Shortest Paths and Negative Cycle Detection in Graphs with Negative Weights

—

I: The Bellman-Ford-Moore Algorithm Revisited

Stefan Lewandowski

Universität Stuttgart, FMI
Universitätsstr. 38
70569 Stuttgart, Germany
Lewandowski@fmi.uni-stuttgart.de

Technical Report No. 2010/05

August 3, 2010

Abstract: Since the mid 1950's when Bellman, Ford, and Moore developped their shortest path algorithm various attempts were made to beat the $O(n \cdot m)$ barrier without success. For the special case of integer weights Goldberg's algorithm gave a theoretical improvement, but the algorithm isn't competative in praxis. This technical report is part one of a summary of existing n-pass algorithms and some new variations. In this part we consider the classical algorithm and variations that differ only in the data structure used to maintain the set of nodes to be scanned in the current and following pass. We unify notation and give some experimental results for the average case on various graph classes.

Keywords: shortest path, negative weights, negative cycle detection

1 Introduction

Since Bellman, Ford, and Moore ([Bel58], [For56], [Moo59]) developped their $O(nm)$ single source shortest path algorithm it's an open problem whether there exist $o(nm)$ algorithms for the general case. Dijkstra ([Dij59], [FT87]) introduced his near linear time algorithm, but this is valid for graphs with non-negative weights only. Goldberg's scaling algorithm [Gol95] for graphs with integer weights greater $\gamma_{\min} \leq \; -2$ gives a theoretical improvement ($O(\sqrt{n}m \log(|\gamma_{\min}|))$) unless $|\gamma_{\min}|$ is very large.

The negative cycle detection problem is closely related to the shortest paths problem – in fact most algorithms solve the negative cycle detection problem by implicitely solving a shortest path problem from an external node that's adjacent to all other nodes by zero cost arcs. The solution of a shortest path problem gives a proof that there's no negative cycle – if there's a negative cycle the shortest path algorithm also has to find it. Recent literature has introduced the name shortest path feasability problem (FP). We will describe all algorithms from this point of view.

This paper is organized as follows: section 2 introduces the basic definitions, section 3 presents the n-pass algorithm, negative cycle detection strategies, and the considered variations. Section 4 describes the experimental setup, section 5 the experimental results. Section 6 closes this report with the conclusions and an outlook.

2 Definitions and Notation

Let $\mathbb{R}$ be the set of real numbers. A directed weighted graph $G = (V, E, \gamma)$ consists of the set of nodes V, the set of edges $E \subseteq V \times V$, and the weight function $\gamma : E \rightarrow \mathbb{R}$. The shortest distance from source node s to any node v is denoted by $\mathrm{d}_s(v)$ and upper bounds (used in the algorithms) by $\mathrm{D}_s(v)$ (most often we omit the subscript s as we use an additional start node v_0 or s is given by context). Let π be a potential function, we denote the reduced weights with $\gamma_\pi(u, v) = \pi(u) + \gamma(u, v) - \pi(v)$.

The solution of the shortest path feasibility problem (FP – given a graph $G = (V, E, \gamma)$) is either a negative cycle in G or a proof that there is none (by a potential function π which holds $\forall \; (u, v) \in E : \; \gamma_\pi(u, v) \geq 0$).

The solution of a shortest path problem in the graph $G' = (V \,\dot{\cup}\, \{v_0\}, E \cup \{(v_0, v) \mid v \in V\}, \gamma')$, $\gamma'(u, v) = \gamma(u, v), \; (u, v) \in E, \; \gamma'(v_0, v) = 0, \; v \in V$ implies a solution for FP because $\pi(v) = \mathrm{d}_{v_0}(v), \; v \in V$ along with the triangle inequality $\mathrm{d}_{v_0}(v) \leq \mathrm{d}_{v_0}(u) + \gamma(u, v)$ implies $\pi(u) + \gamma(u, v) - \pi(v) \geq 0$ (non-negative reduced weights for all arcs). Note, that we won't explicitely construct G', but will modify the initialisation of the algorithms as if these were running on G'.

The reduced cost for paths $\gamma_\pi(v_1, \ldots, v_k) = \pi(v_1) + \gamma(v_1, \ldots, v_k) - \pi(v_k)$ implies that G has a negative cycle if and only if it has one with reduced weight arcs using any potential function.

3 Algorithms – a Detailed Survey

The classical algorithm by Bellman, Ford, and Moore (BFM) scans nodes first-in-first-out. Scanning a node u is to check for every succeeding node v whether the path length to v decreases via u. If the path length decreased, we add v to the queue of nodes to be scanned if it isn't already in there.

This naturally leads to n-pass algorithms

- we scan each node at most once per pass
- init the queue with start node (external node in the FP context)
- pass i scans all nodes that were inserted into the queue during pass $i-1$

In the n-pass algorithm we explicitely distinguish between the set `A` to be scanned in the current pass and set `B` to be scanned in the following pass. The BFM algorithm doesn't need this splitting, but other algorithms will use those two sets separately.

Most information used by the algorithms is stored directly in the graph – arc lengths are stored in the data type for arcs, additional info for nodes include distance estimates D, parent pointers of the shortest path tree, flags for efficient testing whether a node is in a set of nodes, and even the queues are stored within the nodes (note, that each node is in a queue at most once at a time, hence, we need just one pointer per node). This gives better performance – we totally avoid repeated memory allocation (also recursion is often avoided). Additional memory used is linear in the number of nodes.

Though most data is stored within the nodes, in the algorithms data is written like it was an array or a function.

```
function n_pass_algorithm
  in: graph $G = (V, E, \gamma)$ without negative cycles
  out: potential function $\pi$ with $\gamma_\pi(u,v) \geq 0,\ \ (u,v) \in E$

  variables:
    set of nodes A,B
    node $u$,$v$
begin
  B := $V$; $\mathrm{D}(v)$ := 0, $v \in V$; parent($v$) := null, $v \in V$;
  repeat
    A := B; B := $\emptyset$;
    repeat
      $u$ := getElement(A); A := A$\setminus\{u\}$; // remove an arbitrary node
      for all edges $(u,v) \in E$ // and "scan it"
        if $\mathrm{D}(u) + \gamma(u,v) < \mathrm{D}(v)$ then
          $\mathrm{D}(v)$ := $\mathrm{D}(u) + \gamma(u,v)$; parent($v$) := $u$;
```

```
             if $v \notin$ A$\cup$B then
               B := B$\cup\{v\}$;
             end if;
20         end if;
         end for;
       until A=$\emptyset$;
     until B=$\emptyset$;
     $\pi(v)$ := D$(v)$, $v \in V$;
25   end;
```

Algorithm 3.1: a generic n-pass algorithm

The first-in-first-out order of the BFM algorithm doesn't matter for correctness, nodes within one pass may be scanned in any order. This generalization was first introduced by Glover et. al [GKP85]. It unifies correctness proofs for all n-pass algorithms: after pass k shortest paths with k edges are correct (by induction the second last node on that path was set to the correct value in pass $j < k$ and was scanned either in pass j (in case it was in A at that moment or it was added to B and scanned in pass $j+1 \leq k$ (and set the last node on the path to the correct value)). At any time each node is in the queue at most once, so each node is scanned at most once per pass, therefore each arc is used at most once per pass and we get $O(n+m)$ per pass ($O(m)$ if we assume the graph to be connected). We can have at most $n-1$ passes, otherwise there's a negative cycle (through any node that's still in B at that moment) – we'll take care of detecting negative cycles in section 3.1.

Instead of always adding a decreased node to B, we may also add it to A provided that it hasn't been scanned in this pass yet (replace lines 17–19 of algorithm 3.1 with the code of algorithm 3.2). The proof for correctness doesn't change. But we have to maintain a flag scanned for every node and set it to false at the beginning of each pass. After the distance of a node v was decreased it has to be added to either A or B.

```
if $v \notin$ A$\cup$B then
  if not scanned($v$) then
    A := A$\cup\{v\}$;
  else
    B := B$\cup\{v\}$;
  end if;
end if;
```

Algorithm 3.2: modified insertion of node in sets A and B

Note, that all our implementations ensure that a node v is always in at most one of the sets A and B. Having v in both sets means that after scanning it in A it would possibly be scanned again in the next pass while its distance is unchanged – this wouldn't make sense.

3.1 Negative cycle detection

Waiting for pass n is an easy way to detect negative cycles, but it's not practical unless we expect that most graphs have no negative cycles (so this would just make the algorithm fail-proof without cost, we disregard this method here). We review two general strategies for finding negative cycles.

The **parent pointers** used in algorithm 3.1 define a subgraph of G with $O(n)$ arcs. We can adapt depth-first search [Tar72] to detect a cycle on parent pointers in $O(n)$ (a cycle of parent pointers corresponds to a negative cycle in G). So we can check once for negative cycles after each pass without increasing the asymptotic $O(nm)$ running time (after line 22 in algorithm 3.1).

The number of nodes scanned in each pass may differ and be small, so for practical performance it's important to count node scans and check for negative cycles only if the number of node scans since the last check is $\Omega(n)$ (see experiments in section 4 and 5).

Because every node has just one outgoing arc in the subgraph, we can further simplify the depth-first search algorithm.

```
function negative_cycle_detected
  in: subgraph $G_{\text{parent}}$ with at most one outgoing arc per node
      // given by parent pointers of the shortest path tree
  out: true if $G_{\text{parent}}$ has a negative cycle, false otherwise

  variables:
    node p;
begin
  scanned($v$) := -1, $v \in V$; // -1 means "not scanned yet"
  // scanned($u$) = x means
  // "scanned when followed parents started from node x"
  for all nodes $v \in V$
    p := $v$;
    while (p$\neq$null) loop
      if (scanned(p)=-1)
        scanned(p) := $v$;
        p := parent(p);
      elsif (scanned(p)=$v$)
        return true; // negative cycle found
      else
        break; // p was already visited by another node
      end if;
    end loop;
  end for;
  return false;
end;
```

Algorithm 3.3: negative cycle detection by following parent pointers

Tarjan [Tar81] introduced **subtree disassembly** in 1981. When we intend to scan a node v, but have its parent u in the queue, then u's distance was reduced after v was added to the queue, therefore the distance of v will be reduced again and needn't to be scanned now. Following this idea scanning any successor in the shortest path subtree of v is unnecessary, i. e., when we add v to the queue, we can remove all of its successors.

As a side effect subtree disassembly gives us an easy way to detect negative cycles. When we scan u, update the distance of v (making it a new child of u) and find u in the subtree of v, we have a negative cycle, because on the path from v to u no distance changed (otherwise we would have removed u from v's subtree), i. e., the reduced weight distance (with $\mathrm{D}(\cdot)$ used as potential function) of u is equal to that of v, but now the edge (u, v) reduces v's distance, hence there's a negative cycle. We remove all nodes x of v's subtree from the sets `A` or `B`, respectively. While we scan node u (iteration over successor nodes v), we keep u's flag (for being in set `A`) `true`. If that flag is set to `false` during subtree disassembly starting at one of the successor nodes v, then u was removed from v's subtree and we can detect the negative cycle by one single `if` statement (otherwise we continue and set u's flag to `false` after the scan is completed as we removed u from `A`).

The time for subtree disassembly doesn't increase asymptotic running times. In fact we have amortized $O(1)$ cost for maintaining the tree structure (paying $O(1)$ for the actual insertion and $O(1)$ for building a potential). Deleting a subtree is paid from the built potential (linear in size of the subtree). Subtree disassembly reduces the number of node scans on a wide variety of graphs.

We have implemented two data structures for subtree disassembly, details are given in section 3.2 (note that the parent pointers follow the wrong direction – we can't use them here).

Each algorithm can be combined with any negative cycle detection strategy, but of course we'll use the one that uses the least overhead (depending on the algorithm). When we do subtree disassembly (to reduce the expected number of node scans), we get negative cycle detection almost for free (it's just that one single `if` statement). Algorithms that use depth-first search to scan set `A` in a specific order will use this to detect negative cycles with little overhead (the algorithms considered in this report don't use depth-first search, though). Algorithms that neither use subtree disassembly nor depth-first search will use the adapted depth-first search (algorithm 3.3). None of these methods will increase the $O(nm)$ total running time.

3.2 Subtree disassembly

The algorithms that use subtree disassembly maintain a representation of the shortest path tree. We need only two operations on this tree:

- adding a leaf
- deleting a node and its complete subtree

We give details of two possible implementations.

