Chapter 10

Graph Search

The term *graph search* or *graph traversal* refers to a class of algorithms that can be used to compute various properties of graphs. In this chapter, we will introduce the concept of a graph search, describe a generalized algorithm for it, and describe a particular specialization, called priority-first search, that still remains relatively general. In the next several chapters, we will consider further specializations of the general graph-search algorithm, specifically the breadth-first-search and the depth-first-search algorithms, each of which is particularly useful in computing certain properties of graphs.

To motivate graph search, let's first consider the kinds of properties that we might be interested in computing.

Question 10.1. *Can you remember some properties that we have discussed in the previous chapter?*

For example, in Chapter 9 we covered relations such as, reachability, and connectivity. Let's remind us about reachability and connectivity.

Definition 10.2 (Reachability)**.** *Given a graph $G = (V, E)$ and two vertices $u, v \in V$, a vertex v is reachable from another vertex u if there is a path from u to v. We also say sometimes that u can reach v.*

Definition 10.3 (Connectedness)**.** *A (directed) graph is* (strongly) connected *if there is a (directed) path between every pair of vertices.*

Question 10.4. *Is reachability a relation itself?*

Note that reachability is a relation itself and can thus be represented as a graph on the same set of vertices as the original graph—but with different edges. To compute such properties we can use algorithms that process the graph is some way.

Question 10.5. *Can you think of ways of determining whether a given graph satisfies a given property such as connectivity?*

The basic tool for computing such properties is to perform a *graph search* or a *graph traversal*.

10.1 Vertex Hopping

Before we start graph search, let's first consider a technique that we call *vertex hopping*. As we will see vertex hopping is not helpful in many cases but in many cases it offers a good starting point.

Algorithm 10.6 (Vertex hopping)**.** *Repeat until all vertices are visited:*

1. *select an (or some) unvisited vertex (vertices)*
2. *visit the selected vertex (vertices)*
3. *visit all the edges coming out of the selected vertex (vertices).*

Note that the algorithm as expressed is naturally parallel: we can visit many vertices at the same time. Depending on the property we are computing, however, this may not always be possible.

Question 10.7. *When the algorithm terminates, does it visit all the edges?*

Note that the algorithm visits all the vertices. Since any edge (u, v) comes out of a vertex u, it is visited when the algorithm visits u. Since the algorithm visits all the vertices, it will visit all the edges.

Question 10.8. *Given that when the algorithm terminates, it visits all the vertices and edges, can you solve the reachability problem using vertex hopping?*

We can solve the reachability problem by using vertex hopping. To do so, we keep a table mapping each visited vertex to the set of vertices reachable from it. Every time we visit a vertex u, we extend the table by finding all the vertices that can reach u extending their reach by adding v for every edge (u, v).

Question 10.9. *Is this algorithm efficient?*

Unfortunately, this algorithm is not efficient. In reachability, we are interested in finding out whether a particular vertex is reachable from another. This algorithm is building the full the reachability information for all vertices.

Question 10.10. *Can you see why vertex hopping is not effective for reachability computations? The algorithm is not taking advantage of something about graphs. Can you identify what that is and suggest a way to solve the reachability problem more efficiently?*

Since the nondeterministic algorithm jumps around the graph every time it visits a vertex, it does not reveal much information about the global structure of the graph. In particular, it does not follows paths in the graph. As we will see in later, vertex hopping, especially its parallel version, can be useful in many contexts but for problems such as reachability, the algorithm is not able to recover efficiently the path information.

10.2 Graph Search

For properties such as reachability, we use a different technique variously called *graph search* or *graph traversal* that imitates a walk (traversal) in the graph by following paths. Some graph search algorithms have parallel versions where multiple paths can be explored at the same time, some are sequential and explore only one path at a time.

Question 10.11. *Can you think of a way of refining vertex hopping to yield an algorithm for graph search?*

We can refine the vertex-hopping algorithm to perform a graph search by selecting the vertices to be visited more carefully. Specifally, what we need is a way to determine the next vertex to jump to in order to traverse paths rather than (possibly) unrelated vertices. To this end we can maintain a set of vertices that we call a *frontier* and visit only the vertices in the frontier. As the name implies, the frontier consists of vertices that are connected via an edge to visited vertices.

Definition 10.12. *A vertex v is in the frontier if v is not yet visited but it is connected via one edge (u, v) to a visited vertex u.*

We can then state graph search as follows.

Algorithm 10.13 (Graph Search 1/2)**.**
Given: a graph $G = (V, E)$ and a start vertex s.
Frontier $F = \{s\}$.
Visited set $X = \emptyset$.
While there are unvisited vertices
  Pick a set of vertices U in the frontier and visit them.
  Update the visited set $X = X \cup U$.
  Extend F with the out-edges of vertices in U.
  Delete visited vertices from F, $F = F \setminus X$.

Note that our algorithm violates our definition of frontier slightly when it starts with the start vertex in the frontier (start vertex is not one hop away from a visited vertex). We can update our definition to handle this corner case but for the sake of simplicity, we will ignore this slight inconsistency.

Question 10.14. *Can you explain why this algorithm follows paths rather than unrelated edges?*

The graph-search algorithm follows paths because it only visits vertices that are in the frontier, which are one hop away from other visited vertices. Note that this does not mean that the algorithm always follows a single path as far is it can and then switches to another path. Rather, the algorithm can follow multiple paths simultaneously or in an interleaved fashion. (We refer to algorithms such as graph search as non-deterministic.)

Question 10.15. *How can a graph-search algorithm can determine that all vertices are visited?*

The graph search algorithm can determine the time to terminate by checking that the number of visited vertices is equal to the number of vertices in the graph (the cardinality of the visited set). Another simpler way is to check that the frontier is empty. But that works, only when all vertices are reachable from the source. If some are not, then the frontier can become empty even when unvisited vertices remain. We will check for termination by checking that the frontier becomes empty. In the case unvisited vertices remain, graph search can be invoked on such vertices, until all are visited.

Algorithm 10.16 (Graph Search 2/2)**.**
Given: a graph $G = (V, E)$ and a start vertex s.
Frontier $F = \{s\}$.
Visited set $X = \emptyset$.
While the frontier is not empty
Pick a set of vertices U in the frontier and visit them.
Update the visited set $X = X \cup U$.
Extend F with the out-edges of vertices in U.
Delete visited vertices from F, $F = F \setminus X$.

10.3 Priority First Search

The graph search algorithm that we described does not specify the vertices to visit next (the set U). This is intentional, because graph search algorithms such as breadth-first search and depth-first search, differ exactly on which vertices they visit next. While graph search algorithms differ on this point, many are also quite similar: there is often a method to their madness, which we will desribe now.

Consider a graph search algorithm that assigns a priority to every vertex in the frontier. You can imagine such an algorithm giving a priority to a vertex v when it inserts v into the frontier. Now instead of picking some unspecified subset of the frontier to visit next, the algorithm picks, the algorithm visits the highest (or the lowest) priority vertices. We refer to such an algorithm as a *priority first search* or a *best-first search*. The priority order can either be determined statically (a priori) or it can be generated on the fly by the algorithm.

Algorithm 10.17 (Priority Search)**.**
Given: a graph $G = (V, E)$ and a start vertex s.
Frontier $F = \{s\}$.
Visited set $X = \emptyset$.
While the frontier is not empty
Select the set of vertices $U \subseteq F$ with highest priority and visit them.
Update the visited set $X = X \cup U$.
Extend F with the out-edges of vertices in U.
Delete visited vertices from F, $F = F \setminus X$.

