A Scrutiny of Frederickson's Distributed Breadth-First Search Algorithm

Victor van der Veen
Faculty of Computer Science
Vrije Universiteit Amsterdam
vvdveen@cs.vu.nl

September 2008

Abstract

Frederickson outlined a distributed breadth-first search (BFS) algorithm which constructs a BFS tree in levels. The description of the algorithm is short and first-in-first-out (FIFO) channels are assumed. In this paper, we present a detailed description of a modified verison of the algorithm and we show that the algorithm functions in a non-FIFO environment. To allow a programmer to implement the algorithm, pseudocode of the adapted algorithm is provided.

1 Introduction

Consider a problem in which information must be routed among nodes of a network. A fundamental question in distributed computing is how to route this information, involving a minimum number of nodes or paths having lowest costs. For this, we need algorithms that construct such a "best" path between nodes in a network. Once optimal paths have been determined, messages can be routed through the network efficiently.

To route information using a path with lowest costs, all-pairs shortest-path algorithms are used. Routing information using the minimum number of nodes can be done with a breadth-first search (BFS) algorithm.

Frederickson [1985] outlined a distributed BFS algorithm which he used for constructing an all-pairs shortest-path algorithm. However, this BFS algorithm is used as a "building block" and its description is not very detailed. The description assumes that messages in the algorithm are processed in a first-in-first-out (FIFO) fashion.

Tel [2000] reviewed and modified Frederickson's BFS algorithm. His notation of the algorithm, however, is incomplete and still assumes a FIFO-environment. Tel [2006] outlined a fix, but again, the FIFO assumption remained. Van Moolenbroek [2006] suggested another fix, but no further research was done on the subject.

In this paper, we present a detailed description of Frederickson's distributed BFS algorithm, filling in the remaining gaps. We also provide pseudocode of the algorithm to illustrate how the algorithm operates. The correctness of the algorithm when messages are processed in a non-FIFO fashion is demonstrated.

The outline of this paper is as follows. We first provide some preliminaries in Section 2, followed by some related research on BFS algorithms in Section 3. Then, as in the original paper, we divide the description of Frederickson's BFS algorithm in a "simple" (Section 4) and an "advanced" part (Section 5). Section 4 contains a description, example and complexity analysis of the Simple Algorithm. Section 5 provides, besides a description and complexity analysis of the Advanced Algorithm, also more information about previous notations of the algorithm. We also show that the Advanced Algorithm functions in a non-FIFO environment. Finally, we state our conclusions in Section 6.

Note that since some figures used in this paper were too large to fit with the text, we moved them to the appendix.

2 Preliminaries

In this paper, we outline a distributed algorithm for constructing BFS trees. BFS trees provide useful building blocks for a number of routing and control functions in communication networks. Such a network can be seen as a graph, and hence graph theory is relevant to the subject of this paper. Although it is assumed that the reader has basic knowledge about graph theory, we provide some important definitions in this section.

For a formal definition of a BFS tree, the following two definitions are required. We denote a graph as $G = (V, E)$, V being the set of nodes and E being the set of edges between these nodes.

Definition 1 (Tree)**.** A *tree* is an undirected, connected, acyclic graph, having only one root node.

Definition 2 (Spanning Tree)**.** Every connected graph $G = (V, E)$ contains a *spanning tree*; that is, a set $E' \subseteq E$ can be chosen, such that (V, E') is a tree.

Definition 3 (Minimum-Hop Path)**.** The shortest path between two nodes is the minimum-hop path.

With these building blocks, the formal definition of a BFS tree is the following.

Definition 4 (Breadth-First Search Tree)**.** A spanning tree T, starting from u, of network G is a *Breadth-First Search (BFS) Tree* if, for each node, the tree path to u is a minimum-hop path in G.

A BFS algorithm is thus an algorithm that constructs a BFS tree.

In this paper, we also have closer look at the behavior of the BFS algorithm when non-FIFO channels are used. To describe different situations that might occur when such channels are used, the terms *delayed message* and *overtaken* are introduced. Consider two nodes u and v which are connected by one non-FIFO channel c:

- At time t_1, node u sends a message m to v.
- At time $t_2 > t_1$, node u sends a message m' to v.
- At time $t_3 > t_2$, the message m' arrives at v.
- At time $t_4 > t_3$, the message m arrives at v.

It is now said that m is a *delayed message* and that m is *overtaken* by m'.

We assume in this paper that channels between nodes are reliable, but not necessarily FIFO. It is also assumed that the network on which the algorithm is applied is modeled by an undirected graph $G = (V, E)$ with V being the nodes of the network and E being the edges between the nodes. Nodes initially are unaware of the network topology but are aware of their adjacent edges. They have copies of the algorithm and know whether they are the initiator of the algorithm or not.

3 Related Work

In this section, we present some related research on distributed BFS algorithms.

Note that there are differences between distributed algorithms and centralized (uniprocessor) algorithms. A centralized BFS algorithm is rather simple: the root node of the tree sends messages to its neighbors. When a node receives messages, it picks one of the senders as its parent and forwards the message to its unknown neighbors. Since there is centralized control, sent messages arrive instantly. Sending and receiving of a message is thus one atomic operation. It is said that communication is synchronous.

To determine the efficiency of algorithms, we examine the complexity of it. The above BFS algorithm has a perferct worst-case message complexity of $O(2|E|)$. This means that the number of messages used in this algorithm is in the order of 2 times the number of edges. Unfortunatly, above centralized algorithm cannot be applied to a distributed system like for example the Internet. For such systems, distributed algorithms are needed. This is due to the fact that in a distributed system, there is no centralized control. Sending and receiving messages are different operations, and communiction is thus asynchronous.

Table 1 Worst-case message complexity of existing BFS algorithms

Algorithm	Messages
Toueg [1980]	$\Theta(\lvert V\rvert\lvert E\rvert)$
Chandy and Misra [1982]	$O(\lvert V\rvert^2\lvert E\rvert)$
Cheung [1983]	$O(\lvert V\rvert^3)$
Frederickson [1985]	$O(\lvert V\rvert^2)$
Frederickson [1985]	$O\left(\lvert V\rvert\sqrt{\lvert E\rvert}\right)$
Awerbuch and Gallager [1985]	$O\left(\lvert E\rvert 2^{\sqrt{\log\lvert V\rvert\log\log\lvert V\rvert}}\right)$
Zhu and Cheung [1987]	$O(\lvert V\rvert^2)$
Awerbuch and Gallager [1987]	$O(\lvert V\rvert^{1.6} + \lvert E\rvert)$
Awerbuch [1989]	$O\left(\lvert E\rvert^{1+\frac{1}{\sqrt[4]{\log\lvert V\rvert}}}\right)$
Awerbuch et al. [1989]	$O\left(\lvert E\rvert^{1+\sqrt{\log\log\lvert V\rvert/\log\lvert V\rvert}}\right)$
Makki [1996]	$O(\lvert E\rvert)$

Distributed BFS algorithms have been studied extensively by many researchers in the past. We now provide a short overview of existing Distributed BFS algorithms. Table 1 provides a list of these algorithms and their worst-case message complexity. Note that the following overview is not complete. Since BFS trees are used in a wide variety of practical applications, many more BFS algorithms[1] may exist.