Data structure 1 uses 4 pointers per node and stores the tree in a straight forward way:

- pointer to parent (identical to the `parent`-pointer in any n-pass algorithm)
- pointer to one of the children
- pointers to left and right siblings (circular doubly-linked list)

Adding a leaf means to either add a first child to its parent (left and right siblings of the leaf point to itself) or add the leaf as an additional child (into the circular doubly-linked list). Deleting a node with its subtree means to remove the node from its circular doubly-linked list (maybe changing the first child pointer of its parent) and traverse the subtree (recursive calls for all children). Costs are just setting very few pointers per node and it's straight forward to implement.

```
function delete_a_subtree
  in: node $v$ // its subtree has to be deleted
  effect: shortest path tree data structure modified accordingly

  variables:
    node $v_c$;
begin
  if ($v \neq$ null)
    remove $v$ from sets A and B;
    for all children $v_c$
      delete_a_subtree($v_c$);
    end for;
    remove $v$ from shortest path tree;
  end if;
end;
```

Algorithm 3.4: deleting a subtree (subtree disassembly, data structure 1)

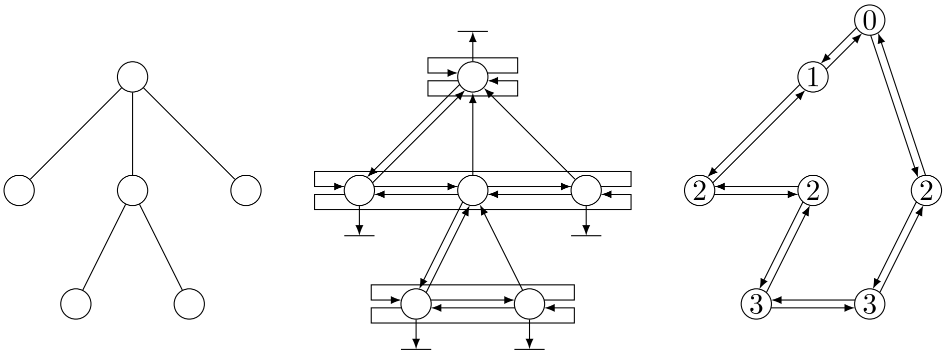
Data structure 2 directly maintains a preorder traversal of the tree, so it uses just two pointers for previous and next node in the traversal. To avoid special cases like deleting the first or last node of the preorder, we suggest inserting a dummy node as first / last node in the traversal. Inserting a new leaf is easy as it can be inserted directly after its parent. Deleting all nodes of a subtree means to begin with the subtree's root and delete the succeeding nodes in the list. In order to be able to detect the end of the subtree, we maintain the level (number of nodes on the path to the root) for each node: when inserting a leaf its level is one more than its parent's level. A subtree ends when the succeeding node has a level lower (or equal) than the subtree's root.

```
function delete_a_subtree
  in: node $v$ // its subtree has to be deleted
      with shortest path tree preorder
  effect: shortest path tree preorder modified accordingly

  variables:
    node in shortest path tree preorder prev,curr;
    integer lowlevel;
begin
  prev := $v$->prev;
  lowlevel := level($v$);
  remove $v$ from sets A and B;
  curr := $v$->next;
  while (level(curr)>lowlevel) loop
    remove curr from sets A and B;
    curr := curr->next;
  end loop;
  curr->prev := prev;
  prev->next := curr;
end;
```

Algorithm 3.5: deleting a subtree (subtree disassembly, data structure 2)

Data structure 2 has an interesting property – we don't need to maintain any parent pointers. We could omit changing the `parent` pointers in any of the n-pass algorithms and – if needed – reconstruct them after the algorithm terminates (traverse the preorder and store every node in an array at the index corresponding to its level, the pointer to its parent can be read from the array at the index minus one (at index 0 we store the null pointer), this can be done in $O(n)$ time).



a) original tree   b) data structure 1   c) data structure 2

Figure 3.1: data structures for subtree disassembly

```
function reconstruct_parent_pointers
  in: shortest path tree preorder $sptp$ // beginning with dummy node,
      providing levels
  effect: parent($v$) correctly set, $v \in V$

  variables:
    array (0..$n$) of nodes help;
    node in shortest path tree preorder dummy;
begin
  help(0) := null;
  dummy := $sptp$; $sptp$ := $sptp$->next;
  while $sptp \neq$dummy loop
    help(level($sptp$)) := $sptp$;
    parent($sptp$) := help(level($sptp$)-1);
    $sptp$ := $sptp$->next;
  end loop;
end;
```

Algorithm 3.6: reconstruct parent pointers

On some algorithms it doesn't matter which version of subtree disassembly we use, on others it makes a difference. The first data structure inserts a new leaf as a last child in the circular doubly-linked list, the second data structure inserts it as a first child. Hence the preorder traversal differs, and depending on this and on the way the sets A and B are maintained in the n-pass algorithm, the order in which nodes are scanned can change. We can easily change data structure 1 to behave like the second one with setting the parents first child pointer to the newly added leaf (just one additional assignment) – the other way round would cause too much overhead. We consider the effect in the experiments in section 4 and 5.

3.3 The BFM algorithm and its naturally derived heuristics

There have been several evaluation reports on shortest paths and the feasibility problem, e. g., [CGR96], [CG99], [NPX99], [WT05], and [CGG$^+$09]. Though some abbreviations are common, it's not always obvious which version the authors are talking about. The most confusing ones are the BFM algorithms derivated from the n-pass algorithm 3.1. There are two sets A and B, each can be either a queue or a stack. Furthermore we can either always add a decreased node to B or add it to A if possible. When we always insert decreased nodes to the set B, it doesn't make a difference whether set A is a queue or a stack – we scan the nodes from top to bottom or first to last, respectively. So, there's six combinations: two queues and insert to B only (this is the original BFM algorithm), same, but B maintained as a stack, four more combinations of queue or stack with queue or stack and nodes are added to set A if not scanned in the current pass yet. Each of these six can be combined with any negative cycle detection strategy: waiting for pass n, following parent pointers, or using subtree disassembly. We name these algorithms by BFM-XYZC with X, Y being S or Q (stack and queue for sets A and B, respectively), Z $\in$ {1,2} (algorithm may insert into first set A or always inserts into second set B). And finally negative cycle detection strategy C=W for simply waiting for pass n, C=P for depth-first seach on parent pointers (algorithm 3.3) or C=T for Tarjan's subtree disassembly (use a subscript to distinguish between different subtree disassembly implementations). Hence, BFM-SQ1T is a n-pass algorithm using a stack for A, a queue for B, allows decreased nodes to be inserted in A where appropriate, and uses subtree disassembly. The family of BFM algorithms using subtree disassembly is denoted by BFM-T (omitting the specifications for the sets A and B). Subfamilies are denoted by ommitting the corresponding value. Subvariants are denoted by subscripts.

The classical Bellman-Ford-Moore algorithm (denoted as BFM in many papers) is now BFM-QQ2W when we wait for pass n to detect negative cycles or BFM-QQ2P. It maintains the sets to be scanned in first-in-first-out order (nodes that are already in the queue remain at their position instead of being moved to the end of the queue or added twice). We also consider the five other natural variants BFM-QS2P, BFM-QQ1P, BFM-QS1P, BFM-SQ1P, and BFM-SS1P with detecting negative cycles by following parent pointers.

Note, that the classical algorithms by Pape [Pap74] and Pallottino [Pal84] are similar to BFM-SQ1 and BFM-QQ1, respectively, but not identical. Those two algorithms don't use passes in the sense of n-pass algorithms. A decreased node is inserted to A if it was scanned at least once before (if it was labeled, i. e., its distance got a finite value, but not scanned yet, it's not added twice). It's not checked how often a node was scanned yet. While these algorithms are practical on many graphs, their running times may deteriorate to $O(n^2m)$ for Pallottino's algorithm and even to $O(n2^n)$ for Pape's algorithm.

A simple variant of Tarjan's subtree disassembly is the parent heuristic BFM_{PH} introduced by Cherkassky et. al [CGR96]. When we intend to scan node u, but find its parent p in the queue, p's distance was reduced after u was added to the queue. Therefore we can skip u, it will be added to the queue again by scanning p. Because we maintain a flag for any node v being in the queue anyway, this parent heuristic is easy to implement by an additional `if` statement (enclosing the node scan, lines 14–21 in algorithm 3.1) and the condition can be checked in constant time. There's no need to maintain any additional data for using this heuristic.

The BFM_{PH} heuristic significantly reduces the number of node scans on most graphs. Instead of disabling node u (we could also have removed it from the queue as soon as u's parent was added to the queue again) the BFM-T algorithm uses subtree disassembly to remove all nodes of the shortest path subtree of the node that we intend to add to the queue. Instead of actually removing nodes from the queue we just set a flag to mark those nodes deactivated. When a deactivated node is added to the queue again, we rather remove that flag than adding that node to the end of the queue. The subtree disassembly heuristic often reduces the number of node scans by a large amount that more than compensates the necessary more costly maintainance of the shortest path tree.

The BFM-T_{UP} heuristic (BFM-T with updates) is a natural variant of BFM-T. When we remove all nodes of v's shortest path subtree after we decreased v's distance by Δ, we know that the distances of all nodes in the subtree will later decrease by at least Δ. We can update the distances while removing the subtree's nodes during disassembly with little overhead (note, that we have to traverse the tree also in BFM-T to adjust flags, it's not sufficient to skip the subtree in the shortest path tree data structure). Cherkassky et. al ([CGR96],[CGG$^+$09]) suggest to decrease the distances by $\Delta - 1$ to avoid additional bookmarking which nodes must eventually be scanned later at least once more. By this trick they achieve that those nodes are unlikely unnecessarily added to the queue, ensuring at the same time that any of those nodes will be added to the queue (and scanned) once more (the update by $\Delta - 1$ can be inserted before lines 11 and 14 in algorithms 3.4 and 3.5, respectively, the decrease by Δ of the subtree's root node is already done outside in the main algorithm). The number of node scans is often a bit lower than in BFM-T, but the updates need an additional overhead – see details in our experiments.

Note, that $\Delta - 1$ assumes integer arc lengths (as used in all our experiments), otherwise we'd have to replace -1 by the smallest possible difference of path lengths.

Also note, that it's not possible to update distances by Δ and simply keep the nodes in the tree (and queue). While correctness wouldn't be affected, we couldn't guarantee $O(nm)$ anymore. Remember that traversing the subtree is paid from the built potential which can be used just once. Therefore we have to update the distances by some smaller value than Δ. Otherwise we'd have to ensure that the subtree isn't traversed again and would have to save all those nodes to be scanned once more or ensure the $O(nm)$ running time by some other amortization of maintaining the shortest path tree. Updating by Δ and removing the subtree also implies that those nodes may not be added to the queue again and the shortest path tree wouldn't be rebuilt. Removing the subtree and keeping the nodes in the queue could result in subsequently trying to add a node v as new child of its parent p while p itself isn't in the tree.

As the scan order doesn't depend on the representation of the shortest path tree (remember that we (de)activate a node in the queue or stack by setting a flag) the number of node scans doesn't differ between data structure 1 and 2 (ref. subsection 3.2). Running time differences are in the order of measuring accuracy.

3.4 More heuristics using other data structures for the sets A and B

With other data structures, like heaps or arrays, it's not useful to consider all combinations. If set A is a heap, the order of nodes in set B doesn't matter, so we'll use a method with least overhead (if the heap is array based, B will be an array). Combining anything with a heap as set B doesn't seem to make sense (furthermore it wouldn't be clear whether to sort nodes in set B first). We'll consider a family of BFM-HA algorithms with different strategies for negative cycle detection. The robust dijkstra algorithm – suggested by [CGG$^+$09] – is BFM-H$_{\text{RD}}$A1T$_{\text{UP}}$. The heap order used is by biggest improvement since the last scan, so we have to save another value with each node, used as reference value (we use subscript RD, an abbreviation used for the algorithm in [CGG$^+$09]). At the beginning of the algorithm we initialize this reference value to 0 for all nodes. In each iteration after swapping the sets A and B we'll heapify the nodes in the array A. The idea is to scan nodes with big improvements first, so that these improvements will propagate through the graph. Another natural heap order considered is by lowest potential (use subscript LP). A third and fourth variant passes on a heap order – these are BFM-AA (which behaves very similar to BFM-SS – we copy new nodes to the end and remove from the end of the arrays, the only difference is when using subtree disassembly where we remove nodes from the array, but only disable them in the stack), and BFM-RA which chooses the node to be scanned next randomly from set A (remove a random element and fill the position with the array's last element). And finally we try to avoid the logarithmic cost of heap operations in BFM-H$_{\text{RD}}$A by making the array a heap but work with it like an array afterwards – we call it BFM-A$_{\text{RD}}$A (the heap order will be more and more void as the algorithm proceeds until the next pass begins with a correct heap again). All five variants are reviewed with always adding decreased nodes into set `B` or allowing to insert into set `A` combined with detecting negative cycles by parent pointers or with both versions of subtree disassembly without and with update (making this a total of 50 variants). Unlike in the queue and stack versions we actually remove nodes from the heap or array during subtree disassembly as using flags in the array is unnecessary (we just copy the last element at the deleted position), and deleting elements from the heap keeps the heap smaller. With arrays and heaps the order in which nodes are removed has an effect on the resulting data structure and therefore on the order in which nodes are scanned.

