

## 1 Introduction

In these scribe notes, we examine the problem of routing data from all stations to a particular destination in a graph  $G$ . Traditional routing protocols like distance vector routing and link state routing work well for networks such as the internet, where links change infrequently, and optimality is a major concern. However, in ad hoc mobile networks, the design goals are different. Because the devices are mobile, links change frequently. As the links are formed and broken, a large number of packets are needed for all stations to modify their routing tables correctly. This wastes precious energy and is not a great solution, especially since the mobile stations are typically battery powered.

We would like a solution, where topology changes do not have *global* consequences; meaning that all stations in the network need not know about every single topology change. In other words, the solution should be *local*.

**Problem Statement:** All stations are trying to find a route to a single destination. Construct a directed acyclic graph (DAG), where the edges are directed towards the destination.

We should note here that this solution has nothing to do with trees. Although we intuitively expect a tree like structure to solve this data dissemination problem, trees are expensive to build, maintain and update. Therefore, all we need is a DAG. As a result, we do not consider reverse multicast trees as a solution to this problem.

**Chronology of Work:** The first work on *link reversal* techniques was published in 1981 by Gafni and Bertsekas, in their seminal paper: *Distributed Algorithms for Generating Loop-Free Routes in Networks with Frequently Changing Topology*. In 1997, Park and Corson implemented this idea in *TORA* and published the results in the paper: *A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks*. In 2003, Busch and Tirthapura analyzed the protocol and published their paper: *Analysis of Link Reversal Routing Algorithms for Mobile Ad Hoc Networks*.

## 2 Link Reversal

**Assumption:** We assume that every station knows its current neighbors. As discussed

in class, this is a fair assumption to make, especially in wireless networks. There are many practical approaches to a station determining its neighbor list. Another assumption is that the channel is symmetric. in other words, if station  $A$  can hear station  $B$ , then  $B$  can also hear station  $A$ .

The basic idea is the following: Each station knows its list of neighbors. We start with a graph  $G$ , where each station  $i$  in the network is represented by a node  $v_i$  in the graph  $G$ . For every pair of stations  $i$  and  $j$  that can hear each other, the graph  $G$  has an edge  $(v_i, v_j)$ . We start off by the edges being undirected. Then, we arbitrarily direct the edges, such that the graph has no cycles. Then, we perform *link reversal* operations to get a DAG that is oriented toward the destination node. Figure 1 shows a DAG on graph  $G$ , while figure 2 shows the corresponding destination oriented DAG. We say that a graph  $G$  is *destination oriented* if there is a directed path from every node to the destination.

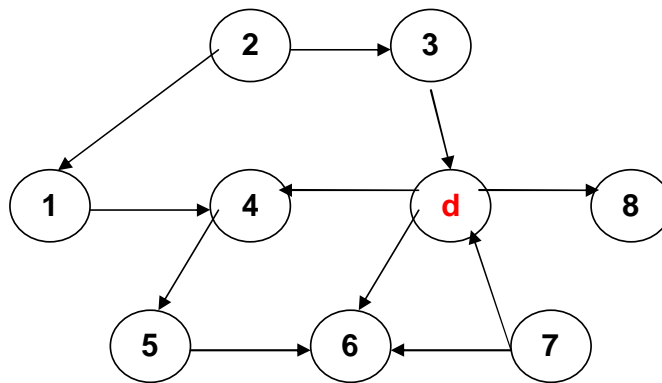


Figure 1: Example of a Directed Acyclic Graph

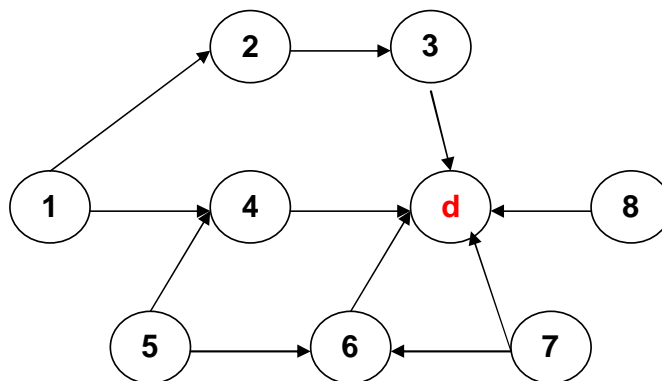


Figure 2: Directed Acyclic Graph - Destination Oriented

Through these operations, we have to ensure that no cycles are formed. How do we do this? To get an intuitive idea, imagine the following: Every station  $u$  is associated with a height  $h_u$ .

Consider two stations  $i$  and  $j$  that are in direct range of each other. If  $h_i > h_j$ , then let the edge  $(v_i v_j)$  be oriented as  $v_i \rightarrow v_j$ , else the edge is oriented as  $v_j \rightarrow v_i$ .

**Basic Fact:** If the heights associated with the stations are unique, then the graph will have no cycles. The proof is intuitive and simple, and hence has been omitted from these notes. In the remainder of these scribe notes, we do not revisit the concept of *height* associated with each node.

