Chapter 2

Tree Algorithms

In this chapter we learn a few basic algorithms on trees, and how to construct trees in the first place so that we can run these (and other) algorithms. The good news is that these algorithms have many applications, the bad news is that this chapter is a bit on the simple side. But maybe that's not really bad news?!

2.1 Broadcast

Definition 2.1 (Broadcast)**.** *A* broadcast *operation is initiated by a single node, the source. The source wants to send a message to all other nodes in the system.*

Definition 2.2 (Distance, Radius, Diameter)**.** *The* distance *between two nodes u and v in an undirected graph G is the number of hops of a minimum path between u and v. The* radius *of a node u is the maximum distance between u and any other node in the graph. The radius of a graph is the minimum radius of any node in the graph. The* diameter *of a graph is the maximum distance between two arbitrary nodes.*

Remarks:

- Clearly there is a close relation between the radius R and the diameter D of a graph, such as $R \leq D \leq 2R$.

Definition 2.3 (Message Complexity)**.** *The* message complexity *of an algorithm is determined by the total number of messages exchanged.*

Theorem 2.4 (Broadcast Lower Bound)**.** *The message complexity of broadcast is at least $n-1$. The source's radius is a lower bound for the time complexity.*

Proof: Every node must receive the message.

Remarks:

- You can use a pre-computed spanning tree to do broadcast with tight message complexity. If the spanning tree is a breadth-first search spanning tree (for a given source), then the time complexity is tight as well.

Definition 2.5 (Clean)**.** *A graph (network) is* clean *if the nodes do not know the topology of the graph.*

Theorem 2.6 (Clean Broadcast Lower Bound)**.** *For a clean network, the number of edges m is a lower bound for the broadcast message complexity.*

Proof: If you do not try every edge, you might miss a whole part of the graph behind it.

Definition 2.7 (Asynchronous Distributed Algorithm)**.** *In the asynchronous model, algorithms are event driven ("upon receiving message ..., do ..."). Nodes cannot access a global clock. A message sent from one node to another will arrive in finite but unbounded time.*

Remarks:

- The asynchronous model and the synchronous model (Definition 1.8) are the cornerstone models in distributed computing. As they do not necessarily reflect reality there are several models in between synchronous and asynchronous. However, from a theoretical point of view the synchronous and the asynchronous model are the most interesting ones (because every other model is in between these extremes).
- Note that in the asynchronous model, messages that take a longer path may arrive earlier.

Definition 2.8 (Asynchronous Time Complexity)**.** *For asynchronous algorithms (as defined in 2.7) the* time complexity *is the number of time units from the start of the execution to its completion in the worst case (every legal input, every execution scenario), assuming that each message has a delay of at most one time unit.*

Remarks:

- You cannot use the maximum delay in the algorithm design. In other words, the algorithm has to be correct even if there is no such delay upper bound.
- The clean broadcast lower bound (Theorem 2.6) directly brings us to the well known *flooding* algorithm.

Algorithm 2.9 Flooding

1: The source (root) sends the message to all neighbors.
2: **Each other node** v upon receiving the message the first time forwards the message to all (other) neighbors.
3: Upon later receiving the message again (over other edges), a node can discard the message.

Remarks:

- If node v receives the message first from node u, then node v calls node u *parent*. This parent relation defines a spanning tree T. If the flooding algorithm is executed in a synchronous system, then T is a breadth-first search spanning tree (with respect to the root).
- More interestingly, also in asynchronous systems the flooding algorithm terminates after R time units, R being the radius of the source. However, the constructed spanning tree may not be a breadth-first search spanning tree.

2.2 Convergecast

Convergecast is the same as broadcast, just reversed: Instead of a root sending a message to all other nodes, all other nodes send information to a root (starting from the leaves, i.e., the tree T is known). The simplest convergecast algorithm is the echo algorithm:

Algorithm 2.10 Echo

1: A leave sends a message to its parent.
2: If an inner node has received a message from each child, it sends a message to the parent.

