Chapter 14

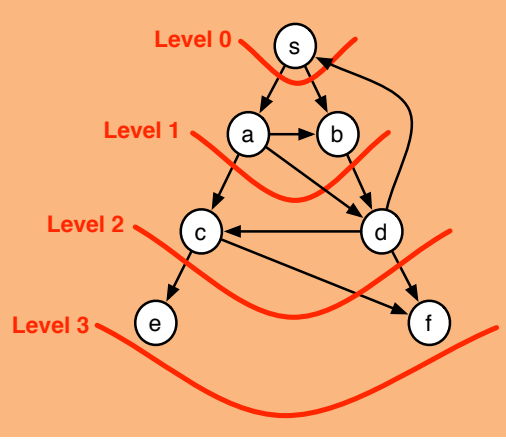
Breadth-First Search

The breadth-first algorithm is a particular graph-search algorithm that can be applied to solve a variety of problems such as finding all the vertices reachable from a given vertex, finding if an undirected graph is connected, finding (in an unweighted graph) the shortest path from a given vertex to all other vertices, determining if a graph is bipartite, bounding the diameter of an undirected graph, partitioning graphs, and as a subroutine for finding the maximum flow in a flow network (using Ford-Fulkerson's algorithm). As with the other graph searches, BFS can be applied to both directed and undirected graphs.

To understand how BFS operates, consider a graph $G = (V, E)$ and a source vertex $s \in V$. We define the *level* of a vertex $v \in V$ as the shortest distance from s to v, that is the number of edges on the shortest path connecting s to v.

Breadth first search (BFS) starts at the given *source* vertex s and explores the graph outward in all directions level by level, first visiting all vertices that are the (out-)neighbors of s (i.e. have distance 1 from s), then vertices that have distance two from s, then distance three, etc.

Example 14.1. *A graph and its levels illustrated. BFS visits the vertices in level 0, 1, 2, and 3 in that order.*



MICROSOFT CORP.
EXHIBIT 1023

BFS is a graph-search algorithm in a precise sense: it can be expressed as a specialization of the graph search algorithm (Algorithm 13.8). To understand the structure of BFS, it is instructive to go through the exercise of convincing ourselves that this is the case.

One way to show that BFS is an instance of graph-search is to specify the way we select the vertices to be visited next (the set U).

Question 14.2. *Can specify the selection of vertices to visit in Algorithm 13.8 to make sure that the vertices are visited in breadth-first order?*

We can do this by enriching the frontier with levels, for example, by tagging each vertex with its level, and selecting, at each round, the vertices at the current level to visit. But this would work only if we can make sure to insert all vertices "in time" to the frontier set, i.e., all vertices at level i should be inserted before we get to that level.

Question 14.3. *How can we determine the set of vertices at level i?*

Note that since the vertices at level i are all connected to vertices at level $i-1$ by one edge, we can find all the vertices at level i when we find the outgoing neighbors of level-$i-1$ vertices.

Question 14.4. *Is the set of vertices that are at level i equal to the vertices reachable from level-$i-1$ vertices by one edge?*

The outgoing neighbors may be at level i or less. But since we have visited all the vertices at level less than i, the unvisited vertices will be exactly the level-i vertices.

Exercise 14.5. *Based on the discussion, complete the specification of BFS and modify Algorithm 13.8 to perform BFS.*

Syntax 14.6. *We use the syntax*

```
(X_1, ..., X_k) =
   start (v_1, ..., v_k) and
   while e_c do
      e_w
```

as syntactic sugar for

```

function loop (X_1, ..., X_k) =
   if e_c then
      let e_w in loop (X_1, ..., X_k) end
   else
     (X_1, ..., X_k)

(X_1, ..., X_k) = loop (v_1, ..., v_k).
```

The BFS-algorithm can be described as a specialization of the graph-search algorithm (Algorithm 13.8) where we choose the whole frontier in each round. In BFS we may also want to keep around the number of levels we explore since this tells us the radius of the graph from the source s.