Observe that a BFS tree is in fact a tree of shortest-paths from a given root node to all other nodes of a network under the assumption that each edge has an equal weight. This means that we can use existing distributed shortest-path algorithms for weighted graphs for constructing BFS trees. The algorithm outlined by Chandy and Misra [1982] is such a shortest-path algorithm. Since this algorithm is short and easy to understand, we provide a short description of the algorithm below.

Each node u has a variable D_u to store the distance to the initiator i and a variable Nb_u to store which neighbor u uses to get to i. Initially, $D_i = 0$ and $Nb_i = \perp$. $D_u = \infty$ for each $u \neq i$. Initiator i starts the algorithm by sending $\langle \mathbf{mydist}, 0 \rangle$ messages to its neighbors. When a node u receives $\langle \mathbf{mydist}, d \rangle$ from a neighbor v and if $d + 1 < D_u$, then

- $D_u \leftarrow d + 1$
- $Nb_u \leftarrow v$
- u sends $\langle \mathbf{mydist}, D_u \rangle$ messages to its neighbors (except to v).

Other algorithms for constructing shortest-path algorithms, for example the one outlined by Toueg [1980], a distributed version of the Floyd-Warshall algorithm (Cormen et al. [1990]), may also be used for constructing BFS trees.

Cheung [1983] outlined an algorithm that constructs a BFS tree of a *directed* graph. The algorithm uses layering properties and labeled vertices in such a way that each vertex is assigned its proper layer number by its parents.

As mentioned before, we will discuss in this paper two BFS algorithms outlined by Frederickson [1985]. Awerbuch and Gallager [1985] outlined an extension to Frederickson's advanced BFS algorithm. This algorithm also uses synchronization rounds, called strips, but within each strip, the exploration is synchronized, instead of only a global synchronization between every two strips. This reduces the worst-case message complexity. Awerbuch and Gallager [1987] outlined another extension based on Frederickson's advanced BFS algorithm. The difference here is that not a static number of ℓ levels are explored each round, but that each round the number of levels to explore is calculated as a function of the number of nodes to discover.

Zhu and Cheung [1987] outlined a BFS algorithm which seems to be quivalent to Frederickson's simple algorithm. More research is however necessary to examine whether the algorithms are entirely the same. If so, the paper by Zhu and Cheung [1987] may be regarderd superfluous.

A very complex algorithm for constructing BFS trees was outlined by Awerbuch [1989]. In this algorithm, multiple nodes start building a BFS trees that are thus generated in parallel. When each node is part of a number of trees, the partial trees are merged into a final BFS tree. To reduce the worst-case message complexity even more, the algorithm outlined by Awerbuch et al. [1989] is based on the same principle, but uses clusters of BFS trees which are formed into bigger clusters. This would finally result in the desired BFS tree.

[1]From now on, the BFS algorithms we discuss are distributed ones, unless stated otherwise

Finally, Makki [1996] outlines a less harder to understand algorithm with a better worst-case message complexity than previous algorithms. This improvement is achieved by a more sophisticated synchronization.

For a more detailed introduction on above algorithms, consider Makki [1996], Section 3.

If one is interested in the use of BFS algorithms, consider Bratislav and Miroslaw [2007] which outlines a BFS algorithm used for cut-edge detection in graphs. This technique is used in wireless networks, which seems to become an important field of interest at the moment. Another example can be found in the paper by Yoo et al. [2005]. In this paper a new parallel distributed BFS algorithm for Blue Gene is presented. Such a parallel algorithm may be used more in the future when grid computing becomes more important.

4 The Simple Algorithm

In this section, we outline the “simple” algorithm for constructing a BFS tree as provided by Frederickson [1985] and Tel [2000]. In Section 4.1 we provide a detailed description of the algorithm. To illustrate how the algorithm works we explain an example in Section 4.2. In Section 4.3 we provide a complexity analysis for the algorithm.

We tried to let the detailed description be as generic as possible, so that we can adopt it later for the “advanced” algorithm. Although Frederickson’s and Tel’s version of the Simple Algorithm are nearly the same, we based the following description on Tel’s notation.

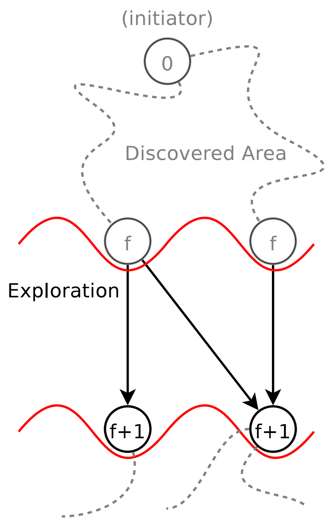


Figure 1: Concept of the Simple Algorithm

The main concept of the Simple Algorithm is illustrated in Figure 1. An initiator $i \in V$ is picked and starts the algorithm. Each round the next level of the tree is explored. At some level k, the initiator sends ⟨**forward**⟩ messages to the front nodes (nodes at the latest explored level), followed by these front nodes sending ⟨**explore**⟩ messages to their neighbors. These “to-be-explored nodes" send replies, followed by the front nodes sending their reports back to the initiator. The initiator now may decide to start round $k + 1$ or to terminate the algorithm.

4.1 Detailed description of the Simple Algorithm

Algorithm 1 Simple Algorithm – Initialization

```
var  $level_u$                  : integer **init** $\infty$
     $level_u[n]$               : integer **init** $\infty$ for each $n \in Neigh_u$
     $parent_u$                 : process **init** $udef$
     $Child_u$                  : process **init** $\emptyset$
     $k$                        : integer **init** 0
     $bvalue_u$                 : boolean **init** $false$
     $expectedreplies_u[n]$     : integer **init** 0 for each $n \in Neigh_u$

For the initiator $i$ only, execute once:
begin
    $level_i \leftarrow 0$
    $k \leftarrow 1$
    forall $n \in Neigh_i$ do
        $Child_u \leftarrow Child_u \cup \{n\}$
        send $\langle \textbf{explore}, k \rangle$ to $n$
        $expectedreplies_i[n] \leftarrow 1$
end
```

The Simple algorithm is started by picking an initiator, followed by the initiator assigning itself level 0, after which the construction of level 0 of the BFS tree is finished. To start the next round, the initiator sends $\langle \textbf{explore}, 1 \rangle$ messages to its neighbors. This behavior is described in detail in Algorithm 1.