In the 1981 paper by Gafni and Bertsekas, the authors talk about three link reversal algorithms:

- Full reversal
- Partial reversal
- General algorithm

The focus of today's class was only on the *full reversal* technique. The following discussion introduces the concepts, and follows it up with a few theorems and their intuitive proofs. In the following discussion, we assume that the network of stations and the graph  $G$  are equivalent; Hence, we make no distinction between them.

We know that in mobile ad hoc networks, the network topology is highly dynamic; meaning the links change often. Let us assume that at some point of time, the network is in steady state; meaning that the network is stable, but the DAG is not *destination oriented*. We say that a graph  $G$  is *destination oriented* if there is a directed path from every node to the destination. By *stable*, we mean that no topology changes are currently taking place. Now, if a topology change takes place, the graph may not be destination oriented. At this time, the *link reversal* technique is applied to make the graph destination oriented. While this algorithm is running, we assume that no further topology changes take place.

**Sink Node:** We should not get confused between the destination node and the sink node. The destination node is the node which is the intended recipient of all the data that is generated in the network. We use the term *sink* to represent a node with the following properties:

- $in\_degree \geq 1$
- $out\_degree = 0$ ;

We would ideally like the graph  $G$  to have only one sink node (destination).

**Lemma:** A DAG  $G$  is not destination oriented if and only if it contains a sink node other than the destination.

**Proof:** We consider the proof in both directions.

- **Case 1: If the graph  $G$  contains a sink node other than the destination**, then  $G$  is not destination oriented. Recall that we say that  $G$  is *destination oriented* if there is a directed path from every node to the destination. If  $G$  has a sink node  $u$  other than the destination, then clearly  $u$  has no path to the destination. Therefore  $G$  is not destination oriented.
- **Case 2: If  $G$  is not destination oriented**, that means that there exists a node  $u$  such that there is no path from  $u$  to the destination. Recall that since  $G$  is a DAG, it can not have a cycle. This means that there is a path of length  $\geq 0$  from  $u$  to a sink node  $v$ . Therefore,  $G$  has at least one sink node, apart from the destination  $d$ . Clearly,  $v$  is not the same as the destination  $d$  because there is no path from  $u$  to  $d$ . From these two cases, the proof is complete.

### 3 Full Reversal Algorithm:

This is the simplest algorithm and performs best in the worst case. As an aside, the average case performance of partial reversal is better than that of full reversal. As mentioned before, we assume that the neighbor list does not change during the course of execution of the algorithm. Also, every station knows its current set of neighbors at any point of time.

The algorithm works as follows: Recall that a sink is a node that has only incoming edges, but no outgoing edges. Whenever a non destination node detects that it has become a sink, it runs this algorithm. It considers each of its incoming edges and reverses their direction to make them outgoing edges. This process of link reversal is also known as *firing*. Figure 3 explains this clearly. we note that this is a distributed algorithm and all nodes run the same algorithm.

**Claim 1:** If two nodes  $u$  and  $v$  reverse in the same step, then they are not neighbors.

**Proof:** Consider two nodes  $u$  and  $v$  as shown in figure 4. Since they are neighbours, let the edge between them be from  $u \rightarrow v$ .

Now, assume  $u$  and  $v$  fire at the same time. This means that they both are *sink* nodes, which is a contradiction, as there is an edge between them. Neighboring nodes cannot have an outdegree of 0 because there is an edge between them. Only one of them could have fired. Therefore, for two nodes to fire at the same time, they cannot be neighbors. Hence proved.

**Claim 2:** At each step of the algorithm, the graph  $G$  will be acyclic.

**Proof:** Though this is not a formal proof, it is intuitive enough to understand, and for us to leave out the messy details. We know that before the algorithm was run,  $G$  was in a steady state, meaning that it did not have any cycles. Now, what happens when the algorithm is run? Consider the graph shown in figure 5. Initially, there are two paths from  $u$  to  $v$ . The first path is a direct edge, while the second path is a multi hop path ( $u \cdots av$ )

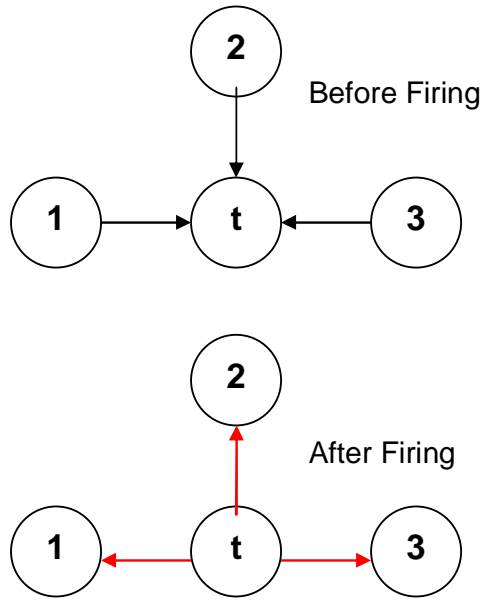


Figure 3: Sink Node Firing

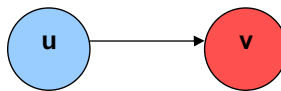


Figure 4: Diagram for Claim 1

Now, clearly, node  $v$  is a sink. Therefore, it fires and reverses the edges incident on it. Does this create a cycle? If only the  $v \rightarrow u$  edge was reversed, then it could potentially create a cycle. But since all edges incident on  $v$  are reversed, a cycle cannot be formed. Therefore, at every step of the algorithm, the graph  $G$  will be acyclic.

