Massachusetts Institute of Technology
6.895: Advanced Distributed Algorithms
Professor Nancy Lynch

Lecture 13
April 3, 2006

Point-to-point routing

Readings:
Point-to-point message routing:
Perkins, Royer: Ad hoc on-demend distance-vector routing (AODV).
Chen, Murphy: Enabling disconnected transitive communication in mobile ad hoc networks.
Link-reversal algorithms:
Gafni, Bertsekas: Distributed algorithms for generating loop-free routes in networks with frequently changing topology.
Busch, Surapaneni, Tirthapura: Analysis of lin reversal algorithms for mobile ad hoc networks.

Next time: Park, Corson TORA paper.
Rao et al. Geographical routing without location info.
Fang et al. GLIDER: Gradient Landmark-based distributed routing.
Fonseca et al. Beacon vector routing.

1 Introduction

We are studying the problem of routing a message from a particular source to a named destination, in a mobile ad hoc network. The main problem is that we don't initially know where that named node is. Also, the network may be changing, due to failures, new nodes joining, and mobility.

Last time we studied Dynamic Source Routing (DSR). Today, we'll talk about Ad-Hoc On-Demand Distance Vector Routing (AODV), and (briefly) a little paper proposing a variation that tries to *take advantage* of mobility. Then, we will study a particular strategy that traces back to the 80s, work by Gafni and Bertsekas: link-reversal algorithms. We'll see how this idea has evolved into algorithms like TORA that have been proposed for wireless networks.

Next time, we'll move on to what is sometimes called "location-free routing", by which they mean routing without explicit use of geographical information. Instead, these methods use some kind of substitute for geography.

Next week, we will move to approaches that actually use real location information, such as Geocasting, location-aided routing (LAR), compass routing, etc. After Patriot's Day, we will also consider a related problem: Keeping track of the geographical locations of particular nodes ("location services").

2 AODV

2.1 Introduction

AODV can be considered a *response* to DSR and DSDV (an older, simpler algorithm), arguably two of the most important ad hoc routing protocols around the time this paper was published.

Both DSR and DSDV have advantages:

1. *DSDV doesn't bloat packets.* Source routing algorithms, on the other hand, put the whole route in packets, adding to their size, increasing the chance of collisions, and reducing throughput.
2. *DSR discovers routes only as needed.* As an on-demand protocol, storage space isn't wasted on little-trafficed nodes, and control traffic overhead is reduced.

Both DSR and DSDV also have disadvantages (just swap and invert the statements above):

1. DSR *bloats* packets.
2. DSDV discovers routes even if they are *not* needed.

AODV, in a sense, is trying to combine these advantages while avoiding the disadvantages. Notice that the core of DSR's advantage is that it's on-demand. The core of DSDV's advantage is that it's distance vector based. AODV, therefore, is an on-demand version of a distance vector algorithm. Nodes maintain distance-vector based routing tables, and route messages according to these tables. The tables are populated on an on-demand fashion; next hops are only added as routes are needed, and eventually timeout if not used or detected as broken.

AODV then gives us the best of both worlds...not to mention that the combination of on-demand and distance vector approaches reduces local storage space requirements even lower than DSR, as only next hop entries are needed for each active destination, rather than the whole route. This combination (no bloat, small local storage, less control traffic) makes AODV more *scalable* than DSR and DSDV.

2.2 Assumptions

The assumptions are basically the same as for DSR:

- The network is connected.
- Nodes are willing to "participate fully".
- Mobility is slower than routing.
- Nodes are in promicuous receive mode.

However, unlike DSR, the network is not assumed to be small. AODV is intended to be scalable.

2.3 Basic Operation

Consider the network in Figure 1. Say that S wants to talk to D and does not have an entry for D in its routing table. S initiates *route discovery* by locally broadcasting a *route request (RREQ)*, which contains the following info: <source_addr, source_seq, bcast_id, dest_addr, dest_seq, hops>. The source_addr and bcast_id uniquely identify this RREQ.

When a node receives this RREQ:

1. If it has already seen this RREQ then ignore it. (You know you've seen it before if it the same source/bcast id as something you've seen before.)
2. If you have "relevant" information then "reply". (Both to be explained soon.)

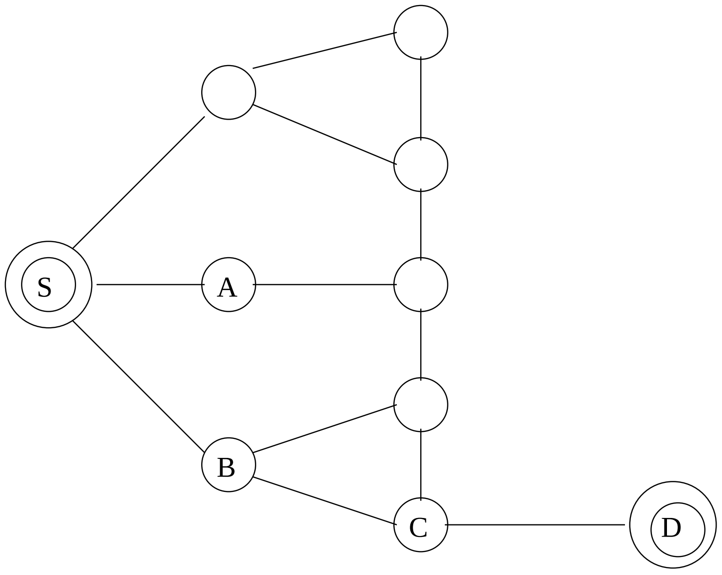

Figure 1. Sample network.