As can been seen from Algorithm 1, each node u keeps track of some administration, namely its own level in the BFS tree ($level_u$); the levels of its neighbor nodes ($level_u[]$); its parent in the BFS tree ($parent_u$); its set of children ($Child_u$); the round the algorithm is in (k, only relevant for the initiator); and two extra variables ($bvalue_u$ and $expectedreplies_u[]$) to perform some checks in the algorithm. Note that it is assumed that each node u has knowledge about its neighbor nodes in the set $Neigh_u$.

We now describe the algorithm by explaining what happens when a message from node v arrives at node u. The three events that can occur are:

1. Node u receives $\langle \textbf{forward}, f \rangle$ from node v.
2. Node u receives $\langle \textbf{explore}, f \rangle$ from node v.
3. Node u receives $\langle \textbf{reverse}, b \rangle$ from node v.

These events are now discussed in detail.

1. Node u receives $\langle \textbf{forward}, f \rangle$ from v.

Algorithm 2 Simple Algorithm – Handle $\langle$**forward**$\rangle$ messages

```
For each process $u$, upon receipt of $\langle \mathbf{forward}, f \rangle$ from $v$:
begin
    $bvalue_u \leftarrow false$
    $\forall n \in Neigh_u : reply_u[n] \leftarrow 0$
    if $level_u < f$ then
        forall $c \in Child_u$ do
            send $\langle \mathbf{forward}, f \rangle$ to $c$
            $reply_u[c] \leftarrow reply_u[c] + 1$
    if $level_u = f$ then
        forall $n \in neigh_u \wedge level_u[n] \neq f - 1$ do
            send $\langle \mathbf{explore}, f + 1 \rangle$ to $n$
            $expectedreplies_u[n] \leftarrow +1$
end
```

$\langle \mathbf{forward}, f \rangle$ messages are used to let the front nodes know they must send $\langle \mathbf{explore}, f+1 \rangle$ messages to their unknown neighbors. If a node u at $level_u < f$ receives such a $\langle \mathbf{forward}, f \rangle$ message, it forwards it to each child [Algorithm 2, line 5-8].
When $level_u = f$, u sends $\langle \mathbf{explore}, f+1 \rangle$ messages to its neighbors not already known to be at level $f - 1$ [Algorithm 2, line 9-12].
When u has sent all its $\langle \mathbf{forward}, f \rangle$ or $\langle \mathbf{explore}, f+1 \rangle$ messages, it must wait for an answer from each node u sent such a message to, before reporting back to its parent. We do this in the pseudocode by storing the number of expected reply messages in the array $expectedreplies_u[]$. This array consists of neighbors from u and stores for each neighbor n how many replies u expects from n.
The report that u sends to its parent depends on the replies it received. When at least one of the replies was a $\langle \mathbf{reverse}, true \rangle$ message, $\langle \mathbf{reverse}, true \rangle$ must be reported to $parent_u$. To store the arrival of a $\langle \mathbf{reverse}, true \rangle$ message, we use the boolean $bvalue_u$, which is thus set to *false* at this moment and will become *true* if a $\langle \mathbf{reverse}, true \rangle$ message arrives.

2. Node u receives $\langle \mathbf{explore}, f \rangle$ from node v.

When an $\langle \mathbf{explore}, f \rangle$ message arrives at some node u for the first time (i.e. when $level_u = \infty$), node u is discovered by v and will be added to the subtree of v. Node v becomes the parent of u and the level of u (the depth of u in the BFS tree) is bound to f. To let v know that u has become its child, u sends back a reply $\langle \mathbf{reverse}, true \rangle$ [Algorithm 3, line 3-6].
Subsequently arriving $\langle \mathbf{explore}, f \rangle$ messages at node u where $level_u = f$ indicate that the senders of those messages are at level $f - 1$. To reduce the number of $\langle \mathbf{explore}, f \rangle$ messages that will be sent in the next round, u stores that these nodes are at level $f - 1$ and sends back a $\langle \mathbf{reverse}, false \rangle$ to let them know it will not become their child [Algorithm 3, line 7-9].
A node u at level f may also receive $\langle \mathbf{explore}, f+1 \rangle$ messages from a neighbor v also at level f. Since u itself will also send an $\langle \mathbf{explore}, f+1 \rangle$ message to v, no reply is sent back by u and such an $\langle \mathbf{explore}, f+1 \rangle$ message is interpreted as a $\langle \mathbf{reverse}, false \rangle$ message [Algorithm 3, line 10-11].

Algorithm 3 Simple Algorithm – Handle $\langle$**explore**$\rangle$ messages

```
For each process $u$, upon receipt of $\langle \textbf{explore}, f \rangle$ from $v$:
begin
    if $level_u = \infty$ then
        $parent_u \leftarrow v$
        $level_u \leftarrow f$
        send $\langle \textbf{reverse}, true \rangle$ to $v$
    else if $level_u = f$ then
        $level_u[v] \leftarrow f - 1$
        send $\langle \textbf{reverse}, false \rangle$ to $v$
    else if $level_u = f - 1$ then
        Interpret as $\langle \textbf{reverse}, false \rangle$ message
end
```

3. Node u receives $\langle$**reverse**$, b\rangle$ from v.

Algorithm 4 Simple Algorithm – Handle $\langle$**reverse**$\rangle$ messages

```
For each process $u$, upon receipt of $\langle \textbf{reverse}, b \rangle$ from $v$:
begin
    $expectedreplies_u[v] \leftarrow expectedreplies_u[v] - 1$
    if $b = true$ then
        $Child_u \leftarrow Child_u \cup \{v\}$
        $bvalue_u \leftarrow true$
    if $\forall n \in Neigh_u : reply_u[n] = 0$ then
        if $parent_u \neq udef$ then
            send $\langle \textbf{reverse}, bvalue_u \rangle$ to $parent_u$
        else if $bvalue_u = true$ then
            $k \leftarrow k + 1$
            forall $c \in Child_u$ do
                send $\langle \textbf{explore}, k \rangle$ to $c$
                $expectedreplies_i[n] \leftarrow 1$
        else
            terminate
end
```

A node u at level $f > 0$ waits for a reply to all $\langle$**explore**$, f\rangle$ or $\langle$**forward**$, f\rangle$ messages it sent. Such a reply is of the form $\langle$**reverse**$, b\rangle$ where $b = true$ if and only if a new node was added to the subtree of u.
When all $\langle$**explore**$, f\rangle$ or $\langle$**forward**$, f\rangle$ messages have been replied, u sends a $\langle$**reverse**$, b\rangle$ itself to its parent. The value of b is *true* if one of the received replies was a $\langle$**reverse**$, true\rangle$ message.
The initiator at level $f = 0$ receives $\langle$**reverse**$, b\rangle$ messages from its children. If a $\langle$**reverse**$, true\rangle$ message was among them, the initiator starts a new round by sending $\langle$**forward**$, f + 1\rangle$ messages to its children. When only $\langle$**reverse**$, false\rangle$ messages arrived at the initiator, no new nodes were found in the BFS tree

and thus no new nodes will be discovered in a next round, resulting in the termination of the algorithm.
If a $\langle \textbf{reverse}, \textit{true} \rangle$ message arrives at u it is certain that v is a child of u since the two cases in which such a message can arrive are:

1. v is just discovered by u, and thus becomes its child
2. v just discovered a new node in its subtree, and reports this back to its parent u, so v is already a child of u.

