Chapter 11

Depth-First Search

In the last chapter we saw that breadth-first search (BFS) is effective in solving certain problems, such as shortest paths. In this chapter we will see that another graph search algorithm called *depth-first search* or *DFS* for short, is more effective for other problems such as topological sorting, cycle detection, and the finding connected components of a graph.

11.1 Topological Sort

As as example, we will consider what a rock climber must do before starting a climb so that she can 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. The example is illustrative of many situations which require a set of actions and has dependencies among the tasks. Figure 11.2 illustrates the actions (tasks) that a climber must take, along with the dependencies between them. This is a directed graph with the vertices being the tasks, and the directed edges being the dependences between tasks. 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.

We note that 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.

Definition 11.1 (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.

Figure 11.1: Before starting, climbers must carefully put on gear that protects them in a fall.

Definition 11.2 (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.*

There are many possible topological orderings for the DAG in Figure 11.2. 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, 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. See Figure 11.3 for an example. 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.

We will soon see how to use DFS as an algorithm to solve topological sorting. Before we move on, however, note that BFS is not an effective way to implement topological sort since it

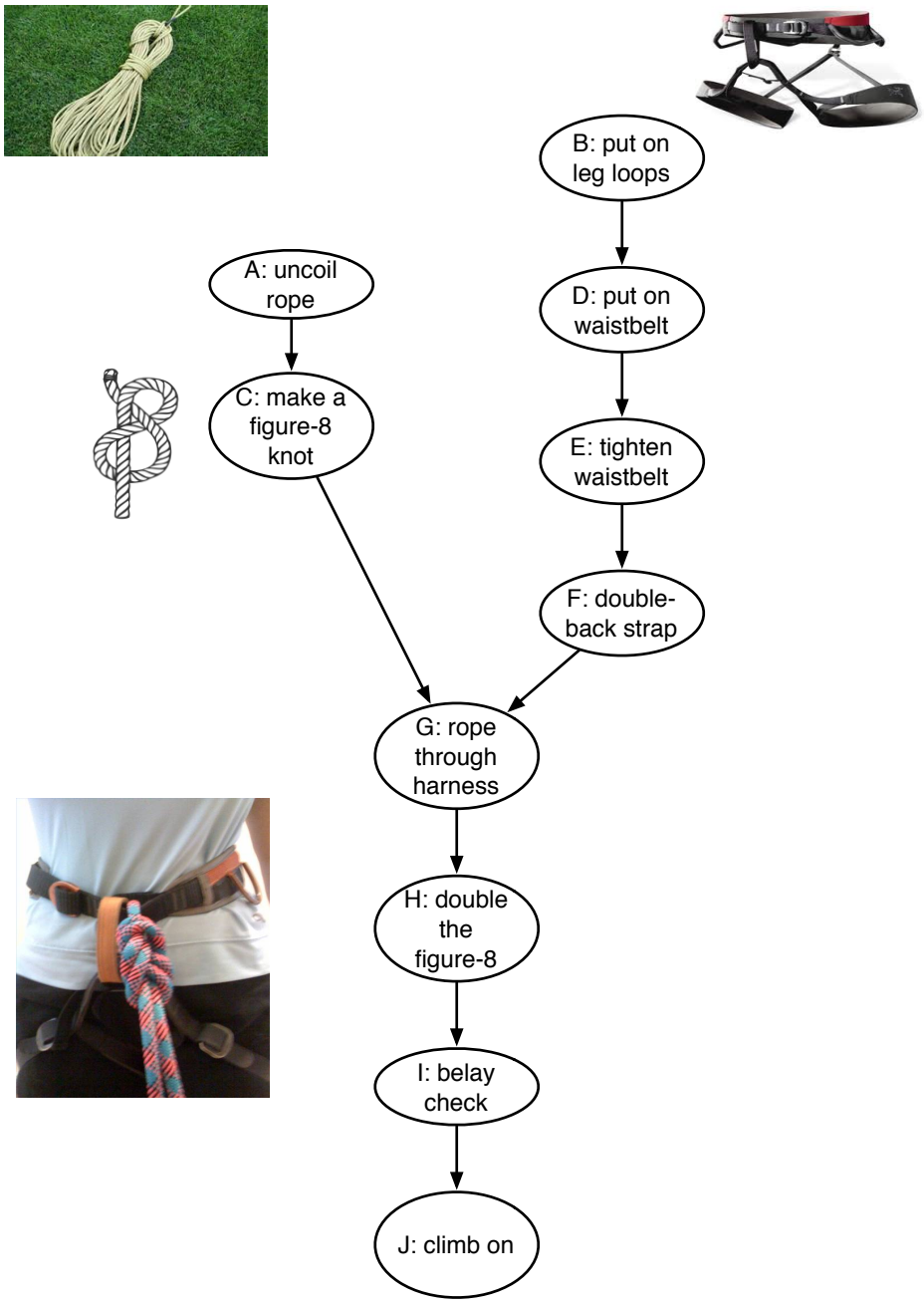


Figure 11.2: A simplified DAG for tying into a rope with a harness.

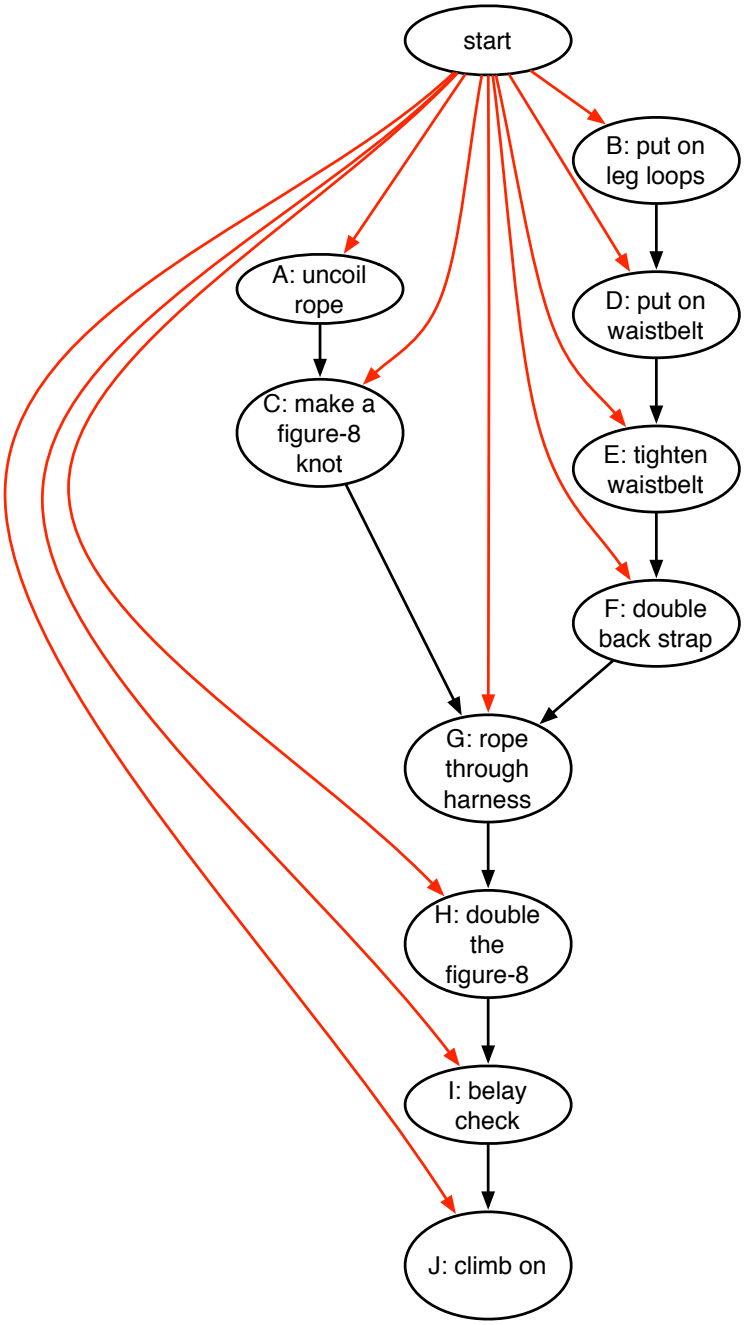


Figure 11.3: Climber's DAG with a start node.

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.

11.2 DFS: Depth-First Search

Recall that in graph search we have the freedom to pick a any (non-empty) subset of the vertices on the frontier to visit. The DFS algorithm is a specialization of graph search that picks the single vertex on the frontier that has been most recently added. 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).