**Claim 3:** If the graph  $G$  is connected, the algorithm terminates, ending in a destination oriented DAG.

**Proof:** Define for each node  $u$ , the *reversal distance*  $rd(u)$  to be the minimum of number of edges on path  $p$  oriented the wrong way, over all *undirected* paths  $p$  from  $u$  to  $d$ . Look at figure 6. The edges in the *good* direction are shown using black solid arrows, while the edges in the *bad* direction are shown using red broken arrows.

It is easy to observe the following: The path  $uabd$  has one edge in the *bad* direction. Path  $ucd$  has one edge in the *bad* direction. Path  $ued$  has two edges in the *bad* direction. By the definition of  $rd(u)$ , we say that  $rd(u) = 1$ .

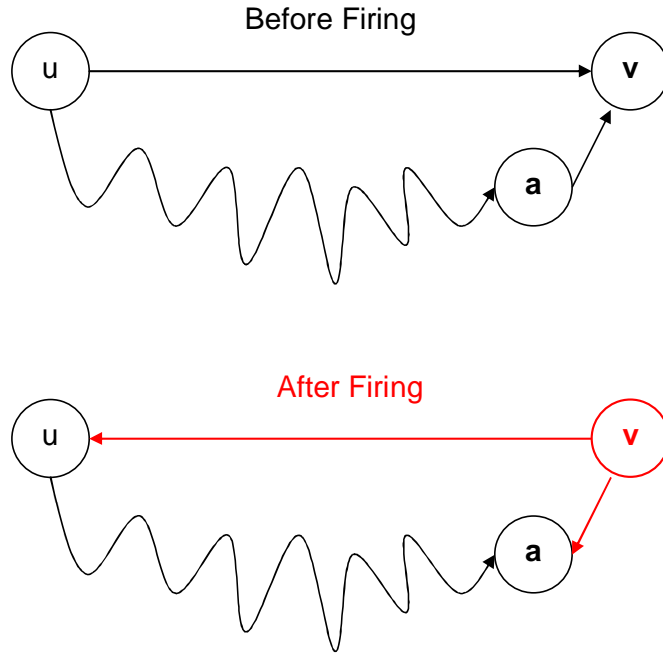


Figure 5: Diagram for Claim 2

Next, we define the reversal distance of the graph  $G$  as follows:

$$rd = \sum_u rd(u) \quad (1)$$

It basically tells us how *bad* the graph is. We say that a node is *bad*, if there is no path from it to the destination. Let  $b$  denote the number of *bad* nodes in  $G$ . Based on the definition of reversal distance, we see that:

$$\begin{cases} rd(u) = 0, & u \text{ is a good node} \\ rd(u) \geq 1, & u \text{ is a bad node} \end{cases}$$

If the node  $u$  has a directed path to the destination  $d$ , then no edges need to be reversed. Therefore,  $rd(u) = 0$ . However, if there is no path from  $u$  to the destination  $d$ , then one or more edges have to be reversed in order for  $u$  to be able to reach  $d$ . Therefore, in this case,  $rd(u) \geq 1$ . It is easy to see that in any state  $rd \leq b^2$ ; Recall that  $b$  is the number of bad nodes and  $rd$  is the sum of all  $rd(u)$  values over all nodes  $u$  in  $G$ . This gives an upper bound of  $O(b^2)$  work.

**Convergence Time:** Now, let us take a brief (but interesting) look into how long the algorithm takes to bring  $G$  back into a steady state, after a topology change has taken place. The cornerstone of our argument is that with every iteration, the number of *bad* nodes will not increase.

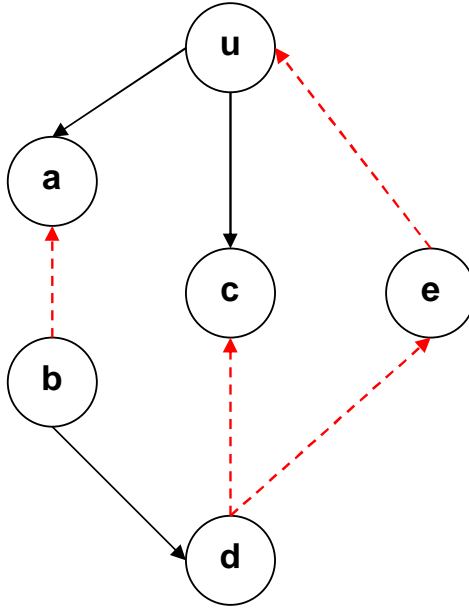


Figure 6: Diagram for  $rd(u)$

In other words, every iteration reduces  $rd$ . From figure 7, take a look at what happens when node  $u$  fires.