For BFM-H$_{\text{RD}}$AT we also implemented versions that disable nodes in the heap during subtree disassembly, but keep them there (just like we did in the queue and stack versions) – when a node to be scanned is disabled, we simply skip it – these versions are denoted with a subscript BFM-H$_{\text{RD,D}}$AT. We'll give details in section 5. Note, that this has a special effect in the BFM-H$_{\text{RD}}$A2T algorithms – when we remove a node from set `A` and try to add it again in the same pass, the BFM-H$_{\text{RD}}$A2T algorithm will add it to set `B`, but BFM-H$_{\text{RD,D}}$A2T just flips the flag and the node is back in set `A` again and can still be scanned in the current pass.

4 Experimental Setup

To be able to get reproducible results we use the pseudo random number generator TT800 (Mersenne Twister, [MK94]).

The main focus of this series of technical reports is on difficult graphs, so we tried to give a class of graphs which have deep shortest path trees which are well hidden. A hamiltonian path is randomly generated to avoid any (dis-) advantages concerning the order of nodes in the adjacency list. Arcs on this path have cost -1. All other arcs have cost in the interval $[n, 2n[$. In order to have many negative arcs, but no negative cycle, we use a high random potential to hide the hamiltonian path. The potential is chosen randomly from the interval $[0, n^2[$ – so almost half of the edges have negative cost after applying the potential. Besides the complete graphs, we also have graphs with a hamiltonian path, but less arcs.

This class of graphs (which we call “deep tree”) is tested with

- 500, 1000, 2500, 5000, and 10000 nodes on a complete graph
- 2500, 5000, 10000, 25000, and 50000 nodes, with $m = n \cdot \lfloor\sqrt{n}\rfloor$ edges
- 10000, 25000, 50000, 100000, and 250000 nodes, with $m = 4n$ edges

See results in subsection 5.2. It’s possible to have a hamiltonian cycle instead. The graphs generated are identical (when TT800 is initialized with identical seed) except the one edge closing the hamiltonian path. The cost of this one edge is chosen to have a cycle of length -1. See results in subsection 5.3.

We also tested graphs without deep hidden paths, with the same number of nodes and edges. Arc weights are chosen randomly from the interval $[0, n[$ with potential in $[0, n^2[$.

Two more graph classes were tested (to compare with the difficult graph with $m = 4n$ arcs): We have 4-regular graphs (the adjacent nodes are chosen randomly) and a lattice on a torus (also 4 arcs per node, one in each direction). Arc costs and potential are chosen from $[0, n[$ and $[0, n^2[$, respectively. All these are tested without negative cycles in subsection 5.4 and with negative cycles in subsection 5.5.

All these combinations were tested on 250 graphs each.

In the FP context we initialize set `B` with all nodes (in the first pass these are scanned in ascending order of their ids – this is different from other evaluation papers that initialize set `B` only with nodes with negative outgoing arcs, but don’t consider the time for this preprocessing – the way we do it, the first iteration does this “preprocessing”, we think this better reflects the total work needed in the algorithms). The hamiltonian path of the deep-tree graphs is a random permutation, hence, we won’t find it at the first glance. Nevertheless, the order of arcs in the adjacency list may matter – we consider two natural orders, that is concerning ascending and decreasing ids of the arcs’ tail nodes.

The first experiments consider graphs without negative cycles. To get graphs with negative cycles, we change the deep-tree class to close the hamiltonian path to a cycle with cost -1. On the other random graphs we reduce all edge weights by an offset, i. e., choosing edge weights from the interval $[\text{offset}, \text{offset} + n[$. With offset 0 there can’t be a negative cycle (this is the setting in subsection 5.4), with offset $-n$ there will be a negative cycle. The critical offset that leads to negative cycles depends on the graph class. We’ll then examine the correlation between offset and running times: First going from the critical offset to more negative arcs (to analyse influence of negative arcs – the smaller the offset, the more and shorter (in terms of number of edges) negative cycles can be found in the graphs. Then going from the critical offset to offset zero (so there's still many negative arcs due to the high potential used, but no negative cycles), and finally going from offset zero to positive offsets (so the graphs will have less negative arcs). This is done in subsection 5.5.

All tests were performed on an Intel Dual Xeon 3.06 GHz processor system with 4 GB RAM running Linux (Gentoo) with kernel 2.6.28. All programs were compiled with gcc 4.1.2 using the `-O4` optimization option.

5 Experimental Results

Instead of giving some tables with average running times or number of node scans, we tried to determine a function $t(n) = c \cdot n^k$ that describes the data well. We used linear regression on a doubly-logarithmic chart. With $\log(t(n)) = \log c + k \cdot \log(n)$ we can use the method of least squares (the coefficient of determination R^2 was > 0.997 for all experiments (> 0.9995 for most), so the method seems reasonable enough). As the cost per node scan differs (using heaps is more costly than arrays, subtree disassembly has an overhead that's different from negative cycle detection by following parent pointers, ...), we should base our analysis on real running times rather than using node scans. Nevertheless there may be slight variations that can't be totally avoided. In fact, the experiments showed effects of the processor cache (more cache hits on smaller graphs) that lead to some distorition. So we begin with functions describing the number of node scans for the considered graph classes and will take into account the real running times in a final comparison.

For every algorithm and graph class we used five quantiles (0.1, 0.3, 0.5, 0.7, and 0.9), determined the values for c and k by the method of least squares for each quantile, and displayed these as a line in the figures. A short line means that there's low variation on the number of node scans, a longer line means higher variation. Note, that due to limitations on the number of experiments, we have to take care when interpreting the results. Often there's just a slight difference in c and k where k is slightly larger and c is slightly lower, which may be rather a result of randomness due to the random graphs considered. Also polynomials of less degree affect the constants. To give an idea of the effect, we consider the "measured points" (500,247500), (1000,995000), (2500,6237500), (5000,24975000), (10000,99950000) (i. e., $t(n) = n^2 - 5n$). The function determined here is $t(n) = 0.973 \cdot n^{2.003}$. To give an idea of the effect that cache hits may have on the evaluation, we consider a 20% reduction on small graphs ($n = 500$ and $n = 1000$) for points (n, n^2) (i. e., (500,20000), (1000,800000), (2500,6250000), (5000,25000000), (10000,100000000)), we get $t(n) = 0.454 \cdot n^{2.09}$. If we assumed a 50% reduction for $n = 500$ and $n = 1000$, we even get $t(n) = 0.086 \cdot n^{2.281}$, which shows an obviously wrong bias. Note, that the latter effect concerns running times only and the effect is similar on all algorithms! We can compare the determined degrees for node scans and running times – these should differ by about the average degree (number of arcs considered per node scan) in terms of n^k, but as we will see the effect of the processor cache is much larger.

5.1 General results

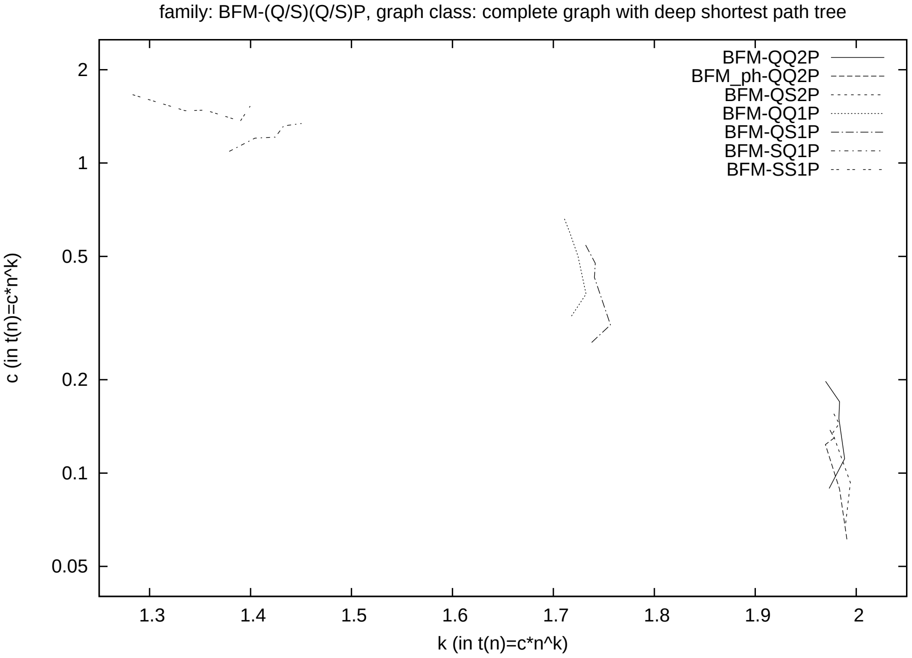
Tests concerning the order of arcs in the adjacency list showed no significant difference. It's just important to take care, that a constructed shortest path is hidden by permutation (otherwise we could find it in the first iteration by following the n nodes for example in ascending order – the way we initialized set `B`).

Also there was no significant difference comparing the two data structures for subtree disassembly. The number of node scans may be affected only in algorithms that explicitely remove nodes from the sets `A` and `B`. But also there, we measured no significant difference. Running times were also almost identical (within accuracy of measurement). Data structure 2 has a little more overhead during initialization, but slightly less on each insertion or removing of a node. The memory requirements are the same when we also use parent pointers for the shortest path tree along with data structure 2 (maintainance of the preorder traversal). Taking the code, data structure 2 should show a slight advantage – but we could measure this (small) effect only on bigger graphs.

An interesting detail is the comparison of family BFM-SS with BFM-AA. The first implements the stack via pointers that are part of the node data structure, the second uses an array of pointers and behaves like a stack (the node to be scanned is taken from the end of the array, newly added nodes are copied to the end). As the stack scans the nodes in increasing order of their ids in the first iteration, but the array in decreasing order, we have a comparison if the order in the initialization makes a difference – but there was no significant difference. The algorithms differ when using subtree disassembly because the stack versions disable nodes (but leave them at their position in the stack) while the array versions actually remove the node (by copying the last node of the array to the deleted position). The number of node scans needed is almost identical on the average. Furthermore, there's no real difference in terms of running time (very slight advantage for the stack version using a pointer within the node data structure – this may be different on other computer architectures).

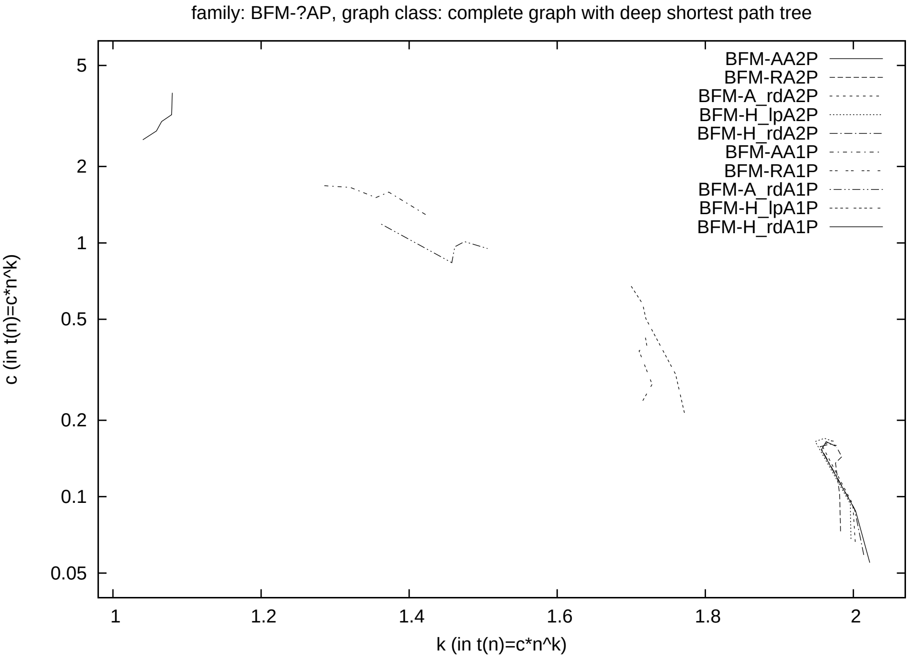
5.2 Graphs with deep shortest path tree

We considered graphs with a deep shortest path tree with $n(n-1)$, $n\sqrt{n}$, and $4n$ arcs, respectively. It's constructed by a hamiltonian path (random permutation) with arcs of cost -1, the remaining arcs have cost in $[n, 2n[$ (as arc costs are $\geq n$ there can't be negative cycles), and it's modified by random potential in $[0, n^2[$. Due to this random potential the actual shortest path tree depth is often slightly below n (the 0.1-quantile is at $\approx 0.98n$ on complete graphs, and $\approx 0.93n$ on graphs with $4n$ arcs – this reduction in depth happens when potential leads to positive reduced cost of the first arcs of the hamiltonian path). This leads to many passes in the classical Bellman-Ford-Moore algorithm. And in fact, the classical algorithm needed $\Theta(nm)$ node scans and time on all of these graphs. Let's first take a more detailed look on the class BFM-P with queues and / or stacks and following parent pointers for detecting negative cycles.



