Lecture 11 — Depth-First Search and Applications

Parallel and Sequential Data Structures and Algorithms, 15-210 (Qatar-Spring 2014)

Lectured by Kemal Oflazer — 16 February 2014

What was covered in this lecture:

- Review of graphs and breadth-first search.
- Depth-first Search.
- Using DFS for topological sort and cycle detection (undirected and directed graphs).

1 Review

This section is a detailed review of the material in Lectures 9 and 10. I suggest you read it to refresh your graph basics.

Graphs. Last week you have learned about one of the most important and fascinating topics in computer science: graphs.

Question 1.1. *What makes graphs so special?*

Graphs represent relationships. As you will (or might have) discover (discovered already) relationships between things from the most abstract to the most concrete, e.g., mathematical objects, things, events, people are what makes everything interesting. Considered in isolation, hardly anything is interesting. For example, considered in isolation, there would be nothing interesting about a person. It is only when you start considering his or her relationships to the world around the person becomes interesting. Challenge yourself to try to find something interesting about a person in isolation. You will have difficulty. Even at a biological level, what is interesting are the relationships between cells, molecules, and the biological mechanisms.

Question 1.2. *Trees captures relationships too, why are graphs more interesting?*

Graphs are more interesting simply because they are more expressive. For example, you cannot have cycles in a graph, you cannot have parallel paths between two nodes.

Question 1.3. *What do we mean by a relationship? Can you think of a mathematical way to represent relationships.*

†Lecture notes by Umut Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan, with additional edits by Kemal Oflazer

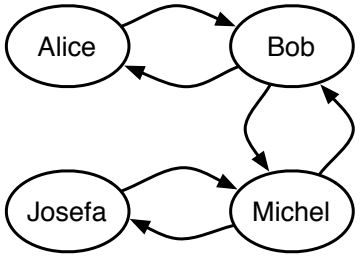


Figure 1: Friendship relation {(Alice, Bob), (Bob, Alice), (Josefa, Michel), (Michel, Josefa), ... } as a graph.

The mathematical notion of a *relation*, which is simply defined as a subset of the Cartesian product of two sets can formally represent any relationship.

Exercise 1. *You can represent the friendship relation between people as a subset of the Cartesian product of the people, e.g, {(Alice, Bob), (Bob, Alice), (Josefa, Michel), (Michel, Josefa), (Bob, Michel), (Michel,Bob), ... }.*

Now what is cool is that you can represent any mathematical relation (and thus) any relationship with a graph.

Question 1.4. *Can you see how to represent a (mathematical) relation with a graph?*

All we have to do is to use the set of vertices to represent the domain and the range of the relationship and connect the vertices with edges as described by the relationship as shown for example in Figure 1.

Representing graphs. How we want to represent a graph largely depends on the operations we intend to support.

Question 1.5. *What kind of operations would we want to support?*

Based our discussions so far, we might want to support the following operations on a graph $G = (V, E)$:

(1) Build and modify a graph by for example inserting and deleting vertices.

(2) Map over the vertices $v \in V$.

(3) Map over the edges $(u, v) \in E$.

(4) Map over the neighbors of a vertex $v \in V$, or in a directed graph the in-neighbors or out-neighbors.

(5) Determine if the edge (u, v) is in E.

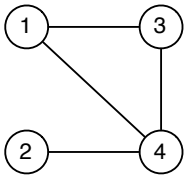
(6) Return the degree of a vertex $v \in V$.

Question 1.6. *Can you remember the different ways for representing graphs?*

Representing graphs, traditional approaches. Traditionally, there are four main representations, all of which assume that vertices are numbered from $1, 2, \ldots, n$ (or $0, 1, \ldots, n-1$):

- **Adjacency matrix.** An $n \times n$ matrix of binary values in which location (i, j) is 1 if $(i, j) \in E$ and 0 otherwise.

  An example:

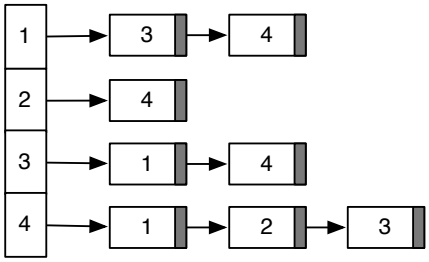


$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

  The main problem with adjacency matrices is their space demand of $\Theta(n^2)$. Graphs are often sparse, with far fewer edges than $\Theta(n^2)$.

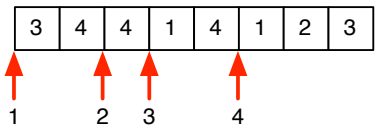
- **Adjacency list.** An array A of length n where each entry $A[i]$ contains a pointer to a linked list of all the out-neighbors of vertex i. In an undirected graph with edge $\{u, v\}$ the edge will appear in the adjacency list for both u and v.

  An example:



- **Adjacency array.** Similar to an adjacency list, an adjacency array keeps the neighbors of all vertices, one after another, in an array `adj`; and separately, keeps an array of indices that tell us where in the `adj` array to look for the neighbors of each vertex.

  An example:



- **Edge list.** A list of pairs $(i, j) \in E$.

