = \delta(s, v)$ is true at all times after this execution.*

Proof: By Lemma 1, if $d(u) = \delta(s, u)$ at some point prior to relaxing the edge (u, v), then this equality holds thereafter. In particular, after relaxing the edge (u, v), we have

$$d(v) \leq d(u) + l(u, v) = \delta(s, u) + l(u, v) = \delta(s, v)$$

By Lemma 1 $\delta(s, v)$ bounds $d(v)$ from bellow, from which we conclude that $d(v) = \delta(s, v)$, and this equality is maintained thereafter. ■

Lemma 4 *Let G contains no reachable negative cycles. Then, at the termination of the algorithm, we have $d(v) = \delta(s, v)$ ($d(v)$ is exact) for all vertices $v \in V$ and the parent graph G_p is a shortest-path tree rooted at s.*

Proof: Suppose first that $v \in V$ is reachable from s. According to Lemma 2 the graph G_p forms at the termination of the algorithm a rooted tree with root s. Let us prove the assertion by induction on the depth of a vertex v

in this tree. For the basis we have $d(s) = \delta(s, s) = 0$ after initialisation and by Lemma 1 this equality is maintained thereafter. For the inductive step, we assume that $d(u) = \delta(s, u)$ for every vertex of depth at most $i - 1$. Let v be a vertex of depth i and let $u = p(v)$. Obviously the depth of u is $i - 1$. Once the value $d(u)$ is settled to $\delta(s, u)$ the vertex u is pushed into the queue by the procedure *UPDATE*. Consequently, the edge (u, v) is relaxed and by Lemma 3 we conclude that $d(v)$ is settled to $\delta(s, v)$ and it remains true for the rest of the computation.

Let $v \in V$ is not reachable from s. Then there is no path from s to v, $d(v)$ is never updated, and keeps its initial value. Therefore $d(v) = \infty = \delta(s, v)$.

It remains to prove that G_p is a shortest-path tree, i.e. that for each $v \in V$ the unique simple path $s \rightsquigarrow v$ in G_p is a shortest path from s to v in G. Let the path be $\rho =< v_0, v_1, \ldots, v_k >$, where $v_0 = s$ and $v_k = v$. We know that $d(v_{i-1}) + l(v_{i-1}, v_i) = d(v_)$ whenever $p(v_i) = v_{i-1}$. At the termination of the algorithm we have $d(v_i) = \delta(s, v_i)$ and therefore $l(v_{i-1}, v_i) = \delta(s, v_i) - \delta(s, v_{i-1})$. Summing the lengths along the path ρ yields

$$l(\rho) = \sum_{i=1}^{k} l(v_{i-1}, v_i) = \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) = \delta(s, v_k) - \delta(s, v_0) = \delta(s, v_k)$$

We conclude that $l(\rho) = \delta(s, v_k)$ and thus ρ is a shortest path from s to $v = v_k$.

■

In the sequel we prove that the *DSP* algorithm correctly detects negative cycles. To that end let us define a notion of a *pass* inductively. Pass zero consists of the initial scanning of the source vertex s. Pass i starts as soon as pass $i - 1$ ends, and ends as soon as the *SCAN* operation has been applied to all vertices v which were in processors' queues at the end of pass $i - 1$ and for which $d(v)$ was equal to $\delta(s, v)$ at that time.

Let $A = \{v \in V \mid -\infty < \delta(s, v) < \infty\}$ and $B = \{v \in V \mid \delta(s, v) = -\infty\}$. Let $< v_0, \ldots, v_k >$, where $v_0 = s$ and $v_k = v$, be a shortest simple path from s to a vertex v. It is easy to verify (by induction on k) that after finishing the pass k the vertex v has distance label $d(v) \leq l(< v_0, \ldots, v_k >)$ (for a vertex $v \in A$ equality $d(v) = \delta(s, v)$ holds). If the graph is feasible (i.e. B is empty), then the total number of passes is at most n and at the end of the last pass all distance labels become exact and all queues become empty. If the graph has a negative cycle, then at the end of the last pass all vertices from the set A have exact distance label and all vertices from B are in queues. In the subsequent computation (let us call this phase the *final pass*) only vertices from the set B are scanned.