3. Else, increment hop count, rebroadcast and record the following:
   a. who forwarded RREQ to you
   b. its source
   c. its destination
   d. expiration time

What does "Relevant" information mean (option 2)? If you *are* the destination, or have a better route with the same destination sequence number, or a route with a better destination sequence number, then you have useful information and should reply.

We will discuss what it means to reply in a second. For now, assume no node knows any route to D. As the RREQ propagates through the network, the information nodes record about whom they received it from sets up a bunch of temporary reverse pointers (temporary because after expiration time they are discarded). See Figure 2.

How does a node with relevant information reply to an RREQ? It generates a *route reply (RREP)* of the following format: < source_addr, dest_addr, dest_seq, hops, lifetime>.
This is then passed back along the reverse pointers.

These RREP's are somewhat seperated from RREQs. (Note, they have no broadcast ID). The basic rule is that for a given <source, dest> pair, you pass an RREP along the reverse pointer for this pair if and only if you haven't already passed along such an RREP for a larger destination sequence number, or the same sequence number with the same or better hop count. This implies book-keeping about RREPs handled for each <source,dest> pair.

Also, this assumes a single reverse link for a given <source, dest> pair; this is reasonable, since if you have a reverse link that didn't time out, then you are on active path between source and destination. If the source was to send out a new RREQ for this dest, this means the current route was broken, so you should throw out this information and accept the new RREQ (and accompanying reverse pointer).

As the RREP passes back (in a unicast fashion), nodes that handle it set up the appropriate forward pointers in their routing table entry for D. See Figure 3.

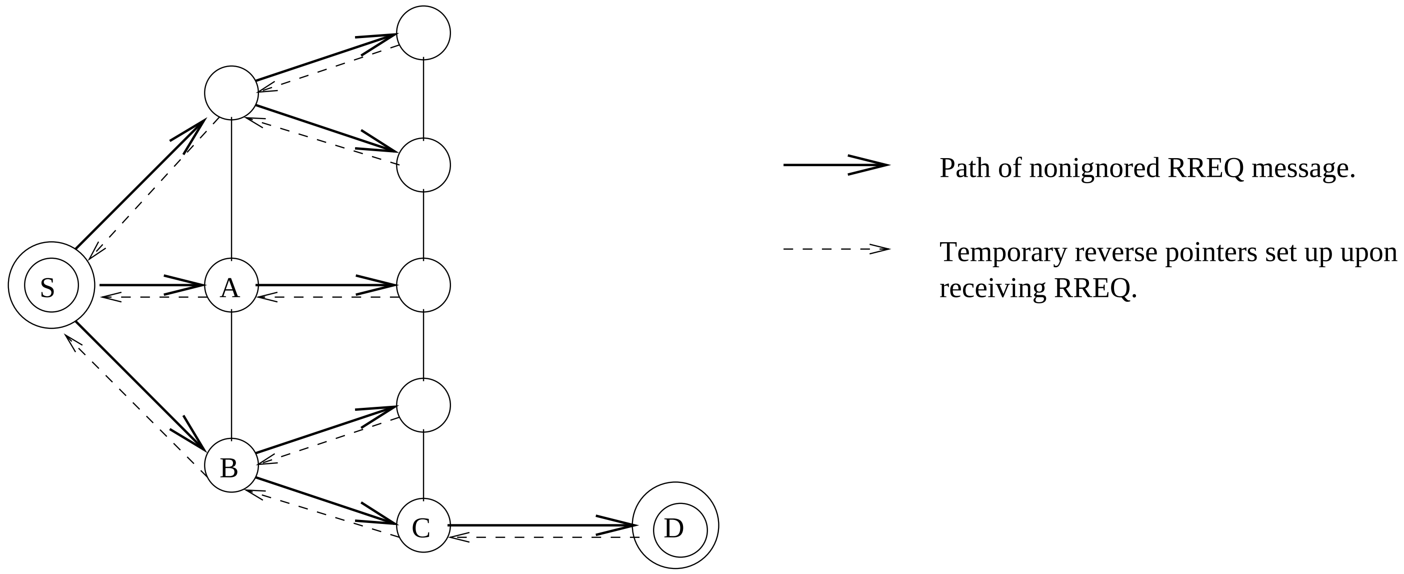
S
A
B
C
D
Path of nonignored RREQ message.
Temporary reverse pointers set up upon receiving RREQ.

Figure 2. Path discovery.

Each routing table entry is indexed on destination, and holds:

1. Next Hop.
2. Number of Hops.
3. Destination sequence number.
4. Active neighbors (the nodes upstream from you, used for local connectivity management).
5. Expiration time.

Entries 2 and 3 are used if new routes are heard; they are used to update the routing table.

2.4 Local Connectivity Management

Let's shake things up, by having node D move. See Figure 4.

