4.2 Breadth-First Search

The next problem we consider is that of performing a breadth-first search (BFS) in a network based on an arbitrary strongly connected directed graph having a distinguished source node. More precisely, we consider how to establish a breadth-first spanning tree for the digraph. The motivation for constructing such a tree comes from the desire to have a convenient structure to use as a basis for broadcast communication. The BFS tree minimizes the maximum communication time from the process at the distinguished node to all other processes in the network (under the simplifying assumption that it takes the same amount of time for a message to traverse each communication channel).

The BFS problem and its solutions are somewhat simpler in the case where all pairs of neighbors have bidirectional communication, that is, where the network graph is undirected. We will indicate the simplifications for this case.

4.2.1 The Problem

We define a directed spanning tree of a directed graph $G = (V, E)$ to be a rooted tree that consists entirely of directed edges in $E$, all edges directed from parents to children in the tree, and that contains every vertex of $G$. A directed spanning tree of $G$ with root node $i$ is breadth-first provided that each node at distance $d$ from $i$ in $G$ appears at depth $d$ in the tree (that is, at distance $d$ from $i$ in the tree). Every strongly connected digraph has a breadth-first directed spanning tree.

For the BFS problem, we suppose that the network is strongly connected and that there is a distinguished source node $i_0$. The algorithm is supposed to output the structure of a breadth-first directed spanning tree of the network graph, rooted at $i_0$. The output should appear in a distributed fashion: each process other than $i_0$ should have a parent component that gets set to indicate the node that is its parent in the tree.

As usual, processes only communicate over directed edges. Processes are assumed to have UIDs but to have no knowledge of the size or diameter of the network.
4.2.2 A Basic Breadth-First Search Algorithm

The basic idea for this algorithm is the same as for the standard sequential breadth-first search algorithm. We call this algorithm \textit{SynchBFS}.

\textbf{SynchBFS algorithm:}

At any point during execution, there is some set of processes that is "marked," initially just \(i_0\). Process \(i_0\) sends out a \textit{search} message at round 1, to all of its outgoing neighbors. At any round, if an unmarked process receives a \textit{search} message, it marks itself and chooses one of the processes from which the \textit{search} has arrived as its parent. At the first round after a process gets marked, it sends a \textit{search} message to all of its outgoing neighbors.

It is not hard to see that the \textit{SynchBFS} algorithm produces a BFS tree. To show this formally, we can prove the invariant that after \(r\) rounds, every process at distance \(d\) from \(i_0\) in the graph, \(1 \leq d \leq r\), has its parent pointer defined; moreover, each such pointer points to a node at distance \(d - 1\) from \(i_0\). This invariant can, as usual, be proved by induction on the number of rounds.

\textbf{Complexity analysis.} The time complexity is at most \(diam\) rounds. (Actually, this analysis can be refined a little, to the maximum distance from the particular node \(i_0\) to any other node.) The number of messages is just \(|E|\)—a \textit{search} message is transmitted exactly once on each directed edge.

\textbf{Reducing the communication complexity.} As for the \textit{FloodMax} algorithm, it is possible to reduce the number of messages slightly: a newly marked process need not send a \textit{search} message in the direction of any process from which it has already received such a message.

\textbf{Message broadcast.} The \textit{SynchBFS} algorithm can easily be augmented to implement message broadcast. If a process has a message \(m\) that it wants to communicate to all of the processes in the network, it can simply initiate an execution of \textit{SynchBFS} with itself as the root, \textit{piggybacking} message \(m\) on the \textit{search} message it sends in round 1. Other processes continue to piggyback \(m\) on all their \textit{search} messages as well. Since the tree eventually spans all the nodes, message \(m\) is eventually delivered to all the processes.

\textbf{Child pointers.} In an important variant of the BFS problem, it is required that each process learn not only who its parent in the tree is, but also who all of its children are. In this case, it is necessary for each process receiving a \textit{search}
message to respond to that message with a parent or non-parent message, telling
the sender whether or not it has been chosen by the recipient as the parent.