Lemma 5 *If G contains a reachable negative cycle, then after the first vertex update of the final pass, G_p always has a cycle.*

Proof: Let v be a vertex which distance label is updated as the first in the final pass. Since $d(v)$ is non-increasing, the new value of $d(v)$ is less than the length of a shortest simple path from s to v. Suppose we start at v and follow the parent pointers. If we find a cycle of parent pointers in this process, we are done. Otherwise we reach s and $d(s) = 0$. This indicates the existence of a simple path ρ with $l(\rho) = d(v)$ which contradicts our assumption about v. ■

Lemma 6 *If the* WTR *procedure detects a cycle, then there is a negative cycle in G.*

Proof: The *WTR* procedure initiated in a vertex u with stamp i detects a cycle by finding a vertex marked with the same vertex u and the same stamp i. The equality of stamps ensures that this vertex has already been visited during the current *WTR*. From the proof of Lemma 2 it follows that any cycle in G_p is a negative cycle in G. ■

Lemma 7 *If G contains a reachable negative cycle, then some* WTR *procedure detects a cycle.*

Proof: Suppose the distributed algorithm is in the final pass and no cycle has been detected up to this moment. Arguments similar to that used in the proof of Lemma 5 manifest that from every vertex which distance label has been updated a cycle in G_p is reachable. The *WTR* procedure is called on line 7 of *UPDATE* procedure for some $u \in B$ and stamp i. *WTR* backtracks the parent graph systematically. The procedure can "meet" another walk to root initiated concurrently for an origin z, i.e. it gets to a vertex already marked by $[z, j]$. Then for $u > z$ or $u = z, i \neq j$ the marks of the second walk to root are overwritten (line 15) by $[u, i]$. Otherwise the current walk to root is terminated, its marks are removed by the procedure *REM* and the other walk continues. However, in such a case the cycle in the parent graph is the same for both walks. We now have a similar situation as before for the walk started at the origin $z > u$. As the number of vertices is finite, walk to root with highest priority cannot be terminated by another one, detecting thus the cycle. ■

The previous lemmas give the proof of the correctness of the distributed algorithm.

Theorem 1 (Correctness of DSP) *If G contains no negative cycles that are reachable from the source vertex s, then the algorithm terminates, $d(v) = \delta(s, v)$ for all vertices $v \in V$, and the parent graph G_p is a shortest-path tree rooted at s. If G does contain a negative cycle reachable from s, then the algorithm terminates and reports "negative cycle found".*

Complexity

We use a simple model [GS93, CKP+93] to analyse the performance of the *DSP* algorithm. We view the distributed algorithm as a sequence of local computations interleaved with communication steps, where we allow computation and communication to overlap. We define the computation time $T_{comp}(n, p)$ as the maximum time it takes a processor to perform all the local computation steps.

Theorem 2 (Computational Complexity of DSP) *Let the input graph with n vertices is distributed over P processors each of which owns $\mathcal{O}(\frac{n}{P})$ vertices. Then the worst case computational complexity of the* DSP *algorithm is $T_{comp}(n, P) = \mathcal{O}(n^2 \cdot \frac{n}{P})$.*

Proof: The analysis of the time complexity proceeds in two steps. Every processor performs two basic operations: relaxation of edges and marking of vertices. First of all we state the time complexity of relaxation operations and then proceed by analysing the complexity of marking operations. Recall that the computation can be divided into at most $n + 1$ passes. At the beginning of each pass (with the exception of the final one) every processor has its own queue of size $\mathcal{O}(\frac{n}{P})$. The pass ends not later than when all vertices from all the queues have been scanned. The scanning of a vertex requires at most n relaxation operations (we consider this as a primitive operation) and the worst time complexity of one pass is $\mathcal{O}(n \cdot \frac{n}{P})$. In the presence of a negative cycle in the input graph G the existence of a cycle in G_p is guaranteed in the final pass (Lemma 5). After that it takes at most n relaxation operations before a new *WTR* procedure is initiated. According to Lemma 7 this procedure must detect the presence of a negative cycle.

Now, let us analyse the complexity of the *WTR* and the *REM* procedures. The negative cycle search is performed (on various processors) sequentially and its time complexity is proportional to the length of the searched path. On the searched path *WTR* cannot enter a vertex more than once as entering a vertex with the same mark indicates the presence of a cycle and causes the termination of the whole computation. Consequently, every *WTR* procedure takes at most $\mathcal{O}(n)$ time. The complexity of the subsequent removing of marks is asymptotically the same. Every processor initiates the *WTR* procedure only when it has done n relaxation operations. Therefore the time complexity of *WTR* and *REM* is amortised over the time complexity of relaxation operations. ■

5 Application to Model Checking of LTL

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 identified 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. This problem can be effectively solved using nested depth first search (*NDFS*) algorithm [HPY96] incorporated in the SPIN verification tool [Hol97]. The practical limitation of this algorithm is the amount of the randomly accessed memory which the algorithm requires. A very natural way how to overcome the memory limitation is to distribute the given graph onto several processors (computers) and to perform a distributed computation. As the depth first search is P-complete, promising parallel depth-first-search-based algorithm is unlikely to exist [Rei85]. A completely different approach to distributed emptiness problem is needed.

The connection between the negative cycle problem and the Büchi automaton emptiness problem is the 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.

Our objective was to compare the performance of the *DSP* algorithm with the nested depth first search (*NDFS*) algorithm. In the distributed version of *NDFS* 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.