We see that node  $u$  was a sink node (it has no path to the destination) , and hence it fires. Assuming that there exists an *undirected* path from  $u$  to  $d$ , it is easy to see that  $rd(u)$  decreases by 1. Now consider node  $v$ , such that  $v$  is a one-hop neighbor of  $u$ . It is easy to notice that node  $v$  is *affected* by  $u$  reversing its edges. Suppose  $v$  was not a bad node prior to  $u$  firing, it means that  $v$  had a path to  $d$  that does not involve  $u$ . Therefore, the reversal of the  $uv$  link does not affect  $rd(v)$ . But, if the shortest undirected path from  $v$  to  $d$  passes through  $u$ , then the reversal of the  $uv$  link makes  $v$  one *bad* link further away from  $d$ . However, by the reversal, node  $u$  has reduced the number of *bad* edges between itself and  $d$  by 1. Therefore,  $rd(v)$  remains unchanged.

In summary, we say that when a node  $u$  reverses its links,  $rd(u)$  reduces by 1. However, for all other nodes  $v$ , under no circumstances does  $rd(v)$  change when the links of node  $u$  are reversed.

Now, let us look at the worst case performance of this algorithm. if  $b$  is the number of bad nodes, then we get the proper destination oriented graph after  $b^2$  reversals. Intuitively, the worst case performance of this algorithm is for a chain graph, where every edge points in the wrong direction, away from the destination. Using figure 8 and figure 9, we explain the working of the algorithm. This worst case example beautifully shows how the reversals take place, along with the associated complexity.

**Conventions:** These are the conventions used in figure 8. As before, nodes are shown by black

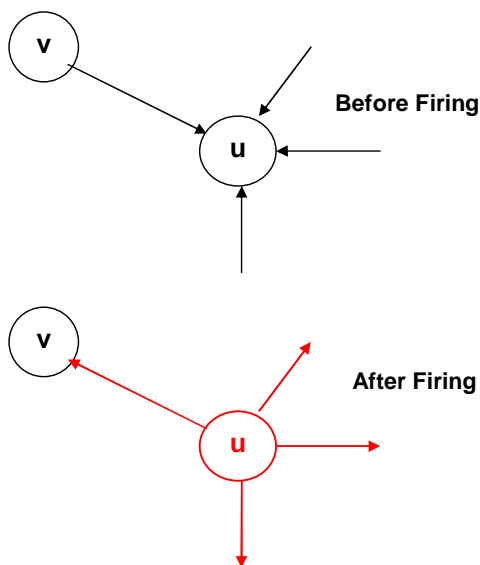


Figure 7: An iteration cannot increase the *badness*

circles. Directed links are shown by black solid arrows. The nodes which fire in a particular iteration are shown in red. The links which change their direction are shown in red dotted arrows. Nodes which have established paths to the destination are shown in green; These nodes do not reverse their edges any more. Similarly, those directed links which are *finalized* are shown using green solid arrows. Finally, the iteration number is shown using the blue rectangle, which is next to the chain graph. Although this convention seems complicated, it is my fond hope that the figure is self explanatory. Therefore, I strongly urge you to take a color print out of these scribe notes, if you prefer a hard copy.

From figure 8, observe iteration 0. We see that there are 6 nodes, apart from the destination node. The nodes of this chain graph all point away from the destination. Therefore, the number of *bad* nodes is equal to 6. The number of link reversals that take place can be counted from figure 8 and 9. The red color solid arrows represent the reversals that are taking place in a particular iteration. If we simply count the number of red solid arrows from all the iterations, we see that the count is 36. Note that figure 9 is a continuation of figure 8. This verifies our claim that if  $G$  has  $b$  *bad* nodes, then it needs  $O(b^2)$  reversals to correctly orient the graph.

**Claim 4:** If a node  $s$  starts out with a directed path to  $d$ , then it never fires.

**Proof:** The proof is rather intuitive. Consider a node  $a$  which has a directed path to the destination node  $d$ . Let this path be:  $a, b, c, \dots d$ . Now, since  $a$  has a path to  $d$  and  $a$  is not a sink node, it never fires. Let us look at the various neighbor topology changes that *may* cause  $a$  to fire. Let one of its neighbors (Apart from  $b$ ) fire. Therefore, some links incident on  $a$  will change. However, the path to  $d$  still remains untouched. Therefore,  $a$  will not fire.

So we can say that  $a$  will fire only if  $b$  fires, because the  $a \rightarrow b$  link is the first hop used by  $a$  to

reach  $d$ . Under what circumstances will  $b$  fire? By a similar argument, we say that  $b$  will fire only if the first hop node in the  $b$  to  $d$  path fires; Otherwise,  $b$  will not fire. We go on arguing this way till we reach the node  $\alpha$ , which is one hop away from the destination node  $d$ . There is an  $\alpha \rightarrow d$  edge, and hence,  $\alpha$  will never fire. This is the base case. By an inductive argument, we see that if any node has a path from itself to  $d$ , it will never fire. Only those nodes which are *bad* will fire.