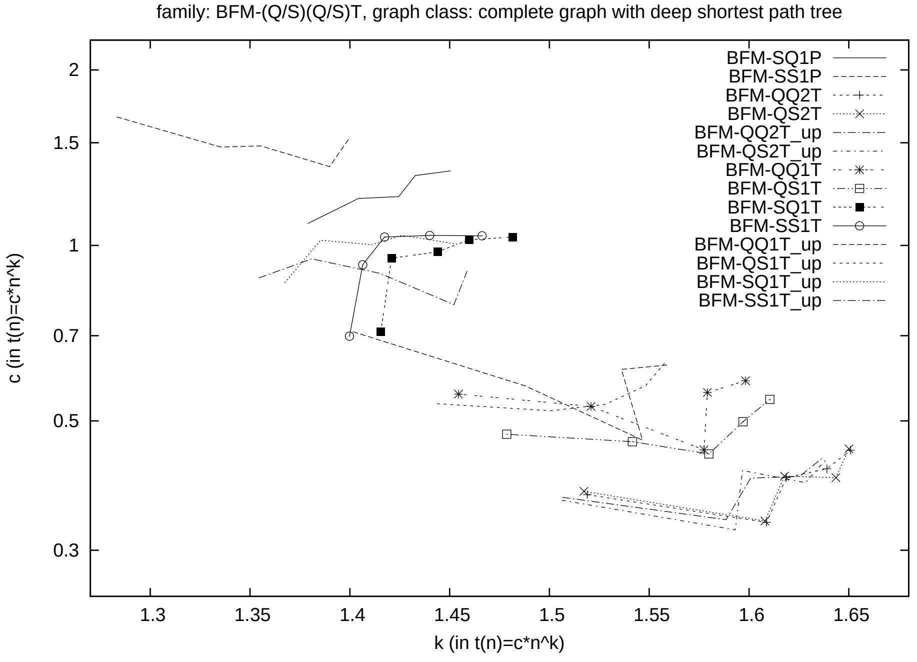
The first picture shows the result for complete graphs. The y-axis shows the constant factor, the x-axis the exponent of the polynomial (keep in mind that a node scan takes $O(n)$ steps on complete graphs). So the closer to the point of origin, the better. We see significant differences between algorithms that always insert into the second queue (worst), and furthermore a significant advantage of the BFMSQ1 and BFMSS1 algorithm. The picture is the same for graphs with deep shortest path trees with $n\sqrt{n}$ nodes (the exponent ranges from 1.26 via 1.52 to 1.74) and $4n$ nodes (exponent 1.18 via 1.30

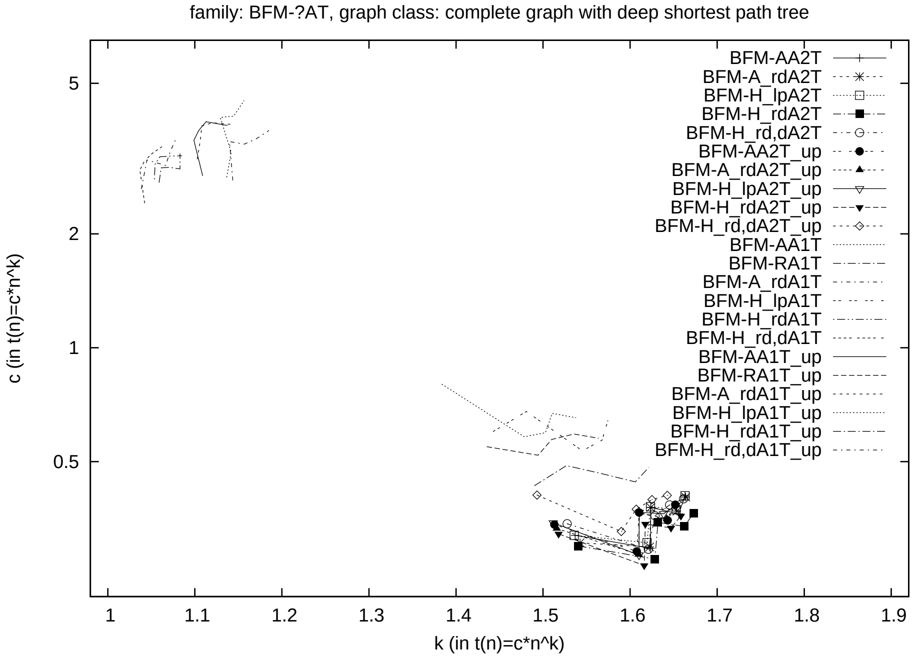
to 1.44). We also see that the parent heuristic just gives a constant factor over the original algorithm of about 12% on thin graphs to about 21% for complete graphs.



The heap and array combinations also show significant differences. Again algorithms which insert decreased nodes always into set B are worst. From the remaining variants the BFM-H$_{\text{RD}}$A1P performs best and shows least variance (short line in the picture). This is still true by far when we take into account the running times (despite of the more costly heap operations). The second best BFM-AA1P is almost identical to BFM-SS1P – followed closely by BFM-A$_{\text{RD}}$A1P. Using heaps (with lowest potential having highest priority) and choosing nodes randomly are worse. Again on graphs with less arcs we get similar pictures with the same clusters (exponents range from 1.08 via 1.28 and 1.50 to 1.75 and 1.08 via 1.19 and 1.29 to 1.43 for graphs with $n\sqrt{n}$ and $4n$ nodes, respectively).

When we add subtree disassembly, we hope to reduce the number of node scans. In the family BFM-T this gives an advantage over BFM-P for all algorithms but the previous best ones using a stack for set A (BFM-SS1P and BFM-SQ1P) where number of node scans and running times are almost identical. The algorithm BFM-H$_{\text{RD,D}}$A1T performs best (with the other BFM-H$_{\text{RD}}$A1 algorithms being about 15% behind) both in terms of node scans and running times. Again the same picture for graphs with $m = n\sqrt{n}$ and $m = 4n$ nodes (BFM-H$_{\text{RD,D}}$A1T and BFM-H$_{\text{RD}}$A1T$_{\text{UP}}$ being the fastests, re-



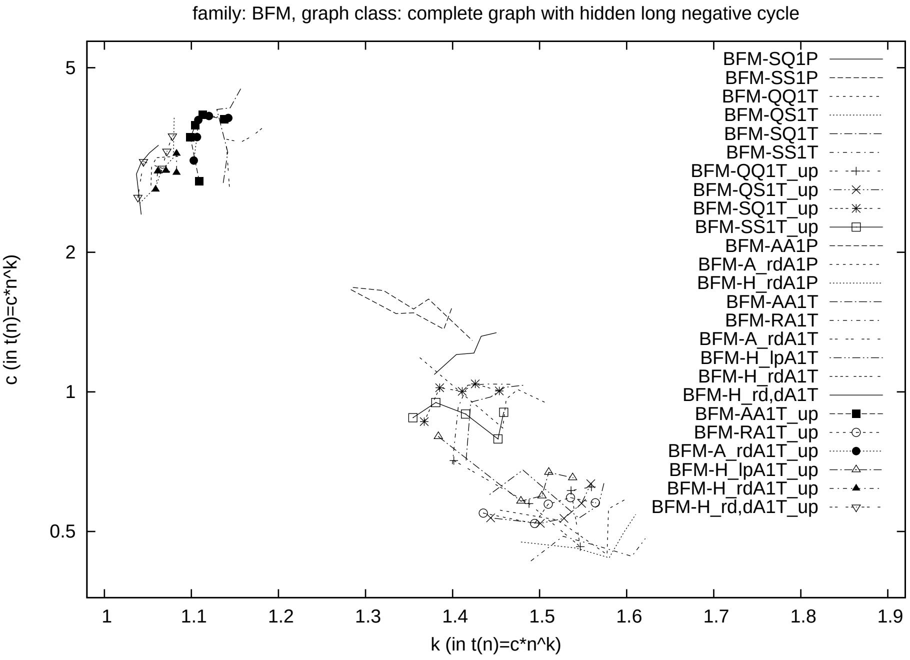


spectively, but without a perfomance gap that we measured on complete graphs). The algorithm BFM-H$_{\text{RD}}$A1P without subtree disassembly has no disadvantage! The hope that BFM-A$_{\text{RD}}$A1P could help getting rid of logarithmic factors wasn't appropriate. We may have the right heap order at the beginning of a pass, but new nodes are just added to the top of the "heap" instead of being inserted at the right position, so it resembles more a stack which corresponds with our experiments on graphs with deep shortest path trees.

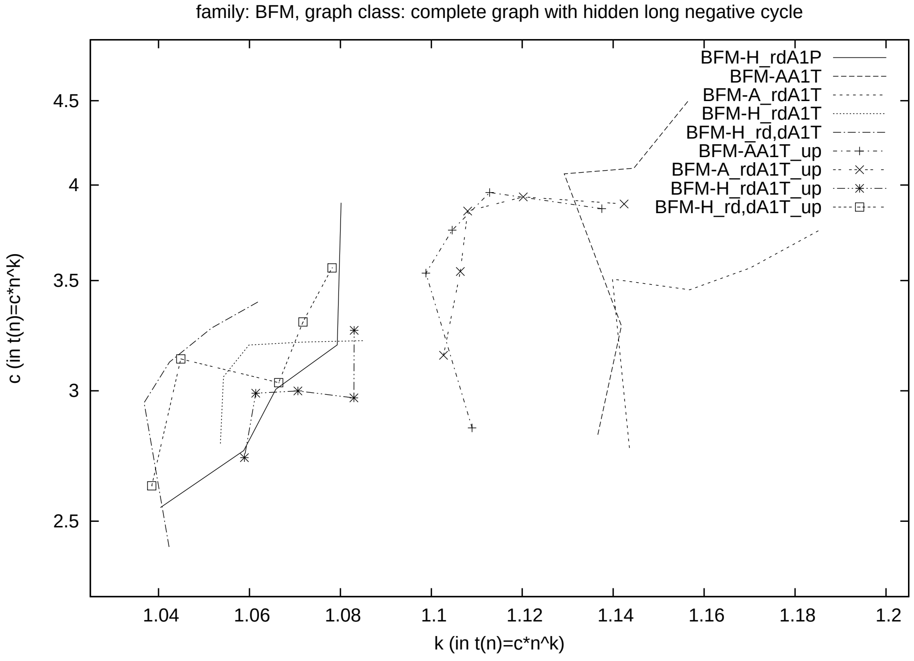
While subtree disassembly shows more benefit when using other data structures than H$_{\text{RD}}$, it shows no real advantage for the BFM-H$_{\text{RD}}$A1 family on these graphs with deep shortest path tree.

5.3 Graphs with hard to find negative cycles

This class is almost identical to the class with deep shortest path tree – we just change the length of the arc that closes the hamiltonian path, such that the cycle gets length -1. It's obviously the only negative cycle. We compared only the algorithms which allow to insert a node to set `A` where approriate (family BFM-1 dominated BFM-2 (using identical data structures) on this class of graphs). We take a closer look at the figure for complete graphs again.



We see some clusters, all BFM-H$_{\text{RD}}$A1 are best, followed closely by BFM-AA1T$_{\text{(UP)}}$ and BFM-A$_{\text{RD}}$A1T$_{\text{(UP)}}$, from the remaing algorithms those with stack-like data structures for set `A` show better performance, but all those are by far worse than BFM-H$_{\text{RD}}$A1.



A close-up of the best algorithms shows a picture almost identical to the one in graphs without negative cycle – all BFM-H$_{\text{RD}}$A1 algorithms are in front, the only of these that shows slightly better performance is – again – BFM-H$_{\text{RD,D}}$A1T (without update only – advantage is about 15%, the other three subtree disassembly variants are within a 3% range). Also when considering runtime, it's the best – comparing it with the best non-heap algorithm BFM-AA1T$_{\text{UP}}$ the difference of node scans (21653 opposed to 44700 on an average for $n = 5000$) and runtime (2.41 opposed to 4.9 seconds) suggests that the heaps are small on these graphs and won't lead to much overhead, but in fact, the heaps are not small, but nodes don't move much levels in the heap, and often nodes aren't moved at all).

