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 and thus avoids using any additional data structures. At the same time the algorithm uses a novel 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.

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.

With respect to the intended application even in the distributed environment the space requirements are the main limiting factors. Therefore we have been

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

looking for a distributed algorithm compatible with other space-saving techniques (e.g. *on-the-fly* technique or *partial order* technique). Our distributed algorithm is therefore 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 potential negative length cycle.

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 looked 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 $n =| V |$ and $m =| E |$. 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 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 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 DFS 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. This operation creates a cycle in G_p if and only if u is an ancestor of v in the current parent graph. 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. However, to preserve the correctness the behavior of *walk to root* has to be significantly modified. The amortization is used in the distributed algorithm and is described in detail in Section 5.

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 recomputations 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. 1. Demonstration of the reverse search

Fig. 1 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 some operation on its children. But *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 present the sequential algorithm (Fig. 2) and prove its correctness and complexity first. This algorithm forms the base of 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). 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. Due to the space limits the proofs are only sketched.

```
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. 2. Pseudo-code of the sequential algorithm

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. 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 let $h(w) = k - 1$, $\{a_1, a_2, \ldots, a_r\} = \{u \mid (w, u) \in E\}$. Since $h(a_i) = k$ for all $i \in \{1, \ldots, r\}$, we can use the induction hypothesis for each a_i and show that the value of the variable u in RGS is equal to a_i exactly once. Therefore RGS returns *u does not exist* for w after a finite number of steps and the sequence S is 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 $v \in V$ 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 $d(v_0) = d(s) = \delta(s,s) = 0$ 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 afterwards 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 afterwards.

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 $C = \max\{\mid l(u,v) \mid : (u,v) \in E\}$.

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

Proof: Each shortest path consists of at most $n-1$ edges and $-C(n-1) \leq \delta(s,v) \leq C(n-1)$ holds for all $v \in V$. Each vertex v is visited only after $d(v)$ is lowered. Therefore each vertex is 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 not unavoidable and the algorithm can be easily 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

For the distributed algorithm we suppose that the set of vertices is divided into disjoint subsets. The distribution is determined by the function *owner* which assigns every vertex v to a processor i. Processor i is responsible for the subgraf determined by the owned subset of vertices. Good partition of vertices among processors is important because it has direct impact on communication complexity and thus on run-time of the program. We do not discuss it here because it is itself quite a difficult problem and depends on the concrete application.

The main idea of the distributed algorithm (Fig. 3) can be summarized as follows. The computation is initialized by the processor which owns 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 vertices basically in the same manner as the sequential algorithm does.

While relaxation can be performed in parallel, the realization of *walk to root* requires more careful treatment. Even if adding the edge initiating the *walk to root* does not create a cycle in the parent graph, the parent graph can contain a cycle on the way to root created in the meantime by some other processor. The *walk to root* we used in the sequential algorithm would stay in this cycle forever. Amortization of walk brings similar problems. We propose a modification of the *walk to root* which solves both problems.

Each processor maintains a *counter* of started *WTR* procedures. The *WTR* procedure marks each node through which it proceeds by the name of the vertex where the walk has been started (*origin*) and by the current value of the processor *counter* (*stamp*). When the walk reaches a vertex that is already marked with the same *origin* and *stamp* a negative cycle is detected and the computation is terminated. In distributed environment it is possible to start more than one walk concurrently and it may happen that the walk reaches a vertex that is already marked by some other mark. In that case we use the ordering on vertices to decide whether to finish the walk or to overwrite the previous mark and continue. In the case that the walk has been finished (i.e. it has reached the root or a vertex marked by higher *origin*, line 9 of *WTR*) we need to remove its marks. This is done by the *REM* (REmove Marks) procedure which follows the path in the parent graph starting from the *origin* in the same manner as *WTR* does. The values $p(v)$ of marked vertices are not changed (line 6 of *RGS*) and therefore the *REM* procedure can find and remove the marks. However, due to possible overwriting of walks, it is possible that the *REM* procedure does not remove all marks. Note that these marks will be removed by some other *REM* procedure eventually. The correctness of cycle detection is guaranteed as for the cycle detection the equality of both the *origin* and *stamp* is required.

The modifications of *walk to root* enforces the *Trace* procedure to stop when it reaches a marked vertex and to wait till the vertex becomes unmarked. Moreover, *walk to root* is not called during each relaxation step (*WTR_amortization* condition becomes true every n-th time it is called).

Whenever a processor has 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 4 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*. Anytime 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. *REM*) procedure.