**Lower Bound by Busch et al:** Here, we introduce the concept of a *layer*. In a graph  $G$ , consider a node  $u$ ; We explain how to find out which layer node  $u$  belongs to. Look at all the undirected paths from  $u$  to  $d$  and ask the question: *How many bad direction edges do we have to cross to reach the destination?* The answer to this question gives us the layer of a node  $u$ ; This is nothing but  $rd(u)$ . Therefore, all nodes at level  $k$  need a minimum of  $k$  edges reversed to reach  $d$ . Figure 10 shows a simple example of nodes present in layer 1 of a graph  $G$ .

For figure 10, we note the following convention. The destination node is  $d$ . The nodes 1, 2 and 3 (which are enclosed in the big blue circle) all have directed paths to the destination. These paths are shown by the bold black arrow. Note that these black arrows represent entire multi hop paths, not just single links. Now, the nodes encapsulated in the blue rectangle are those nodes which need one edge reversed, so that they can reach the destination. These nodes are therefore, in layer 1. Note that the red dotted arrow represents a single directed link, and not a multi hop path.

**Theorem by Busch et al:** Every node in layer  $k$  must fire  $k$  times to become a good node.

**Proof Idea:** Though the following discussion really doesn't prove the result, it gives us a good intuition of what is really happening. We try to find the worst case execution, where if a node reverses its edges, it needs to reverse them again, at some point of time. We are trying to find out the worst case number of reversals that a node has to perform. Given this algorithm, we are trying to find the worst case execution. Let us analyze the algorithm, where every *bad* node fires once. Call this execution as: *Segment*  $e_1$ . We then continue with the sequence  $e_2, e_3, e_4, \dots e_k$ .

If the *bad* node is at layer 1, it becomes a good node. If the bad node is at layer 2, it goes to layer 1. This way, we say that if the bad node is at layer  $i$ , it jumps to layer  $i - 1$ . Let us understand this better with the help of an example, shown in figure 11, 12 and 13. Note that figure 12 is a continuation of figure 11, and figure 13 is a continuation of figure 12.

Initially, the network is like the first graph of figure 11. From this, it is easy to observe that node 1 is in layer 1. Nodes 2, 3, 4, 5, 6 and 7 are in layer 2. Finally, node 8 is in layer 3. From figure 11, 12 and 13, observe how many times each node fires. We see that node 1 fires just once before it becomes a *good* node, and is shown in green. Nodes 2, 3, 4, 5, 6 and 7 fire twice each before they become *good* nodes. Node 8 fires 3 times before it becomes a good node. This was an informal *proof* by example of the theorem by Busch et al, which states that: *Each node in layer  $k$  must fire  $k$  times to become a good node.*

**Complexity Analysis:** We have performed very simple complexity analysis on this algorithm, and summarized it in this paragraph. We have shown that every node in layer  $k$  must fire  $k$  times to become a good node. If there are  $n$  nodes in the graph, then the maximum value of  $k$  is equal to  $n$ . It is simple to see that the total number of reversals has a lower bound of  $\Omega(n^2)$ . In the earlier part of the scribe notes, we showed that the upper bound on the number of reversals is of the order:  $O(n^2)$ . Combining these two results, we conclude that this algorithm runs in  $\Theta(n^2)$ .

**Final Discussion:** All this time, we have been talking about the worst case complexity, and the amount of work involved. Now, let us be a little more optimistic and see what sort of network topologies lend themselves to efficient use by this algorithm. Consider a clique of size  $n$ . In this case, it is easy to see that the time needed to create a destination oriented DAG is linear in the size of the clique. In other words, it is  $\Theta(n)$ .

