Chapter 4

Distributed breadth-first search

Here we describe some algorithms for building a **breadth-first search** (**BFS**) tree in a network. All assume that there is a designated **initiator** node that starts the algorithm. At the end of the execution, each node except the initiator has a parent pointer and every node has a list of children. These are consistent and define a BFS tree: nodes at distance k from the initiator appear at level k of the tree.

In a synchronous network, **flooding** (§3.1) solves BFS; see [AW04, Lemma 2.8, page 21] or [Lyn96, §4.2]. So the interesting case is when the network is asynchronous.

In an asynchronous network, the complication is that we can no longer rely on synchronous communication to reach all nodes at distance d at the same time. So instead we need to keep track of distances explicitly, or possibly enforce some approximation to synchrony in the algorithm. (A general version of this last approach is to apply a synchronizer to one of the synchronous algorithms using a **synchronizer**; see Chapter 7.)

To keep things simple, we'll drop the requirement that a parent learn the IDs of its children, since this can be tacked on as a separate notification protocol, in which each child just sends one message to its parent once it figures out who its parent is.

4.1 Using explicit distances

This is a translation of the AsynchBFS automaton from [Lyn96, §15.4]. It's a very simple algorithm, closely related to Dijkstra's algorithm for shortest paths, but there is otherwise no particular reason to use it. Not only does it not detect termination, but it is also dominated by the $O(D)$ time and $O(DE)$ message complexity synchronizer-based algorithm described in §4.3. (Here D is the **diameter** of the network, the maximum distance between any two nodes.)

The idea is to run flooding with distances attached. Each node sets its distance to 1 plus the smallest distance sent by its neighbors and its parent to the neighbor supplying that smallest distance. A node notifies all its neighbors of its new distance whenever its distance changes.

Pseudocode is given in Algorithm 4.1

```
initially do
    if pid = initiator then
        distance ← 0
        send distance to all neighbors
    else
        distance ← ∞

upon receiving $d$ from $p$ do
    if $d + 1 <$ distance then
        distance ← $d + 1$
        parent ← $p$
        send distance to all neighbors
```

Algorithm 4.1: AsynchBFS algorithm (from [Lyn96])

(See [Lyn96] for a precondition-effect description, which also includes code for buffering outgoing messages.)

The claim is that after at most $O(VE)$ messages and $O(D)$ time, all distance values are equal to the length of the shortest path from the initiator to the appropriate node. The proof is by showing the following:

Lemma 4.1.1. *The variable* $\mathsf{distance}_p$ *is always the length of some path from initiator to p, and any message sent by p is also the length of some path from* initiator *to p.*

Proof. The second part follows from the first; any message sent equals p's current value of distance. For the first part, suppose p updates its distance; then it sets it to one more than the length of some path from initiator to p', which is the length of that same path extended by adding the pp' edge. □

We also need a liveness argument that says that $\mathsf{distance}_p = d(\mathsf{initiator}, p)$ no later than time $d(\mathsf{initiator}, p)$. Note that we can't detect when distance stabilizes to the correct value without a lot of additional work.

In [Lyn96], there's an extra $|V|$ term in the time complexity that comes from message pile-ups, since the model used there only allows one incoming message to be processed per time units (the model in [AW04] doesn't have this restriction). The trick to arranging this to happen often is to build a graph where node 1 is connected to nodes 2 and 3, node 2 to 3 and 4, node 3 to 4 and 5, etc. This allows us to quickly generate many paths of distinct lengths from node 1 to node k, which produces k outgoing messages from node k. It may be that a more clever analysis can avoid this blowup, by showing that it only happens in a few places.

4.2 Using layering

This approach is used in the *LayeredBFS* algorithm in [Lyn96], which is due to Gallager [Gal82].

Here we run a sequence of up to $|V|$ instances of the simple algorithm with a distance bound on each: instead of sending out just 0, the initiator sends out $(0, \mathsf{bound})$, where bound is initially 1 and increases at each phase. A process only sends out its improved distance if it is less than bound.

Each phase of the algorithm constructs a partial BFS tree that contains only those nodes within distance bound of the root. This tree is used to report back to the root when the phase is complete. For the following phase, notification of the increase in bound increase is distributed only through the partial BFS tree constructed so far. With some effort, it is possible to prove that in a bidirectional network that this approach guarantees that each edge is only probed once with a new distance (since distance-1 nodes are recruited before distance-2 nodes and so on), and the bound-update and acknowledgment messages contribute at most $|V|$ messages per phase. So we get $O(E + VD)$ total messages. But the time complexity is bad: $O(D^2)$ in the worst case.

4.3 Using local synchronization

The reason the layering algorithm takes so long is that at each phase we have to phone all the way back up the tree to the initiator to get permission to go on to the next phase. We need to do this to make sure that a node is only recruited into the tree once: otherwise we can get pile-ups on the channels as in the simple algorithm. But we don't necessarily need to do this globally. Instead, we'll require each node at distance d to delay sending out a recruiting message until it has confirmed that none of its neighbors will be sending it a smaller distance. We do this by having two classes of messages:[1]

- $\mathsf{exactly}(d)$: "I know that my distance is d."
- $\mathsf{more\text{-}than}(d)$: "I know that my distance is $> d$."