In any case, we add v to the set of children of u (again) and we set $bvalue_u$ to *true*, to store that a $\langle \textbf{reverse}, \textit{true} \rangle$ arrived [Algorithm 4, line 4-6].
When u received all its expected replies, we distinguish two situations:

1. $parent_u \neq udef$
2. $parent_u = udef$ (i.e. u is the initiator of the algorithm)

Since $bvalue_u = true$ if and only if at least one $\langle \textbf{reverse}, \textit{true} \rangle$ arrived at u, the first case causes u to send a reply $\langle \textbf{reverse}, bvalue_u \rangle$ to $parent_u$. In the second case, when $bvalue_u = true$, a new round is started by the initiator. If $bvalue_u = false$, the algorithm terminates [Algorithm 4, line 7-16].

4.2 Example

We will now have a look at an example of the Simple Algorithm. Consider Figure 7. The algorithm just started the construction of level 3: the initiator r sent $\langle \textbf{forward}, 2 \rangle$ messages to nodes a and b.

In Figure 7a $\langle \textbf{forward}, 2 \rangle$ messages travel from node a to c, and from node b to d and e. Note that since each node can only have one parent in this algorithm, node a did not send a $\langle \textbf{forward}, 2 \rangle$ message to node d.

When the $\langle \textbf{forward}, 2 \rangle$ messages arrives at nodes at level 2, $\langle \textbf{explore}, 3 \rangle$ messages are sent by the nodes d and e as depicted Figure 7b. $\langle \textbf{explore}, 3 \rangle$ messages travel to each neighbor of d and e, except for those that are at level $(2-1=)1$. Since c only has one neighbor, a, which is at level 1, c does not send any $\langle \textbf{explore}, 3 \rangle$ messages and reports back a $\langle \textbf{reverse}, \textit{false} \rangle$ message to a, to tell a that no more nodes were added to its subtree.

In Figure 7c, nodes f and g send back their replies to d and e. It seems that the $\langle \textbf{explore}, 3 \rangle$ from d to g arrived earlier than the $\langle \textbf{explore}, 3 \rangle$ from e to g, and thus g sends $\langle \textbf{reverse}, \textit{true} \rangle$ to d and $\langle \textbf{reverse}, \textit{false} \rangle$ to e. Note that the $\langle \textbf{explore}, 3 \rangle$ messages d and e exchanged were both interpreted as $\langle \textbf{reverse}, \textit{false} \rangle$ messages, and thus were not replied. Meanwhile, since a received a reply from all its children, a replies back to the initiator r that no new nodes were added to its subtree by sending a $\langle \textbf{reverse}, \textit{false} \rangle$ message.

In Figure 7d, $\langle \textbf{reverse} \rangle$ messages travel from d and e back to b. f and g are now children from d and are added to its subtree. This round continues with b sending $\langle \textbf{reverse}, \textit{true} \rangle$ back to r, which will trigger the start of the next round.

4.3 Complexity Analysis

The construction of the BFS tree uses at most $|V|$ exploration rounds. In each round, an edge of the BFS tree carries at most one $\langle \textbf{forward} \rangle$ and one replying $\langle \textbf{reverse} \rangle$ message. In total, each edge carries at most one $\langle \textbf{explore} \rangle$ message and one replying ⟨**reverse**⟩ message. The worst-case message complexity is thus $O(|V|^2)$. As level $f+1$ is computed in $2(f+1)$ time units, the worst-case message and time complexity are the same:

$$O(|V|^2)$$

5 The Advanced Algorithm

In this section, we outline a more advanced BFS algorithm. This Advanced Algorithm extends the Simple Algorithm to reduce its worst-case message complexity.

We start this section with some discussion about the existing versions of the Advanced Algorithm in Section 5.1 and 5.2 as they were outlined by Frederickson [1985] and Tel [2000]. In Section 5.3 we outline the improved version of the Advanced Algorithm. We discuss the algorithm and present a complexity analysis of it in Section 5.4 and 5.5.

The main concept of the Advanced Algorithm is that not one level per round, but ℓ levels are to be explored in each round of the algorithm. This will reduce the message overhead in the already explored tree since less ⟨**forward**⟩ and ⟨**reverse**⟩ messages shall be sent. A drawback is that this complicates the algorithm: nodes can suffer from parent switches, since the first arrival of an ⟨**explore**⟩ message does no longer guarantee the best minimum-hop path to the initiator. In such a situation, a node must inform its former parent that it will not be its child anymore. This problem is called the "old-parent problem" and different solutions for this problem exists. An example of the old-parent problem is illustrated in Figure 8.

5.1 Frederickson's version

A small difference between Frederickson's version and the versions presented by Tel and in this paper is that Frederickson initiates the set of children of a node u ($Child_u$) to each of its neighbors ($Neigh_u$). During the exploration stage, nodes are removed from this set when it turns out that they are not children of u. In the other versions, $Child_u$ is initiated empty, and nodes are added when they are indeed children of u.

To solve the old-parent problem, Frederickson's version uses an extra ⟨**negative**⟩ message type to indicate a parent-switch. The ⟨**reverse**⟩ messages are only used as an echo mechanism and do not require a variable.

After studying Frederickson's version, we conclude the following:

- *The algorithm works for non-FIFO channels, but this is not stated anywhere.*

After testing different "delayed message situations" like the ones shown in Section 5.4, it can be concluded that the algorithm supports non-FIFO channels. This is due to the fact that the Frederickson's Advanced Algorithm uses a level-variable in each ⟨**negative**⟩ message.

- *The algorithm produces a slight message overhead when a parent-switch occurs.*

As will be proposed in Section 5.3, it is enough to send an ⟨**explore**⟩ message when a parent-switch occurs. The ⟨**negative**⟩ messages introduced by Frederickson are thus superfluous.

- *The algorithm is not fully operating in lockstep.*

When a parent switch occurs, the front nodes do not wait until all information about the tree is updated before reporting back to the initiator. When non-FIFO channels are used, the algorithm may terminate while the BFS tree is incorrect, i.e. two nodes may think they are both the parent of one other node. However, since each node knows which node is its parent, nodes are able to discard messages from their wrong parent. This is thus not a big issue.

We conclude now, that the provided algorithm of Frederickson is correct, but it has still some drawbacks. The provided new version is supposed to overcome these drawbacks.

5.2 Tel's version

After studying Tel's version of the algorithm, we conclude that the description is incomplete. The description in Tel's book is too short and lacks a solution for the old-parent problem: it does not say what to do when a parent-switch occurred and the algorithm could end up with multiples nodes having the same child.