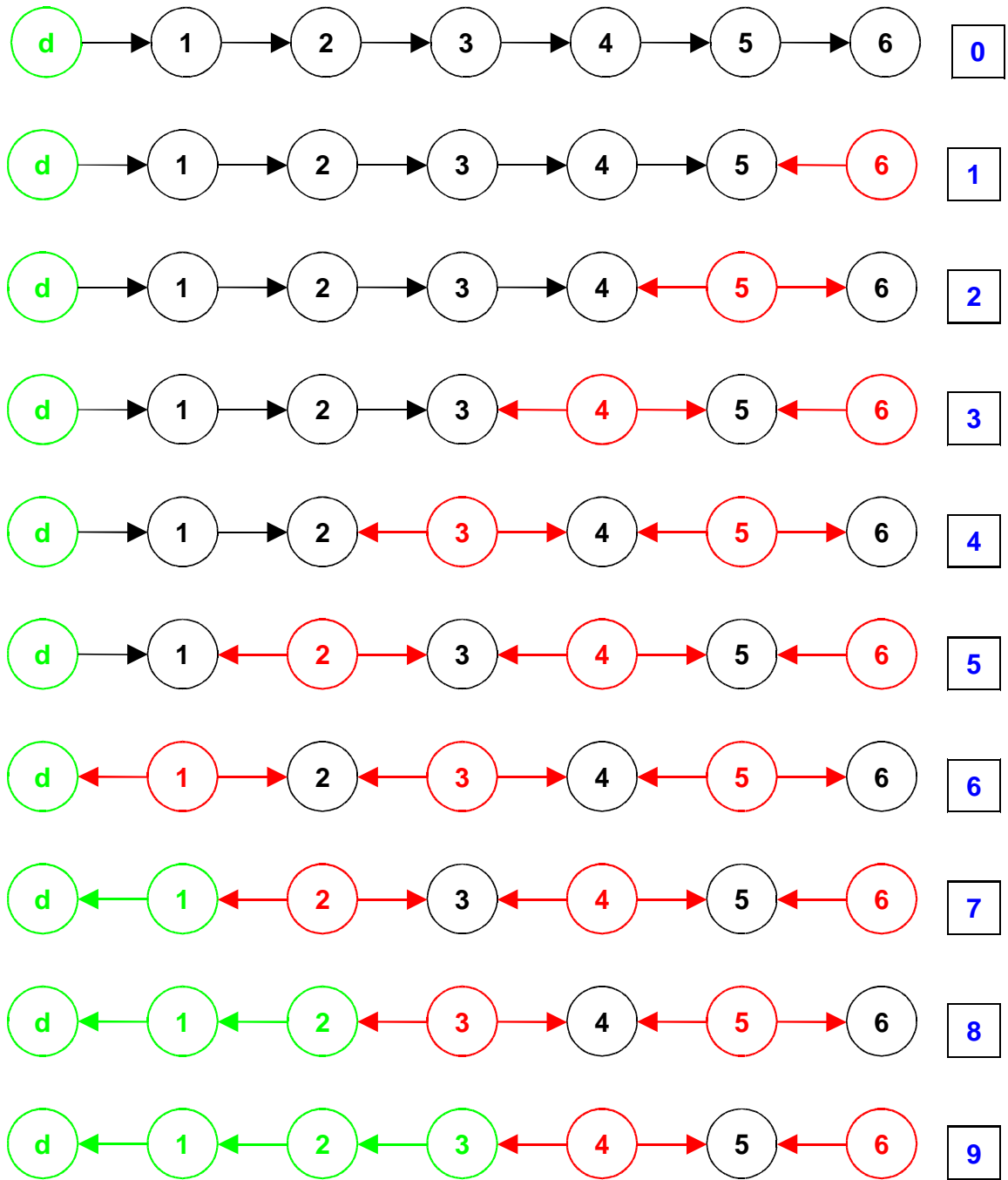


Figure 8: The algorithm running on a chain graph

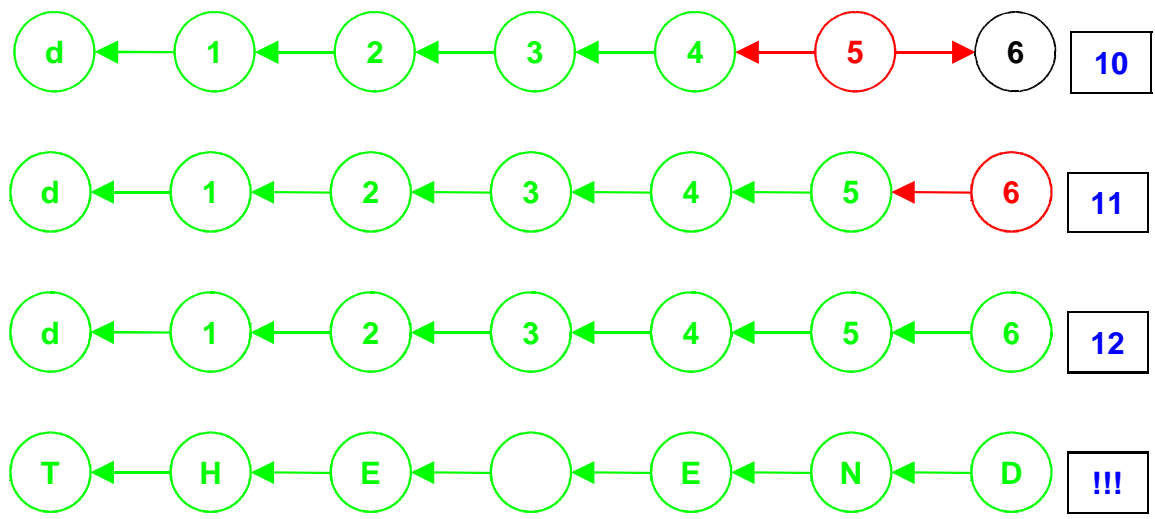


Figure 9: The algorithm running on a chain graph - Cont'd

