Chapter 14

Depth-First Search

In the last chapter we saw that breadth-first search (BFS) is effective in solving certain problems on graphs, such as finding the shortest paths from a source. 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, and cycle detection on directed graphs.

14.1 Topological Sort

As an example, we consider what a rock climber must do before starting a climb to protect herself in case of a fall. For simplicity, we only consider the tasks of wearing a harness and tying into the rope. The example is illustrative of many situations which require a set of actions or tasks with dependencies among them. Figure 14.2 illustrates the 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 the safety of the climber—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 14.1. *There is something interesting about the structure of this graph. Can you see something missing in this graph compared to a general graph?*

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

Figure 14.1: Before starting, climbers must carefully put on gear that protects them in a fall. Shown: climbing legend Lynn Hill climbing.

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 14.3 (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 $(v_i, v_j) \in E$, $j > i$ holds.*

Question 14.4. *Can you come up with a topological ordering of the climbing DAG shown in Figure 14.2.*

There are many possible topological orderings for the DAG in Figure 14.2. 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 to all the other vertices. See Figure 14.3 for an example.

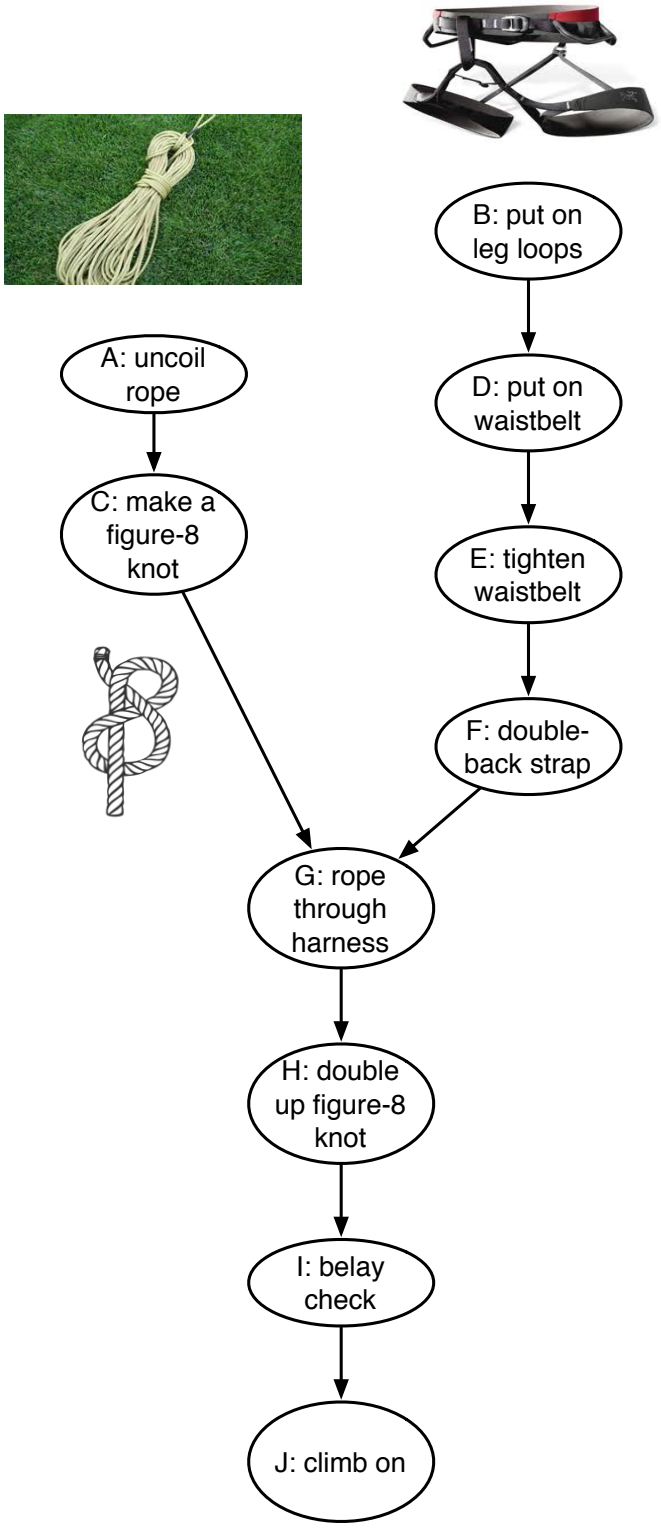


Figure 14.2: A simplified DAG for tying into a rope with a harness.