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 stack, achieves the same effect without having to resort to time stamps since stacks support a LIFO (last in, first out) ordering.

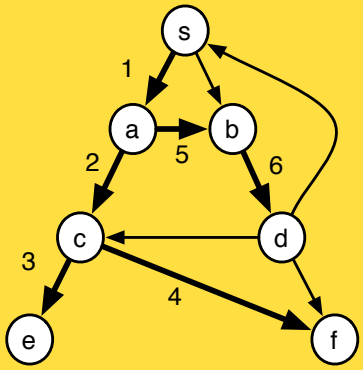
As an intuitive way to think about DFS, think of curiously browsing photos of your friends in a photo sharing site. You start by visiting the site of a friend. You mark it with a white pebble so you remember it has been visited. You then realize that some other people are tagged and visit one of their sites, marking it with a white pebble. You then in that new site see other people tagged and visit one of those sites. You of course are careful not to revisit people's sites that already have a pebble on them. When you finally reach a site that contains no new people, you start pressing the back button. What the back button does is to change the site you are on from a white pebble to a red pebble. The red pebble indicates you are done searching that site—i.e. there are no more neighbors who have not been visited. The back button also moves you to the most recently placed white pebble (i.e. the previous site you visited before the current one). You now check if there are other unvisited neighbors on that site. If there are some, you visit one of those. When done visiting all neighbors you hit the back button again, turning that site into a red pebble. This continues until you turn your original friend from a white pebble to a red pebble.

Exercise 11.3. *Convince yourself that the pebbling-based description and the DFS algorithm described above are equivalent in terms of the ordering you first visit a site (place a white pebble on it).*

We note that DFS and BFS visits vertices in very different orders, and in particular DFS does not visit vertices in order of their levels. Instead it dives down deeply following a single path until there are no longer any unvisited out neighbors to visit. It then backs up along that path until it finds another (unvisited) path to follow. This is why it is called depth-first search. DFS can be more effective in solving some problems exactly because it fully follows a path.

Figure 11.4: 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.

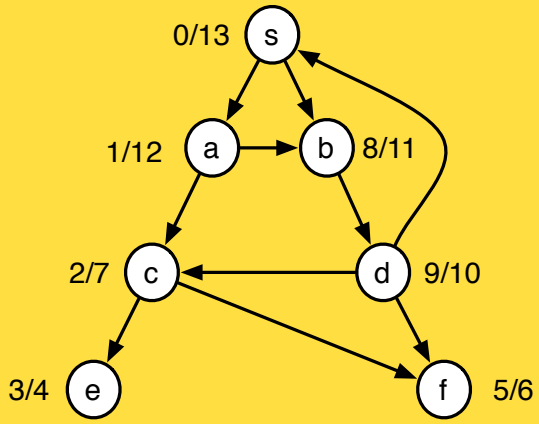
Example 11.4. *An example DFS.*



11.3 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 11.5. *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 11.6. *Can you determine by just using the DFS numbers of a graph whether a vertex is an ancestor or a descendant of another vertex?*

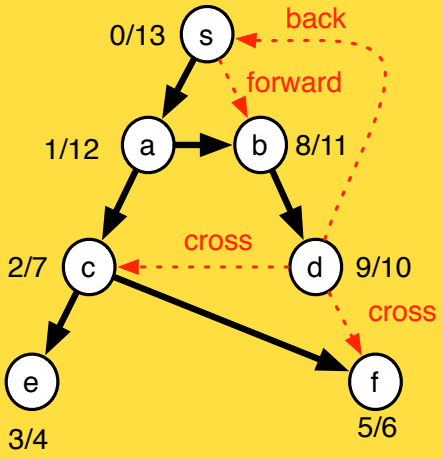
There is an interesting connection between DFS numbers and topological sort: in a DAG, the sort of the vertices from the largest to the smallest finishing time yield a topological ordering of the DAG.