Representing graphs, our approach. The problem with the space-effient, list-based representations is that they are not suitable for parallelism. In this course, we will raise the level of abstraction to enable for more natural parallelism. We will also loosen the restriction on the labeling of vertices from 1 to n, allowing instead for any labels. Conceptually, though, the representations we describe are not much different from adjacency lists and edge lists. We are just going to think parallel and base our view on data types that are parallel.

We will often use one of the following two representations.

Edge Sets. The idea here is to represent a graph simply as a set of edges, i.e., an *edge set*. The representation is similar to an edge list representation mentioned before, but it abstracts away from the particular data structure used for the set—the set could be based on a list, an array, a tree, or a hash table.

Edge Tables. As is edge sets but we use a table of edges. This allows us to associate data with each edge.

Question 1.7. *What is the advantage and disadvantage of the set and table based representations?*

Going back to the operations that we wish to be able to perform on graphs, determining whether an edge is a graph can be performed efficiently in $\Theta(\log m)$ work. Using an edge list would require $\Theta(m)$ work.

The disadvantage is that, as with edge lists, they do not allow direct access to the neighbors of a given vertex v.

Question 1.8. *Can you think of an algorithm for finding the neighbors of a given vertex?*

We can consider all the edges and pick out the ones that have v as an endpoint. This requires $\Theta(\log m)$ span and $\Theta(m)$ work, and thus is not work efficient.

Question 1.9. *Can you think of a way to improve access the neighbors of a vertex?*

Adjacency Tables. In *adjacency table*, representation, we keep a table that maps every vertex to the set of its neighbors. This is simply an edge-set table.

Question 1.10. *What if we want to associate data with the edges?*

If we want to associate data with each edge, we can use an edge-table table, where the table associated with a vertex v maps each neighbor u to the value on the edge (u, v).

Question 1.11. *What is the cost of accessing the neighbors of a vertex?*

In this representation, accessing the neighbors of a vertex v is cheap: it just requires a lookup in the table. Assuming the balanced-tree cost model for tables, this can be done in $O(\log n)$ work and span.

Question 1.12. *Can you give an algorithm for finding an edge (u, v)?*

We can first pull out the adjacency set for u and then look up for v by using a filter. Looking for v in the adjacency set requires $O(d_G(v))$ work and $O(\log d_G(v))$ span. In total, looking up if an edge is in the graph requires the same work and span as with edge sets: $O(\log n)$.

Note that in general, once the neighbor set has been pulled out, we can apply a constant work function over the neighbors in $O(d_G(v))$ work and $O(\log d_G(v))$ span.

Cost Summary. For these two representations the costs assuming the tree cost model for sets and tables can be summarized in the following table. This assumes the function being mapped uses constant work and span.

	edge set		**adj table**	
	work	*span*	*work*	*span*
`isEdge`$(G,(u,v))$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
`map over all edges`	$O(m)$	$O(\log n)$	$O(m)$	$O(\log n)$
`map over neighbors of` v	$O(m)$	$O(\log n)$	$O(\log n + d_G(v))$	$O(\log n)$
$d_G(v)$	$O(m)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Properties of Graphs. Given a graph, we are often interested in finding out certain things about the graph or its vertices.

Question 1.13. *Can you remember some properties that you have talked about?*

You have learned relationships such as, reachability, and connectivity. Let's remind us about reachability and connectivity.

Definition 1.14. 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 1.15. A (directed) graph is *(strongly) connected* if there is a (directed) path between every pair of vertices.

Question 1.16. *Is reachability a relation itself?*

Reachability indeed 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 1.17. *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*.

Vertex hopping. Before we start learning about 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 it offers a good starting point for solving many problems.

Algorithm 1.18 (Vertex hopping)**.** *Repeat until all vertices are visited: pick an (or some) unvisited vertex (vertices), visit that vertex, as well all the edges coming out of that vertex.*

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

Question 1.19. *When the algorithm terminates, does it visit all the edges?*

Every edge (u, v) comes out of a vertex u and thus will be visited when visiting u. Since the algorithm visits all the vertices, all the edges will be visited by the algorithm when it terminates.

Question 1.20. *Given that when the algorithm terminates, it visits all the vertices and edges, can you solve the reachability problem using vertex hopping?*

We can. 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 1.21. *Is this algorithm efficient?*

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 1.22. *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?*

Graph Search. Since the nondeterministic algorithm jumps around the graph every time it visits a vertex, it does not reveal the global structure of the graph. In particular, it does not follows paths in the graph. As we will see in later parts of this course, 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.

For properties such as reachability, we use a different technique variously called *graph search* or *graph reachability* 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 1.23. *Can you think of a way of refining vertex hopping to yield an algorithm for graph search?*

What we need is a way to determine the next vertex to jump to in order to traverse paths rather than (possibly) unrelated edges. To this end we can maintain a set of vertices that we will call a *frontier*.

Definition 1.24. A vertex v is in the frontier if v is not yet visited but but is connected via one edge (u, v) to a visited vertex u.

We can then state graph search as follows.

Algorithm 1.25 (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. We can update our definition to handle this corner case but we will not, for the sake of simplicity.