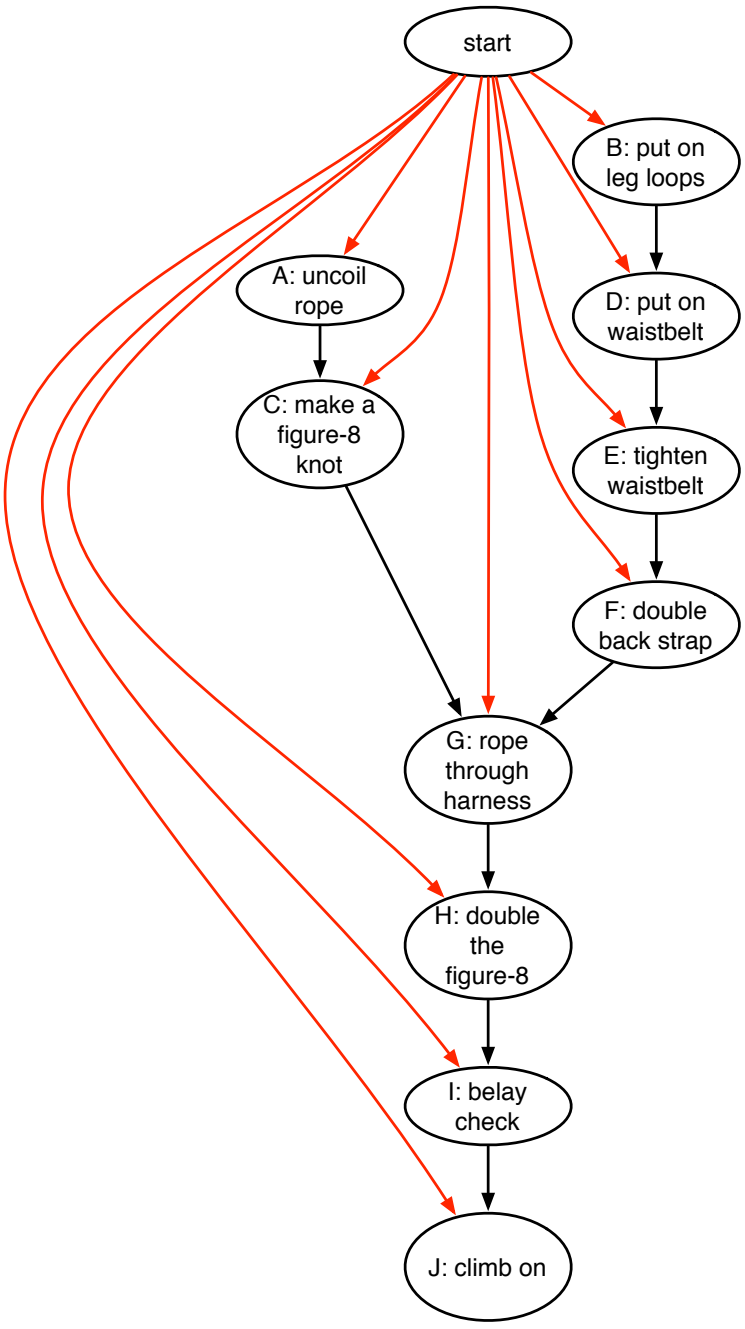


Figure 14.3: Climber's DAG with a start node.

Question 14.5. *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 14.6. *Can you see why BFS is not as natural for topological sorting?*

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 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.

14.2 Depth-First Search (DFS)

Recall that in graph search, we have the freedom to pick any (non-empty) subset of the vertices on the frontier to visit in each round. The DFS algorithm is a specialization of graph search that picks the most recent vertex added to the frontier. Intuitively, when a vertex is visited, we can think of "seeing" all the neighbors, and the DFS algorithm as an algorithm that visits the most recently seen vertex (we can order the out edges based on the ordering supplied by the graph, if any, or arbitrarily if no order is provided).