Tel [2006] suggested a solution: when a node receives a ⟨**forward**⟩ from a node other than its parent, it sends a ⟨**no-child**⟩ message in return. Receivers of a ⟨**no-child**⟩ message remove the sender of such a message from its list of children. Since the last round of the algorithm will only send ⟨**forward**⟩ messages, this would always result in the desired BFS tree and the suggested solution is thus correct.

Van Moolenbroek [2006] suggested another solution: a node which suffers from a parent-switch sends its ⟨**explore**⟩ messages also to its former parent. When the former parent receives this message, it knows that the sender is not its child anymore. This is the solution we present below in further detail.

Besides the lack of a solution for the old-parent problem, it is stated in Tel [2006] that the described algorithm is only working for non-FIFO channels. Although it turns out that the suggested solution implies that the algorithm is also correct for non-FIFO channels, there are no references that show this.

As can be concluded now, Tel's version has some drawbacks. The revisited version will overcome these drawbacks.

5.3 Detailed description

Before the detailed description of the Advanced Algorithm is provided, first consider the next important differences compared to the previous versions:

1. No use of ⟨**negative**⟩ or ⟨**no-child**⟩ messages.
2. Elegant solution for the old-parent problem.
3. Suitable for networks using non-FIFO channels.

Some parts of the detailed description might be similar or the same as in the Simple Algorithm, but they will be restated here for the sake of completeness.

Algorithm 5 Advanced Algorithm – Initialization

```
var  $level_u$                : integer init $\infty$
     $level_u[n]$             : integer init $\infty$ for each $n \in Neigh_u$
     $parent_u$               : process init $udef$
     $Child_u$                : process init $\emptyset$
     $k$                      : integer init 0
     $\ell$                   : integer init ?
     $bvalue_u$               : boolean init $false$
     $expectedreplies_u[n]$   : integer init 0 for each $n \in Neigh_u$
     $sentreverse_u$          : boolean init $true$

For the initiator $i$ only, execute once:
begin
    $level_i \leftarrow 0$
    $k \leftarrow 1$
    forall $n \in Neigh_i$ do
        $Child_u \leftarrow Child_u \cup \{n\}$
        send $\langle \textbf{explore}, k, \ell \rangle$ to $n$
        $expectedreplies_i[n] \leftarrow 1$
end
```

The algorithm is started by picking an initiator i and choosing a value for ℓ, followed by the initiator assigning itself level 0, after which the construction of level 0 is complete. The next round is started by the initiator by sending $\langle \textbf{explore}, 1, \ell \rangle$ messages to each of its neighbors. This behavior is described in detail in Algorithm 5.

As in the Simple Algorithm, each node u keeps track of some administration: its own level in the BFS tree ($level_u$); the levels of its neighbor nodes ($level_u[]$); its parent in the BFS tree ($parent_u$); its set of children ($Child_u$); the round the algorithm is in (k, only relevant for the initiator); the value for ℓ (ℓ) and three extra variables ($bvalue_u$, $expectedreplies_u[]$ and $sentreverse_u$) to perform checks in the algorithm. Note that in the following description two assumptions are made:

- Each node of the network has knowledge about its neighbor nodes in the set $Neigh_u$.
- Each node of the network has knowledge about the decided value for ℓ.

The last assumption could be removed by adjusting the algorithm a bit: let the initiator calculate the best value for ℓ and send it at round $k > 2$ within each $\langle \textbf{forward}, f \rangle$ message.

We now describe the algorithm by explaining what happens when a message from node v of type m arrives at node u. The events that can occur are:

1. Node u receives $\langle \textbf{forward}, f \rangle$ from node v.
2. Node u receives $\langle \textbf{explore}, f, m \rangle$ from node v.
3. Node u receives $\langle \textbf{reverse}, b \rangle$ from node v.

These events are now discussed in detail.

1. Node u receives $\langle \mathbf{forward}, f \rangle$ from node v.

Algorithm 6 Advanced Algorithm – Handle $\langle \mathbf{forward} \rangle$ messages

```
For each process $u$, upon receipt of $\langle \mathbf{forward}, f \rangle$ from $v$:
begin
    $bvalue_u \leftarrow false$
    $\forall n \in Neigh_u : expectedreplies_u[n] \leftarrow 0$
    if $level_u < f$ then
        forall $c \in Child_u$ do
            send $\langle \mathbf{forward}, f \rangle$ to $c$
            $expectedreplies_u[c] \leftarrow 1$
    if $level_u = f$ then
        forall $n \in Neigh_u \wedge level_u[n] \neq f - 1$ do
            send $\langle \mathbf{explore}, f+1, \ell \rangle$ to $n$
            $expectedreplies_u[n] \leftarrow 1$
end
```

$\langle \mathbf{forward}, f \rangle$ messages are used to tell the front nodes they must send $\langle \mathbf{explore}, f, \ell \rangle$ messages. The forward procedure is pretty straightforward: if a node u at $level_u < f$ receives a $\langle \mathbf{forward}, f \rangle$ message, the message is forwarded to each child and u sets the number of expected replies to 1 for each child [Algorithm 6, line 5-8].
If $level_u = f$, u sends $\langle explore, f, \ell \rangle$ messages to its neighbors not already known to be at level $f - 1$. In this case also, the number of expected replies is set to 1 for each neighbor [Algorithm 6, line 9-12].
After sending all $\langle \mathbf{forward}, f \rangle$ or $\langle \mathbf{explore}, f, \ell \rangle$ messages, u waits for replies to those messages. When all replies arrived, u can report back to its parent (see event 3).

2. Node u receives $\langle \mathbf{explore}, f, m \rangle$ from node v.

$\langle \mathbf{explore}, f, m \rangle$ messages with $m > 1$ and $level_u > f$ are forwarded to neighbors of u. If the network is suitable, this decreases the number of messages sent and thus decreases the worst-case message complexity of the BFS algorithm (see Section 5.5).
For administration reasons, u first stores at what level its neighbors are in the BFS tree. The algorithm ignores delayed $\langle \mathbf{explore}, f, m \rangle$ messages here [Algorithm 7, line 3-4].
When u receives an $\langle \mathbf{explore}, f, m \rangle$ message with $level_u > f$, u found a (better) path to the initiator. u must now update some administration: set *bvalue* to *true*, to indicate that — when all replies are received — $\langle \mathbf{reverse}, true \rangle$ must be sent to $parent_u$; change $parent_u$ to v; adjust $level_u$ to the new level in the tree and reset the list of children $Child_u$. Besides this administration update, u checks whether it already sent a $\langle \mathbf{reverse}, b \rangle$ message to its former parent if it had one. If u did not sent a $\langle \mathbf{reverse}, b \rangle$ yet, this is done now by u sending a $\langle \mathbf{reverse}, false \rangle$ message to the former parent. This technique causes old explore rounds to be terminated quickly [Algorithm 7, line 5-12].
When a (better) path was found and if $m > 1$, u sends $\langle \mathbf{explore}, f + 1, m -$