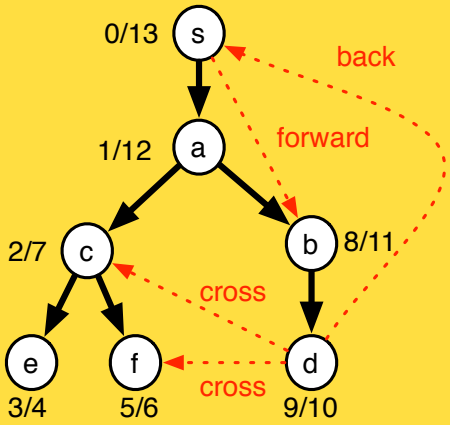
Tree edges, back edges, forward edges, and cross edges. Given a graph and a DFS of the graph, we can classifies the edges of the graph into various categories.

Definition 11.7. *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 11.8. *Tree edges (black), and non-tree edges (red, dashed) illustrated with the original graph and drawn as a tree.*





DFS-based reachability. 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.

Algorithm 11.9 (DFS reachability)**.**

```
1  function reachability(G, s) =
2    let
3      function DFS(X, v) =
4        if (v ∈ X) then
5          val X                          % Touch v
6        else
7          let
8            val X' = X ∪ {v}             % Enter v
9            val 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 $\texttt{DFS'}(X \cup \{v\}, w_1)$ to explore the unvisited vertices reachable from w_1. When the call $\texttt{DFS'}(X \cup \{v\}, w_1)$ 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.

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 11.10. *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.*

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 11.5, 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 11.11. *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 11.12. *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 11.13 (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 $O(n + m)$.*

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

1. reflexivity — $a \leq_p a$,

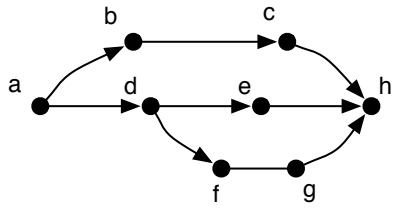
2. antisymmetry — if $a \leq_p b$ and $b \leq_p a$, then $b = a$, and

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 11.14 (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:



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.

Algorithm 11.15 (Topological Sort)**.**

```
1  function topSort(G = (V, E)) =
2    let
3      val s = a new vertex
4      val 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           val X' = X ∪ {v}        % Enter v
11           val (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 11.16. *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.

11.5 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 algorithm.

Algorithm 11.17 (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            val X' = X ∪ {v}    % enter v
9            val (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 11.18. *In the final line of the code the initial "parent" is the source s itself. Why is this OK for correctness?*

11.6 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 algorithm:

Algorithm 11.19 (Directed cycle detection)**.**

```
1  function directedCycle(G = (V, E)) =
2    let
3      val s = a new vertex
4      val 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           val X' = X ∪ {v}          % enter v
11           val Y' = Y ∪ {v}
12           val (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 11.20. *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 11.21. *Prove this theorem.*

11.7 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 11.22 (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 algorithm for generalized DFS for directed graphs can then be written as:

Algorithm 11.23 (Generalized directed DFS)**.**

```
1  function directedDFS(G, Σ0, 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            val Σ' = enter(Σ, v, p)
9            val X' = X ∪ {v}
10           val (X'', Σ'') = iter (DFS v) (X', Σ') (N_G^+(v))
11           val Σ''' = exit(Σ', Σ'', v, p)
12         in (X'', Σ''') end
13   in
14     DFS s ((∅, Σ0), s)
15   end
```

At the end, `DFS` returns an ordered pair (X, Σ) : $Set \times \alpha$, 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 11.24 (Undirected Cycles with generelized undirected DFS)**.**

```
1  Σ0 = false : bool
2  function touch(_) = true
3  function enter(fl,_,_) = fl
4  function exit(_, fl,_,_) = fl
```

For topological sort we have.

Pseudo Code 11.25 (Topological sort with generelized directed DFS)**.**

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

For directed cycle detection we have.

Pseudo Code 11.26 (Directed cycles with generelized directed DFS)**.**

```
1 Σ_0 = ({}, 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.

11.8 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.

Algorithm 11.27 (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            val X' = update(v, true, X)
9            val Σ' = enter(Σ, v, p)
10           val (X'', Σ'') = iter (DFS v) (X', Σ') (G[v])
11           val Σ''' = exit(Σ', Σ'', v, p)
12         in (X'', Σ''') end
13     val X_init = stSeq.fromSeq(⟨ false : v ∈ ⟨0, ..., |G| − 1⟩ ⟩)
14   in
15     DFS s ((X_init, 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.

.