```
proc Main
  while not finished do
        req := pop(queue);
        if req.length = d(req.vertex) then Trace (req.vertex, req.father); fi
  od
end

proc Trace (v, father)
  p(v) := father;
  while v ≠ father do
        Handle_messages;
        u := RGS(v, NULL);
        while u does not exist do
              last := v; v := p(v);
              u := RGS(v, last); od
        v := u;
  od
end

proc RGS (v, last) {Relax and Get Successor}
  u := successor of v greater than last;
  while u exists do
        if u is local then
                          if d(u) > d(v) + l(u, v) then
                             if mark(u) then wait; fi
                             p(u) := v;
                             d(u) := d(v) + l(u, v);
                             if WTR_amortization then WTR([u, stamp], u); inc(stamp); fi
                             return u;
                          fi
                     else send_message(owner(u), "update u, v, d(u) + l(u, v)");
        fi
        u := next successor of v;
  od
  return u does not exist;
end

proc WTR ([origin, stamp], at) {Walk To Root}
  done := false;
  while ¬done do
        if at is local
          then
                if mark(at) = [origin, stamp] →
                  send_message(Manager, "negative cycle found");
                  terminate
                [] (at = source) ∨ (mark(at) > [origin, stamp]) →
                  if origin is local
                    then REM([origin, stamp], origin)
                     else send_message(owner(origin),
                                 "start REM([origin, stamp], origin))" fi
                  done := true;
                [] (mark(at) = nil) ∨ (mark(at) < [origin, stamp]) →
                  mark(at) := [origin, stamp];
                  at := p(at)
                fi
           else send_message(owner(at), "start WTR([origin, stamp], at)");
                done := true
        fi
  od
end
```

Fig. 3. Pseudo-code of the distributed algorithm

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. A *manager* process is used to detect the termination and to finish the algorithm by sending a *termination* signal to all processors.

Theorem 3 (Correctness and complexity of the distributed algorithm). *If G has no negative cycle reachable from the source s 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.*

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

Proof: The proof of the correctness of the distributed algorithm is technically more involved and due to the space limits is presented in the full version of the paper only. The basic ideas are the same as for the sequential case, especially in the case when G has no negative cycle. Proof of the correctness of the distributed *walk to root* strategy is based on the ordering on walks and on the fact that if G contains a reachable negative cycle then after a finite number of relaxation steps G_p always has a cycle.

Complexity is $\mathcal{O}(Cn^3)$ due to the amortization of the *walk to root*. ■

6 Experiments

We have implemented the distributed algorithm. The experiments have been performed on a cluster of seven workstations interconnected with a fast 100Mbps Ethernet using Message Passing Interface (MPI) library.

We have performed a series of practical experiments on particular types of graphs that represent the *LTL model checking problem*. The LTL model checking problem is defined as follows. Given a finite system and a LTL formula decide whether the given system satisfies the formula. This problem can be reduced to the problem of finding an *accepting cycles* in a directed graph [VW86] and has a linear sequential complexity. In practice however, the resulting graph is usually very large and the linear algorithm is based on depth-first search, which makes it hard to distribute. We have reduced the model checking problem to the SSSP problem with edge lengths 0, -1. Instead of looking for accepting cycles we detect negative cycles.

The experimental results clearly confirm that for LTL model checking our algorithm is able to verify systems that were beyond the scope of the sequential model checking algorithm.

Part of our experimental results is summarized in the table below. 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.

	Number of Computers						
No. of 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

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.

[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.

[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.