We see a difference between disabling (and maybe later reenabling) a node in the heap and removing (and maybe later reinserting). This needs some further research. One possible explanation is a different behaviour for heap elements with equal priority. Removing and later reinserting may change the order of two nodes with equal priority, while disabling and reenabling doesn't change it. At least for these complete graphs with hamiltonian negative cycles, this seems to be an advantage.

On graphs with less arcs, we have the same picture, but we see a bit more impact of the heap operations on graphs with higher number of nodes. Comparing the best one (here BFM-H$_{\text{RD}}$A1T$_{\text{UP}}$, the other BFM-H$_{\text{RD}}$A1 algorithms are behind within a 20% range) with the best non-heap algorithm BFM-AA1T$_{\text{UP}}$ the difference of node scans (981094 opposed to 1912686 on an average for $n = 100000$) and runtime (1.39 opposed to 1.68 seconds) shows the higher cost per node scan on graphs with many nodes.

The performance of all algorithms included for finding hamiltonian negative cycles was almost identical to those without this negative cycle. While this isn't too astonishing at the first glance, we could have hoped that, e. g. the algorithms that detect negative cycles by following parent pointers could have begun finding the cycle from different points putting the parent pointers together to a cycle, but the subtree disassembly would destroy all but one path that is finally closed to the negative cycle. This hope didn't come true on these graphs with just one hidden long cycle.

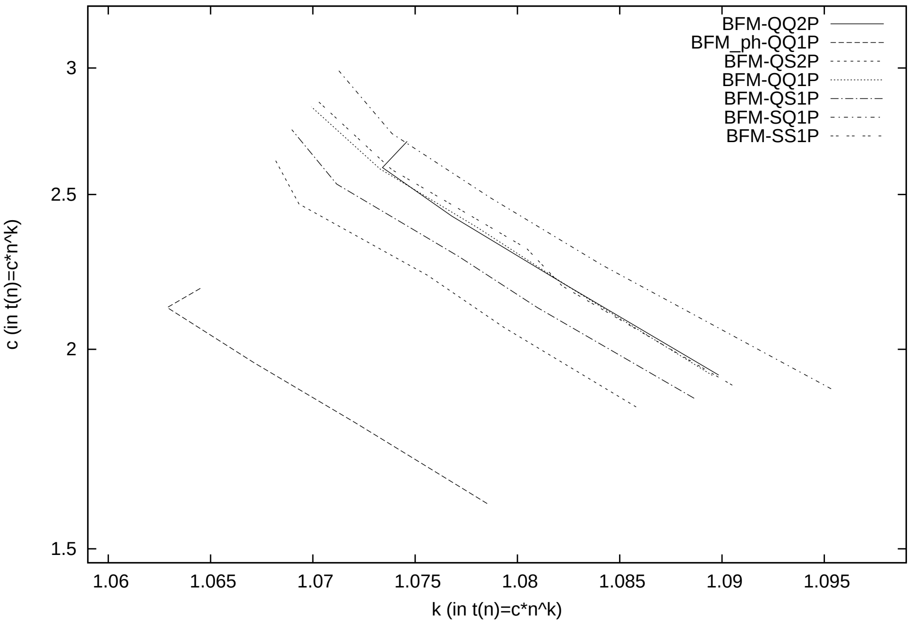
5.4 Random graphs without negative cycles

If we choose all arc lengths randomly, we often get shallow trees. While on complete graphs with deep shortest path tree we had depth of nearly n, on random graphs we had only measured about $0.26 \log^2 n$ (on graphs with $4n$ arcs the depth measured was about $1.6 \log n$). So we'll have very few passes compared to the number nodes and the differences between the algorithms are much less apparent. We show the figures for graphs with $m = 4n$ arcs (those for $n\sqrt{n}$ arcs and complete graphs look very similar).

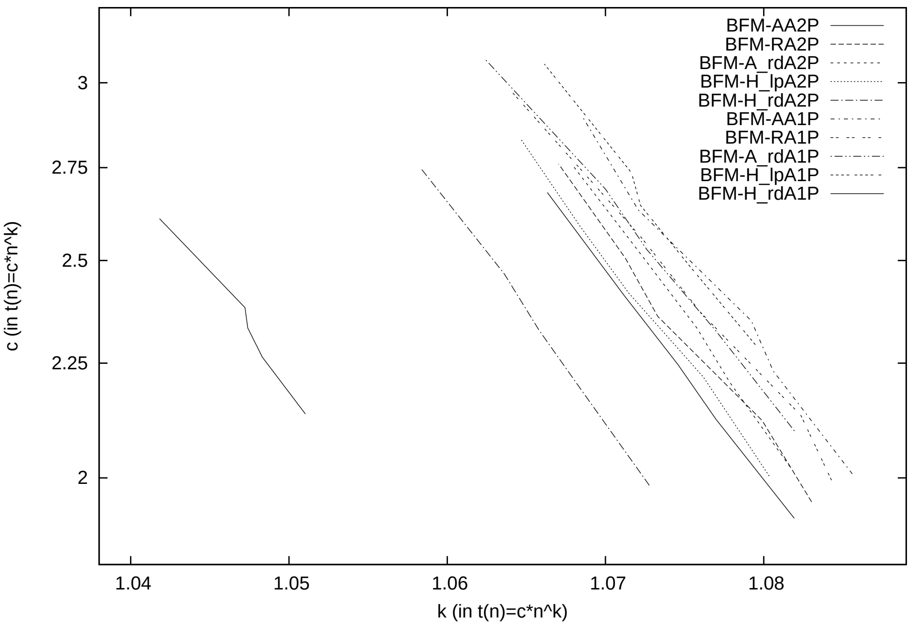
The algorithms without subtree disassembly are all dominated by others and all have identical performance within a small range. The BFM$_{\text{PH}}$ gives a slight advantage but it's still dominated by the subtree disassembly versions.

Using arrays and heaps, but no subtree disassembly BFM-H$_{\text{RD}}$A1P is the best once more. While this is true for the number of node scans, we must be more specific concerning running time. On dense graphs we see that the advantage in terms of node scans remains in terms of running time. When we compare BFM-H$_{\text{RD}}$A1P to BFM-AA1P on graphs with $m = n\sqrt{n}$ arcs, we have less node scans, but these even out with expenses of the heap operations. With thin graphs ($m = 4n$ arcs), let's give numbers for $n = 100000$ nodes, BFM-H$_{\text{RD}}$A1P needs about 400000 node scans and 0.81 seconds while BFM-AA1P needs about 585000 node scans, but only 0.51 seconds. On graphs with shallow shortest path trees the advantage of node scans for BFM-H$_{\text{RD}}$A1P isn't big enough to also have faster running times. On graphs with deep shortest path tree BFM-H$_{\text{RD}}$A1P was in a family of the fastest algorithms.

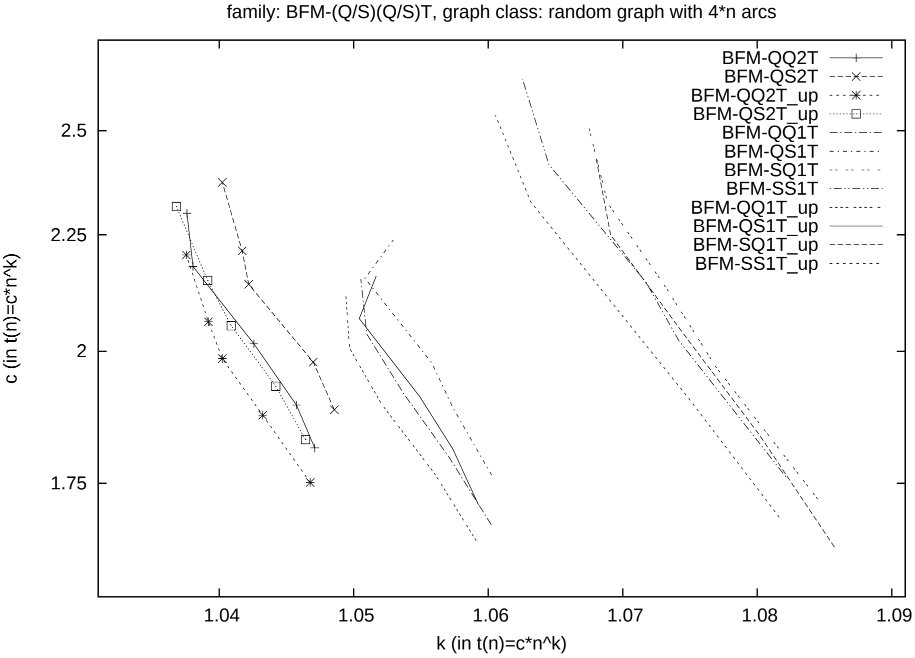
If we add subtree disassembly to the stack and queue versions, we note that on the random graphs there's no advantage to add decreased nodes to set `A`. While the difference is very small in terms of node scans, it get's more visible in terms of running times. When we always insert into set `B`, we don't have family: BFM-(Q/S)(Q/S)P, graph class: random graph with 4*n arcs



family: BFM-?AP, graph class: random graph with 4*n arcs



to maintain a flag whether a node was already scanned in the current pass. In our experiments this effect seemed to have been multiplied with the effect

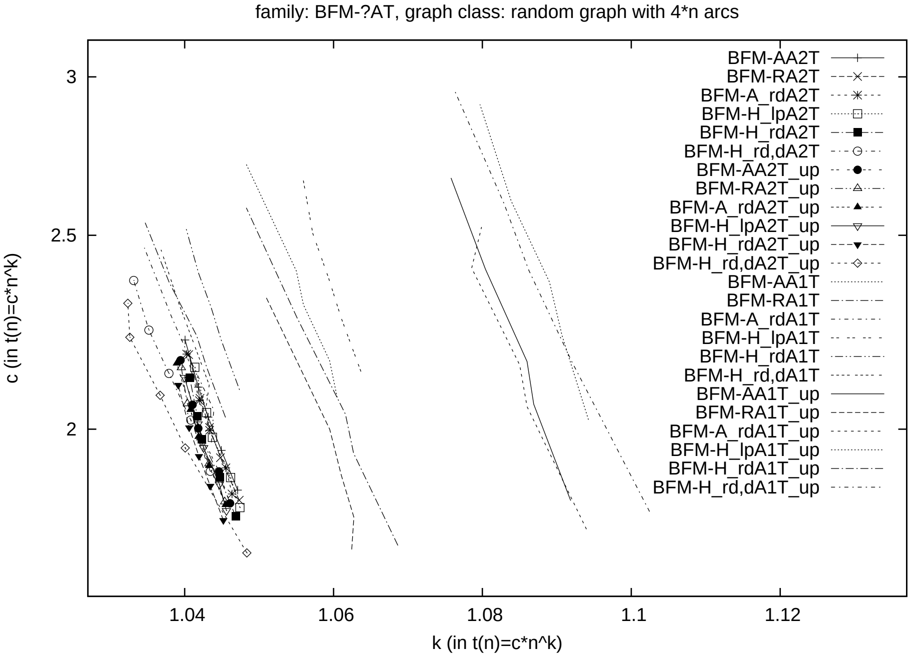


of cache hits / misses. While the number of node scans differs just slightly between BFM-QQ2T$_{\text{UP}}$ and BFM-QQ1T$_{\text{UP}}$ (e. g. about 8% less), savings in running times are more than 25% on an average.

From the remaining algorithms BFM-H$_{\text{RD}}$A2T$_{\text{UP}}$ has the least number of node scans (note, that the on the picture BFM-H$_{\text{RD}}$A1T$_{\text{UP}}$ has a smaller exponent, but bigger constant – for all graph sizes used in these experiments the small advantage in the exponent isn't big enough to compensate the bigger constant factor – this may also be an effect of the linear regression model along with variations in the measured data). Again the running times are worse by a factor due to higher cost of the heap operations.

Overall experiments on random graphs without negative cycles the simplest algorithm (the original BFM-QQ2) combined with subtree disassembly with update gives the best results in terms of node scans and running times.