Algorithm 14.7 (BFS: reachability and radius)**.**

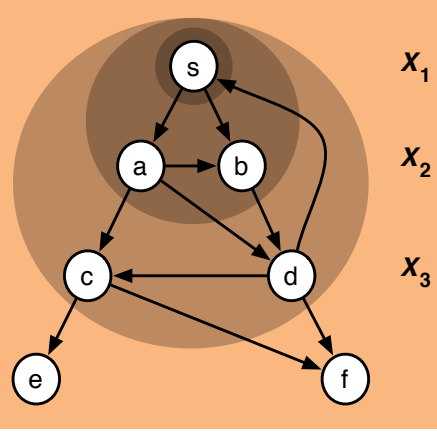
```
function BFSReach(G = (V, E), s) =
   let
      (X, F, i) =
         start ({}, {s}, 0 and
         while (|F| > 0)
            invariant: X = {u ∈ V | δ_G(s, u) < i} ∧
                       F = {u ∈ V | δ_G(s, u) = i}
            X = X ∪ F
            N = N_G^+(F)
            F = N \ X
            i = i + 1
   in (X, i) end
```

The algorithm maintains a visited set, a frontier, and the current level being visited. We refer to all the visited vertices at the start of level i as X_i. Since in round i we visit vertices at a distance i, and since we visit levels in increasing order, the vertices in X_i are exactly those with distance less than i from the source. At the start of level i the frontier F_i contains all unvisited neighbors of X_i, which is all vertices in the graph with distance exactly i from s. In each

round we visit all vertices in the frontier and mark newly visited vertices by simply adding the frontier to the visited set, i.e, $X_{i+1} = X_i \cup F_i$. To generate the next set of frontier vertices, the search takes the neighborhood of F and removes any vertices that have already been visited, i.e., $F_{i+1} = N_G(F) \setminus X_{i+1}$. Recall that for a vertex v, $N_G(v)$ are the neighbors of v in the graph G (the out-neighbors for a directed graph) and for a set of vertices F, that $N_G(F) = \bigcup_{v \in F} N_G(v)$.

As the algorithm visits vertices, it simply marks them visited. In general, a BFS-based algorithm can perform other actions depending on the specific application.

Example 14.8. *The figure below illustrates the BFS visit order by using overlapping circles from smaller no higher larger. Initially, X_0 is empty and F_0 is the single source vertex s, as it is the only vertex that is a distance 0 from s. X_1 is all the vertices that have distance less than 1 from s (just s), and F_1 contains those vertices that are on the middle ring, a distance exactly 1 from s. The outer ring contains vertices in F_2, which are a distance 2 from s. The neighbors $N_G(F_1)$ are the central vertex and those in F_2. Notice that vertices in a frontier can share the same neighbors, which is why $N_G(F)$ is defined as the* union *of neighbors of the vertices in F to avoid duplicate vertices.*



To prove that the algorithm is correct we need to prove the invariant that is stated in the algorithm. Recall that the notation $\delta(s, u)$ indicates the shortest path from s to u.

Lemma 14.9. *In algorithm BFS, when calling BFS$'(X, F, i)$, we have* $X = \{v \in V_G \mid \delta_G(s, v) < i\} \wedge F = \{v \in V_G \mid \delta_G(s, v) = i\}$

Proof. This can be proved by induction on the level i. For the base case (the initial call) we have $X_0 = \{\}$, $F_0 = \{s\}$ and $i = 0$. This is true since no vertex has distance less than 0 from s and only s has distance 0 from s. For the inductive step, we assume the claims are correct for i and want to show it for $i + 1$. For X_{i+1} we are simply taking the union of all vertices at distance less than i (X_i) and all vertices at distance exactly i (F_i). So this union must include exactly the vertices a distance less than $i + 1$. For F_{i+1} we are taking all neighbors of F_i and removing the X_{i+1}. Since all vertices F_i have distance i from s, by assumption, then a neighbor

v of F must have $\delta_G(s, v)$ of no more than $i+1$. Furthermore, all vertices of distance $i+1$ must be reachable from a vertex at distance i. Therefore, the neighbors of F_i contain all vertices of distance $i+1$ and only vertices of distance at most $i+1$. When removing X_{i+1} we are left with all vertices of distance $i+1$, as needed. □

To argue that the algorithm returns all reachable vertices, we note that if a vertex v is reachable from s and has distance $d = \delta(s, v)$ then there must be another vertex u with distance $\delta(s, u) = d - 1$. Therefore, BFS will not terminate without finding v. Furthermore, for any reachable vertex v, $\delta(s, v) < |V|$ so the algorithm will terminate in at most $|V|$ rounds (levels).