Algorithm 7 Advanced Algorithm – Handle $\langle$**explore**$\rangle$ messages

```
For each process $u$, upon receipt of $\langle \textbf{explore}, f, m\rangle$ from $v$:
begin
    if $level_u[v] \neq f - 1$ then
        $level_u[v] \leftarrow f - 1$
    if $level_u > f$ then
        $bvalue_u \leftarrow true$
        if not $sendreverse_u$
            send $\langle \textbf{reverse}, false\rangle$ to $parent_u$
            $sendreverse_u \leftarrow true$
        $parent_u \leftarrow v$
        $level_u \leftarrow f$
        $Child_u \leftarrow \emptyset$
        if $m > 1$
            $sendreverse_u \leftarrow false$
            forall $n \in Neigh_u \wedge n \neq parent_u$ do
                send $\langle \textbf{explore}, f+1, m-1\rangle$ to $n$
                $expectedreplies_u[n] \leftarrow expectedreplies_u[n] + 1$
        else
            send $\langle \textbf{reverse}, true\rangle$ to $v$
    else if $(level_u = f) \vee (level_u = f - 1)$ then
        $Child_u \leftarrow Child_u \setminus \{v\}$
        send $\langle \textbf{reverse}, false\rangle$ to $v$
    else if $level_u < f - 1$ then
        send $\langle \textbf{reverse}, false\rangle$ to $v$
end
```

1$\rangle$ messages to each of its neighbors except to its parent. The counter array *expectedreplies*$_u$ is hereby increased for each node, to make sure that all replies will be received before reporting back to the parent [Algorithm 7, line 13-17]. When $m = 1$, u sends a $\langle$**reverse**,*false*$\rangle$ message back, indicating v will not becomes u's parent [Algorithm 7, line 18-19].
Note that when u received an $\langle \textbf{explore}, f, m\rangle$ message, but already had another parent x, m must be greater than 1. This means that when u already sent a $\langle$**reverse**, *true*$\rangle$ to its former parent x, u will sent an $\langle \textbf{explore}, f+1, m-1\rangle$ to x as well:

Conjecture 1. *When a node u in round k issues a parent change from x to v, it is certain that eventually, but still in round k, an $\langle$**explore**$\rangle$ is sent to x.*

When the former parent x receives the $\langle \textbf{explore}, f', m'\rangle$ message, it knows that u found another parent. Two situations can be distinguished now:

- $level_x > f$. Now u becomes the parent of x and the $\langle \textbf{explore}, f', m'\rangle$ message is interpreted as described above. Note that it is thus necessary to reset $Child_x$ in such a situation.
- $level_x = f$ or $level_x = f - 1$. This means that u found a shorter path to the initiator, but x will not become u's child. Node x simply deletes u

from its list of children and replies with a $\langle$**reverse**, *false*$\rangle$ [Algorithm 7, line 20-22].

When a node y which does not have any relation with u whatsoever, receives an $\langle$**explore**$, f, m\rangle$ message whereby $level_y = f$ or $level_y = f - 1$, those statements do not change anything about y's state.
Using the above technique, it could be the case that inside a wave of $\langle$**explore**$\rangle$ and $\langle$**reverse**$\rangle$ messages, nodes are experiencing parent switches. However, before the front nodes sent their reports back to the initiator, no malicious parent-child relationships well be left. This is due to Conjuncture 1 and to the fact that each node waits until it received a reply to all sent $\langle$**explore**$, f, m\rangle$ messages. This shows that the old-parent problem is solved.
Finally, if u received an $\langle$**explore**$, f, m\rangle$ message with $level_u = f-1$ $\langle$**reverse**$, false\rangle$ is replied since no better path was found [Algorithm 7, line 23-24].

3. Node u receives $\langle$**reverse**$, b\rangle$ from node v.

Algorithm 8 Advanced Algorithm – Handle $\langle$**reverse**$\rangle$ messages

```
For each process $u$, upon receipt of $\langle$reverse$, b\rangle$ from $v$:
begin
    $expectedreplies_u[v] \leftarrow expectedreplies_u[v] - 1$
    if $level_u[v] \leq level_u$ then
        $b \leftarrow false$
    if $b = true$ then
        $Child_u \leftarrow Child_u \cup \{v\}$
        $bvalue_u \leftarrow true$
    if $\forall i \in expectedreplies_u : expectedreplies_u[i] = 0$ then
        if $parent_u \neq udef$ then
            send $\langle$reverse$, bvalue_u\rangle$ to $parent_u$
            $sendreverse_u \leftarrow false$
        else if $bvalue_u = true$ then
            $k \leftarrow k + 1$
            forall $c \in Child_u$ do
                send $\langle$forward$, k\rangle$ to $c$
                $expectedreplies_u[c] \leftarrow 1$
        else
            terminate
end
```

Processing a $\langle$**reverse**$, b\rangle$ message is rather simple. The process is very similar to the one given in Algorithm 4, except that delayed $\langle$**reverse**$, b\rangle$ messages must be detected. When a delayed $\langle$**reverse**$, b\rangle$ is found, it is interpreted as a $\langle$**reverse**, *false*$\rangle$ message [Algorithm 8, line 4-5].

5.4 Discussion

As stated in Section 5.3, we added the next three improvements to the Advanced Algorithm:

1. No use of $\langle$**negative**$\rangle$ or $\langle$**no-child**$\rangle$ messages.
2. Elegant solution for the old-parent problem.
3. Suitable for networks using non-FIFO channels.

As can be adopted from the detailed description, the first improvement does not require further explanation.

The working of the second improvement is also explained in the detailed description. The main reason for stating that the new solution for the old-parent problem is more elegant than Frederickson's one is that in this solution the BFS tree is correct at the moment that the front nodes start sending their reports back to the initiator. This is not necessarily the case in Frederickson's version, since nodes in that version do not wait until all $\langle$**negative**$\rangle$ messages are processed, because they do not have knowledge about these messages being sent.

It is hard to claim that the third improvement is correct. In this subsection, we now provide a list of examples that strike out the different possible situations that may occur when a message is delayed. Observe that only the next two situations are relevant to investigate:

- A delayed $\langle$**explore**$\rangle$ message arrives.
- A delayed $\langle$**reverse**$\rangle$ message arrives.

A $\langle$**forward**$\rangle$ message cannot be delayed, since there is a synchronization step between each wave of $\langle$**forward**$\rangle$ messages.

Although it should be enough to investigate only above two situations, we present some more examples to give a better idea of how the algorithm functions in a non-FIFO environment.

Note that in the following figures, the numbers before a message represent the order of sending. A gray, dotted line represents a delayed message.

- An $\langle\mathbf{explore}, f, m\rangle$ message is overtaken by an $\langle\mathbf{explore}, f', m'\rangle$ message (figure 2).