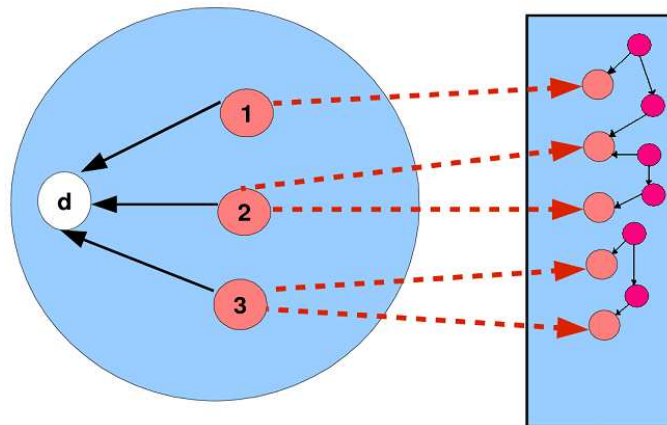


Figure 10: Layer 1 nodes - Inside the rectangle

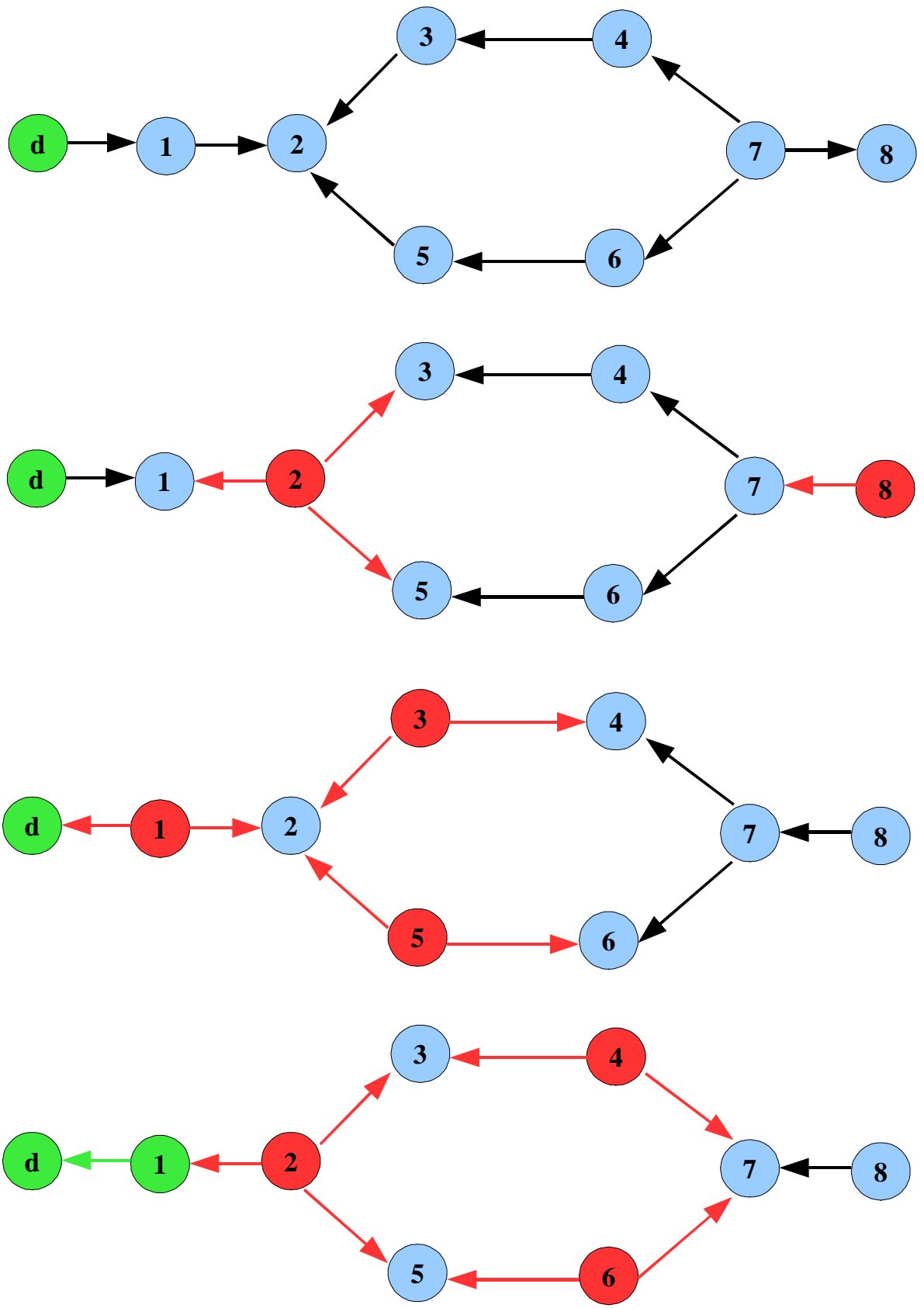


Figure 11: Relation between the Layer of a node and the Number of times it fires