Exercise 14.10. *In general, from which frontiers could the vertices in $N_G(F_i)$ come when the graph is undirected? What if the graph is directed?*

14.1 Distances and BFS Trees

Thus far we have used BFS primarily for reachability. As a graph-search technique BFS can be used to compute other interesting properties of graphs. For example, we may want to compute the distance of each vertex from the source, or the shortest path from the source to some vertex, i.e., the sequence of vertices on the path. It is relatively straightforward to extend BFS for these purposes, for example by modifying the representation for the visited set and the frontier.

For example the following algorithm takes a graph and a source and a returns table mapping every reachable vertex v to $\delta_G(s, v)$. To compute the distances, the algorithm uses a table mapping each vertex to its distance, which is set as the level at the time that the vertex is visited.

Algorithm 14.11 (BFS-based Unweighted Shorted Paths)**.**

```
function BFSDistance(G, s) =
  let
    (X, F, i) =
      start ({}, {s}, 0 and
      while (|F| > 0)
        X = X ∪ {v ↦ i : v ∈ F}
        F = N⁺_G(F) \ domain(X)
        F = N \ X
        i = i + 1
  in (X, i) end
```

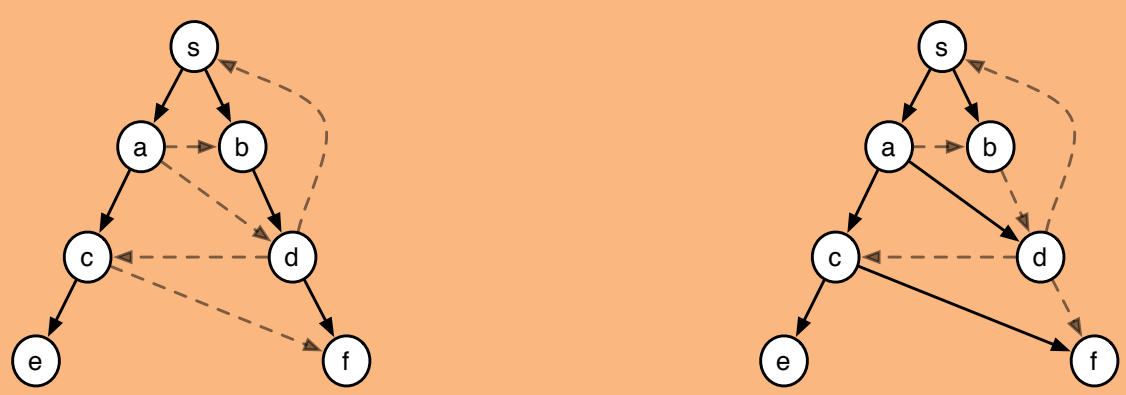
To compute the shortest paths (as for example as a sequence of vertices), we can generate a ***shortest path tree***, which contains the shortest path from the source to each vertex. Such a tree

can be represented as a table mapping each reachable vertex to its parent in the tree. Gives such a tree, we can then compute the shortest path to a particular vertex by walking in the tree from that vertex up to the root. Note that there could be more than one shortest-path trees. Indeed, Example **??** shows two possible trees that differ while still yielding the same shortest distances.

We can compute such a shortest-path tree using BFS, because in BFS, vertices are visited in their level order, which is the same as their distance to source. In particular, a shortest path to a vertex becomes complete when that the vertex is visited. To construct the tree, all we need to do is to determine a parent for each vertex when it is visited. We call such a tree, where the parent of a vertex v is a vertex u through which v is discovered, as a ***BFS Tree***. Note that (in unweighted graphs) a BFS Tree is the same as a shortest path tree.

To find a parent for each vertex, in addition to finding the next frontier $F' = N_G(F)/X'$ on each level, we also identify for each vertex in the frontier $v \in F'$, one vertex $u \in F$ such that $(u, v) \in E$. More specifically, we can represent the visited vertices X and the frontier F as tables that map vertices to their parents. At each round, we update visited table by merging the tables X and F. Finding the next frontier requires tagging each neighbor with its parent (via the edge leading to it) and then merging the results. That is, for each $v \in F$, we generate a table $\{u \mapsto v : u \in N(v)\}$ that maps each neighbor of v back to u. When merging tables, we have to decide how to break ties since vertices in the frontier might have the several (parent) neighbors; we can break such ties arbitrarily, for example by taking the first vertex.