Question 14.7. *How can we determine the next vertex to visit?*

One way to keep track of the most recently seen vertices is to time-stamp the neighbors when we visit a vertex, and then use these time stamps as a priority (latest first). Since time stamps increase monotonically and since we always visit the greatest one, we can implement this by using a stack, which would assign implicitly the time-stamps. We can refine this solution one-more level and in fact represent the stack implicitly by using recursion.

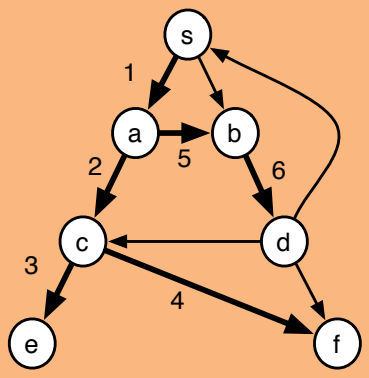
Algorithm 14.8 (DFS reachability)**.**

```
1 function reachability(G, s) = let
2   function DFS (X, v) =          % X is the set of visited vertices, initially empty
3     if v ∈ X then X              % if v is already visited, return
4     else iter DFS (X ∪ {v}) (N_G(v))   % else mark as visited, and visit neighbors
5 in DFS({}, s) end
```

In the algorithm, if the vertex v has not already been visited by $\texttt{DFS}(X, v)$, we mark it as visited by adding it to the set X (i.e. $X \cup \{v\}$), and then immediately iterate over the neighbors recursively running `DFS` on each of them, in turn. The algorithm returns all visited vertices (the

final value of X). The frontier is not kept explicitly, but is implicitly in the "recursion stack".

Example 14.9. *An example of DFS on a graph where edges are ordered counterclockwise, starting from left.*



v	X
s	$\{\}$
a	$\{s\}$
c	$\{s, a\}$
e	$\{s, a, c\}$
f	$\{s, a, c, e\}$
b	$\{s, a, c, e, f\}$
d	$\{s, a, c, e, f, b\}$
c	$\{s, a, c, e, f, b, d\}$
f	$\{s, a, c, e, f, b, d\}$
s	$\{s, a, c, e, f, b, d\}$
b	$\{s, a, c, e, f, b, d\}$

Each row corresponds to the arguments to one call to `DFS` *in the order they are called. In the last four rows the vertices have already been visited, so the call returns immediately without revisiting the vertices since they appear in* X.

Exercise 14.10. *Convince yourself that using generic graph search with a stack visits the vertices in the same order as the recursive implementation of DFS.*

The recursive formulation of DFS has an important property—it makes it easy not just to identify when a vertex is first visited (i.e., when adding v to X), but also to identify when everything that is reachable from v has been visited (i.e., when the `iter` is done).

As an intuitive way to understand 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 are ready to press the back button. However, before you press the back button you change the pebble on the site you are on from white to red. The red pebble indicates you are done searching that site—i.e. there are no more neighbors who have not been visited. You then press the back button, which moves you to the site you visited immediately before first visiting the current one (also the most recently visited site that still has a white pebble on it). 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 change that site from a white to a red pebble, and hit the back button

Figure 14.4: 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.

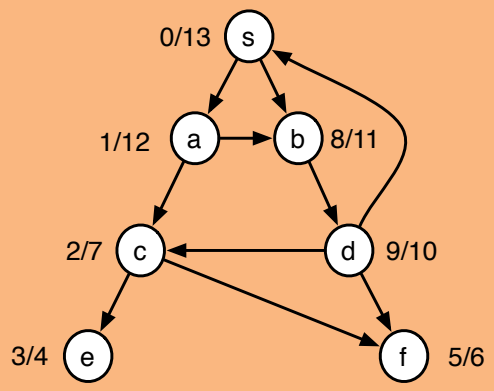
again. This continues until you change the site of your original friend from a white pebble to a red pebble.

This process of turning a vertex white and then later red is a conceptual way to identify two important points in the search. As we will see many applications of DFS require us to do something at each of these points.

14.3 Cost of DFS

At first sight, we might think that DFS can be parallelized by searching the out edges in parallel. This might work if the searches on each out edge never "meet up" as would be the case for a tree. However, when portions of the graph reachable through the outgoing edges are shared, visiting them in parallel creates complications. This is because it is important that each vertex is only visited (discovered) once, and in DFS it is also important that the earlier out-edge discovers any shared vertices, not the later one.