Question 10.18. *Can you imagine how you can use a priority search to explore the web from a start page so that the web sites that are of higher importance to you are visited first.*

You can use priority search to explore the web in a way that gives priority to the sites that you are interested in. The idea would to implement the frontier as a priority queue data structure containing the unvisited outgoing links based on your interest on the link (which we assume to be known). You can then choose what page to visit next by simply selecting the link with the highest priorirty. After visiting the page, you would remove it from the priority queue and add all its unvisited outgoing links to the list with their corresponding priorities.

Priority First Search is a greedy technique since it greedily selects among the choices in front of it (the frontier) based on some cost function (the priorities) and never backs up. Algorithms based on PFS are hence often referred to as greedy algorithms. As you will see soon, several famous graph algorithms are instances of priority first search. For example, Dijkstra's algorithm for finding single-source shortest paths, Prim's algorithm for finding Minimum Spanning Trees.

Chapter 11

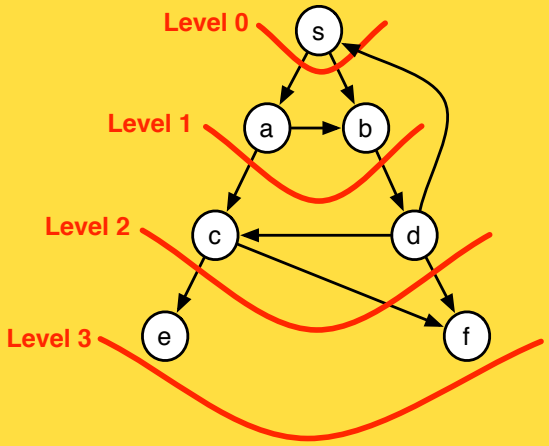
Breadth-First Search

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.

11.1 The BFS Algorithm

The idea of *breadth first search* is to start at a *source* vertex s and explore 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. More precisely, suppose that we are given a graph G and a source s. We define the *level* of a vertex v as the shortest distance from s to v, that is the number of edges on the shortest path connecting s to v. Example 11.1 below shows an example.

Example 11.1. *A graph and its levels.*



BFS is a graph-search algorithm in a precise sense: it can be expressed as a specialization of the graph search algorithm (Algorithm 10.16). Let's see how we can do this.

Question 11.2. *Can you think of a way of modifying the graph-search algorithm to make sure that the vertices are visited in breadth-first order?*

One way would be to enrich the frontier with levels, for example, by tagging each vertex with its level and visit vertices in level order. This would help only if we can make sure to insert all vertices "in time" to the frontier set.

Question 11.3. *How can we determine the set of vertices at level $i+1$?*

Since the vertices at level $i+1$ are all connected to vertices at level i by one edge, we can find all the vertices at level $i+1$ when we find the outgoing edges of level-i vertices.

Question 11.4. *Is the set of vertices that are at level $i+1$ equal to the vertices reachable from level-i vertices by one hop?*

Not all vertices reachable from level-i vertices have level $(i+1)$, because some vertices may be at levels i or less (but not at level $i+2$ or greater).

Question 11.5. *Can you see when this happens in the example in Example 11.1?*

For instance, in Example 11.1, the edges (a, b) and (d, s) lead to vertices at earlier levels.

Question 11.6. *How can we find and eliminate such vertices?*

Such vertices are easy to identify when visiting vertices in increasing levels: they are already visited. Thus if we exclude visited vertices, then vertices at level $i+1$ are exactly those that are reachable from level-i vertices by one hop.

The BFS-algorithm can thus be described as a specialization of the graph-search algorithm (Algorithm 10.16) as follows.

Algorithm 11.7 (BFS 1/2)**.**
Given: a graph $G = (V, E)$ and a start vertex s.
Frontier $F = \{(s, 0)\}$.
Visited set $X = \emptyset$.
Current level $i = 0$
While $F \neq \emptyset$ do
Pick the vertices $U \subseteq$ at the level i and visit them.
Update the visited set $X = X \cup U$.
Extend F with the out-edges of vertices in U: $F = F \cup \{(v, i + 1) : v \in$ out-neighbors of $U\}$
Delete visited vertices from F.
$i = i + 1$.

Question 11.8. *We can simplify this algorithm a bit. Can you see how?*

We can simplify the algorithm by observing that vertices added to the frontier at each level are at the next level, after we delete all the visited vertices. Since we are visiting all the vertices at the same level in parallel, the frontier will contain only those vertices at the next level. The breadth first algorithm can thus be simplified by throwing away the level information.

Algorithm 11.9 (BFS 2/2)**.**
Given: a graph $G = (V, E)$ and a start vertex s.
Frontier $F = \{s\}$.
Visited set $X = \emptyset$.
While $F \neq \emptyset$ do
Visit all the vertices in the frontier.
Update the visited set $X = X \cup F$.
Set F to be the out-neighbors of vertices in F.
Delete visited vertices, $F = F \setminus X$.

Based on this discussion, we can express `BFS` with the following pseudo code. The algorithm returns the set of vertices reachable from a vertex s as well as the shortest distance to the furthest reachable vertex, i.e., the radius of the graph from s. We define $\delta_G(s, u)$ to be the shortest distance from vertex s to vertex u, and $R_G(s)$ the set of vertices reachable from s in the graph G.

Pseudo Code 11.10.

```
1  function reachabilityAndRadius(G = (V, E), s) =
2    let
3      % requires: X = {u ∈ V | δ_G(s, u) < i} ∧
4                  F = {u ∈ V | δ_G(s, u) = i}
5      % returns: (R_G(s), max{δ_G(s, u) : u ∈ R_G(s)})
6      function BFS(X, F, i) =
7        if |F| = 0 then (X, i)
8        else
9          let
10           X' = X ∪ F      (* Visit the Frontier *)
11           N = N_G(F)      (* Determine the neighbors of the frontier *)
12           F' = N \ X'     (* Remove vertices that have been visited *)
13         in BFS(X', F', i + 1) end    (* Next level *)
14   in BFS({}, {s}, 0) end
```

If we are using a adjacency table representation of the graph, we can find $N_G(F)$, all the neighbors of the frontier F, as

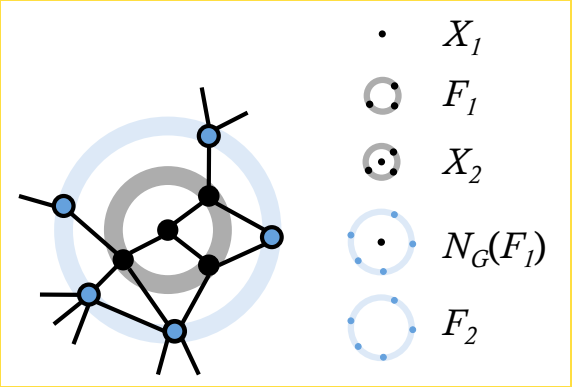
```
1 function N_G(F) = Table.reduce Set.Union {} (Table.extract(G, F))
```

The full SML code for the algorithm is given in the appendix at the end of these notes.

We refer to all the visited vertices at the start of level i as X_i. Since on level i we visit vertices at a distance i away, the vertices in X_i are exactly those with distance less than i from the source. On each level the search also maintains a frontier. 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. On each level we visit all vertices in the frontier. This differs from the depth-first-search algorithm, DFS, (Chapter 12), which only visits one. What we do when we visit depends on the particular application of BFS. But for now we assume we simply mark the vertices as visited. We mark newly visited vertices by simply adding the frontier to the previously visited vertices, 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 $N_G(F) = \bigcup_{v \in F} N_G(v)$.

Example 11.11 illustrates `BFS` on an undirected graph where s is the central vertex. 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 inner concentric ring, a distance exactly 1 from s. The outer concentric 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 some vertices in F_1 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. For the graph in the figure, which vertices are in X_2?