Now, assume A goes looking for D. A will get RREPs from S, B, and D. It will accept D's, of course, because it has a better distance sequence number. HOWEVER, B hear's D's RREP. This RREP has a better destination sequence number (and hop count) so B changes its routing table entry. See Figure 5.

How does C determine its path is broken? Active neighbors are upstream neighbors for this route (those with reverse pointers to you). You are expected to send a HELLO message every so often to these active neighbors. If active neighbors don't hear such a HELLO for a long enough amount of time (a system parameter) they assume you're long gone, and propagate back a special RREP with infinite hop count and one greater destination sequence number. The effect: Anyone who has not found a more recent route to the destination will remove it from their table. See Figure 6.

If D, on the other hand, had moved far away then C's infinite RREP would have propagated all the way back to S, forcing them to rediscover from scratch.

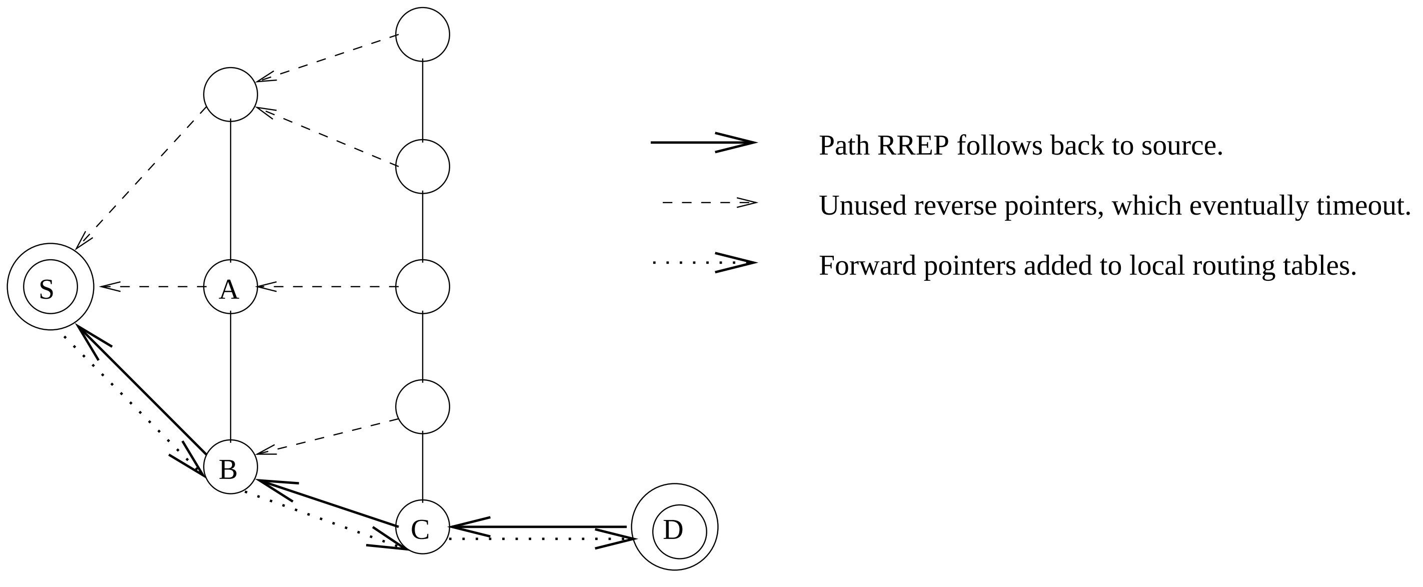
S
A
B
C
D
Path RREP follows back to source.
Unused reverse pointers, which eventually timeout.
Forward pointers added to local routing tables.

Figure 3. Path discovery continued.

2.5 Simulation Results

They evaluated AODV by running it on an event-driven, packet-level simulator named PARSEC. However, it was not very realistic (they over-simplified the physical layer):

1. Strict range cut-off.
2. Perfect physical carrier sensing with BEB.
3. Collision always detected (losing both messages), if two neighbors broadcast at the same time.

NS-2, for example, is more realistic. It implements 802.11 DCF on top of a radio model that simulates SNR at any point.

First, they experiment with some parameters; this was not that interesting. Then, they ran some experiments with 50, 100, 500, and 1000 nodes. These last two are very large compared to most MANET simulations, and serves as a proof of concept of scalability.

For small node numbers, they got very high "goodput", or, roughly, the good bits minus the bad. They had around 98% for 50-100 nodes. For more nodes got around 72%. At this point up to 25% of messages were lost to collisions. Collisions are a very important factor with routing protocols.

For some real world AODV results, consider the following experiment and results:

- Approximately 40 people carrying laptops, and moving randomly.
- Playing fields, 225 x 365m.
- Traffic generation: 1200 bite packets, 5.5 packets per stream, 3 seconds between packets, 15 seconds between streams (the numbers were derived from a military application prototype).
- Around 425 bytes per second (modest).