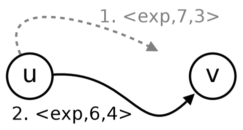


Figure 2: $\langle\mathbf{explore}, 7, 3\rangle$ overtaken by $\langle\mathbf{explore}, 6, 4\rangle$

Since u sends an $\langle\mathbf{explore}, f', m'\rangle$ message only if its level is decreased, the value of f in the delayed $\langle\mathbf{explore}, f, m\rangle$ message will always be greater than f'. It is thus not necessary to adjust the algorithm for this situation, since v will ignore any $\langle\mathbf{explore}, f, m\rangle$ message where $f < level_v$.
In this particular situation, v will send two $\langle\mathbf{reverse}, b\rangle$ messages, one of them being $\langle\mathbf{reverse}, \mathit{true}\rangle$, which will be interpreted by u in such a way that v becomes u's child.

- An $\langle\mathbf{explore}, f, m\rangle$ message is overtaken by a $\langle\mathbf{reverse}, b\rangle$ message (Figure 3).

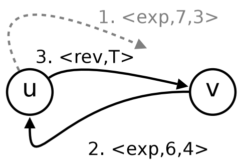


Figure 3: $\langle$**explore**$, 7, 3\rangle$ overtaken by $\langle$**reverse**$, \textit{true}\rangle$

This case is similar to the previous one. When the value f from the delayed $\langle$**explore**$, f, m\rangle$ message is greater than $\textit{level}_v$, v will send back a $\langle$**reverse**$, false\rangle$. When $f < \textit{level}_v$, v understands that u must becomes its parent and replies with a $\langle$**reverse**$, \textit{true}\rangle$. This type of message delay does thus not change the operation of the algorithm.

- A $\langle$**reverse**$, b\rangle$ message is overtaken by an $\langle$**explore**$, f, m\rangle$ message (Figure 4).

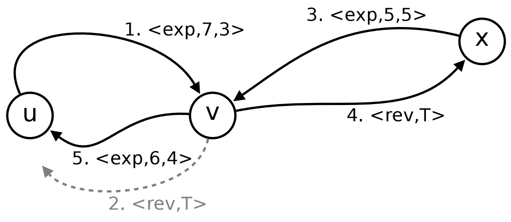


Figure 4: $\langle$**reverse**$, \textit{true}\rangle$ overtaken by $\langle$**explore**$, 6, 4\rangle$

This case is only interesting when the value b from the delayed $\langle$**reverse**$, b\rangle$ message is *true*, since $\langle$**reverse**$, \textit{false}\rangle$ messages do not affect the state of the receiver (i.e. its list of children).
When a $\langle$**reverse**$, \textit{true}\rangle$ message from v to u is delayed and meanwhile u becomes v's child, u has to ignore this message. This is done by u by checking what v's level is: when $\textit{level}_v \leq \textit{level}_u$ — which u only knows when it received an $\langle$**explore**$\rangle$ message from v — u interprets the received $\langle$**reverse**$, \textit{true}\rangle$ message as a $\langle$**reverse**$, \textit{false}\rangle$ message.

- A $\langle$**reverse**$, \textit{true}\rangle$ message is overtaken by a $\langle$**reverse**$, \textit{false}\rangle$ message (Figure 5).

This is in fact the same situation as the previous one. When a $\langle$**reverse**$, \textit{true}\rangle$ message arrives at u, a check is performed to test whether this message might be delayed.
In the situation described in Figure 5, when the delayed $\langle$**reverse**$, \textit{true}\rangle$ arrives at u, u will mistakenly assume that v is its child, but eventually, the $\langle$**explore**$, 6, 4\rangle$ message will tell u that v found another parent and that u thus must remove v from its list of children.

- A $\langle$**reverse**$, \textit{false}\rangle$ message is overtaken by a $\langle$**reverse**$, \textit{true}\rangle$ message (Figure 6).

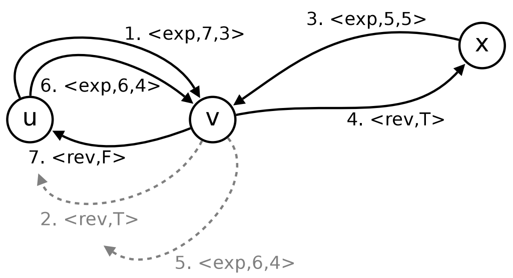


Figure 5: ⟨**reverse**, *true*⟩ overtaken by ⟨**reverse**, *false*⟩

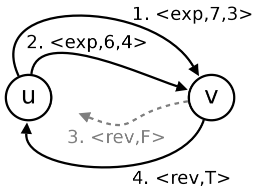


Figure 6: ⟨**reverse**, *false*⟩ overtaken by ⟨**reverse**, *true*⟩

This is the opposite situation as the previous one. It is rather unimportant since ⟨**reverse**, *false*⟩ messages do not change anything about the receivers state (i.e. its list of children).

We now conclude that the above three improvements of Frederickson's BFS algorithm are true and correct.

5.5 Complexity Analysis

Since the different versions of the Advanced Algorithm rely on the same concept, the worst-case message and time complexity are the same for each different version.
The construction of the BFS tree uses at most $\lceil \frac{|V|}{\ell} \rceil$ exploration rounds. In each round, an edge of the BFS tree carries at most one ⟨**forward**⟩ and replying ⟨**reverse**⟩ message. In total, each edge carries at most ℓ ⟨**explore**⟩ messages and ℓ replying ⟨**reverse**⟩ messages. The worst-case message complexity is thus $O(\frac{|V|^2}{\ell} + \ell|E|)$. As levels $f+1$ through $f+\ell$ are computed in $O(f+\ell)$ time units, the worst-case time complexity is $O(\frac{|V|^2}{\ell})$.

By solving the algebraic equation $\frac{|V|^2}{\ell} = \ell|V|$, the best value for ℓ can be obtained, which is $\frac{|V|}{\sqrt{E}}$. Filling this value for ℓ in the worst-case message and time complexity formulas, they become the same:

$$O(|V|\sqrt{|E|})$$

So for sparse graphs (i.e. graphs with only a few edges), the Advanced Algorithm has a better worst-case message complexity than the Simple Algorithm. Note however, that to achieve this, the algorithm needs to know $|V|$ and $|E|$ in advance. If $|V|$ and $|E|$ are not known beforehand, extra messages are required.

6 Conclusions

We extended the existing notations of Frederickson's BFS algorithm in three ways. First, we showed that the revisited version improves the functionality of the algorithm for two reasons: the algorithm is now works in lockstep; and the use of ⟨**explore**⟩ messages instead of ⟨**negative**⟩ messages gives the algorithm more transparency. Second, we showed that the algorithm operates correctly in a non-FIFO environment, while in previous notations, it was always assumed that FIFO channels were used. Third, we provided pseudocode along with the description of the algorithm to give the programmer a fundamental of how to implement the algorithm.