Example 11.11. *BFS on an undirected graph with the source vertex at the center.*



Exercise 11.12. *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?*

To prove that the algorithm is correct we need to prove the assumptions that are stated in the algorithm. In particular:

Lemma 11.13. *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).

11.2 BFS Cost

So far in the class we have mostly calculated costs using recurrences. This works well for divide-and-conquer algorithms, but, as we will see, most graph algorithms do not use divide-and-conquer. Instead, for many graph algorithms we can calculate costs by counting, i.e., adding the costs across a sequence of rounds of an algorithm. Different rounds can take a different amount of work or span.

Since `BFS` works in a sequence of rounds, one per level, we can add up the work and span across the levels. The problem, however, is that the work done at each level varies, since it depends on the size of the frontier at that level—in fact it depends on the number of outgoing edges from the frontier for that level. What we do know, however, is that every reachable vertex only appears in the frontier exactly once. Therefore, all their out-edges are processed exactly once only. If we can calculate the cost per edge W_e and per vertex W_v regardless of the level, then we can simply multiply these by the number of edges and vertices giving $W = W_v n + W_e m$ (recall that $n = |V|$ and $m = |E|$). For the span we can determine the largest span per level S_l and multiply it by the number of levels $d = \max_{v \in V} \delta(s, v)$, giving $S = S_l d$,.

If we use the tree representation of sets and tables, we can show that the work per edge and per vertex is bounded by $O(\log n)$ and the span per level is bounded by $O(\log^2 n)$. Therefore we have:

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

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

Now let's show that the work per vertex and edge is $O(\log n)$. We can examine the code and consider what is done on each level. In particular the only non-trivial work done on each level is 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 will depend on the size of the frontier, and in fact in the number of out-edges from the frontier. We will 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 level are as follows

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

The first and last lines fall directly out of the cost spec for the set interface. The second line is a bit more involved. Recall that it is implemented as

function $N_G(F) = \texttt{Table.reduce Set.Union } \{\} \ (\texttt{Table.extract}(G, F))$

Let $G_F = \texttt{Table.extract}(G, F)$. The work to find G_F is bounded by $O(|F| \log n)$. For the cost of the union we can use Lemma 2.1 from lecture 6. In particular, union satisfies the conditions of the Lemma. Therefore, 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.

Now we see that work per vertex and edge is $O(\log n)$, since each vertex and its out-edges appear on only one frontier and on level i we process $||F_i||$ vertices and their out-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.

11.3 BFS Extensions

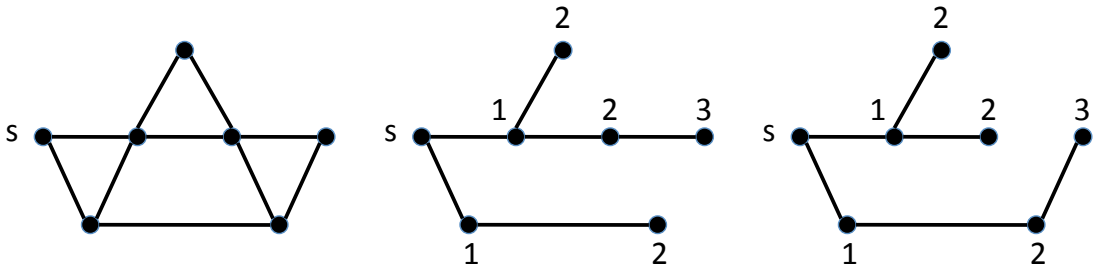
So far we have specified an algorithm that returns the set of vertices reachable from s and the longest length of all shortest paths to these vertices. Often we would like to know more, such as the distance of each vertex from s, or the shortest path from s to some vertex v, i.e., the actual sequence of vertices in the path. It is easy to extend BFS for these purposes. For example the following algorithm returns a table mapping every reachable vertex v to $\delta_G(s, v)$.

Pseudo Code 11.14.

```
1  function distanceFromSource(G, s) =
2    let
3      function BFS(X, F, i) =
4        if |F| = 0 then
5          X
6        else
7          let
8            X' = X ∪ {v ↦ i : v ∈ F}
9            F' = N_G(F) \ domain(X')
10         in BFS(X', F', i + 1) end
11   in BFS({}, {s}, 0) end
```

To report the actual shortest paths, we can generate a shortest path tree, which can be represented as a table mapping each reachable vertex to its parent in the tree. We can then report the shortest path to a particular vertex by following from that vertex up the tree to the root.

Example 11.15. *An undirected graph and two possible BFS trees with distances from s.*



We note that to generate the pointers to parents requires that we not only find the next frontier $F' = N_G(F)/X'$ of the frontier on each level, but that we identify for each vertex $v \in F'$ one vertex $u \in F$ such that $(u, v) \in E$. There could be multiple such edges to the vertex v. Indeed, Example **??** shows two possible trees that differ in what the parent is for the vertex at distance 3.

There are various ways to identify the parent. One is to post-process the result of the BFS that returns the distance. In particular, for every vertex we can pick one of its (in-)neighbors with a distance one less than itself. Another way is to identify the parent when generating the neighbors of F. The ideas is that both the visited vertices X and the frontier F are tables that map each of its vertices to a parent vertex. Finding the next visited table is easy since it just merges two tables X and F. Finding the next frontier requires tagging each neighbor from where it came and then merging the results. That is, for each $v \in F$, it generates a table $\{u \mapsto v : u \in N(v)\}$ that maps each neighbor of v back to v. When merging these tables, it has decide how to break ties since vertices in the frontier might have the several (parent) neighbors. The code takes the first vertex.

11.4 BFS with Single Threaded Sequences

Here we consider a version of BFS that uses sequences instead of sets and tables. The advantage is that it runs in $O(m)$ total work and $O(d \log n)$ span.

We refer to a graph $G = (V, E)$ where $V = \{0, 1, \ldots, n-1\}$ as an *integer labeled* (IL) graph. For an IL graph we can use the sequences to represent a graph. Each vertex identifier is an integer and the data for the vertex is stored at that index in the sequence. In this way, if the sequence is array-based, looking up a vertex is only constant work. At each index we store the neighbors of the vertex, where the neighbors can be represented as an array sequence containing

the neighbors' integer identifiers. An IL graph can therefore be implemented with type:

```
(int seq) seq.
```

Because the graph does not change during BFS, this representation is efficient.

The set of visited vertices X, however, does change during the course of the algorithm. Therefore, we 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. We can use either a Boolean to indicated whether a vertex has been visited on not, or an `(int option)` if we want to map each vertex to its parent, which will return the so-called BFS-tree. In the BFS-tree the option *NONE* indicates the vertex has not been visited, and *SOME*(v) indicates it has been visited and its parent is v. Each time we visit a vertex, we map it to its parent in the BFS tree. As the updates to this sequence are potentially small compared to its length, using an `stseq` is efficient. On the other hand, because the set of frontier vertices is new at each level, we can represent the frontier simply as an integer sequence containing all the vertices in the frontier, allowing for duplicates.

To simplify the algorithm we change the invariant a bit. In particular on entering *BFS'* the sequence *XF* contains parent pointers for both the visited and the frontier vertices instead of just for the visited vertices. F is an integer sequence containing the frontier.