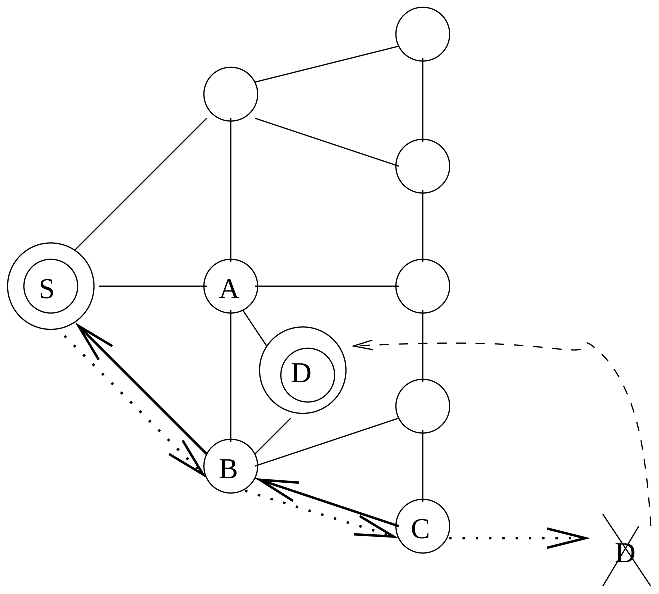

Figure 4. Route to D changes.

- Compare to 64 byte packets with delay of 20 msecs between.
- MESSAGE DELIVERY RATIO: 50%.
- Best Latency compared to others tested (ODMRP, APRL, STARA).
- Second best hop count.
- Smallest amount of overhead.

This seems to indicate that fluctuating links are a concern; you might think you are somebody's neighbor because you got a msg, but the link quality was bad. Also, control traffic is important to consider. If you are too aggressive in trying to find good routes, you can actually make things worse.

3 Distributed Transitive Communication

Previous routing algorithms we discussed assumed a connected network that stayed connected–only the topology of the connection changed. This paper (Chen/Murphy) wants to assume a more sparse model. The network is not always fully connected. Two nodes are never guarenteed to be transitively connected at any instant.

Instead, imagine nodes in clusters, with clusters intermittently connected. The kinds of applications that are okay in this setting have highly asynchronous communication (think minutes and hours delay, not milliseconds and seconds). (Don't use this to run a ssh connection, for example.) This can be fine to update two military camps about a change in the battleplan for a battle coming up in a few days.

3.1 Basic Idea

The basic idea is simple: try to pass a message to a node in your cluster that is "closest" to the intended destination. Figuring out who is "closest" (utility) is the whole trick, and might

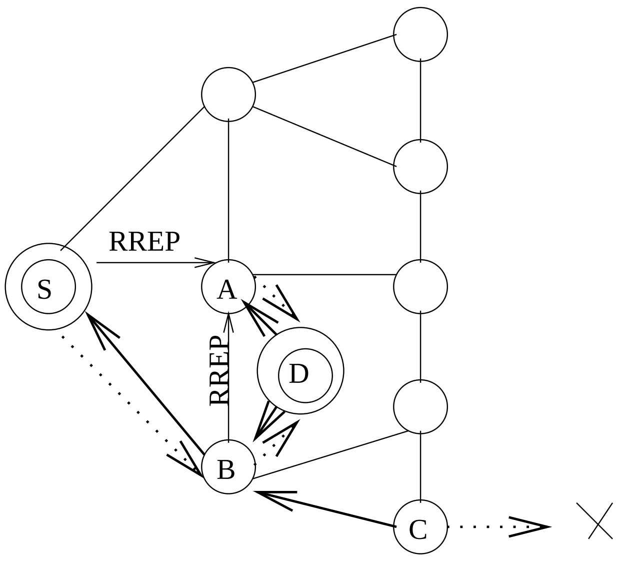

Figure 5. A looks for D; B hears better route.

require application-layer information. Also, we will use a low-level routing protocol, like DSR, to communicate within clusters. Accordingly, they consider this middleware.

Summary: if you have a message to pass on, broadcast a probe to your connected component, and recipients will test their utility for that destination. If high enough, they will respond (both broadcasting and response uses low-level protocol like DSR or AODV), and pass message on to person with best utility.

```
Basic Algorithm for a node S with a message m to send to D:
   FindNextHop(message m)
     while(true)
       delay(RDI)
       broadcast(<m.dest, m.timeOut, m.weights, m.utilityComp,
                 threshold>).
       delay(ResponseTimeOut)
       myUtility = localUtility(m)
       if \exists response x s.t. x.utility > myUtility then
           send(x,m)
           break.
```

Here are some other important pieces:

- If you RECEIVE a PROBE you “calculate utility” (to be explained soon), and if better than probe threshold, respond to probe originator through unicast.
- If receive a RESPONSE to PROBE, store it for consideration later.
- If you RECEIVE a MESSAGE then if you are destination, pass it to the application, else call FindNextHop for message.

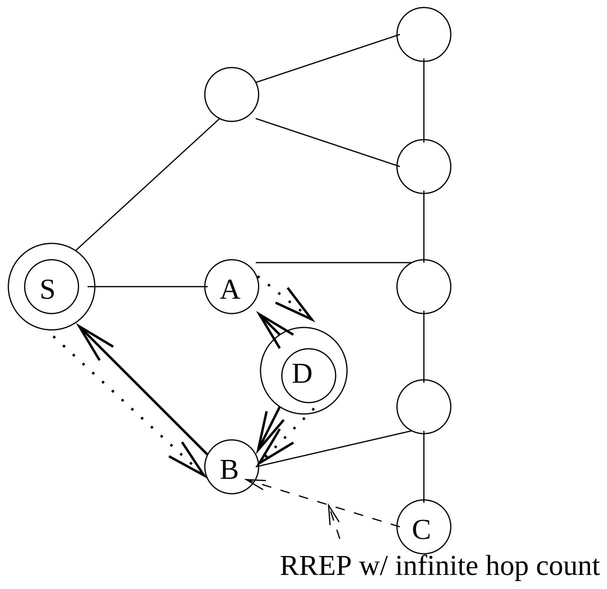

Figure 6. C detects break, clears entry, propagates RREP.