Remarks:

- Usually the echo algorithm is paired with the flooding algorithm, which is used to let the leaves know that they should start the echo process; this is known as flooding/echo.
- One can use convergecast for termination detection, for example. If a root wants to know whether all nodes in the system have finished some task, it initiates a flooding/echo; the message in the echo algorithm then means "This subtree has finished the task."
- Message complexity of the echo algorithm is $n-1$, but together with flooding it is $\mathcal{O}(m)$, where $m = |E|$ is the number of edges in the graph.
- The time complexity of the echo algorithm is determined by the depth of the spanning tree (i.e., the radius of the root within the tree) generated by the flooding algorithm.
- The flooding/echo algorithm can do much more than collecting acknowledgements from subtrees. One can for instance use it to compute the number of nodes in the system, or the maximum ID, or the sum of all values stored in the system, or a route-disjoint matching.
- Moreover, by combining results one can compute even fancier aggregations, e.g., with the number of nodes and the sum one can compute the average. With the average one can compute the standard deviation. And so on ...

2.3 BFS Tree Construction

In synchronous systems the flooding algorithm is a simple yet efficient method to construct a breadth-first search (BFS) spanning tree. However, in asynchronous systems the spanning tree constructed by the flooding algorithm may be far from BFS. In this section, we implement two classic BFS constructions—Dijkstra and Bellman-Ford—as asynchronous algorithms.

We start with the Dijkstra algorithm. The basic idea is to always add the "closest" node to the existing part of the BFS tree. We need to parallelize this idea by developing the BFS tree layer by layer. The algorithm proceeds in phases. In phase p the nodes with distance p to the root are detected. Let T_p be the tree in phase p.

Algorithm 2.11 Dijkstra BFS

1: We start with T_1 which is the root plus all direct neighbors of the root. We start with phase $p = 1$:
2: **repeat**
3: The root starts phase p by broadcasting "start p" within T_p.
4: When receiving "start p" a leaf node u of T_p (that is, a node that was newly discovered in the last phase) sends a "join $p + 1$" message to all quiet neighbors. (A neighbor v is quiet if u has not yet "talked" to v.)
5: A node v receiving the first "join p+1" message replies with "ACK" and becomes a leaf of the tree T_{p+1}.
6: A node v receiving any further "join" message replies with "NACK".
7: The leaves of T_p collect all the answers of their neighbors; then the leaves start an echo algorithm back to the root.
8: When the echo process terminates at the root, the root increments the phase
9: **until** there was no new node detected

Theorem 2.12. *The time complexity of Algorithm 2.11 is $\mathcal{O}(D^2)$, the message complexity is $\mathcal{O}(m + nD)$, where D is the diameter of the graph, n the number of nodes, and m the number of edges.*

Proof: A broadcast/echo algorithm in T_p needs at most time $2D$. Finding new neighbors at the leaves costs 2 time units. Since the BFS tree height is bounded by the diameter, we have D phases, giving a total time complexity of $\mathcal{O}(D^2)$. Each node participating in broadcast/echo only receives (broadcasts) at most 1 message and sends (echoes) at most once. Since there are D phases, the cost is bounded by $\mathcal{O}(nD)$. On each edge there are at most 2 "join" messages. Replies to a "join" request are answered by 1 "ACK" or "NACK" , which means that we have at most 4 additional messages per edge. Therefore the message complexity is $\mathcal{O}(m + nD)$.

Remarks:

- The time complexity is not very exciting, so let's try Bellman-Ford!

The basic idea of Bellman-Ford is even simpler, and heavily used in the Internet, as it is a basic version of the omnipresent border gateway protocol (BGP). The idea is to simply keep the distance to the root accurate. If a

neighbor has found a better route to the root, a node might also need to update its distance.