Example 14.11. *In our example graph:*



if we search the out-edges of s in parallel, we would visit the vertices a, c and e in parallel with b, d and f. This is not the DFS ordering, which requires b, d and f to be visited via a. In fact it is BFS ordering. Furthermore the two parallel searches would have to synchronize to avoid visiting vertices, such as b, twice.

Remark 14.12. *Depth-first search is known to be* **P***-complete, a class of computations that can be done in polynomial work but are widely believed not to admit a polylogarithmic span algorithm. A detailed discussion of this topic is beyond the scope of this book, but it provides evidence that DFS is unlikely to be highly parallel.*

We therefore assume there is no parallelism in DFS and focus on the work. The work required by 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 this by the cost of each operation. In particular we have the following

Lemma 14.13. *For a graph $G = (V, E)$ with m edges, and n vertices, `DFS` in Algorithms 14.8 and 14.22 will be called at most $n + m$ times and a vertex will be discovered (and finished) at most n times.*

Proof. Since each vertex is added to X when it is first discovered, every vertex can only be discovered once. It follows that every out edge will only be traversed once, invoking a call to `DFS`. Therefore at most m calls will be made to `DFS` through an edge. In topological sort, an additional n initial calls are made starting at each vertex. □

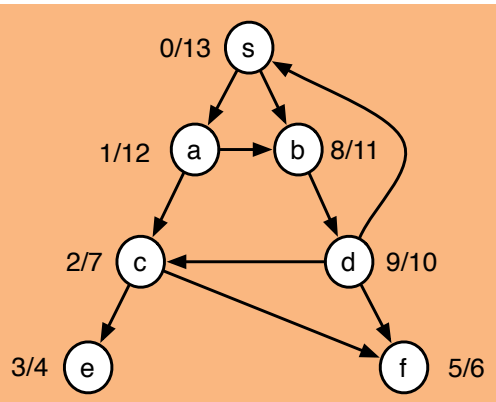
Every time we call `DFS` we perform one find for $X[v]$. Every time we discover a vertex we do one insertion of v into X. We therefore perform at most n insertions and $m + n$ finds. This gives:

Cost Specification 14.14 (DFS). *The `DFS` algorithm on a graph with m out edges, and n vertices, and using the tree-based cost specification for sets runs in $O((m + n) \log n)$ work and span. Later we will consider a version based on single threaded sequences that reduces the work and span to $O(n + m)$.*

14.4 DFS Numbers and DFS Tree

To formalize this notion of turning a vertex white and red in DFS, we can assign two timestamps to each vertex. The time at which a vertex receives its white pebble is called the *discovery time*. The time at which a vertex receives its red pebble is called *finishing time*. We refer to the timestamps cumulatively as *DFS numbers*.

Example 14.15. *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.*



Note that vertex a *gets a finished timed of* 12 *since it does not finish until everything reachable from its two out neighbors,* c *and* b*, have been fully explored. Vertices* d*,* e *and* f *have no unvisited out neighbors, and hence their finishing time is one more than their discovery time.*

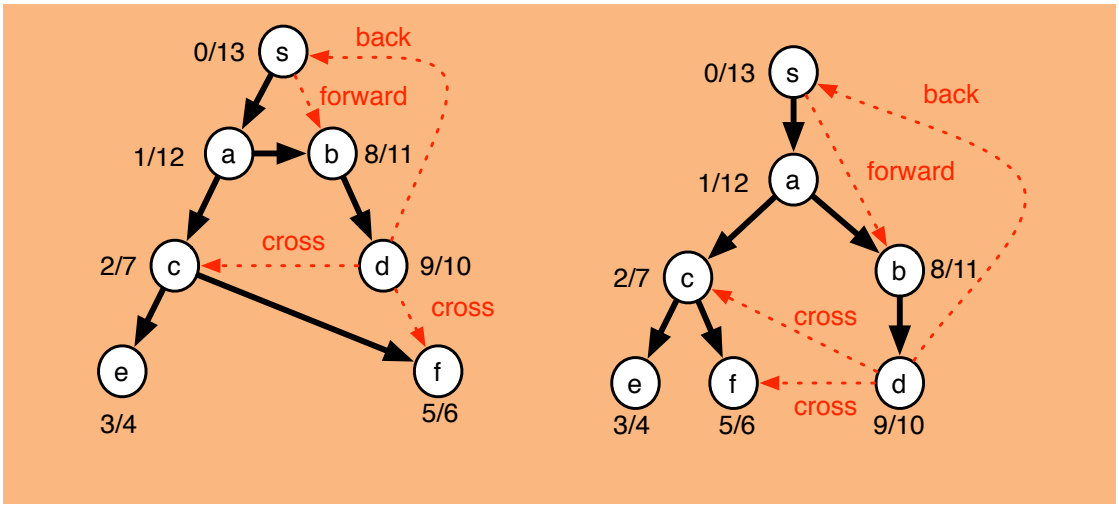
Given a graph and a DFS of the graph, it is can be useful to classify the edges of into categories.