3.2 Utility

They defined the following potential utility metrics:
(History-based:)
1) Most Recently Noticed – higher utility if destination noticed more recently.
2) Most Frequently Noticed – higher utility if destination noticed more often.
(Future-based:)
3) Future Plans – assumes the application will tell you the next time it expects to see a certain host (i.e. by reading a calendar app).
(Status-based:)
4) Power – more power, better utility.
5) Rediscovery Interval – smaller RDI means better chance of discovery of destination or someone closer. The actual equations are in paper, and not that interesting. In some cases (i.e. power) they are arguably not well-defined.

The receiver uses the “Weights” component of a probe to figure out how much weight to give each of these metrics in the sum it calculates. IN ADDITION to these five, they give the sender of the probe the ability to send its own utility calculation function to take advantage of special, scenario specific information. If there is a utilityComp function in the probe, we use that, else we use weighted sum of the five presented before.

There are a couple details. Choosing RDI is important. If it is too BIG, you miss potential connections, but if it is too SMALL, you create too much traffic. Their solution is to double RDI after each successful probe. If a new node is discovered, drop back to initial.

How do they "discover new nodes"? They use hello messages. If a hello reveals a new neighbor:
a) reset RDI, and
b) set a boolean flag in your next HELLO to 1.
If receive a HELLO with flag = 1, do the two steps above.
In order to avoid an infinite loop of RDI resets, we must ignore the next cycle of HELLOs after a reset.

4 Link Reversal Algorithms

Link state and distance vector algorithms are too expensive to maintain, in the face of network changes as in mobile ad hoc networks. Dynamic source routing (DSR) is less expensive, since it operates on-demand, but it still costs communication to flood packets to find a route. It is also somewhat hard to maintain in a large network, since it requires finding an entire route ahead of time.

Now we will see another idea, dating back to 1981, which maintains loop-free routes in the face of network changes, while executing only *local* operations. The routes are not guaranteed to be optimal. However, they are always loop-free.

4.1 Overview

The technique was first presented in the classical 1981 paper by Gafni and Bertsekas. It has been studied and analyzed since then, perhaps most interestingly by Busch et al., in SPAA 2003. It has also been adapted and elaborated into proposals for practical mobile network routing, most notably, TORA (Temporally-Ordered Routing Algorithm).

The setting considered is a mobile ad hoc network (in 1981, this was called a "packet radio network"). The network is assumed to be subject to change as described earlier. We construct an undirected graph, representing a bidirectional neighbor relationship: Nodes must be able to figure out if they are neighbors at any particular time, and a node that transmits should be heard by all its current neighbors.

These are rather strong assumptions for a real network. In particular, this is saying that network changes amount to atomic steps, in which the required communications are guaranteed, and in which both endpoints of an edge simultaneously become aware of the correct status of the edge.
Q: How can we achieve these assumptions in a real network? (They have some comments at end of paper about using an agreement protocol to do this.)

However, it seems like the algorithms still make some sense with other versions of these assumptions, e.g., that the network eventually, or periodically, stabilizes so that changes stop and the correct information is known everywhere. The TORA algorithm, in fact, uses weaker assumptions, allowing a discrepancy in time between when two nodes learn they are, or are not, neighbors. For now, for studying Gafni-Bertsekas and Busch, we will accept the stronger assumptions.

In the scenario considered, all the nodes in the dynamic network want to establish and maintain routes to a single destination node d. They do this by examining their neighbors' states and applying a simple function to their own states (we assume they can do this atomically). The result is that, at all times, each edge is *oriented* in one direction or the other (TORA also allows some edges to exist without being oriented). The orientations are arranged so that the entire structure *never* has any directed cycles. That is, the graph with its directed edges forms a DAG (directed acyclic graph). It need not be a tree—it can have multiple paths between the same two nodes. In fact, the papers feature this possibility, saying that it can provide redundant routes for forwarding messages.

The fact that the graph is always acyclic does not mean that all directed paths are always directed toward the destination d. Some paths can lead nowhere—dead-ending in a *sink* node that isn't the destination d. The problem is, basically, to start with a DAG, and modify it by simple local rules to end up with a new DAG in which all the edges are parts of routes directed towards d.

The algorithms in the papers are described in two ways. First, they are described in terms of explicit link directions, guaranteed always to form a DAG, and explicit *link reversals*—changes in direction of some of the links. The link reversal operation must be designed carefully so that it can never create a cycle. For example, one trick that appears in GB is that, when a node performs a reversal, it simultaneously makes all its adjacent edges outgoing. Clearly these new edges cannot be part of any cycle, so such an operation cannot create a cycle. (Nobody had edges into this node.) However, this trick doesn't suffice to explain why all their algorithms guarantee acyclicity.