The comparison of the results of random graphs (with $4n$ arcs) with 4-regular graphs suggests that the additional structure is too weak to have an effect on node scans and / or running times. The differences are within accuracy of measurement. So again, on these graphs with shallow shortest path trees the simple BFM-QQ2T$_{\text{UP}}$ algorithm is the best one. As on graphs with $4n$ arcs the difference between the best and worst algorithm tested considering number of node scans is a mere factor of about 2 (for $n = 100000$). And again, the algorithm BFM-H$_{\text{RD}}$A2T$_{\text{UP}}$ needs about the same number of node scans as BFM-QQ2T$_{\text{UP}}$, but running time is much slower (factor 2.85 on an

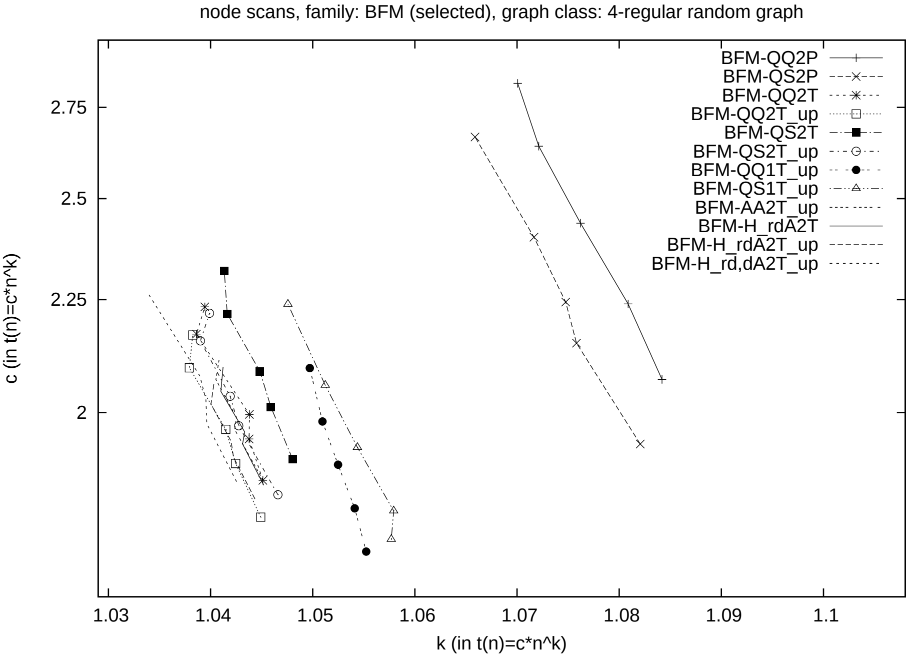


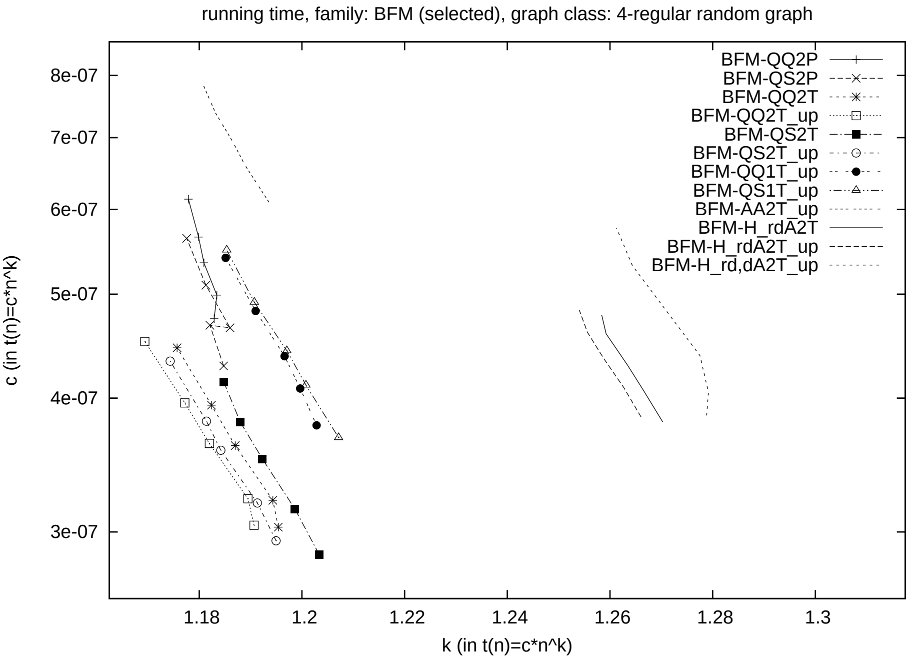
average for $n = 100000$ nodes). We show two figures – one showing the number of node scans, the other running times – for 12 algorithms (which include the algorithms which needed the least number of node scans and the fastest ones). We can see how all variants using heaps suffer from the higher cost per node scan due to the necessary heap operations.

On lattice graphs (square grid on a torus) the simple algorithms BFM-QQ2T and BFM-QS2T (with and without updates during subtree disassembly) dominate all others – in terms of node scans as well as in terms of running time. The algorithms that use heaps are again competative in terms of node scans but suffer from higher cost per node scan.

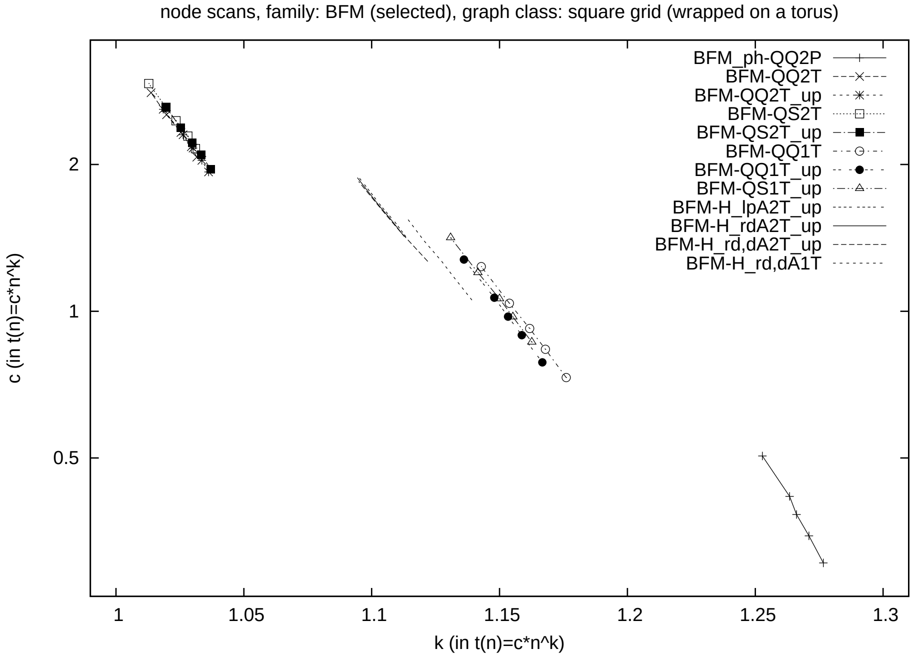
Both comparisons between number of node scans and running time show quite big effects of processor cache. On small graphs bigger parts of the graphs are held within the cache and reduce running time. Due to this effect the determined polynomials have a higher exponent on running time than on node scans. The logarithmic factor (250000 nodes on the biggest graphs opposed to 10000 nodes on the smallest for these two graph classes) is ≈ 1.35 which would be ≈ 0.09 in the calculated exponent. But most nodes move only very few levels in the heap (if any at all) – the real error in the exponent (due to heap operations) is much less.

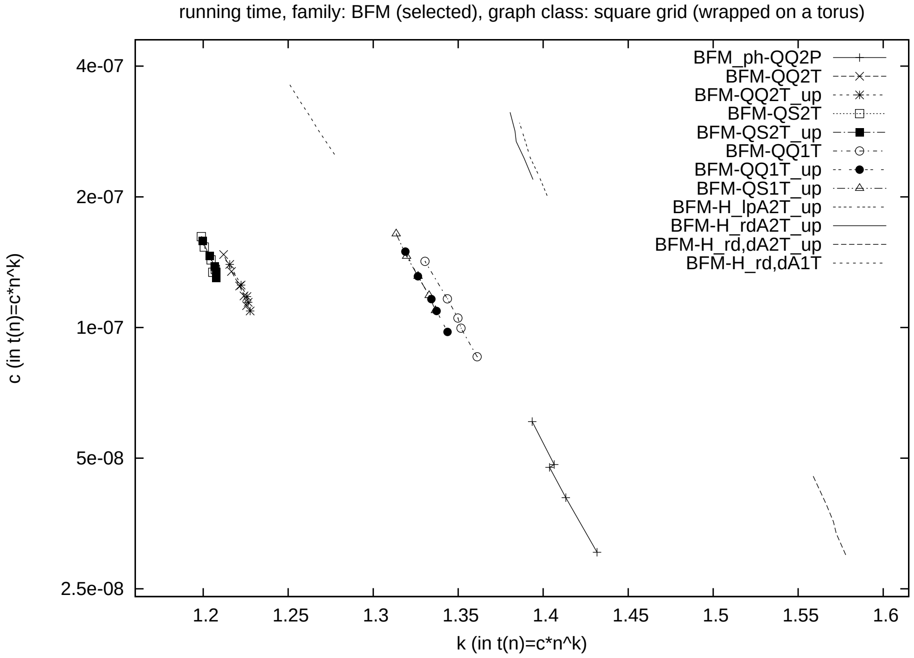
While the determined polynomials (concerning node scans) can be reproduced, the effects of processor cache depend much on what else the computer is doing and even on what was done on it before the experiment started.





On the figure that shows running times on lattice graphs the bigger exponent on the BFM-H$_{\text{RD,D}}$A2T$_{\text{UP}}$ algorithm is mostly due to a smaller running time on small graphs (compared to the other algorithms using heaps). So making 250 experiments on each graph class in a row may also reflect the





state of memory before the experiment – more than the real effect of the data structure. The number of node scans (and therefore almost all figures shown in this report) isn't affected. We will consider the effect of processor cache on experiments like these in another technical report [Lew10].

5.5 Random graphs with negative cycles

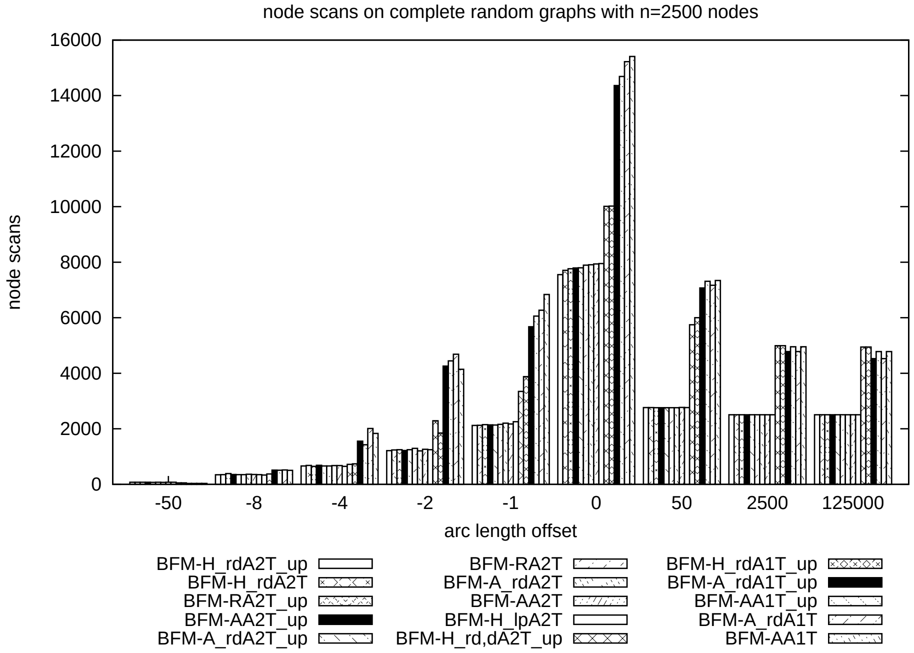
This section shows the correlation between "negativeness" of a graph and the node scans needed to detect negative cycles (or prove there is none). In our graph generator we can shift all arc weights up or down by any constant. What we do is to determine the lowest offset for a graph, such that it has no negative cycle, but it does have some when the offset is one less. From this "critical offset" we examine how the algorithms' performance changes with the offset going in both directions.