Algorithm 2.13 Bellman-Ford BFS

1: Each node u stores an integer d_u which corresponds to the distance from u to the root. Initially $d_{\text{root}} = 0$, and $d_u = \infty$ for every other node u.
2: The root starts the algorithm by sending "1" to all neighbors.
3: **if** a node u receives a message "y" with $y < d_u$ from a neighbor v **then**
4: node u sets $d_u := y$
5: node u sends "$y + 1$" to all neighbors (except v)
6: **end if**

Theorem 2.14. *The time complexity of Algorithm 2.13 is $\mathcal{O}(D)$, the message complexity is $\mathcal{O}(nm)$, where D, n, m are defined as in Theorem 2.12.*

Proof: We can prove the time complexity by induction. We claim that a node at distance d from the root has received a message "d" by time d. The root knows by time 0 that it is the root. A node v at distance d has a neighbor u at distance $d-1$. Node u by induction sends a message "d" to v at time $d-1$ or before, which is then received by v at time d or before. Message complexity is easier: A node can reduce its distance at most $n-1$ times; each of these times it sends a message to all its neighbors. If all nodes do this, then we have $\mathcal{O}(nm)$ messages.

Remarks:

- Algorithm 2.11 has the better message complexity and Algorithm 2.13 has the better time complexity. The currently best algorithm (optimizing both) needs $\mathcal{O}(m + n \log^3 n)$ messages and $\mathcal{O}(D \log^3 n)$ time. This "trade-off" algorithm is beyond the scope of this chapter, but we will later learn the general technique.

2.4 MST Construction

There are several types of spanning trees, each serving a different purpose. A particularly interesting spanning tree is the minimum spanning tree (MST). The MST only makes sense on weighted graphs, hence in this section we assume that each edge e is assigned a weight ω_e.

Definition 2.15 (MST)**.** *Given a weighted graph $G = (V, E, \omega)$, the MST of G is a spanning tree T minimizing $\omega(T)$, where $\omega(G') = \sum_{e \in G'} \omega_e$ for any subgraph $G' \subseteq G$.*

Remarks:

- In the following we assume that no two edges of the graph have the same weight. This simplifies the problem as it makes the MST unique; however, this simplification is not essential as one can always break ties by adding the IDs of adjacent vertices to the weight.
- Obviously we are interested in computing the MST in a distributed way. For this we use a well-known lemma:

Definition 2.16 (Blue Edges)**.** *Let T be a spanning tree of the weighted graph G and $T' \subseteq T$ a subgraph of T (also called a* fragment*). Edge $e = (u, v)$ is an* outgoing edge *of T' if $u \in T'$ and $v \notin T'$ (or vice versa). The minimum weight outgoing edge $b(T')$ is the so-called* blue edge *of T'.*

Lemma 2.17. *For a given weighted graph G (such that no two weights are the same), let T denote the MST, and T' be a fragment of T. Then the blue edge of T' is also part of T, i.e., $T' \cup b(T') \subseteq T$.*

Proof: For the sake of contradiction, suppose that in the MST T there is edge $e \neq b(T')$ connecting T' with the remainder of T. Adding the blue edge $b(T')$ to the MST T we get a cycle including both e and $b(T')$. If we remove e from this cycle, then we still have a spanning tree, and since by the definition of the blue edge $\omega_e > \omega_{b(T')}$, the weight of that new spanning tree is less than than the weight of T. We have a contradiction.

Remarks:

- In other words, the blue edges seem to be the key to a distributed algorithm for the MST problem. Since every node itself is a fragment of the MST, every node directly has a blue edge! All we need to do is to grow these fragments! Essentially this is a distributed version of Kruskal's sequential algorithm.
- At any given time the nodes of the graph are partitioned into fragments (rooted subtrees of the MST). Each fragment has a root, the ID of the fragment is the ID of its root. Each node knows its parent and its children in the fragment. The algorithm operates in phases. At the beginning of a phase, nodes know the IDs of the fragments of their neighbor nodes.

Remarks:

- Algorithm 2.18 was stated in pseudo-code, with a few details not really explained. For instance, it may be that some fragments are much larger than others, and because of that some nodes may need to wait for others, e.g., if node u needs to find out whether neighbor v also wants to merge over the blue edge $b = (u, v)$. The good news is that all these details can be solved. We can for instance bound the asynchronicity by guaranteeing that nodes only start the new phase after the last phase is done, similarly to the phase-technique of Algorithm 2.11.