Second, they describe them in more concrete terms, using *heights* associated with the nodes (not the edges), h_u with node u. Heights are chosen from some totally-ordered set, and, at any point in time, all the heights are different. Link (u, v) is defined to be directed from u to v provided that $h_u > h_v$. Think of a ball rolling downhill from u to v in the direction of the link, from higher to lower height. Obviously, if unique heights are used to define orientations of edges, the digraph must always be acyclic.

Now we will proceed to the algorithms, generally following GB but with some material extracted from Busch et al. (BST) along the way, since BST present some of the same ideas with clearer definitions and proofs. There are three algorithms in GB: Full reversal, partial reversal, and a general height-based algorithm.

4.2 Gafni-Bertsekas algorithm assumptions

GB describe their three algorithms for a synchronous model, in which all the nodes take their steps in synchronous rounds. Generally, they are considering the case where the network is stable, but the digraph initially might not be directed toward d, that is, might not be "destination-oriented". In this case, their job is simply to reorient the edges (eventually) toward d.

The same algorithms also make sense in adaptive settings, where nodes and edges get added and removed. You have to be careful with initialization conditions for this analysis; look at the Busch paper. This analysis focuses on the stable case.

The following basic property of DAGs with a destination node d is useful:

Lemma: A DAG G is not destination-oriented iff it contains a sink node other than the destination d.
Proof: Obviously if it has such a sink node, it's not destination-oriented.
Now suppose it it not destination-oriented. Then there is some node u from which there is no path to d. If u is a sink we are done; otherwise, follow an outgoing edge from u. Keep following such edges until we reach a sink. We must eventually reach one because there are only finitely many nodes and we can't have any loops in a DAG. The resulting sink isn't d since we assumed G contains no path from u to d. QED Lemma

4.3 The full reversal algorithm

This is BG's simplest algorithm. It's presented first abstractly, in terms of directed edges and reversals, and then concretely, in terms of heights.

4.3.1 Abstract full reversal algorithm

Assume we have a DAG. At each step, if there are any nodes $\neq d$ that are sinks (have no outgoing edges), then one or more of these reverses the directions of all its incoming links. Sometimes we'll refer to a node reversing its links as "firing".
Claim 1. If two nodes u and v both reverse in the same step, then u and v aren't neighbors.
Proof: Obvious, since they are both sinks.

Claim 2. At each stage, the graph is acyclic.
Proof: We argued this above—more formally, we argue by induction on the number of steps. No step can create a cycle, because at each step, each node u that fires reverses all its edges to point outward. None of these edges can then be part of a cycle, since there are no edges into u. So the step could not have created a cycle if none existed before.

Claim 3. If the graph is connected, then the algorithm eventually terminates, ending up with a DAG that is destination-oriented (all paths lead to d).
Proof: (This is a new, easy proof, not in GB, though related arguments are used in Busch et al. to prove an upper bound on the number of steps.) Define, for each node u, the "reversal distance" $rd(u)$ to be the minimum, over all undirected paths ρ from u to d, of the number of edges on path ρ that are oriented in the wrong way (that you have to traverse backwards).
Define a metric for the entire state: $rd = \Sigma_u rd(u)$.
Clearly, in any state, $rd \leq b^2$, where b is the number of "bad" nodes—those that don't have directed paths to the route. That's because we have to consider, in the sum, terms for only bad nodes u (the other terms would be 0). For each such node, the smallest number of reversals is at most equal to the number b of bad nodes—after traversing at most that many links, you would have to reach a "good" node, which does have a directed path to d, and then there are no further reversals.
Now we argue that every step reduces this metric. Why? At each step, at least one node fires. If more than one does, the effect is just as if they fired one at a time (since no two can be neighbors—all must be sinks). So let's just consider the case where a single node u fires.
This node u has $rd(u)$ decreased by 1: Any path it followed before, it can still follow after the reversal, but now one fewer edge has to be traversed in the wrong direction. Moreover, no other node v has $rd(v)$ increased: If a path from v went through u before, it had to traverse an edge in and an edge out, first in the right direction and then in the wrong direction. After the reversal, it still can follow the same path, with the same cost: now traversing the first edge in the wrong direction and the second in the right direction.
It follows that after at most b^2 steps, this terminates with a destination-oriented graph.

Claim 4: If a node starts out having a directed path to d, then it never fires.
Proof: We prove this by induction on the length k of the shortest directed path from a node v to d. For length $k = 0$, we have $v = d$, and d never fires.
Now consider a node v with a (shortest) directed path of length $k + 1$, assuming the result for k. The first node along that path in the direction of d has a shortest path of length at most k. So by the inductive hypothesis, it never fires. However, then v can never become a sink, since it always has an outgoing edge, so never fires.

Let's look at an example, Figure 7. Note the order of reversal steps for it:
1. node 6 (only)
2. node 5
3. nodes 4 and 6

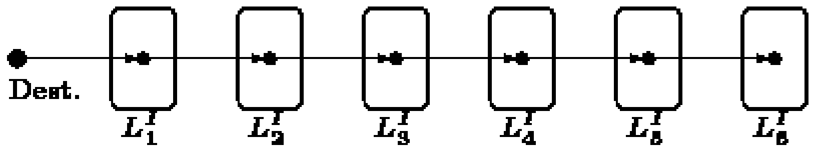

Figure 7. Graph with 6 bad nodes.