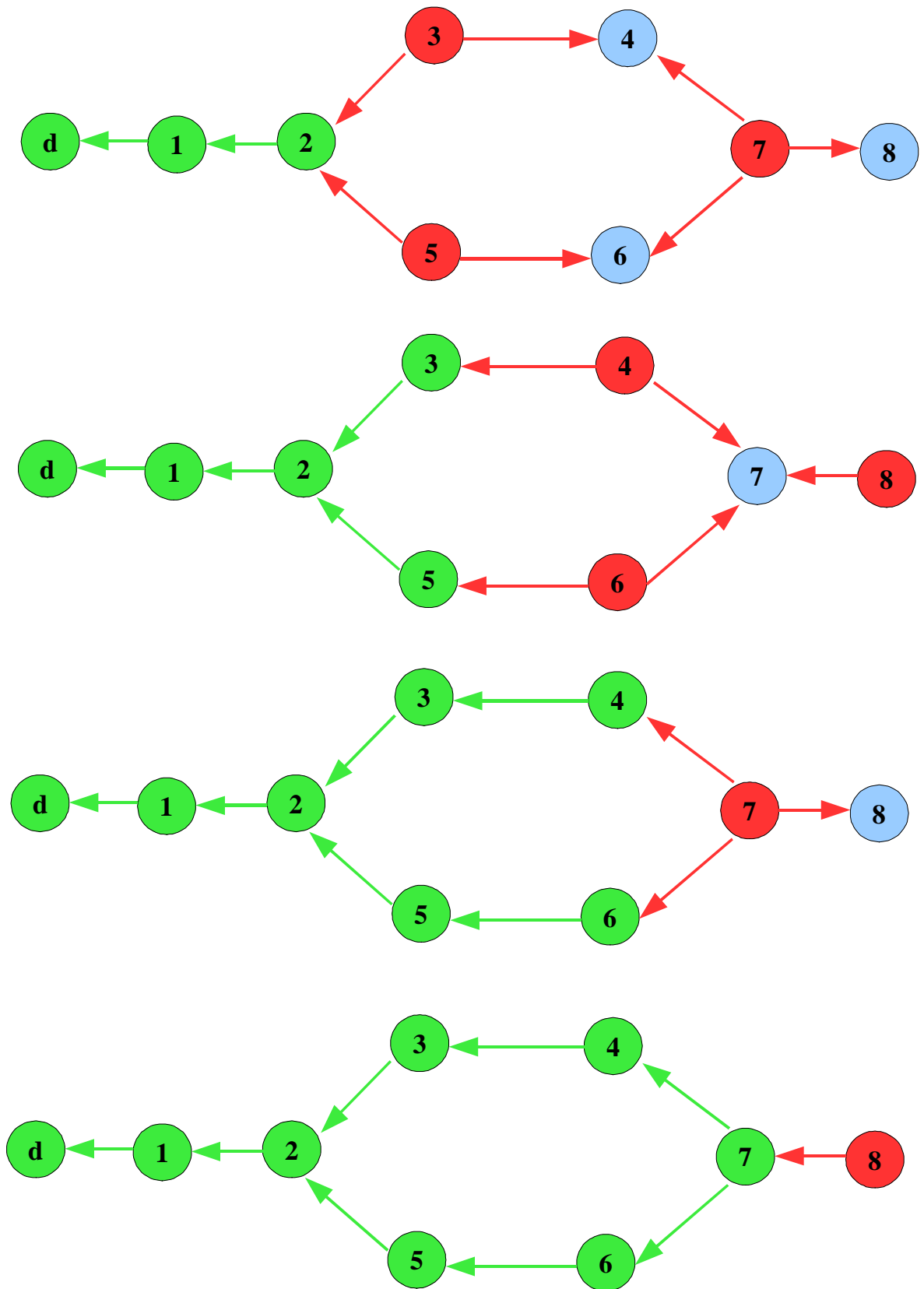
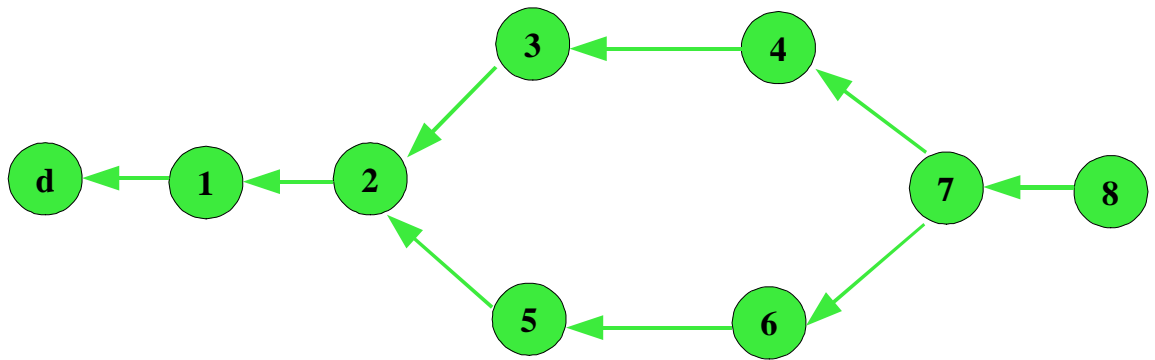


Figure 12: Continuation of Figure 11



**Destination Oriented DAG Successfully Created**

Figure 13: Continuation of Figure 12