Theorem 2.19. *The time complexity of Algorithm 2.18 is $\mathcal{O}(n \log n)$, the message complexity is $\mathcal{O}(m \log n)$.*

Proof: Each phase mainly consists of two flooding/echo processes. In general, the cost of flooding/echo on a tree is $\mathcal{O}(D)$ time and $\mathcal{O}(n)$ messages. However, the diameter D of the fragments may turn out to be not related to the diameter of the graph because the MST may meander, hence it really is $\mathcal{O}(n)$ time. In addition, in the first step of each phase, nodes need to learn the fragment ID of their neighbors; this can be done in 2 steps but costs $\mathcal{O}(m)$ messages. There are a few more steps, but they are cheap. Altogether a phase costs $\mathcal{O}(n)$ time and

Algorithm 2.18 GHS (Gallager–Humblet–Spira)

1: Initially each node is the root of its own fragment. We proceed in phases:
2: **repeat**
3: All nodes learn the fragment IDs of their neighbors.
4: The root of each fragment uses flooding/echo in its fragment to determine the blue edge $b = (u, v)$ of the fragment.
5: The root sends a message to node u; while forwarding the message on the path from the root to node u all parent-child relations are inverted {such that u is the new temporary root of the fragment}
6: node u sends a merge request over the blue edge $b = (u, v)$.
7: **if** node v also sent a merge request over the same blue edge $b = (v, u)$ **then**
8: either u or v (whichever has the smaller ID) is the new fragment root
9: the blue edge b is directed accordingly
10: **else**
11: node v is the new parent of node u
12: **end if**
13: the newly elected root node informs all nodes in its fragment (again using flooding/echo) about its identity
14: **until** all nodes are in the same fragment (i.e., there is no outgoing edge)

$\mathcal{O}(m)$ messages. So we only have to figure out the number of phases: Initially all fragments are single nodes and hence have size 1. In a later phase, each fragment merges with at least one other fragment, that is, the size of the smallest fragment at least doubles. In other words, we have at most $\log n$ phases. The theorem follows directly.

Chapter Notes

Trees are one of the oldest graph structures, already appearing in the first book about graph theory [Koe36]. Broadcasting in distributed computing is younger, but not that much [DM78]. Overviews about broadcasting can be found for example in Chapter 3 of [Pel00] and Chapter 7 of [HKP$^+$05]. For a introduction to centralized tree-construction, see e.g. [Eve79] or [CLRS09]. Overviews for the distributed case can be found in Chapter 5 of [Pel00] or Chapter 4 of [Lyn96]. The classic papers on routing are [For56, Bel58, Dij59]. In a later chapter, we will later learn a general technique to derive algorithms with an almost optimal time and message complexity.

Algorithm 2.18 is called "GHS" after Gallager, Humblet, and Spira, three pioneers in distributed computing [GHS83]. Their algorithm won the prestigious Edsger W. Dijkstra Prize in Distributed Computing in 2004, among other reasons because it was one of the first non-trivial asynchronous distributed algorithms. As such it can be seen as one of the seeds of this research area. We presented a simplified version of GHS. The original paper featured an improved message complexity of $\mathcal{O}(m + n \log n)$. Later, Awerbuch managed to further improve the GHS algorithm to get $\mathcal{O}(n)$ time and $\mathcal{O}(m + n \log n)$ message complexity, both asymptotically optimal [Awe87].

Bibliography

[Awe87] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 230–240, New York, NY, USA, 1987. ACM.

[Bel58] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[Dij59] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[DM78] Y.K. Dalal and R.M. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 12:1040–148, 1978.

[Eve79] S. Even. *Graph Algorithms*. Computer Science Press, Rockville, MD, 1979.

[For56] Lester R. Ford. Network Flow Theory. *The RAND Corporation Paper P-923*, 1956.

[GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983.

[HKP+05] Juraj Hromkovic, Ralf Klasing, Andrzej Pelc, Peter Ruzicka, and Walter Unger. *Dissemination of Information in Communication Networks - Broadcasting, Gossiping, Leader Election, and Fault-Tolerance*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2005.

[Koe36] Denes Koenig. *Theorie der endlichen und unendlichen Graphen*. Teubner, Leipzig, 1936.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Pel00] David Peleg. *Distributed Computing: a Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.