4. nodes 3 and 5
5. nodes 2, 4, and 6
6. nodes 1, 3, and 5
7. nodes 2, 4, and 6
8. nodes 3 and 5
9. nodes 4 and 6
10. node 5
11. node 6

Finally, another interesting result about this algorithm proved by GB:

Claim 5: Any two executions of the full reversal algorithm in a connected network, starting from the same global state (here, this just means the link directions) are "equivalent", in the sense that (from the Busch paper):
Each node performs the same number of reversals in both executions, and the final state is the same.
Proof: (This appears in the BG appendix, and is also proved in the Busch paper.) The basic idea is that nodes that are enabled to fire in the same step aren't neighbors, so the order in which they perform their steps doesn't matter. In fact, it's all the same if they fire one per synchronous step. So WLOG assume that two executions ρ and ρ' from the same initial state both involve only one firing per step.
Next, they consider "canonical form" executions, in which, at each step, only one fires, and moreover, it's a particular one—the one with the minimum index among those that are enabled to fire. Then, starting from any execution with only one firing per step, they can play enough commutativity games, reordering steps so they wind up with an equivalent "canonical form" execution. So if they start with two different executions ρ and ρ', they can put each in canonical form while maintaining equivalence. However, from a given initial state, there is only one canonical form execution—so ρ and ρ' must also be equivalent.

Busch et al. upper bounds:
Recall above I showed that the number of steps is $O(b^2)$, where b is the initial number of bad nodes. Busch et al. prove this. This turns out to be not just an upper bound on the number of steps (time measure) but also, on the total number of reversals (work measure). It is basically the same argument.

Busch et al. lower bounds:
They also give example graphs and initial settings for which the amount of work, and the amount of time, are $\Omega(b^2)$. There are two examples, because the one for work is simpler than the one for time.

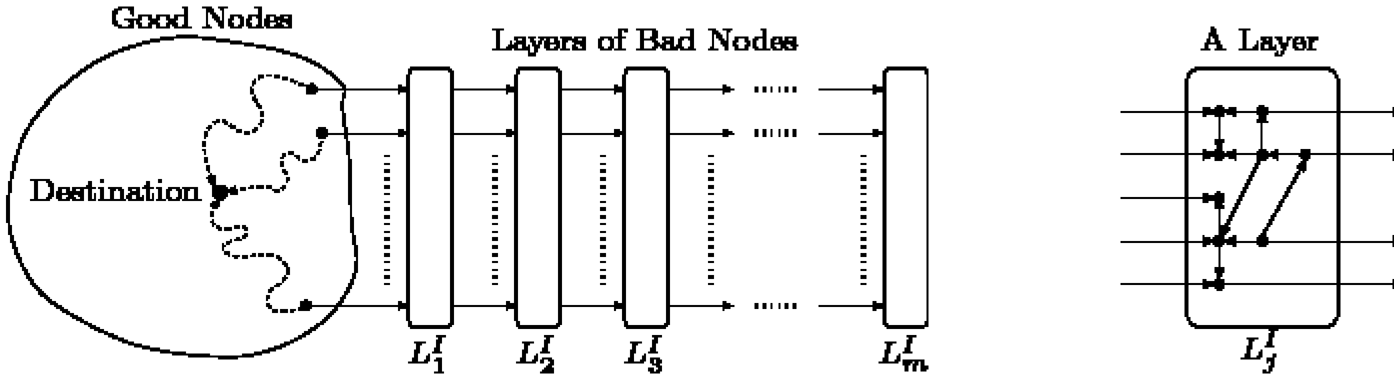

Figure 8. Partitioning nodes into layers.

See Figure 8, for the basic setup. They divide the initial graph into “layers” numbered 1,2,3. The nodes in layer k are those that have a path to d with k edges directed the wrong way, but no path to d with only $k-1$ incorrectly-directed edges. All these nodes are bad nodes, in the initial digraph.

They prove the key Theorem 4.7, asserting that each node in layer L_k must fire k times before it becomes a good node. To prove this, they consider an equivalent canonical execution in which, first, every bad node, in every layer, fires once. The result keeps all the portions of the DAG internal to all the layers the same as before, turns around the arrows beween d and Layer 1, but keeps all the other between-layer arrows pointing the wrong way, as before. In particular, it turns all the nodes in Layer 1 into good nodes, but all the others remain bad. Then they repeat the argument, thus getting k firings for the Level k nodes. Since this holds for a canonical execution, it works in all executions.

To get the work bound, see Figure 7. A series of “layers” each containing exactly one node, with the edges pointed the wrong way, away from d. By Theorem 4.7, each node in Layer k must perform k reversals; this gives the $\Omega(b^2)$ bound on work.

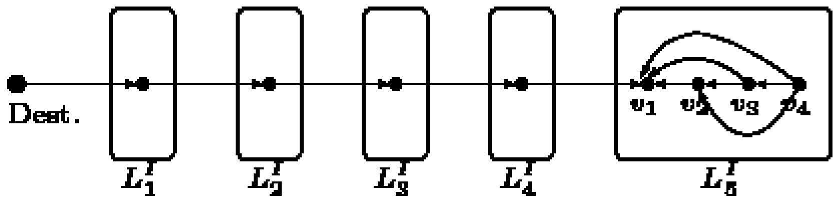

Figure 9. Graph with 8 bad nodes.