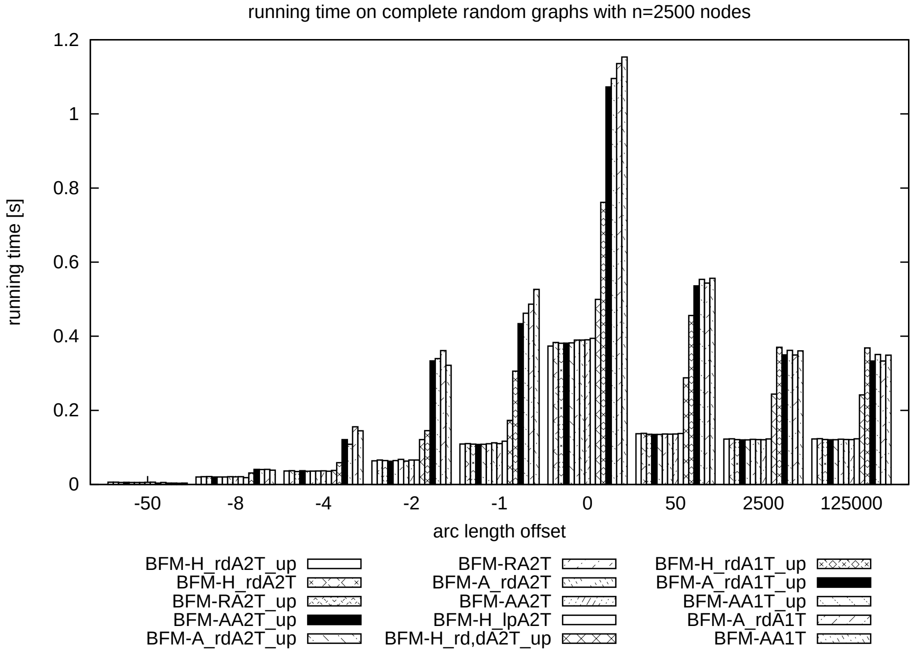
Again, we begin with complete random graphs, choosing arc lengths from $[0, n[$, random potential in $[0, n^2[$. An offset of -1 made the graphs having negative cycles on all instances. We choose the offsets $-\sqrt{n}$, -8, -4, -2, -1, 0, $\sqrt{n}$, n, and $n\sqrt{n}$ for each of 250 graphs and show the average total number of node scans – here for $n = 2500$ nodes.

The choice of the depicted algorithms was done such that it includes those that need the least number of node scans and the fastest – for the critical offset, offset 0 (if different), the lowest and highest offset tested. The algorithms are listed in order of increasing number of node scans needed for graphs with critical offset. For each class of graphs we show one figure with node scans and another with real running times.

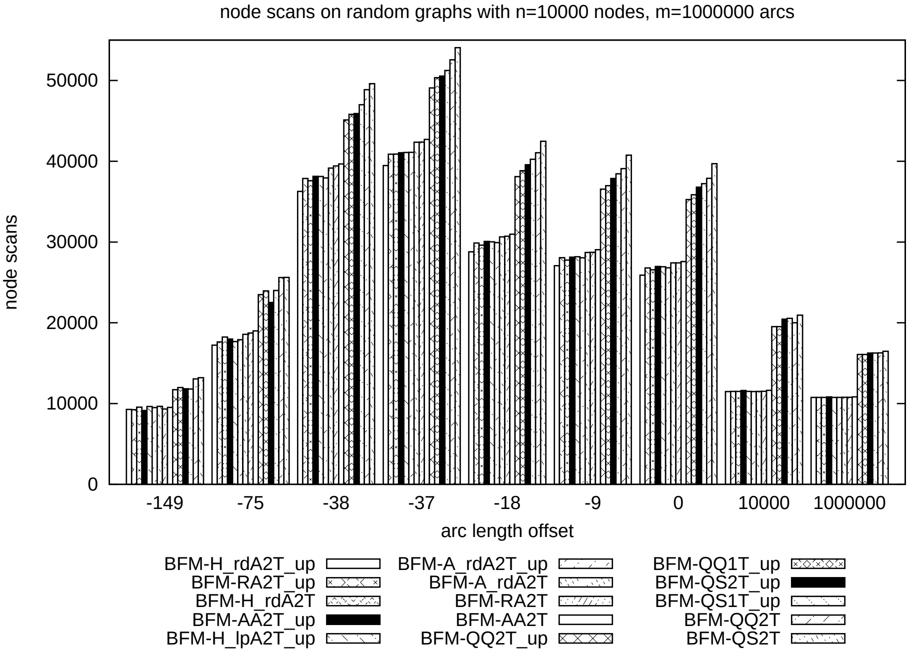
It's obvious how – on complete graphs – we get many cycles even with little negative offset. Algorithms with subtree disassembly benefit more (comparing offset 0 and -1) than those with following parent pointers (not seen in the figure due to the inferiority of following parent pointers on these graphs) – following parent pointers always needs one complete iteration, i. e., scanning at least n nodes before we check for negative cycles for the first time. Subtree disassembly allows to find negative cycles after no more than 2 node scans, on graphs with many negative cylces all algorithms using subtree disassembly get near zero node scans opposed to at least 2500 node scans in our example for algorithms with following parent pointers. Despite of the bigger constant in initialization when using heaps (but still linear) the advantage in the number of node scans of the BFM-H_{RD}A2T family is retained concerning running times on graphs with some negative cycles (we measured an average number of below 350 node scans with offset -8). An explanation for the behaviour of the heaps was already discussed in the previous subsection. It's important to note the difference between graphs with and without negative cycles. While BFM-2 algorithms are faster and need less node scans on graphs without negative cycles, BFM-1 algorithms get superior when the graph has many negative cycles (offset -50 in our example).

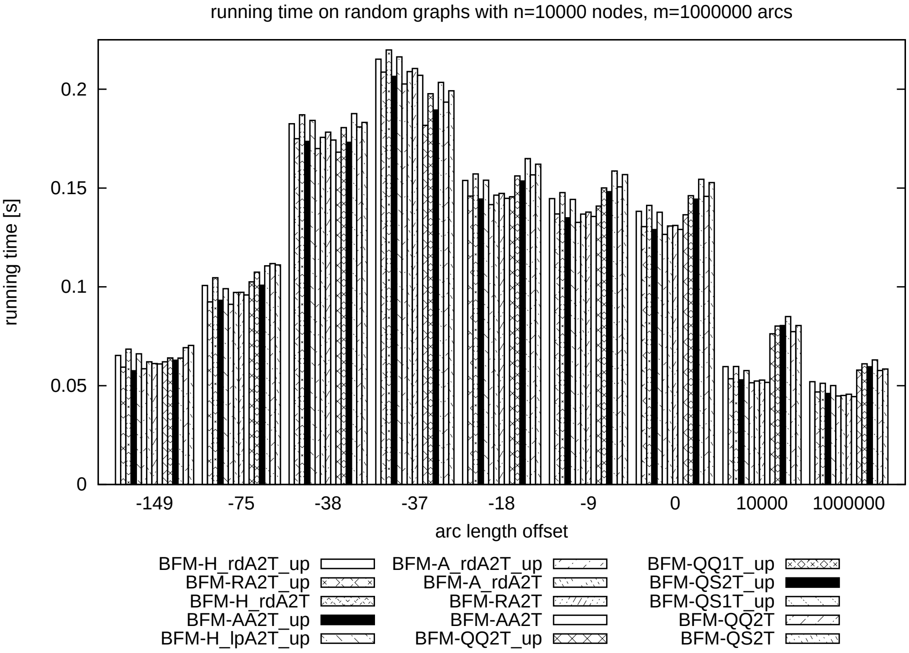
With less arcs ($m = n\sqrt{n}$ and $m = 4n$) the critical offset gets less. The number -37 mustn't be taken literally – we measured this average critical



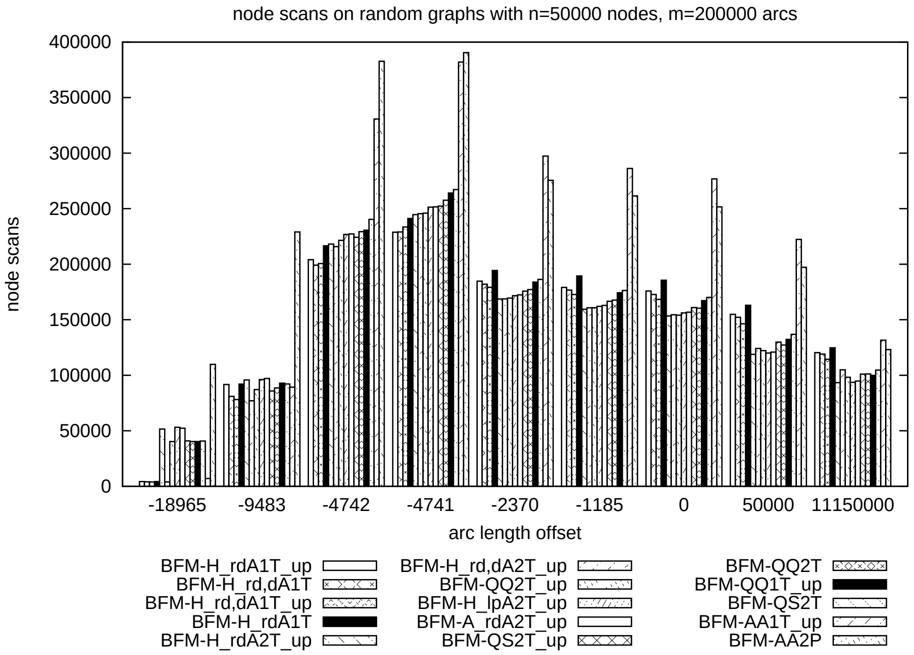


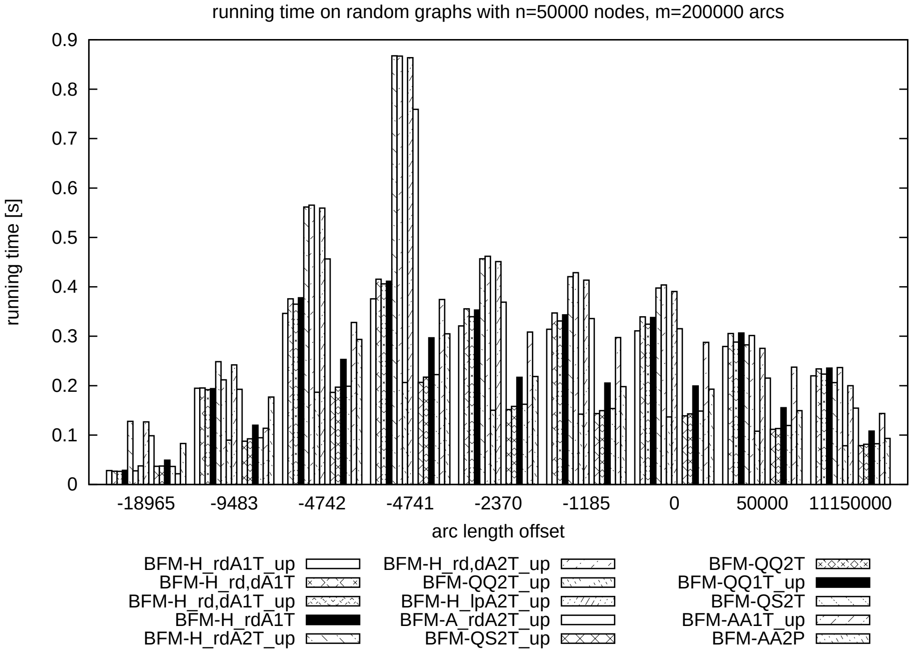
offset of -37 (for graphs with $n = 10000$ and $m = n\sqrt{n} = 10^6$). We determined the critical offset c for each graph and used as offsets for this test suite $4c-1$, $2c-1$, $c-1$, c, $c/2$, $c/4$, 0, n, $n\sqrt{n}$. Those resulted on an average in the numbers shown in the picture, though the individual graphs