Definition 14.16. *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 14.17. *Tree edges (black), and non-tree edges (red, dashed) illustrated with the original graph and drawn as a tree.*



Question 14.18. *Can you think of a way to implement pebbling efficiently?*

Exercise 14.19. *How can you determine by just using the DFS numbers of the endpoints of an edge whether it is a cross edge, forward edge, or backward edge?*

14.5 Revisiting DFS

We can write a version of DFS that corresponds closely to the DFS numbers, making explicit the points at which a vertex is discovered, finished, and revisited (if any). Viewing DFS in this way helps in applying the technique to solve different problems. In fact, later in this chapter we describe a higher-order version of DFS that can be used to solve different problems by simply instantiating it with different arguments.

Algorithm 14.20 (Time-Stamping DFS)**.**

```
1  function TimeStampingDFS(G, s) =
2    let
3      function DFS ((X, t), v) =
4        if v ∈ X then
5          (* Revisit v *)
6          X
7        else
8          let
9            (* Visit v *)
10           td_v = t + 1
11           X' = X ∪ {(v, td_v, ⊥)}
12           (X'', t') =  iter DFS (X', td_v) (N_G^+(v))
```

```
13              (* Finish v *)
14              tf_v = td_v + 1
15              X''' = X'' \ {(v, td_v, ⊥)} ∪ {(v, td_v, tf_v)}
16          in
17              (X''', tf_v)
18          end
19  in
20      DFS ((∅, −1), s)
21  end
```

14.6 Topological Sort with DFS

The DFS numbers have many interesting properties. One of these properties, established by the following lemma, makes it possible to use DFS for topological sorting.

Lemma 14.21. *When running DFS on a DAG, if a vertex u is reachable from v then u will finish before v finishes.*

Proof. This lemma might seem obvious, but we need to be a bit careful. We consider two cases.

1. u is discovered before v. In this case u must finish before v is discovered otherwise there would be a path from u to v and hence a cycle.
2. v is discovered before u. In this case since u is reachable from v it must be visited while searching from v and therefore finish before v finishes. □

Intuitively, the lemma holds because DFS fully searches any unvisited vertices that are reachable from a vertex before returning from that vertex (i.e., finishing that vertex). This lemma implies that if we order the vertices by finishing time (latest first), then all vertices reachable from a vertex v will appear after v in the ordering, since they must finish before v finishes. This is exactly the property we require from a topological sort.

Algorithm 14.22 gives an implementation of topological sort. Instead of generating DFS numbers, and sorting them, which is a bit clumsy, it maintains a sequence S and whenever finishing a vertex v, appends v to the front of S, i.e. $cons(v, S)$. This gives the same result since it orders the vertices by reverse finishing time. The main difference from the `reachability` version of `DFS` is that we thread the sequence S, as indicated by the underlines. In the code we have marked the discovery (Line 7) and finish points (Line 9). The vertex is added to the front of the list at the finish. The last line iterates over all the vertices to ensure they are all included in the final topological sort. Alternatively we could have added a "start" vertex and an edge from it to every other vertex, and then just searched from the start. The algorithm just returns

Algorithm 14.22 (Topological Sort)**.**

```
1  function topSort(G = (V, E)) = let
2       function DFS ((X, S), v) =
3         if v ∈ X then
4           (X, S)   (* Revisit v *)
5         else
6           let
7             X' = X ∪ {v}                                  % Discover v
8             (X'', S') = iter DFS (X', S) (N_G^+(v))       % Visit Neighbors
9           in (X'', cons(v, S')) end                       % Finish v
10   in second (iter DFS ({}, ⟨ ⟩) V) end
```

the sequence S (the second value), throwing away the set of visited vertices X.