For the lower bound on time, they use the digraph in Figure 9. Here, the last layer is big—it contains half the nodes. They form a clique in the undirected graph, which are linearly ordered in the initial digraph. Again BST uses Theorem 4.7 to see that all the nodes in the last layer $k = b/2$ must fire at least (approximately) k times. But now, note that all these must be done sequentially—since the nodes form a clique, no two can be sinks at the same time, so they cannot fire concurrently.

4.3.2 Concrete full reversal algorithm

We have already managed to present a good portion of the algorithms. in BG and results in BST, in terms of just the abstract version of the full reversal algorithm. But BG also have a concrete version of this algorithm:
Associate with each node u, at each time, a pair (α_u, u). Here, u is the node's uid (and a tiebreaker), and α_u is a nonnegative integer sequence number. We assume that these pairs are totally ordered lexicographically. Initially, the pair for u is $(0, u)$. Call this pair the *height* of node u.

The concrete algorithm interprets these values as follows:
An edge (u, v) is said to be directed from u to v iff $(\alpha_u, u) > (\alpha_v, v)$, that is, if the height h_u of u is greater than the height h_v of v. Since these pairs are ordered lexicographically (and are globally unique, because of the second components), this immediately implies absence of cycles. Such an assignment of pairs can be defined initially (based on a topological sort of the DAG).

Now, each node u, to reverse its edges, simply sets α_u to be $max_{v \in N_u} \alpha_v + 1$.

They observe that this implementation faithfully emulates the abstract algorithm described earlier. All the results I described above carry over to this version too, without any new proofs needed.

4.4 The partial reversal algorithm

4.4.1 Abstract partial reversal algorithm

This is another special case algorithm, a bit more complicated than the full reversal algorithm. Again, only sinks can reverse links. Now, however, they needn't reverse all their edges, only some subset.

We have to be careful—we can't allow an arbitrary subset, or we could create cycles where there were none previously: consider nodes d, u, v, w, with directed edges $(u, d), (u, v), (u, w), (v, w)$. This has no cycles. We can see that w is a sink. If we could reverse just one of its edges—in particular, (u, w)—we would get a cycle.

GB Partial reversal: Each node $u \neq d$ maintains a list of its neighbors v that have "fired" since the last time u fired. Initially, all these lists are empty. Then at each step, some set of sinks fire. For each such sink u:
If there exists at least one neighbor of u that isn't on u's current list, then u reverses the directions of exactly the links (v, u) for which v isn't on the list and empties the list.
If there is no such neighbor, that is, u's list contains all of N_u, then u reverses all its incident edges. In either case, every node v such that (v, u) gets reversed adds u to its list.

We have the same claims as before:
Claim 1. If two nodes u and v both reverse in the same step, then u and v aren't neighbors.
Same argument as before.

Claim 2. At each stage, the graph is acyclic.
Proof: Now we can't use the same argument as before, because we might not be reversing all the edges, and hence might create a cycle.
To prove this solely in terms of the abstract formulation, we need some new invariant. (E.g., for any node u, none of its listed neighbors is reachable by a directed path from any of its unlisted neighbors??? Can this be proved by induction. It seems so. Having this seems to allow a proof of

acyclicity. LTTR.)

Claim 3. If the graph is connected, then the algorithm eventually terminates, ending up with a DAG that is destination-oriented.
Proof: The argument given above doesn't work. The following proof of termination (without any time bound) is new, though inspired by the proof of Lemma 1 in BG.
Suppose that an execution doesn't terminate with a destination-oriented graph. Then there is always some sink, and so firing steps continue forever. Then there must be some particular node u that fires infinitely many times.
We know that, at each point where u fires, all of its edges are incoming. Consider three times when u fires; we will show that in between times 1 and 3, every neighbor of u fires at least once. Consider a particular neighbor $v \in N_u$.
Case 1: u reverses the edge to v at the second firing.
Then v must fire between u's second and third firing in order to make u a sink again.
Case 2: u does not reverse the edge to v at the second firing.
Then v must be in u's list at the beginning of the second firing. But u empties its list after its first firing. So, to get onto u's list by the second firing, v must fire between u's first and second firings.

Thus, since u fires infinitely many times, so do all of its neighbors! Continuing this argument step by step in the (undirected) graph, we can conclude that all the nodes fire infinitely many times. But this can't happen for d, a contradiction. QED Claim 3

Exercise: Can you get a time bound from this, by introducing a suitable metric?

The proof of Lemma 1 in BG is actually for their general algorithm. Also, it requires a special hypothesis (A.3) (more on this below). BST do prove termination here, by proving time and work bounds; however, these are stated and proved in terms of the concrete version, below. There are also some funny issues about initialization, as you will see.

Claim 4: If a node starts out having a directed path to d, then it will never reverse the direction of its links.
Proof as before.

Claim 5: Any two executions of the abstract partial reversal algorithm in a connected network, starting from the same global state (which includes not just link directions but also list information) are "equivalent": Each node performs the same number of reversals in both executions, and the final state is the same.
Proof as before.