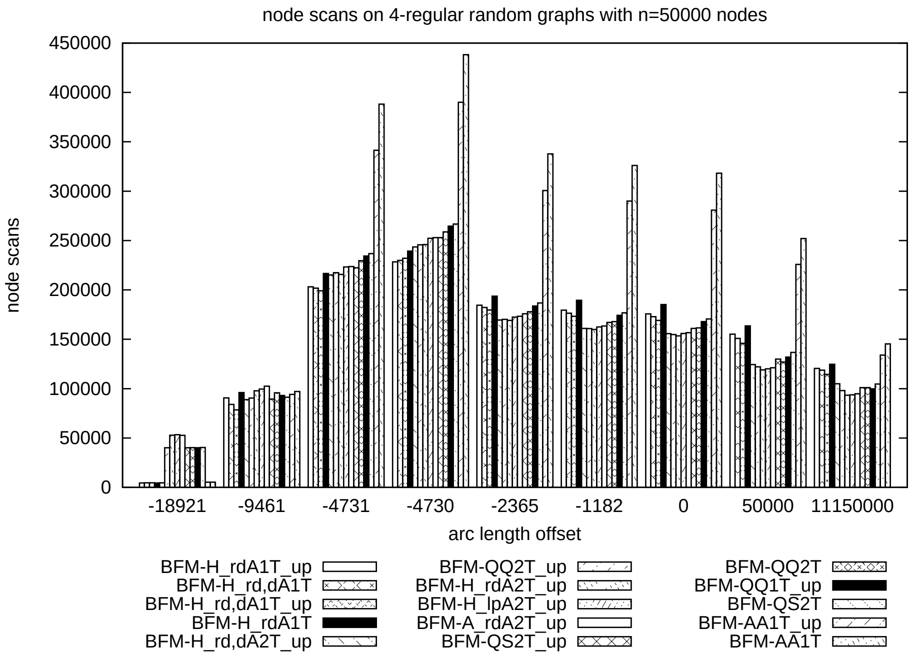


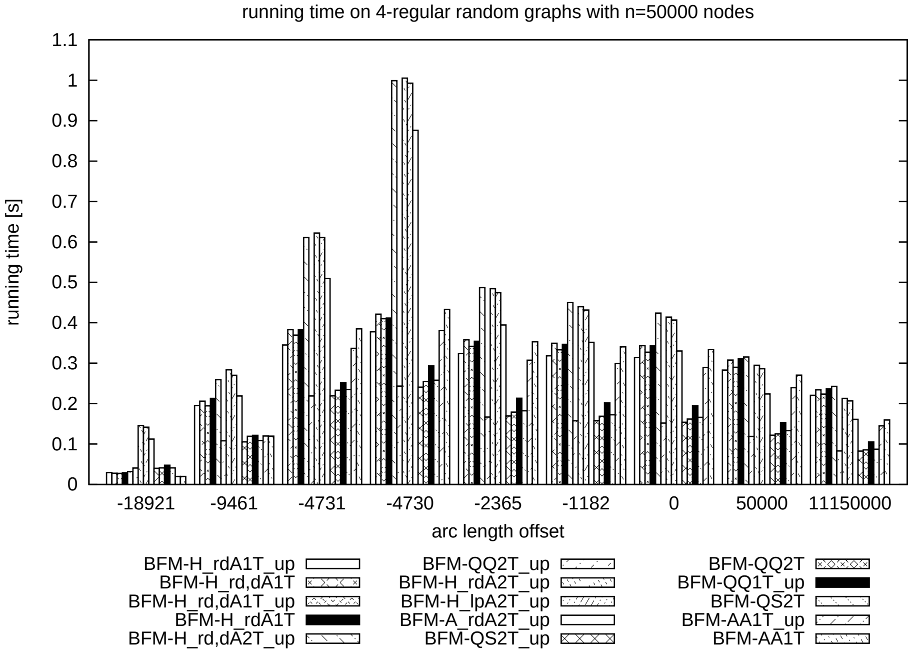
may have had different offsets. Most algorithms perform much better even when the critical offset was exceeded by 1 only, exceptions are the algorithms following parent pointers. The time needed for graphs with the critical offset is about 30% above time needed with offset 0. We discussed the graphs





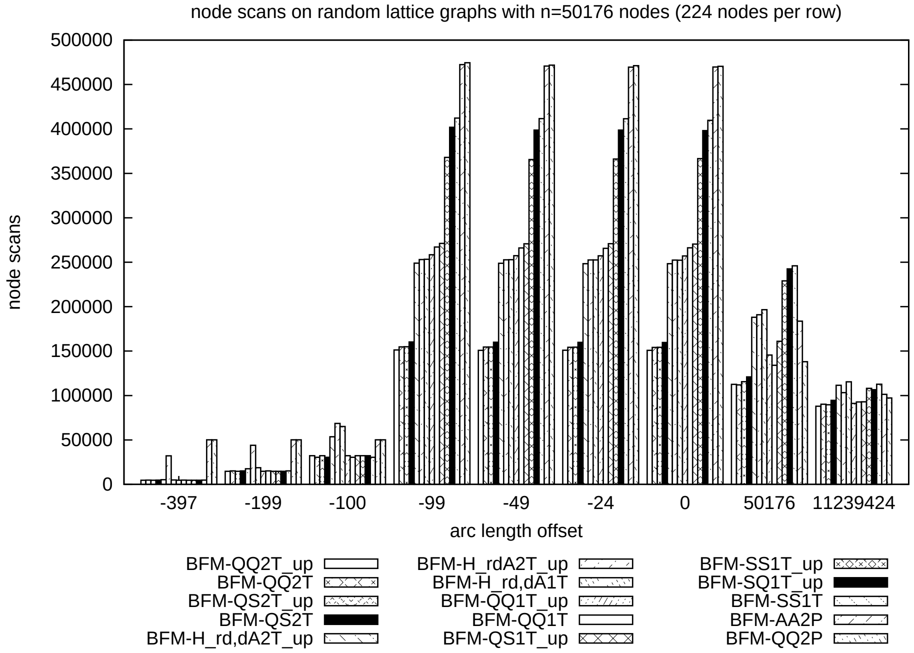
with offset 0 already. Tree depth of graphs with critial offset is higher (58 on an average opposed to 20 for graphs with offset 0 for $n = 10000$ nodes). We observe again differences between number of node scans and running times – we see that BFM-QQ2T$_{\text{UP}}$ and BFM-QS2T$_{\text{UP}}$ perform better than the

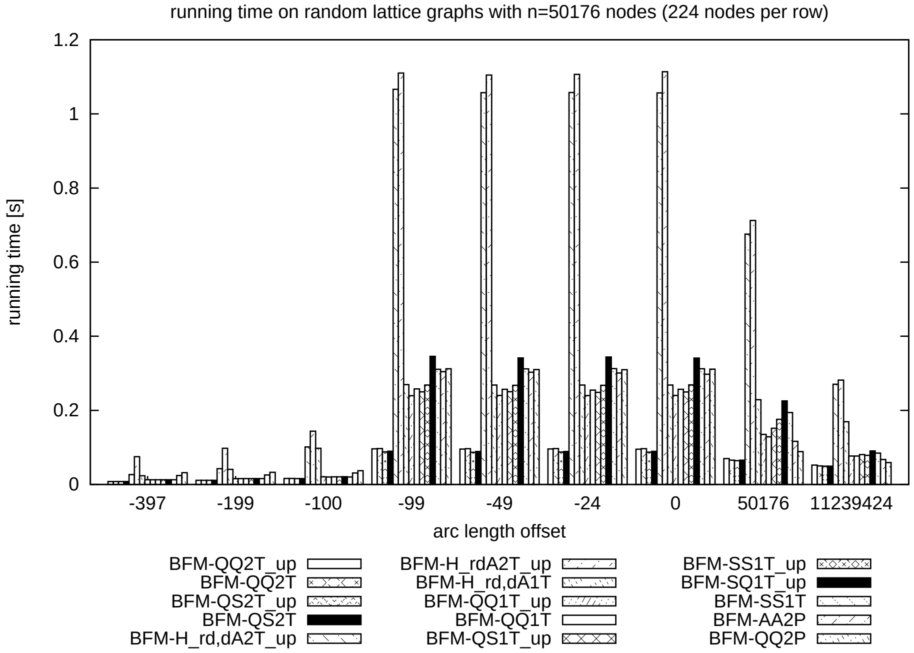




BFM-H$_{\text{RD}}$A2 family though the latter needs less node scans. This advantage dissolves on graphs with many negative cycles. If we lower the offset more, we observe the same phenomenon that BFM-1 algorithms get faster.

On thin graphs the advantage of BFM-1 algorithms is already visible on





graphs with offset near but above the critical offset (graphs without negative cycles). Otherwise we made similar observations – the BFM-H$_{\text{RD}}$A family needs the least number of node scans, but the higher cost per node scan makes the BFM-QQT and BFM-QST families superior. On graphs with many negative cycles BFM-AA1T$_{\text{UP}}$ is the fastest again (though the BFM-H$_{\text{RD}}$A family needs less node scans).

The comparison with 4-regular graphs suggests that the additional structure is too weak to have an effect on node scans and / or running times.

If we switch from 4-regular graphs to the fixed structure of lattice graphs we observe the most extreme behaviour at the critical offset. The average number of node scans decreases by a factor of nearly 10 for most of the algorithms! It's a step from an average tree depth of 122 ($n \approx 50000$) to some cycles with few edges that are found in the first pass by some of the subtree disassembly algorithms. It should be noted, that the variance in the number of node scans is small on graphs without negative cycles, but is big as soon as we go beyond the critical offset. The range of the number of node scans for the best algorithm BFM-QQ2T$_{\text{UP}}$ with critical offset is about 145000 (0.1-quantile) to 160000 (0.9-quantile), but with critical offset minus one we only need 7700 node scans at the 0.1-quantile, but still 102000 at the 0.9-quantile. The behaviour is similar for all algorithms on this class of graphs. Also the behaviour concerning the offset between critical offset and 0 is quite peculiar. While on other graph classes, the algorithms get faster, we don't see this effect on lattice graphs.

The high running times of BFM-H$_{\text{RD}}$A family can't be explained by higher cost of heap operations. As the number of passes is low such that the higher cost of initialization of a pass (building the heap) can't be responsible, we assume some weird effect of cache misses due to removing nodes from the heap and adding nodes in some other memory area.

6 Conclusions and Outlook

We gave some more detailed insight on the running time behaviour of the classical BFM algorithm and its variations. Especially, our experiments showed dependencies on the shortest path tree depth and length of shortest cycles. For graphs with shallow shortest path tree or few negative cycles with few edges BFM-2 algorithms that add a node always to set `B` seem to be superior. On graphs where the algorithm may follow deep shortest paths or long negative cycles BFM-1 algorithms are better despite of the extra maintainance needed to decide whether a node was scanned in the current pass yet. This advantage is especially noticeable when comparing to BFM-2 algorithms that need to build a heap when initializing a pass, those algorithms need much more passes – and time – than their BFM-1 relatives.

While the family BFM-H$_{\text{RD}}$A needs the least number of node scans on many graphs, it's not necessarily the fastest due to higher cost per node scan. Attempts like the BFM-A$_{\text{RD}}$A family to adapt the heap behaviour at lower cost didn't lead to competitive algorithms yet. One may try to develop a relaxed heap property with lower cost per operation.

Further experiments are needed to give some more detailed insight and to show dependencies on tree depth, average degree of the shortest path tree and similar properties.

We've also seen big effects of processor cache on running times, this topic is considered more detailed in [Lew10].

We included only straight forward variations of the classical Bellman-Ford-Moore algorithm in this technical report. Next to well known heuristics, e. g. Goldberg-Radzik [GR93], we also didn't include combined algorithms here like switching from BFM-SS1T$_{\text{UP}}$ to BFM-QQ2T$_{\text{UP}}$ after the first iteration (called HYB in [CGG$^+$09]). These and other algorithms will be considered in another technical report.

References

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

[CG99] Boris V. Cherkassky and Andrew V. Goldberg. Negative-cycle detection algorithms. *Mathematical Programming*, 85:277–311, 1999.

[CGG$^+$09] Boris V. Cherkassky, Loukas Georgiadis, Andrew V. Goldberg, Robert E. Tarjan, and Renato F. Werneck. Shortest-path feasibility algorithms: An experimental evaluation. *J. Exp. Algorithmics*, 14:2.7–2.37, 2009.

[CGR96] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(4):269–271, 1959.

[For56] L. R. Ford. Network flow theory. Technical Report P-923, The Rand Corporation, Santa Monica, CA, August 1956.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[GKP85] F. Glover, D. Klingman, and N. Phillips. A new polynomially bounded shortest path algorithm. *Operations Research*, 33(1):65–73, 1985.

[Gol95] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, June 1995.

[GR93] Andrew V. Goldberg and Tomasz Radzik. A heuristic improvement of the bellman-ford algorithm. *Applied Mathematics Letters*, 6(3):3–6, 1993.

[Lew05] Stefan Lewandowski. *Vereinheitlichte Darstellung von Techniken zur effizienten Kürzeste-Wege-Suche.* Dissertation, Fakultät für Informatik, Elektrotechnik und Informationstechnik, Universität Stuttgart, 2005.

[Lew10] Stefan Lewandowski. Processor cache effects on running time – an experimental study, 2010. unpublished yet.

[MK94] Makoto Matsumoto and Yoshiharu Kurita. Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 4(3):254–266, 1994. http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/emt.html.

[Moo59] E. F. Moore. The shortest path through a maze. In *Proc. of the Int. Symp. on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[NPX99] M. Nonato, S. Pallottino, and B Xuewen. Spt_l shortest-path algorithms: Reviews, new proposals, and some experimental results. Technical Report TR-99-16, Dipartmento di Informatica, Pisa University, Italy, 1999. http://compass2.di.unipi.it/TR/files/TR-99-16.ps.gz.

[Pal84] Stefano Pallottino. Shortest-path methods: Complexity, interrelations and new propositions. *Networks*, 14:257–267, 1984.

[Pap74] U. Pape. Implementation and efficiency of moore-algorithms for the shortest route problem. *Mathematical Programming*, 7:212–222, 1974.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Tar81] R. E. Tarjan. Shortest paths. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, 1981.

[WT05] Chi-Him Wong and Yiu-Cheong Tam. Negative cycle detection problem. In *Algorithms - ESA 2005*, pages 652–663, 2005.