The implementation has been done 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. In the implementation of the *DSP* algorithm we have employed the following optimisation scheme. For more efficient communication between processors we do not send separate messages. The messages are sent in packets of prespecified 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.

		NDFS		DSP	
Vertices	Cross–edges	Time	Messages	Time	Messages
Generated, without cycle					
40398	34854	1:01	79376	0:11	809
71040	1301094	31:13	3008902	0:48	1108
696932	1044739	27:02	2387220	1:31	14029
736400	5331790	126:46	12316618	5:17	48577
777488	870204	21:36	1887252	2:02	13872
1859160	1879786	49:04	4226714	6:00	25396
Generated, with cycle					
18699	22449	0:06	22	0:05	68
33400	2073288	0:37	30824	0:24	555
46956	83110	0:05	108	0:09	702
448875	1863905	0:51	21106	0:56	3435
Random, without cycle					
4000	353247	14:03	1390262	0:17	17868
5000	839679	31:48	3151724	0:32	2489
80000	522327	30:11	2042212	1:39	87002
60000	1111411	57:19	4131210	4:08	98686
947200	5781959	184:23	13338462	9:49	47030
Random, with cycle					
18000	1169438	1:20	104822	0:09	862
Philosophers					
(12) 94578	42154	2:06	168616	0:13	756
(14) 608185	269923	16:11	1079692	1:40	4500

Table 1: Summary of experimental results

We performed several sets of tests on different instances in order to verify how fast is the algorithm in practice, i.e. beyond its theoretical characterisation. Our experiments were performed on two kinds of systems given by random graphs and generated graphs. Graphs were generated using a simple specification language and an LTL formula. In both cases we tested graphs with and without cycles to model faulty and correct behaviour of systems. As our real example we tested the parametrised Dining Philosophers problem. Each instance is characterised 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 and the number of sent messages (communication) as the main metrics. Table 1 summarises the achieved results.
The experiments lead basically to the following two conclusions:

- the *DSP* algorithm is comparable with the *NDFS* one on all graphs.
- the *DSP* algorithm is significantly better on graphs without negative cycles.

The experiments show that in spite of worse theoretical worst time complexity of the *DSP* algorithm its behaviour 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 the *DSP* algorithm 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, the *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 the *DSP* 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* all can compute concurrently. We notice that all mentioned advantages of the *DSP* 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.

We have accomplished yet another set of tests (see Table 2) in order to validate the scalability of the *DSP* algorithm. 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 increased.

Computers	Time	Messages
1	-	-
2	4:37.96	5668
4	3:11.80	16340
6	2:20.17	22057
8	1:59.92	28019
10	1:50.18	32104

Table 2: Scalability for graph with 777488 vertices, without cycle

6 Conclusions

We propose a novel distributed algorithm for the single source shortest path problem for directed graphs which can contain negative length cycles. A unique distributed variant of the walk to root negative cycle detection strategy has been employed. The algorithm is scalable and work-efficient as the total number of operations of the distributed algorithm is of the same order as the number of operations of the best known sequential algorithm [CLR90, CG99].

We have evaluated the empirical performance of our distributed algorithm and applied it to the distributed model checking problem for finite state systems and LTL formulas and these experiments show that we are able to verify systems for which the standard sequential algorithm fails.

References

[BČKP01] L. Brim, I. Černá, P. Krčál, and R. Pelánek. Distributed LTL model checking based on negative cycle detection. Submitted to 21st Foundations of Software Technology and Theoretical Computer Science, 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.

[CKP+93] D Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. v. Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, 1993.

[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.

[Dij59] E.W. Dijkstra. A note on two problems in connexion with graphs. *Num. Math.*, 1:269–271, 1959.

[DKZ94] P. Spirakis D. Kavvadias, G. Pantziou and C. Zaroliagis. Efficient sequential and parallel algorithms for the negative cycle problem. In *Proc. 5th ISAAC'94, Lecture Notes in Computer Science*, volume 834, pages 270–278. Springer-Verlag, 1994.

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

[GS93] A. Gibbons and P. Spirakis, editors. *Lectures on Parallel Computation*. Cambridge Univesity Press, 1993.

[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.

[HTB97] M. Hribar, V. Taylor, and D. Boyce. Performance study of parallel shortest path algorithms: Characteristics of good decompositions. In *Proc. ISUG '97 Conference*, 1997.

[Law76] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.

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

[Rei85] J.H. Reif. Depth-first search is inherrently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[Roo00] S.H. Roosta. *Parallel processing and parallel algorithms*. Springer-Verlag, 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 (IRREGULAR-3)*, volume 1117 of *LNCS*, pages 183–194. Springer-Verlag, 1996.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science, LICS'86, Cambridge, MA, USA, 16–18 June 1986*, pages 332–344. IEEE Computer Society Press, Washington, DC, 1986.

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