It must be stated however that we did not provide a formal proof of the correctness of the algorithm. This may be done in the future when the algorithm is researched in more detail. Also, although we provided a better manner to use Frederickson's BFS algorithm, the worst-case message complexity of the algorithm remained the same.

It must also be stated that more research is possible on Frederickson's original paper. Since we redefine the BFS algorithm that Frederickson used for constructing the all-pairs shortest-path algorithm, this last algorithm may require some updating as well. Also, as we concluded in Section 3, more research is necessary on Zhu and Cheung [1987] to determine whether this paper may be regarderd superfluous.

References

B. Awerbuch. Distributed shortest path algorithms. In *Proceedings on 21st annual ACM Symposium on Theory of Computing (STOC)*, pages 490–500, 1989.

B. Awerbuch and R. G. Gallager. Distributed BFS algorithms. In *Proceedings on Foundations of Computation Theory*, pages 250–256, 1985.

B. Awerbuch and R. G. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Transaction on Information Theory*, IT-33(3):315–322, 1987.

B. Awerbuch, A. V. Goldberg, M. Luby, and S. A. Plotkin. Network decomposition and locality in distributed computation. In *Proceedings on 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 364–369, 1989.

M. Bratislav and M. Miroslaw. Adaptation of breadth first search algorithm for cut-edge detection in wireless multihop networks. In *Proceedings of the 10th ACM-IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 377–386, 2007.

K. M. Chandy and J. Misra. Distributed computation on graphs: Shortest path algorithms. *Communications of the ACM*, 25(11):833–838, 1982.

T.-Y. Cheung. Graph traversal techniques and the maximum flow problem in distributed computation. *IEEE Transactions on Software Engineering*, SE-9 (4):504–512, 1983.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 26.4. McGrauw-Hill/MIT Press, 1990.

G. N. Frederickson. A single source shortest path algorithm for a planar distributed network. In K. Melhorn, editor, *Proceedings on STACS 85 2nd annual symposium on theoretical aspects of computer science*, volume 182 of *Lecture Notes in Computer Science*, pages 143–150, Saarbrücken, 1985. Springer-Verlag.

S. Makki. Efficient distributed breadth-first search algorithm. *Computer Communications*, 19(8):628–636, 1996.

G. Tel. *Introduction to distributed algorithms*, chapter 12.4. Asynchronous BFS Algorithms, pages 414–420. Cambridge University Press, 2000.

G. Tel. Personal communication. March 2006.

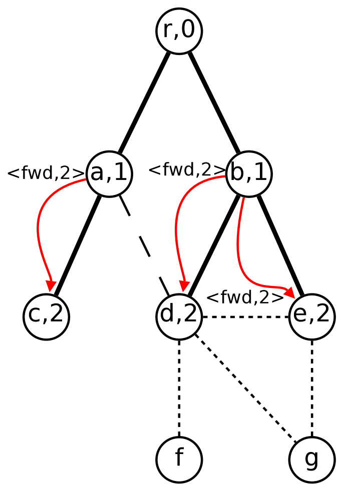
S. Toueg. An all-pairs shortest-path distributed algorithm. *RC-8397, IBM T. J. Watson Research Center*, 1980.

D. Van Moolenbroek. Personal communication. March 2006.

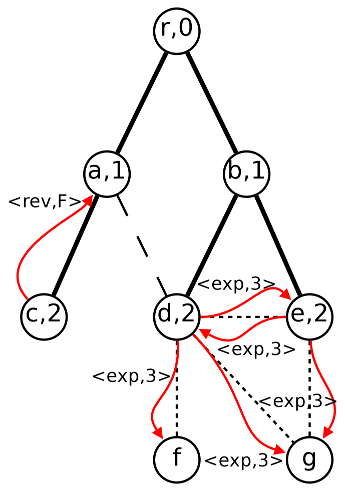
A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 25, 2005.

Y. Zhu and T.-Y. Cheung. A new distributed breadth-first-search algorithm. *Information Processing Letters*, 25(5):329–333, 1987.

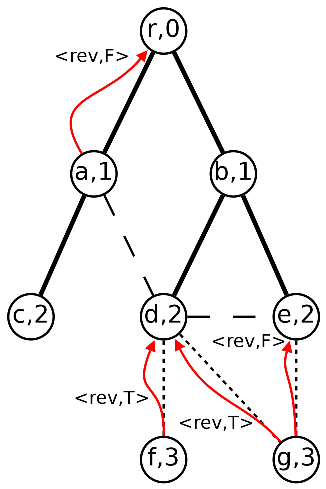
A Example: Simple Algorithm



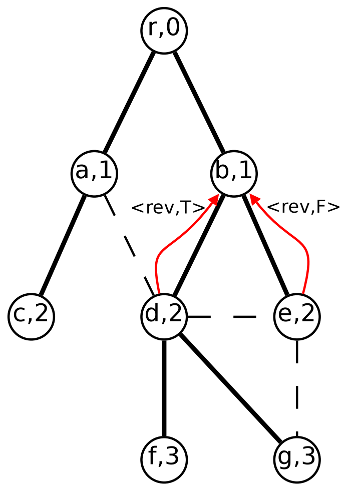
(a) forward messages



(b) explore messages



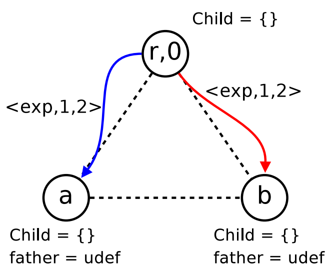
(c) reverse messages



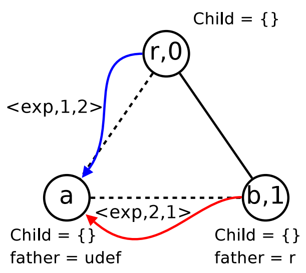
(d) reverse messages

Figure 7: Example of the Simple Algorithm

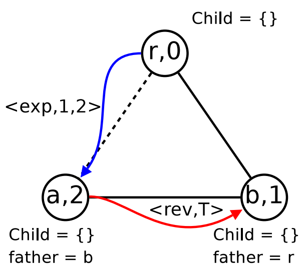
B Example: Old-parent problem



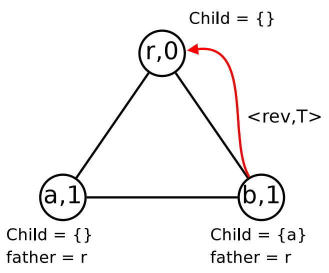
(a) r sends ⟨**explore**⟩ messages to a and b.



(b) b accepts r as its parent and forwards the ⟨**explore**⟩ message to a. The message through $r - a$ is very slow.



(c) a accepts b as its parent and replies to b.



(d) b reports back to r and assumes a is its child. However, the ⟨**explore**⟩ from r to a arrived and a now knows a better route to r. b should somehow be informed about the fact that a has a better parent now. This situation is called the old-parent problem.

Figure 8: Example of the old-parent problem