Pseudo Code 11.16.

```
1  function BFSTree(G : (int seq) seq, s : int) =
2    let
3      function BFS(XF : (int option) stseq, F : int seq) =
4        if |F| = 0 then
5          stSeq.toSeq(XF)
6        else
7          let
8            N = flatten⟨ ⟨ (u, SOME(v)) : u ∈ G[v] ⟩ : v ∈ F ⟩
9            XF' = stSeq.inject(N, XF)
10           F' = ⟨ u : (u, v) ∈ N | XF'[u] = v ⟩
11         in BFS(XF', F') end
12     val X0 = stSeq.fromSeq(⟨ NONE : v ∈ ⟨ 0, ..., |G| − 1 ⟩ ⟩)
13   in
14     BFS(stSeq.update(s, SOME(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
6	$O(\|\|F_i\|\|)$	$O(\log n)$	$O(\|\|F_i\|\|)$	$O(\log n)$
7	$O(\|\|F_i\|\|)$	$O(1)$	$O(n)$	$O(1)$
8	$O(\|\|F_i\|\|)$	$O(\log n)$	$O(\|\|F_i\|\|)$	$O(\log n)$
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 $\sum_{i=0}^{d} ||F_i|| = m$.

Chapter 12

Depth-First Search

The BFS algorithm can be used to solve many interesting problems on graphs.

Question 12.1. *Do you think that the BFS algorithms sufficient to compute all properties of graphs?*

While BFS is effective for many problems but another algorithm called *depth-first search* of *DFS* for short, can be more natural in other problems such as topological sorting, cycle detection, and the finding connected components of a graph. In this lecture, we will consider topological sorting and cycle detection.

12.1 Topological Sort

As as example, we will consider the work that a rock climber must do before starting a climb in order to protect herself in case of a fall. For simplicity, we will consider only a the task of wearing a harness and tying into the rope.

Figure 12.2 illustrates the actions that a climber must take in order to secure herself with a rope and the dependencies between them. Performing each task and observing the dependencies in this graph is crucial for safety of the climber and any mistake puts the climber as well as her belayer and other climbers into serious danger. While instructions are clear, errors in following them abound.

Question 12.2. *There is something interesting about the structure of this graph. Can you see something missing in this graph compared to a general graph?*

This directed graph has no cycles, which is natural because this is a dependency graph and you would not want to have cycles in your dependency graph. We call such graphs directed-acyclic graph or DAG for short.

Figure 12.1: Before starting, climbers must carefully put on gear that protects them in a fall.

Definition 12.3 (Directed Acylic Graph (DAG))**.** *A directed acyclic graph is a directed graph with no cycles.*

Since a climber can only perform one of these tasks at a time, her actions are naturally ordered. We call a total ordering of the vertices of a DAG that respects all dependencies a topological sort.

Definition 12.4 (Topological Sort of a DAG)**.** *The topological sort a DAG (V, E) is a total ordering, $v_1 < v_2 \ldots < v_n$ of the vertices in V such that for any edge $(v_i, v_j) \in E$, if $j > i$. In other words, if we interpret edges of the DAG as dependencies, then the topological sort respects all dependencies.*

Question 12.5. *Can you come up with a topological ordering of the climbing DAG shown in Figure 12.2.*

There are many possible topological orderings for the DAG in Figure 12.2. For example, following the tasks in alphabetical order gives us a topological sort.

For climbing, this is not a good order because it has too many switches between the harness and the rope. To minimize errors, the climber will prefer to put on the harness first (tasks B,

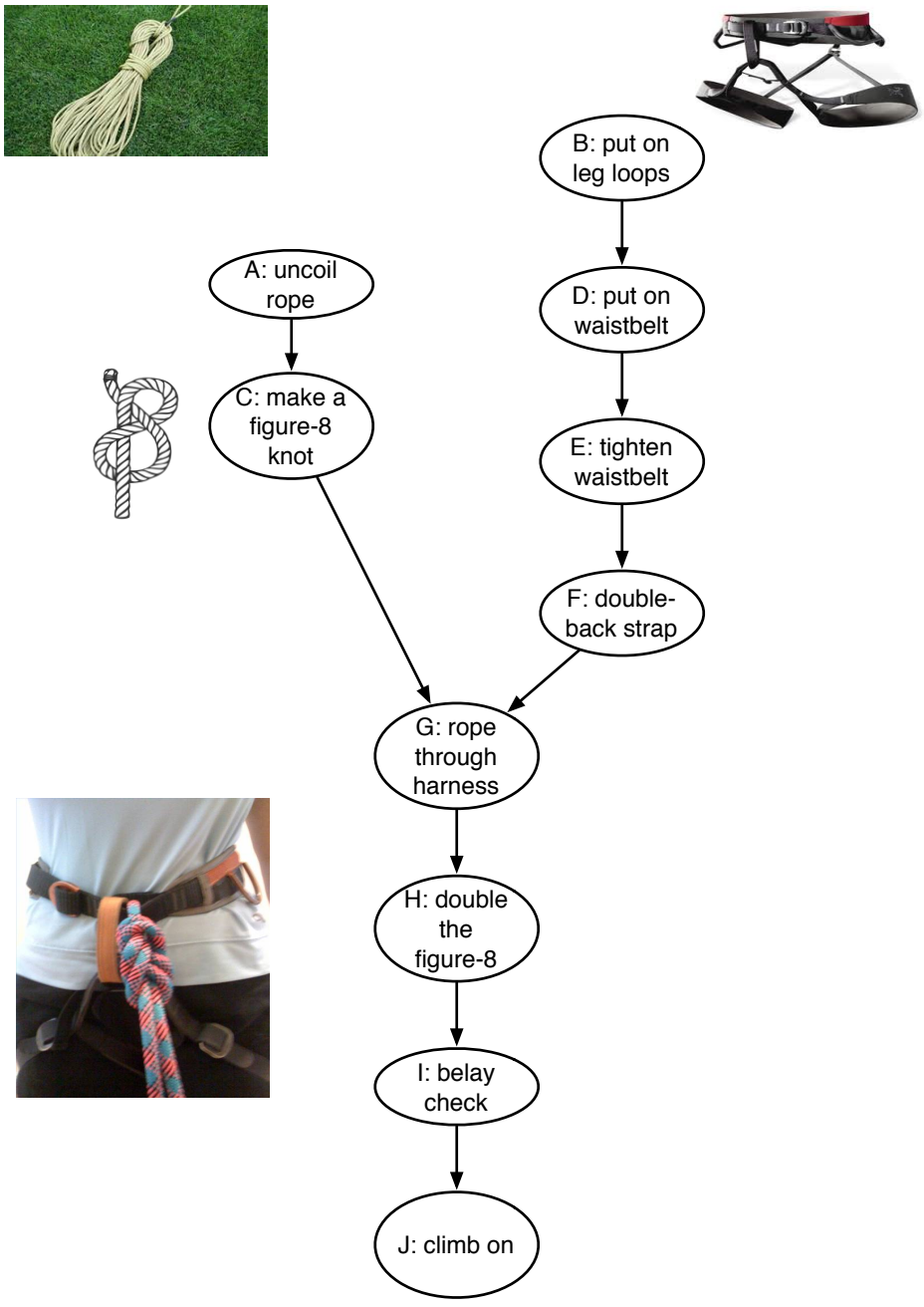


Figure 12.2: A simplified DAG for tying into a rope with a harness.

D, E, F in that order) and then prepare the rope (tasks A and then C), and finally rope through, complete the knot, get her gear checked by her climbing partner, and climb on (tasks G, H, I, J, in that order).

When considering the topological sort of a graph, it is often helpful to insert a "start" vertex and connect it all the other vertices.

To make thinking about this problem easier it is helpful to add a start vertex to the DAG and create a dependency from a start vertex to every other vertex as shown in Figure 12.3.

Question 12.6. *Would this change the set of valid topological sortings for the DAG?*

Adding a start vertex does not change the set of topological sorts. Since all new edges originate at the start vertex, any valid topological sort of the original DAG can be converted into a valid topological sort of the new dag by preceding it with the start vertex.

Question 12.7. *Can you think of an algorithm for finding such a topological sort by using the vertex-hopping algorithm discussed in Chapter 10?*

We can use the vertex-hopping algorithm discussed in Chapter 10 to find a topological sort of DAG. To do this, we have to make sure that a vertex is not visited before all its parents are visited.

Question 12.8. *Can you think of an algorithm for selecting the next vertex to visit?*

We can select the next vertex to visit in several different ways. One possibility is to maintain a priority queue of vertices where each vertex is assigned as priority the number of unvisited parents it has. We will soon see that the DFS algorithm achieves this without using an additional data structure such as a priority queue.

Question 12.9. *Can you see why BFS is not as natural for topological sorting?*

Before we move onto DFS, note that BFS visits a vertex in the quickest possible way, because it visits vertices in level order (shortest distance from source order). Thus in our example, BFS would ask the climber to rope through the harness (task G) before fully putting on the harness.

12.2 DFS: Depth-First Search

Recall that when we search a graph, we have the freedom to pick a vertex in the frontier and visit that vertex. The DFS algorithm is a specialization of graph search that picks the vertex that is most recently added to the frontier, of course making sure that in never revisits visited vertices. For this (the most recently added vertex) to be well-defined, in DFS we insert the out-neighbors of the visited vertex to the frontier in a pre-defined order (e.g., left to right).

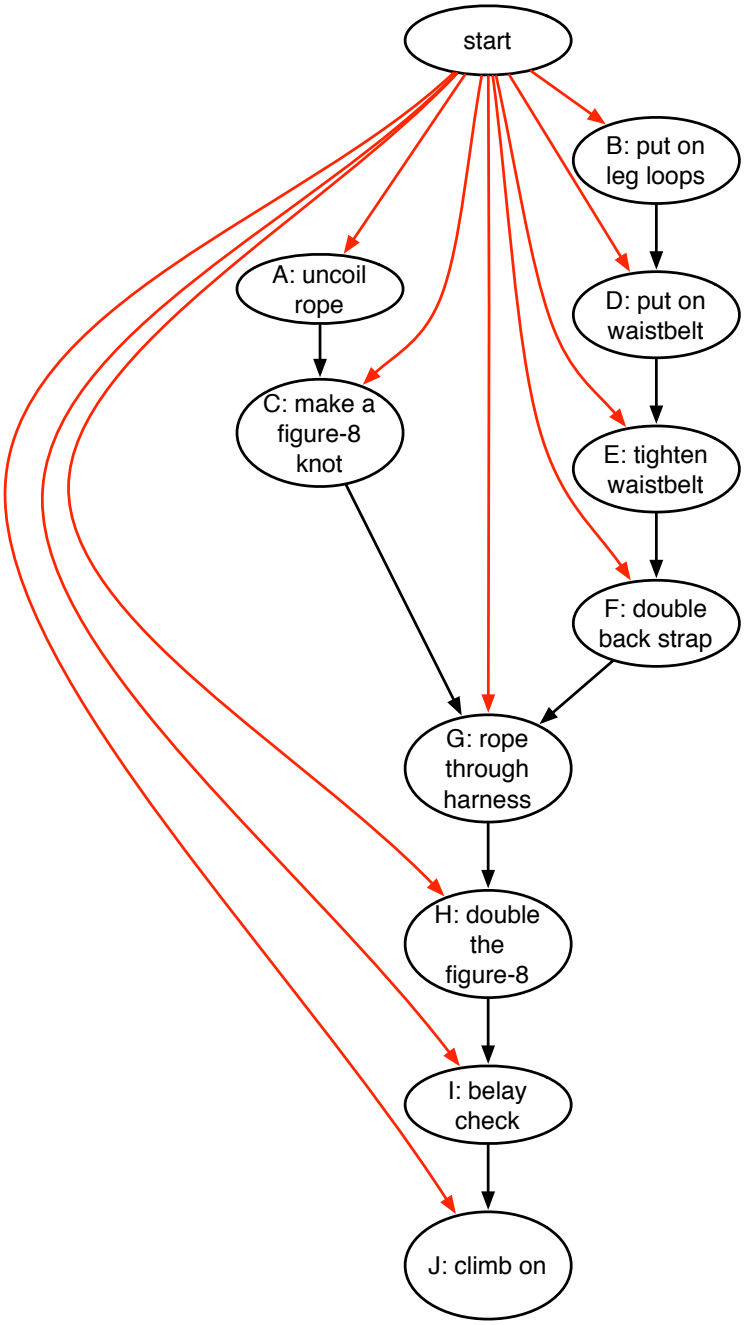


Figure 12.3: Climber's DAG with a start node.

Question 12.10. *Can you come up with an algorithm for determining the next vertex to visit?*

One straightforward way to determine the next vertex to visit is to time-stamp the vertices as we add to the frontier and simply remove the most recently inserted vertex. Maintaining the frontier as a sequence data structure that maintains the vertices in the frontier in an stack order (LIFO: last in first out), achieves the same effect without having to resort to time stamps.

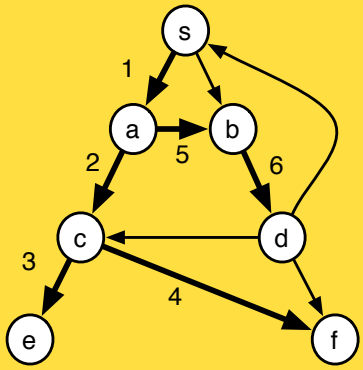
Algorithm 12.11 (Graph Search 2/2)**.**
Given: a graph $G = (V, E)$ *and a start vertex* s*.*
Frontier $F = \langle s \rangle$*.*
Visited set $X = \emptyset$*.*
While the frontier is not empty
  Let $F = \langle v_o, v_1, \ldots, v_m \rangle$*.*
  Visit v_o
  $X = X \cup \{v_o\}$*.*
  $\{u_0, \ldots, u_k\} \in (N_G^+(v_0) \setminus X)$
  $F = \langle u_0, \ldots, u_k, v_1, \ldots, v_m \rangle$

As an intuitive way to think about DFS, think of curiously browsing photos of your friends in a photo sharing site. You start with the site of a friends. You then realize that the some other people are tagged and visit their site. So that you can remember the people that you have visited, you mark their names with a proverbial white pebble. You then see other people tagged and visit their web site. You of course are careful not to revisit people's sites that you have already visited (as in DFS). When you finally reach a site that contains no new people, you start pressing the back button until you find someone that you have visited. By tracking your way back you are essentially searching for the link to the most recently seen (unvisited) person that you have seen tagged. As you back track, you will encounter sites that contain no links to new (unvisited) people. You mark their names with a red pebble.

Exercise 12.12. *Convince yourself that the pebbling-based description and the DFS algorithm described above are identical.*

Figure 12.4: The idea of pebbling is old. In Hänsel and Gretel, one of the folk tales collected by Grimm brothers in early 1800's, the protagonists use (white) pebbles to find their way home when left in the middle of a forest by their struggling parents. In a later adventure, they use bread crumbs instead of pebbles for the same purpose (but the birds eat the crumbs, leaving their algorithm ineffective). They later outfox a witch and take possession of all her wealth, with which they live happily ever after. Tales such as Hänsel and Gretel were intended to help with the (then fear-based) disciplining of children. Consider the ending tough—so much for scare tactics.

Example 12.13. *An example DFS.*



Question 12.14. *How does the order in which DFS visits vertices in this example differ from that of BFS?*

DFS and BFS visits vertices in very different orders: DFS does not visit vertices in order of their levels.

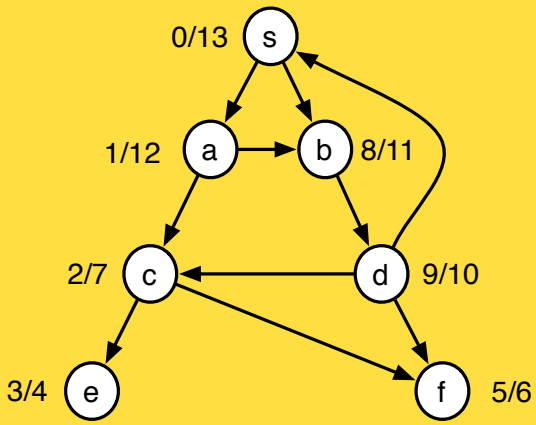
Question 12.15. *Can you think of why DFS might be more effective in solving some problems that BFS cannot.*

DFS can be more effective in solving some problems because as BFS traverses the graph sequentially, it can collect more information about the graph. At any point, it is possible know everything about the vertices and edges that have been visited thus far.

12.3 DFS Numbers

In a DFS, we can assign two timestamps to each vertex that show when the vertex receives its white and red pebble. The time at which a vertex receives its white pebble is called the *discovery time* or *enter time*. The time at which a vertex receives its red pebble is called *finishing time* or *exit time*. We refer to the timestamps cumulatively as *DFS numbers*.

Example 12.16. *A graph and its DFS numbers illustrated; t_1/t_2 denotes the timestamps showing when the vertex gets its white (discovered) and red pebble (finished) respectively.*



DFS numbers have some interesting properties.

Exercise 12.17. *Can you determine by just using the DFS numbers of a graph whether a vertex is an ancestor or a descendant of another vertex?*

Question 12.18. *Can you solve give a topological sort of the graph using the DFS numbers.*

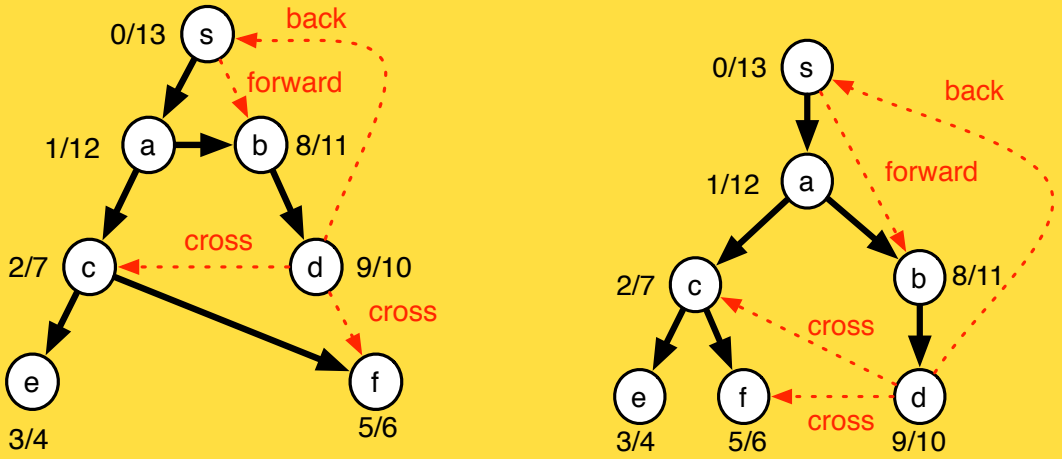
There is an interesting connection between DFS numbers and topological sort: in a DAG, the sort of the vertices from the largest to the smallest finishing time yield a topological ordering of the DAG.

Tree edges, back edges, forward edges, and cross edges. Given a graph and a DFS of the graph, we can classifies the edges of the graph into various categories.

Definition 12.19. *We call an edge (u, v) a* tree edge *if v receives its white pebble when the edge (u, v) was traversed. Tree edges define the* DFS tree.
The rest of the edges in the graph, which are non-tree edges, can further be classified as back edges, forward edges, and cross edges.

- *A non-tree edge (u, v) is a* back edge *if v is an ancestor of u in the DFS tree.*
- *A non-tree edge (u, v) is a* forward edge *if v is a descendant of u in the DFS tree.*
- *A non-tree edge (u, v) is a* cross edge *if v is neither an ancestor nor a descendant of u in the DFS tree.*

Example 12.20. *Tree edges (black), and non-tree edges (red, dashed) illustrated with the original graph and drawn as a tree.*



Question 12.21. *Can you think of a way to implement pebbling efficiently?*

Pseudocode for DFS. It turns out to be relatively straightforward to implement DFS by using nothing more than recursion. For simplicity, let's consider a version of DFS that simply returns a set of reachable vertices, as we did with BFS.

Pseudo Code 12.22 (DFS)**.**