If bidirectional communication is allowed between all pairs of neighbors, that
is, if the network graph is undirected, then there is no difficulty—and little extra
cost—in adding this extra communication. However, since we are allowing pairs
of neighbors with only unidirectional communication, some of the parent and
non-parent messages may need to be sent via indirect routes. For example, a
parent or non-parent message could be sent via a new execution of SynchBFS,
using piggybacking as above. In order for such a message to be recognized by
its intended recipient, the message should also carry the UID of the intended
recipient (plus a local name by which the recipient knows the sender), which
should therefore itself be piggybacked on the original search message. Note that
many executions of these SynchBFS "subroutines" can go on in parallel. In
order to fit our formal model, in which at most one message can be sent on each
link at each round, it may be necessary to combine many messages into one.

For a directed graph with unidirectional communication on some edges, in
addition to outputting parent and child pointers, it may also be useful to have
processes output information about the shortest routes from children to their
parents. Such information could be produced, for example, using additional
executions of SynchBFS.

Complexity analysis. If the graph is undirected, then the total time to com-
pute a BFS tree, including child pointers, is \( O(diam) \), and the communication
complexity is \( O(|E|) \).

Even if some of the pairs of neighbors have unidirectional communication,
the time to compute the tree plus child pointers is still only \( O(diam) \), because
the extra BFS executions can all go on in parallel. In this case, the total number
of messages is \( O(diam|E|) \), because at most \(|E|\) messages can be sent at each
of the \( O(diam) \) rounds. However, because a message might contain information
from up to \(|E|\) concurrent BFS executions, there might be as many as \(|E|b\) bits in
a message, where \( b \) is the maximum number of bits needed to represent a single
UID. This yields a total of \( O(diam|E|^2b) \) bits of communication. A smaller
bound on the total number of bits can be obtained by noting that each of the (at
most \(|E|\)) concurrent BFS executions uses at most \(|E|\) messages, each having at
most \( b \) bits. So the total number of communication bits is at most \( O(|E|^2b) \).

Termination. How can the source process \( i_0 \) tell when the construction of the
tree has been completed? If each search message is answered with either a parent
or non-parent message, then after any process has received responses for all of
its search messages, it knows who all its children in the BFS tree are and knows
that they have all been marked. So, starting from the leaves of the BFS tree, notification of completion can be "fanned in" to the source: each process can send notification of completion to its parent in the tree as soon as (a) it has received responses for all its search messages (so that it knows who its children are and knows that they have been marked), and (b) it has received notification of completion from all its children. This type of procedure is called a convergecast.

If the graph is undirected, then the total time to compute a BFS tree, including child pointers, and to propagate notification of completion back to the source is $O(diam)$ and the communication complexity is only $O(|E|)$. If unidirectional communication is allowed, then the total time, including notification of completion, is $O(diam^2)$. The reason the behavior is quadratic is that the notification has to proceed sequentially, one level at a time in the tree. The total number of messages is $O(diam^2|E|)$ and the total number of communication bits is at most $O(|E|^2 b)$.

### 4.2.3 Applications

Breadth-first search is one of the most basic building blocks for distributed algorithms. We give some examples here of how the SynchBFS algorithm can be used or augmented to help in performing other tasks.

**Broadcast.** As we mentioned earlier, a message broadcast can be implemented along with the establishment of a BFS tree. Another idea is first to produce a BFS tree with child pointers, as described above, and then to use the tree to conduct the broadcast. The message need only be propagated along edges from parents to their children. This allows the work of constructing the BFS tree to be reused, because many messages can be sent on the same tree. Once the BFS tree has been constructed, the additional time to broadcast a single message is only $O(diam)$, and the number of messages is only $O(n)$.

**Global computation.** Another application of BFS trees is the collection of information from throughout the network or, more generally, the computation of a function based on distributed inputs. For example, consider the problem in which each process has a nonnegative integer input value and we want to find the sum of all the inputs in the network. Using a BFS tree, this can be done easily (and efficiently) as follows. Starting from the leaves, "fan in" the results in a convergecast procedure, as follows. Each leaf sends its value to its parent; each parent waits until it gets the values from all its children, adds them to its own input value, and then sends the sum to its own parent. The sum calculated by the root of the BFS tree is the final answer.
4.3. **SHORTEST PATHS**

Assuming that the BFS tree has already been constructed, and assuming bidirectional communication on all tree edges, this scheme requires $O(diam)$ time and $O(n)$ messages. The same scheme can be used to compute many other functions, for example, the maximum or minimum of the integer inputs. (What is required is that the function be associative and commutative.)