Example 14.12. *An undirected graph and two possible BFS trees with distances from s. Non-tree edges, which are edges of the graph that are not on a shortest paths are indicated by dashed lines.*



Exercise 14.13. *Compute a BFS tree (shortest-distance tree) from the distance values returned by* `BFSDistance`

14.2 Cost of BFS

The cost of BFS depends on the particular representation that we choose for graphs. In this section, we consider two representations, one based on sets and tables, and another based on single-threaded sequences, which are more efficient because they support constant-work update operations.

When analyzing the cost of BFS with either representation, a natural method is to sum the work and span over the rounds of the algorithm, each of which correspond to a single iteration of the while loop. In contrast with recurrence based analysis, this approach makes the cost somewhat more concrete but can be made complicated by the fact that the cost per round depends on the structure of the graph. We overcome this difficulty by observing that BFS visits each vertex and traverses each edge no more than once, while visiting each vertex and edge reachable from the source exactly once.

14.2.1 Cost with BST-Sets and BST-Tables

Let's first analyze the cost per round. In each round, the only non-trivial work consists of the union $X' = X \cup F$, the calculation of neighbors $N = N_G(F)$, and the set difference $F' = N \setminus F$. The cost of these operations depends on the number of out-edges of the vertices in the frontier. Let's use $||F||$ to denote the number of out-edges for a frontier plus the size of the frontier, i.e., $||F|| = \sum_{v \in F}(1 + d_G^+(v))$. The costs for each round are then

	Work	Span
$X \cup F$	$O(\lvert F\rvert \log n)$	$O(\log n)$
$N_G(F)$	$O(\lvert\lvert F\rvert\rvert \log n)$	$O(\log^2 n)$
$N \setminus X'$	$O(\lvert\lvert F\rvert\rvert \log n)$	$O(\log n)$.

The first and last lines fall directly out of the tree-based cost specification for the set ADT. The second line is a bit more involved. The union of out-neighbors is implemented as

fun $N_G(F)$ = `Table.reduce Set.Union {} (Table.extract`(G, F))

Let $G_F =$ `Table.extract`(G, F). The work to find G_F is bounded by $O(|F| \log n)$. For the cost of the union, note that the set union results in a set whose size is no more than the sizes of the sets unioned. The total work per level of reduce is therefore no more than $||F||$. Since there are $O(\log n)$ such levels, the work is bound by

$$W(\texttt{reduce union}\ \{\}\ G_F) = O\left(\log |G_F| \sum_{v \mapsto N(v) \in G_F} (1 + |N(v)|)\right) = O\left(\log n \cdot ||F||\right)$$

and span is bounded by

$$S(\texttt{reduce union}\ \{\}\ G_T) = O(\log^2 n)$$

since each union has span $O(\log n)$ and the reduction tree is bounded by $\log n$ depth.

We can now calculate the span as the sum over all rounds, which gives us $O(d \log^2 n)$, where d is the depth of the shortest path tree (longest path in the tree) or equivalently the total number of rounds. For the work, we might want to sum of the work for all rounds, but unfortunately this does not lead to anything interesting because the work at each round depends on the frontier.

Let's focus an single round and note that cost per vertex and edge visited in that round is $O(\log n)$. Furthermore we know that every reachable vertex only appears in the frontier exactly once. Therefore, all the out-edges of a reachable vertex are also processed exactly once only. Thus the cost per edge W_e and per vertex W_v over the algorithm is the same as the cost per round. We thus conclude that $W_v = W_e = O(\log n)$.

Thus, we can write total work as $W = W_v n + W_e m$ (recall that $n = |V|$ and $m = |E|$).

We thus conclude that

$$\begin{aligned} W_{BFS}(n, m, d) &= O(n \log n + m \log n) \\ &= O(m \log n), \text{ and} \\ S_{BFS}(n, m, d) &= O(d \log^2 n). \end{aligned}$$

We drop the $n \log n$ term in the work since for BFS we cannot reach any more vertices than there are edges.

Notice that span depends on d. In the worst case $d \in O(n)$ and BFS is sequential. As we mentioned before, many real-world graphs are shallow, and BFS for these graphs has good parallelism.