```
1  function reachability(G, s) =
2    let
3      function DFS(X, v) =
4        if (v ∈ X) then
5          X                          (* Touch v *)
6        else
7          let
8            X' = X ∪ {v}             (* Enter v *)
9            X'' = iter DFS X' (N_G(v))
10         in X'' end                 (* Exit v *)
11   in DFS({}, s) end
```

The helper function $\texttt{DFS}(X, v)$ does all the work. X is the set of already visited vertices (as in BFS) and v is the current vertex (that we want to explore from). The code first tests if v has already been visited and returns if so (line 5, `Touch` v). . Otherwise it visits the vertex v by adding it to X (line 8, `Enter` v), iterating itself recursively on all neighbors, and finally returning the updated set of visited vertices (line 10, `Exit` v).

Recall that $(\texttt{iter}\ f\ s_0\ A)$ iterates over the elements of A starting with a state s_0. Each iteration uses the function $f : \alpha \times \beta \to \alpha$ to map a state of type α and element of type β to a new state. It can be thought of as:

$S = s_0$
foreach $a \in A$:
  $S = f(S, a)$
return S

For a sequence `iter` processes the elements in the order of the sequence, but since sets are unordered the ordering of `iter` will depend on the implementation.

What this means for the DFS algorithm is that when the algorithm visits a vertex v (i.e., reaches the line 8, `Enter` v, it picks the first outgoing edge (v, w_1), through `iter`, calls `DFS'`$(X \cup \{v\}, w_1)$ to explore the unvisited vertices reachable from w_1. When the call `DFS'`$(X \cup \{v\}, w_1)$ returns the algorithm has fully explored the graph reachable from w_1 and the vertex set returned (call it X_1) includes all vertices in the input $X \cup v$ plus all vertices reachable from w_1.

The algorithm then picks the next edge (v, w_2), again through `iter`, and fully explores the graph reachable from w_2 starting with the the vertex set X_1. The algorithm continues in this manner until it has fully explored all out-edges of v. At this point, `iter` is complete—and X'' includes everything in the original $X' = X \cup \{v\}$ as well as everything reachable from v.

Like the BFS algorithm, however, the DFS algorithm follows paths, making it thus possible to compute interesting properties of graphs.

For example, we can find all the vertices reachable from a vertex v, we can determine if a graph is connected, or generate a spanning tree.

Question 12.23. *Can we use DFS to find the shortest paths?*

Unlike BFS, however, DFS does not naturally lead to an algorithm for finding shortest unweighted paths.

It is, however, useful in some other applications such as topologically sorting a directed graph (TOPSORT), cycle detection, or finding the strongly connected components (SCC) of a graph. We will touch on some of these problems briefly.

Touching, Entering, and Exiting. There are three points in the code that are particularly important since they play a role in various proofs of correctness and also these are the three points at which we will add code for various applications of `DFS`. The points are labeled on the left of the code. The first point is `Touch` v which is the point at which we try to visit a vertex v but it has already been visited and hence added to X. The second point is `Enter` v which is when we first encounter v and before we process its out edges. The third point is `Exit` v which is just after we have finished visiting the out-neighbors and are returning from visiting v. At the exit point all vertices reachable from v must be in X.

Exercise 12.24. *At `Enter` v can any of the vertices reachable from v already be in X? Answer this both for directed and separately for undirected graphs.*

Question 12.25. *Is `DFS` a parallel algorithm? Can you make it parallel?*

At first sight, we might think that `DFS` can be parallelized by searching the out edges in parallel. This would indeed work if the searches initiated never "meet up" (e.g., the graph is a tree so paths never meet up). However, when the graphs reachable through the outgoing edges are shared, visiting them in parallel creates complications because we don't want to visit a vertex twice and we don't know how to guarantee that the vertices are visited in a depth-first manner.

For example in Example 12.16, if we search from the out-edges on s in parallel, we would visit the vertex b twice. More generally we would visit the vertices b, c, f multiple times.

Remark 12.26. *Depth-first search is known to be* P*-complete, a class of computations that can be done in polynomial work but are widely believed that they do not admit a polylogarithmic span algorithm. A detailed discussion of this topic is beyond the scope of our class. But this provides further evidence that DFS might not be highly parallel.*

Cost of DFS. The cost of `DFS` will depend on what data structures we use to implement the set, but generally we can bound it by counting how many operations are made and multiplying it by the cost of each operation. In particular we have the following

Lemma 12.27. *For a graph $G = (V, E)$ with m edges, and n vertices, `DFS'` will be called at most m times and a vertex will be entered for visiting at most $\min(n, m)$ times.*

Proof. Since each vertex is visited at most once, every edge will only be traversed once, invoking a call to `DFS'`. Therefore at most m calls will be made to `DFS'`. At each call of `DFS'` we enter/discover at most one vertex. Since discovery of a vertex can only happen if the vertex is not in the visited set it can happen an most $\min(n, m)$ times. □

Every time we enter `DFS'` we perform one check to see if $v \in X$. For each time we enter a vertex for visiting we do one insertion of v into X. We therefore perform at most $\min m, n$ insertions and m finds. This gives:

Cost Specification 12.28 (DFS). *The `DFS` algorithm a graph with m out edges, and n vertices, and using the tree-based cost specification for sets runs in $O(m \log n)$ work and span. Later we will consider a version based on single threaded sequences that reduces the work and span to $O(n + m)$.*

12.4 DFS Applications: Topological Sorting

We now return to topological sorting as a second application of `DFS`.

Directed Acyclic Graphs. A directed graph that has no cycles is called a *directed acyclic graph* or DAG. DAGs have many important applications. They are often used to represent dependence constraints of some type. Indeed, one way to view a parallel computation is as a DAG where the vertices are the jobs that need to be done, and the edges the dependences between them (e.g. a has to finish before b starts). Mapping such a computation DAG onto processors so that all dependences are obeyed is the job of a scheduler. You have seen this briefly in 15-150. The graph of dependencies cannot have a cycle, because if it did then the system would deadlock and not be able to make progress.

The idea of topological sorting is to take a directed graph that has no cycles and order the vertices so the ordering respects reachability. That is to say, if a vertex u is reachable from v, then v must be lower in the ordering. In other words, if we think of the input graph as modeling dependencies (i.e., there is an edge from v to u if u depends on v), then topological sorting finds a partial ordering that puts a vertex *after* all vertices that it depends on.

To make this view more precise, we observe that a DAG defines a so-called *partial order* on the vertices in a natural way:

For vertices $a, b \in V(G)$, $a \leq_p b$ if and only if there is a directed path from a to b[1]

Remember that a partial order is a relation $\leq_p$ that obeys

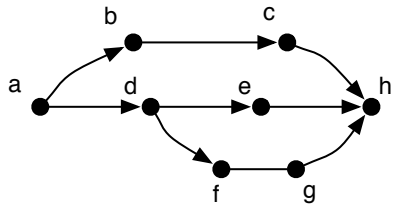
1. reflexivity — $a \leq_p a$,
2. antisymmetry — if $a \leq_p b$ and $b \leq_p a$, then $b = a$, and
3. transitivity — if $a \leq_p b$ and $b \leq_p c$ then $a \leq_p c$.

In this particular case, the relation is on the vertices. It's not hard to check that the relation based on reachability we defined earlier satisfies these 3 properties. Armed with this, we can define the topological sorting problem formally as follows.

Problem 12.29 (Topological Sorting(TOPSORT))**.** *A* topological sort *of a DAG is a total ordering $\leq_t$ on the vertices of the DAG that respects the partial ordering (i.e. if $a \leq_p b$ then $a \leq_t b$, though the other direction needs not be true).*

For instance, consider the following DAG:

[1] We adopt the convention that there is a path from a to a itself, so $a \leq_p a$.



We can see, for example, that $a \leq_p c$, $d \leq h$, and $c \leq h$. But it is a partial order: we have no idea how c and g compare. From this partial order, we can create a total order that respects it. One example of this is the ordering $a \leq_t b \leq_t \leq_t c \leq_t d \leq_t e \leq_t f \leq_t g \leq_t h$. Notice that, as this example graph shows, there are many valid topological orderings.

Solving TOPSORT using `DFS`. To topologically sort a graph, we augment our directed graph $G = (V, D)$ with a new source vertex s and a set of directed edges from the source to every vertex, giving $G' = (V \cup \{s\}, E \cup \{(s, v) : v \in V\})$. We then run the following variant of `DFS` on G' starting at s.

Pseudo Code 12.30 (Topological Sort)**.**