**E lecting a leader.** Using SynchBFS, an algorithm can be designed to elect a leader in a network with UIDs, even when the processes have no knowledge of $n$ or $diam$. Namely, all the processes can initiate breadth-first searches in parallel. Each process $i$ uses the tree thereby constructed and the global computation procedure just described to determine the maximum UID of any process in the network. The process with the maximum UID then declares itself to be the leader, and all others announce that they are not the leader. If the graph is undirected, the time is $O(diam)$ and the number of messages is $O(diam|E|)$, again because at most $|E|$ messages can be sent at each of the $diam$ rounds. The number of bits is at most $O(n|E|b)$, where $b$ is the maximum number of bits used to represent a single UID.

**Computing the diameter.** The diameter of the network can be computed by having all processes initiate breadth-first searches in parallel. Each process $i$ uses the tree thereby constructed to determine $\text{max-dist}_i$, defined to be the maximum distance from $i$ to any other process in the network. Each process $i$ then reuses its breadth-first tree for a global computation to discover the maximum of the $\text{max-dist}$ values. If the graph is undirected, the time is $O(diam)$ and the number of messages is $O(diam|E|)$, the number of bits is $O(n|E|b)$. The diameter thus computed could be used, for example, in the leader-election algorithm FloodMax.

### 4.3 Shortest Paths

Now we examine a generalization of the BFS problem. Again, we consider a strongly connected directed graph, with the possibility of unidirectional communication between some pairs of neighbors. This time, however, we assume that each directed edge $e = (i,j)$ has an associated nonnegative real-valued $\text{weight}$, which we denote by $\text{weight}(e)$ or $\text{weight}_{i,j}$. The weight of a path is defined to be the sum of the weights on its edges. The problem is to find a shortest path from a distinguished source node $i_0$ in the digraph to each other node in the digraph, where a shortest path is defined to be a path with minimum weight.\(^2\) A collection of shortest paths from $i_0$ to all the other nodes in the digraph constitutes a subtree of the digraph, all of whose edges are oriented from parent to child.

\(^2\)The mixture of measures of weight and distance is unfortunate, but traditional.
As for breadth-first search, the motivation for constructing such a tree comes from the desire to have a convenient structure to use for broadcast communication. The weights represent costs that may be associated with the traversal of edges, for instance, communication delay or a monetary charge. A shortest paths tree minimizes the maximum worst-case cost of communicating with any process in the network.

We assume that every process initially knows the weight of all its incident edges, or, more precisely, that the weight of an edge appears in special weight variables at both its endpoint processes. We also assume that each process knows the number \( n \) of nodes in the digraph. We require that each process should determine its parent in a particular shortest paths tree, and also its distance (i.e., the total weight of its shortest path) from \( i_0 \).

If all edges are of equal weight, then a BFS tree is also a shortest paths tree. Thus, in this case, a trivial modification of the simple SynchBFS tree construction can be made to produce the distance information as well as the parent pointers.

The case where weights can be unequal is more interesting. One way to solve the problem is by the following algorithm—a version of the Bellman-Ford sequential shortest paths algorithm.

**BellmanFord algorithm:**

Each process \( i \) keeps track of \( dist \), the shortest distance from \( i_0 \) it knows so far, together with parent, the incoming neighbor that precedes \( i \) in a path whose weight is \( dist \). Initially, \( dist_{i_0} = 0 \), \( dist_i = \infty \) for \( i \neq i_0 \), and the parent components are undefined. At each round, each process sends its \( dist \) to all its outgoing neighbors. Then each process \( i \) updates its \( dist \) by a "relaxation step," in which it takes the minimum of its previous \( dist \) value and all the values \( dist_j + \text{weight}_{j,i} \), where \( j \) is an incoming neighbor. If \( dist \) is changed, the parent component is also updated accordingly. After \( n - 1 \) rounds, \( dist \) contains the shortest distance, and parent the parent in the shortest paths tree.

It is not hard to see that, after \( n - 1 \) rounds, the \( dist \) values converge to the correct distances. One way to argue the correctness of BellmanFord is to show (by induction on \( r \)) that the following is true after \( r \) rounds: Every process \( i \) has its \( dist \) and parent components corresponding to a shortest path among those paths from \( i_0 \) to \( i \) consisting of at most \( r \) edges. (If there are no such paths, then \( dist = \infty \) and parent is undefined.) We leave the details for an exercise.

**Complexity analysis.** The time complexity of the BellmanFord algorithm is \( n - 1 \), and the number of messages is \((n - 1)|E|\).