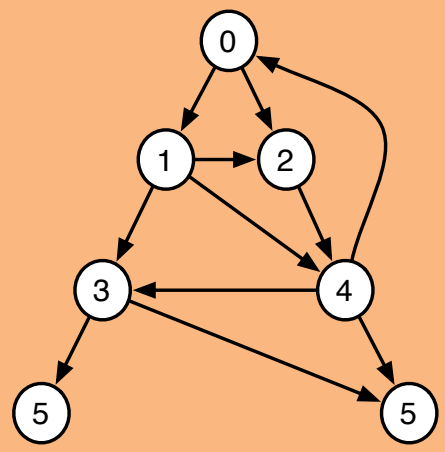
14.2.2 Cost with Single-Threaded Sequences

Consider a graph $G = (V, E)$ where $V = \{0, 1, \ldots, n-1\}$; that is the vertices are labeled starting from 0. We call such graphs *enumerated* graphs. As we shall show next, by using such a representation and by using single-threaded sequences to represent the visited set, we can give an $O(m)$-work and $O(d \log n)$-span implementation for BFS.

For an enumerated graph, we can use sequences as a graph representation by representing a graph as a sequence of sequences A, where the sequence at $A[i]$ is a sequence representing the outedges of vertex i. If the out-edges are ordered, we can order them accordingly; if not, we can choose an arbitrary order.

Example 14.14. *The enumerated graph below can be represented as*

$$\langle\langle 1,2\rangle,\langle 2,3,4\rangle,\langle 4\rangle,\langle 5,6\rangle,\langle 3,6\rangle,\langle\ \rangle,\langle\ \rangle\rangle.$$



This representation supports constant-work lookup operations for finding the out-edges (or out-neighbors) of a vertex. Since the graph does not change during BFS, this representation suffices for implementing BFS. In addition to performing lookups in the graph, the BFS algorithm also needs to determine whether a vertex is visited or not by using the visited set X. Unlike the graph, the visited set X changes during the course of the algorithm. We therefore use a single threaded (ST) sequence of length $|V|$ to mark which vertices have been visited. By using inject, we can mark vertices in constant work per update. For each vertex, we can use either a Boolean flag to indicate its status, or the label of the parent vertex (if any). The latter representation can help up construct a BFS tree.

The sequence-based BFS algorithm is shown below. For simplicity, we change the invariant a bit: on entering `BFS'` the sequence `XF` contains the parents for both the visited and the frontier vertices instead of just for the visited vertices. The frontier F is represented as an integer sequence containing the frontier.

Algorithm 14.15 (BFS Tree).

```
1  function BFSTree (G, s) =
2    let
3      function BFS (XF, F) =
4        if |F| = 0 then
5          stSeq.toSeq(XF)
6        else
7          let
8            N = flatten⟨⟨(u,v) : u ∈ G[v] | XF[u] = ⊥⟩ : v ∈ F⟩
9            XF' = stSeq.inject(N, XF)
10           val F' = ⟨u : (u,v) ∈ N | XF'[u] = v⟩
11         in BFS (XF', F') end
12     val X0 = stSeq.fromSeq(⟨⊥ : v ∈ ⟨0, ..., |G| − 1⟩⟩)
13   in
14     BFS (stSeq.update(s, s, X0), ⟨s⟩)
15   end
```

All the work is done in lines 8, 9, and 10. Also note that the `stSeq.inject` on line 9 is always applied to the most recent version. We can write out the following table of costs:

	XF : `stseq`		XF : `seq`	
line	work	span	work	span
8	$O(\|\|F_i\|\|)$	$O(\log n)$	$O(\|\|F_i\|\|)$	$O(\log n)$
9	$O(\|\|F_i\|\|)$	$O(1)$	$O(n)$	$O(1)$
10	$O(\|\|F_i\|\|)$	$O(\log n)$	$O(\|\|F_i\|\|)$	$O(\log n)$
total across all d rounds	$O(m)$	$O(d \log n)$	$O(m + nd)$	$O(d \log n)$

where d is the number of rounds (i.e. the shortest path length from s to the reachable vertex furthest from s). The last two columns indicate the costs if XF was implemented as a regular array sequence instead of an stSeq. The big difference is the cost of `inject`. As before the total work across all rounds is calculated by noting that every out-edge is only processed in one frontier, so $\sum_{i=0}^{d} ||F_i|| = m$.