The rules for sending these messages for a non-initiator are:

1. I can send $\mathsf{exactly}(d)$ as soon as I have received $\mathsf{exactly}(d-1)$ from at least one neighbor and $\mathsf{more\text{-}than}(d-2)$ from all neighbors.
2. I can send $\mathsf{more\text{-}than}(d)$ if $d = 0$ or as soon as I have received $\mathsf{more\text{-}than}(d-1)$ from all neighbors.

The initiator sends $\mathsf{exactly}(0)$ to all neighbors at the start of the protocol (these are the only messages the initiator sends).

My distance will be the unique distance that I am allowed to send in an $\mathsf{exactly}(d)$ messages. Note that this algorithm terminates in the sense that every node learns its distance at some finite time.

If you read the discussion of synchronizers in Chapter 7, this algorithm essentially corresponds to building the **alpha synchronizer** into the synchronous BFS algorithm, just as the layered model builds in the **beta synchronizer**. See [AW04, §11.3.2] for a discussion of BFS using synchronizers. The original approach of applying synchronizers to get BFS is due to Awerbuch [Awe85].

We now show correctness. Under the assumption that local computation takes zero time and message delivery takes at most 1 time unit, we'll show that if $d(\mathsf{initiator}, p) = d$, (a) p sends $\mathsf{more\text{-}than}(d')$ for any $d' < d$ by time d', (b) p sends $\mathsf{exactly}(d)$ by time d, (c) p never sends $\mathsf{more\text{-}than}(d')$ for any $d' \geq d$, and (d) p never sends $\mathsf{exactly}(d')$ for any $d' \neq d$. For parts (c) and (d) we use induction on d'; for (a) and (b), induction on time. This is not terribly surprising: (c) and (d) are safety properties, so we don't need to talk about time. But (a) and (b) are liveness properties so time comes in.

Let's start with (c) and (d). The base case is that the initiator never sends any $\mathsf{more\text{-}than}$ messages at all, and so never sends $\mathsf{more\text{-}than}(0)$, and

[1]In an earlier version of these notes, these messages where called $\mathsf{distance}(d)$ and $\mathsf{not\text{-}distance}(d)$; the more self-explanatory $\mathsf{exactly}$ and $\mathsf{more\text{-}than}$ terminology is taken from [BDLP08].

any non-initiator never sends $\mathsf{exactly}(0)$. For larger d', observe that if a non-initiator p sends $\mathsf{more\text{-}than}(d')$ for $d' \geq d$, it must first have received $\mathsf{more\text{-}than}(d'-1)$ from all neighbors, including some neighbor p' at distance $d-1$. But the induction hypothesis tells us that p' can't send $\mathsf{more\text{-}than}(d'-1)$ for $d'-1 \geq d-1$. Similarly, to send $\mathsf{exactly}(d')$ for $d' < d$, p must first have received $\mathsf{exactly}(d'-1)$ from some neighbor p', but again p' must be at distance at least $d-1$ from the initiator and so can't send this message either. In the other direction, to send $\mathsf{exactly}(d')$ for $d' > d$, p must first receive $\mathsf{more\text{-}than}(d'-2)$ from this closer neighbor p', but then $d'-2 > d-2 \geq d-1$ so $\mathsf{more\text{-}than}(d'-2)$ is not sent by p'.

Now for (a) and (b). The base case is that the initiator sends $\mathsf{exactly}(0)$ to all nodes at time 0, giving (a), and there is no $\mathsf{more\text{-}than}(d')$ with $d' < 0$ for it to send, giving (b) vacuously; and any non-initiator sends $\mathsf{more\text{-}than}(0)$ immediately. At time $t+1$, we have that (a) $\mathsf{more\text{-}than}(t)$ was sent by any node at distance $t+1$ or greater by time t and (b) $\mathsf{exactly}(t)$ was sent by any node at distance t by time t; so for any node at distance $t+2$ we send $\mathsf{more\text{-}than}(t+1)$ no later than time $t+1$ (because we already received $\mathsf{more\text{-}than}(t)$ from all our neighbors) and for any node at distance $t+1$ we send $\mathsf{exactly}(t+1)$ no later than time $t+1$ (because we received all the preconditions for doing so by this time).

Message complexity: A node at distance d sends $\mathsf{more\text{-}than}(d')$ for all $0 < d' < d$ and $\mathsf{exactly}(d)$ and no other messages. So we have message complexity bounded by $|E| \cdot D$ in the worst case. Note that this is gives a bound of $O(DE)$, which is slightly worse than the $O(E + DV)$ bound for the layered algorithm.

Time complexity: It's immediate from (a) and (b) that all messages that are sent are sent by time D, and indeed that any node p learns its distance at time $d(\mathsf{initiator}, p)$. So we have optimal time complexity, at the cost of higher message complexity. I don't know if this trade-off is necessary, or if a more sophisticated algorithm could optimize both.

Our time proof assumes that messages don't pile up on edges, or that such pile-ups don't affect delivery time (this is the default assumption used in [AW04]). A more sophisticated proof could remove this assumption.

One downside of this algorithm is that it has to be started simultaneously at all nodes. Alternatively, we could trigger "time 0" at each node by a broadcast from the initiator, using the usual asynchronous broadcast algorithm; this would give us a BFS tree in $O(|E| \cdot D)$ messages (since the $O(|E|)$ messages of the broadcast disappear into the constant) and $2D$ time. The analysis of time goes through as before, except that the starting time 0 becomes the time at which the last node in the system is woken up by the broadcast. Further optimizations are possible; see, for example, the paper of Boulinier *et al.* [BDLP08], which shows how to run the same algorithm with constant-size messages.