```
1  function topSort(G = (V, E)) =
2    let
3      s = a new vertex
4      G' = (V ∪ {s}, E ∪ {(s, v) : v ∈ V})
5      function DFS ((X, L), v) =
6        if (v ∈ X) then
7            (X, L)              (* Touch v *)
8        else
9          let
10           X' = X ∪ {v}        (* Enter v *)
11           (X'', L') = iter DFS (X', L) (N_G(v))
12         in (X'', v :: L') end   (* Exit v *)
13   in  DFS ((({}, [ ]), s) end
```

The significant changes from the generic version of DFS' are marked with underlines. In particular we thread a list L through the search. The only thing we do with this list is cons the vertex v onto the front of it when we exit `DFS` for vertex v (line `Exit` v). We claim that at the end, the ordering in the list returned specifies a topological sort of the vertices, with the earliest at the front.

Why is this correct? The correctness crucially follows from the property that `DFS` fully searches any unvisited vertices that are reachable from it before returning. In particular the following theorem is all that is needed.

Theorem 12.31. *On a DAG when exiting a vertex v in `DFS` all vertices reachable from v have already exited.*

Proof. This theorem might seem obvious, but we have to be a bit careful. Consider a vertex u that is reachable from v and consider the two possibilities of when u is entered relative to v.

1. u is entered before v is entered. In this case u must also have exited before v is entered otherwise there would be a path from u to v and hence a cycle.
2. u is entered after v is entered. In this case since u is reachable from v it must be visited while searching v and therefore exit before v exits.

□

This theorem implies the correctness of the code for topological sort. This is because it places vertices on the front of the list in exit order so all vertices reachable from a vertex v will appear after it in the list, which is the property we want.

12.5 DFS Application: Cycle Detection in Undirected Graphs

We now consider some other applications of `DFS` beyond just reachability. Given a graph $G = (V, E)$ *cycle detection* problem is to determine if there are any cycles in the graph. The problem is different depending on whether the graph is directed or undirected, and here we will consider the undirected case. Later we will also look at the directed case.

How would we modify the generic `DFS` algorithm above to solve this problem? A key observation is that in an undirected graph if `DFS`$'$ ever arrives at a vertex v a second time, and the second visit is coming from another vertex u (via the edge (u, v)), then there must be two paths between u and v: the path from u to v implied by the edge, and a path from v to u followed by the search between when v was first visited and u was visited. Since there are two distinct paths, there is a "cycle". Well not quite! Recall that in an undirected graph a cycle must be of length at least 3, so we need to be careful not to consider the two paths $\langle u, v \rangle$ and $\langle v, u \rangle$ implied by the fact the edge is bidirectional (i.e. a length 2 cycle). It is not hard to avoid these length two cycles. These observations lead to the following code.

Pseudo Code 12.32 (Undirected cycle detection)**.**