14.7 Cycle Detection in Undirected Graphs

We now consider some other applications of `DFS`. 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 by removing the parent from the list of neighbors. These observations lead to the following algorithm.

Algorithm 14.23 (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)        % revisit v
6        else
7          let
```

```
8              X' = X ∪ {v}    % discover v
9              (X'', C') = iter (DFS v) (X', C) (N_G(v)\{p})
10           in (X'', C') end          % finish v
11   in DFS s (({}, false), s) end
```

The algorithm 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 $\texttt{DFS}'$ 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 $(\texttt{DFS}'\ v)$. If the code executes the `revisit v` line then it has found a path of length at least 2 from v to p and the length 1 path (edge) from p to v, and hence a cycle.

Exercise 14.24. *In the final line of the algorithm the initial "parent" is the source s itself. Why is this OK for correctness?*

14.8 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 14.25 (Directed cycle detection)**.**

```
1  function directedCycle(G = (V, E)) = let
2       s = a new vertex
3       G' = (V ∪ {s}, E ∪ {(s, v) : v ∈ V})
4       function DFS((X, Y, C), v) =
5         if (v ∈ X then
6           (X, Y, C ∨ Y[v])    % revisit v
7         else let
8               X' = X ∪ {v}          % discover v
9               Y' = Y ∪ {v}
10              (X'', _, C') = iter DFS (X', Y', C) (N_G'(v))
11            in (X'', Y, C') end          % finish v
12      (_, _, C) = DFS(({}, {}, false), s)
```

```
13      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 discovering the vertex and remove it when finishing. 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 14.26. *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 14.27. *Prove this theorem.*

14.9 Higher-Order DFS

As already described there is a common structure to all the applications of `DFS`—they all do their work either when "discovering" a vertex, when "finishing" it, or when "revisiting" 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. More specifically, each function takes the state, the current vertex v, and the parent vertex p in the `DFS` tree, and returns an updated state. The `finish` function takes both the discover and the finish state. The algorithm for generalized DFS for directed graphs can then be written as:

Algorithm 14.28 (Generalized directed DFS)**.**

```
1  function directedDFS (revisit, discover, finish) (G, Σ0, s) =
2    let
3      function DFS p ((X, Σ), v) =
4        if (v ∈ X) then
5          (X, revisit(Σ, v, p))
6        else
7          let
8            Σ' = discover(Σ, v, p)
9            X' = X ∪ {v}
10           (X'', Σ'') = iter (DFS v) (X', Σ') (N_G^+(v))
```

```
11              Σ''' = finish(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` $\times \alpha$, 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 generic algorithm we can easily define our applications of `DFS`. For undirected cycle detection we have:

Algorithm 14.29 (Undirected Cycles with generelized undirected DFS)**.**

$\Sigma_0 =$ `false : bool`
function `revisit`$(\_) =$ `true`
function `discover`$(fl, \_, \_) =$ `fl`
function `finish`$(\_, fl, \_, \_) =$ `fl`

For topological sort we have.

Algorithm 14.30 (Topological sort with generelized directed DFS)**.**

$\Sigma_0 = []$: `vertex list`
function `revisit`$(L, \_, \_) = L$
function `discover`$(L, \_, \_) = L$
function `finish`$(\_, L, v, \_) = v :: L$

For directed cycle detection we have.

Algorithm 14.31 (Directed cycles with generelized directed DFS)**.**

$\Sigma_0 = (\{\},$ `false`$)$: `Set` $\times$ `bool`
function `revisit`$((S,$ `fl`$), v, \_) = (S,$ `fl` $\vee (S[v]))$
function `discover`$((S,$ `fl`$), v, \_) = (S \cup \{v\},$ `fl`$)$
function `finish`$((S, \_), (\_,$ `fl`$), v, \_) = (S,$ `fl`$)$

For these last two cases we need to also augment the graph with the vertex s and add 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.

14.10 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 14.32 (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, revisit(Σ, v, p))
6        else
7          let
8            X' = update(v, true, X)
9            Σ' = discover(Σ, v, p)
10           (X'', Σ'') = iter (DFS v) (X', Σ') (G[v])
11           Σ''' = finish(Σ', Σ'', v, p)
12         in (X'', Σ''') end
13     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.

.