```
1  function undirectedCycle(G, s) =
2    let
3      function DFS p ((X, C), v) =
4        if (v ∈ X) then
5            (X, true)        (* touch v *)
6        else
7          let
8            X' = X ∪ {v}    (* enter v *)
9            (X'', C') = iter (DFS v) (X', C) (N_G(v)\{p})
10         in (X'', C') end      (* exit v *)
11   in DFSs (({}, false), s) end
```

The code returns both the visited set and whether there is a cycle. The key differences from the generic DFS are underlined. The variable C is a Boolean variable indicating whether a cycle has been found so far. It is initially set to *false* and set to *true* if we find a vertex that has already been visited. The extra argument p to DFS$'$ is the parent in the DFS tree, i.e. the vertex from which the search came from. It is needed to make sure we do not count the length 2 cycles. In particular we remove p from the neighbors of v so the algorithm does not go directly back to p from v. The parent is passed to all children by "currying" using the partially applied (DFS$'$ v). If the code executes the Touch v line then it has found a path of at least length 2 from v to p and the length 1 path (edge) from p to v, and hence a cycle.

Exercise 12.33. *In the final line of the code the initial "parent" is the source s itself. Why is this OK for correctness?*

12.6 DFS Application: Cycle Detection in Directed Graphs

We now return to cycle detection but in the directed case. This can be an important preprocessing step for topological sort since topological sort will return garbage for a graph that has cycles. As with topological sort, we augment the input graph $G = (V, E)$ by adding a new source s with an edge to every vertex $v \in V$. Note that this modification cannot add a cycle since the edges are all directed out of s. Here is the code:

Pseudo Code 12.34 (Directed cycle detection)**.**

```
1  function directedCycle(G = (V, E)) =
2    let
3      s = a new vertex
4      G' = (V ∪ {s}, E ∪ {(s, v) : v ∈ V})
5      function DFS((X, Y, C), v) =
6        if (v ∈ X) then
7          (X, Y, C ∨ (v ∈? Y))      (* touch v *)
8        else
9          let
10           X' = X ∪ {v}            (* enter v *)
11           Y' = Y ∪ {v}
12           (X'', _, C') = iter DFS (X', Y', C) (N_G'(v))
13         in (X'', Y, C') end       (* exit v *)
14     (_, _, C) = DFS(({}, {}, false), s)
15   in C end
```

The differences from the generic version are once again underlined. In addition to threading a Boolean value C through the search that keeps track of whether there are any cycles, it threads

the set Y through the search. When visiting a vertex v, the set Y contains all vertices that are ancestors of v is the DFS tree. This is because we add a vertex to Y when entering the vertex and remove it when exiting. Therefore, since recursive calls are properly nested, the set will contain exactly the vertices on the recursion path from the root to v, which are also the ancestors in the DFS tree.

To see how this helps we define a *back edge* in a DFS search to be an edge that goes from a vertex v to an ancestor u in the DFS tree.

Theorem 12.35. *A directed graph $G = (V, E)$ has a cycle if and only if for $G' = (V \cup \{s\}, E \cup \{(s, v) : v \in V\})$ a DFS from s has a back edge.*

Exercise 12.36. *Prove this theorem.*

12.7 Generalizing DFS

As already described there is a common structure to all the applications of DFS—they all do their work either when "entering" a vertex, when "exiting" it, or when "touching" it, i.e. attempting to visit when already visited. This suggests that we might be able to derive a generic version of DFS in which we only need to supply functions for these three components. This is indeed possible by having the user define a state of type α that can be threaded throughout search, and then supplying and an initial state and the following three functions:

Pseudo Code 12.37 (Functions for generalized DFS)**.**

```
1 Σ_0      : α
2 touch    : α × vertex × vertex → α
3 enter    : α × vertex × vertex → α
4 exit     : α × α × vertex × vertex → α
```

Each function takes the state, the current vertex v, and the parent vertex p in the DFS tree, and returns an updated state. The *exit* function takes both the enter and the exit state. The code for generalized DFS for directed graphs can then be written as:

Pseudo Code 12.38 (Generalized directed DFS)**.**

```
1  function directedDFS(G, Σ_0, s) =
2    let
3      function DFS p ((X, Σ), v) =
4        if (v ∈ X) then
5          (X, touch(Σ, v, p))
6        else
7          let
8            Σ' = enter(Σ, v, p)
9            X' = X ∪ {v}
10           (X'', Σ'') = iter (DFS v) (X', Σ') (N_G^+(v))
11           Σ''' = exit(Σ', Σ'', v, p)
12         in (X'', Σ''') end
13   in
14     DFS s ((∅, Σ_0), s)
15   end
```

At the end, `DFS` returns an ordered pair (X, Σ) : *Set* $\times$ α, which represents the set of vertices visited and the final state Σ. The generic search for undirected graphs is slightly different since we need to make sure we do not immediately visit the parent from the child. As we saw this causes problems in the undirected cycle detection, but it also causes problems in other algorithms. The only necessary change to the directedDFS is to replace the $(N_G^+(v))$ at the end of Line 10 with $(N_G^+(v) \setminus p)$.

With this code we can easily define our applications of `DFS`. For undirected cycle detection we have:

Pseudo Code 12.39 (Undirected Cycles with generelized undirected DFS)**.**

```
Σ_0 = false : bool
function touch(_) = true
function enter(fl,_,_) = fl
function exit(_,fl,_,_) = fl
```

For topological sort we have.

Pseudo Code 12.40 (Topological sort with generelized directed DFS)**.**

```
Σ_0 = [] : vertex list
function touch(L,_,_) = L
function enter(L,_,_) = L
function exit(_,L,v,_) = v :: L
```

For directed cycle detection we have.

Pseudo Code 12.41 (Directed cycles with generelized directed DFS)**.**

```
1 Σ0 = ({}, false) : Set × bool
2 function touch((S, fl), v, _) = (S, fl ∨ (v ∈? S))
3 function enter((S, fl), v, _) = (S ∪ {v}, fl)
4 function exit((S,_), (_, fl), v, _) = (S, fl)
```

For these last two cases we need to also augment the graph with the vertex s and add the the edges to each vertex $v \in V$. Note that none of the examples actually use the last argument, which is the parent. There are other examples that do.

12.8 DFS with Single-Threaded Arrays

Here is a version of `DFS` using adjacency sequences for representing the graph and ST sequences for keeping track of the visited vertices.

Pseudo Code 12.42 (DFS with single treaded arrays)**.**

```
1  function directedDFS(G : (int seq) seq, Σ0 : state, s : int) =
2    let
3      function DFS p ((X : bool stseq, Σ : state), v : int) =
4        if (X[v]) then
5          (X, touch(Σ, v, p))
6        else
7          let
8            X' = update(v, true, X)
9            Σ' = enter(Σ, v, p)
10           (X'', Σ'') = iter (DFS v) (X', Σ') (G[v])
11           Σ''' = exit(Σ', Σ'', v, p)
12         in (X'', Σ''') end
13     Xinit = stSeq.fromSeq(⟨ false : v ∈ ⟨0, ..., |G| − 1⟩ ⟩)
14   in
15     DFS s ((Xinit, Σ0), s))
16   end
```

If we use an `stseq` for X (as indicated in the code) then this algorithm uses $O(m)$ work and span. However if we use a regular sequence, it requires $O(n^2)$ work and $O(m)$ span.

.