Question 1.26. *Can you explain why this algorithm follows paths rather than unrelated edges?*

The algorithm always picks a vertex that is one hop (away) from the frontier. Since frontier contains only the visited vertices, the algorithm follows paths.

Question 1.27. *How can a graph-search algorithm can determine that all vertices are visited?*

One way is to check the number of visited vertices is equal to the number of vertices in the graph (the cardinality of the visited set). A better way is to check that the frontier is not empty. These approaches are identical if all the vertices are reachable from the source, if not, then they do behave differently. We will use the version that terminates when the frontier is empty.

Algorithm 1.28 (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, $F = F \setminus X$.

Breadth-First Search (BFS) algorithm. The BFS algorithm is a graph-search algorithm that picks the next set of vertices to visit such that the vertices are visited in a breadth-first order, that is in order of increasing hop distances from the source vertex.

To see what we mean by "breadth-first", let's 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. Figure 2 shows an example.

Question 1.29. *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 so that it has information about the levels of the vertices by tagging each vertex with its level. We can then visit vertices in level order.

Question 1.30. *How can we determine the set of vertices at level $i + 1$?*

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

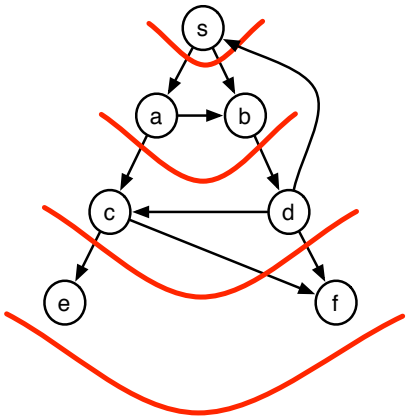


Figure 2: A graph and its levels.

Question 1.31. *Can we thus say that the set of vertices that are at level $i+1$ are those that are reachable from level-i vertices by one hop?*

Not quite because we may also reach vertices that are at levels i or less. (Note that we cannot reach vertices at level $i+2$ or greater.)

Question 1.32. *Can you see when this happens in the example in Figure 2 ?*

In the example in Figure 2, this happens when considering the edges (a,b) and (d,s).

Question 1.33. *How can we find and eliminate such vertices?*

Since we are visiting vertices in increasing levels, we must have visited all such vertices. Thus if we exclude visited vertices, then vertices at level $i+1$ are those that are reachable from level-i vertices by one hop.

The BFS-algorithm can thus be described as follows.

Algorithm 1.34 (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 1.35. *We can simplify this algorithm a bit. Can you see how?*

Note that vertices added to the frontier 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 breath first algorithm can thus be simplified by throwing away the frontier information.

Algorithm 1.36 (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, $F = F \setminus X$.

This gives us the following pseudo code.

Pseudo Code 1.37 (BFS)**.**

```
1   fun BFS(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     fun BFS'(X, F, i) =
7       if |F| = 0 then (X, i)
8       else let
9           X' = X ∪ F          % Visit the Frontier
10          N = N_G(F)          % Determine the neighbors of the frontier
11          F' = N \ X'         % Remove vertices that have been visited
12      in BFS'(X', F', i+1) % Next level
13      end
14  in BFS'({}, {s}, 0)
15  end
```

Question 1.38. *Do you recall the cost specification of this algorithm?*

Cost Specification 1.39 (Cost specification for BFS)**.** *For BFS, we have the following cost specification.*

- *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.

- *Using single-threaded sequences, we can reduce the total work to $O(m)$ total work and span to $O(d\log n)$.*

BFS Extensions: reachability. You can use BFS to compute various properties of graphs. Let's for example go back to our problem that originally motivated this algorithm, reachability.

Question 1.40. *Can you extend the BFS algorithm to determine whether a vertex v is reachable from another vertex u?*

Yes, we can do so by starting BFS at u (pick u as the source) and stopping as soon as we reach v or we visit all the vertices in the graph.

Exercise 2. *Give the pseudocode for a reachability algorithm using BFS.*

BFS Extensions: shortest distance. We sometimes want to know the (shortest) 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.

Question 1.41. *Can you think of a way of extending the BFS algorithm to find the shortest distances from the source to all the vertices reachable from the source?*

We can extend the algorithm to return us a table mapping vertices to their distances. When we visit the frontier at level i, we insert the vertices visited to the table with distance i.

Exercise 3. *Give the pseudocode for a shortest-distances algorithm using BFS.*

Summary 1.42. *A few important points that we have so far reviewed.*

- *Graphs succinctly and efficiently represent (mathematical) relationships. Since relationships are often naturally very interesting, graphs find many applications.*
- *Some properties of graphs that we will often talk about include reachability, connectivity, and the shortest distance between two vertices.*
- *The vertex hopping algorithm is a simple algorithm for exploring graphs that works by repeatedly picking an arbitrary (set of) vertex (vertices) to visit next. The algorithm does not reveal much about the global structure of the graph. It is therefore difficult to use this algorithm to compute properties such as reachability.*
- *A graph search algorithm is specialization of vertex-hopping that visits vertices that are one-hop away adjacent to visited vertices.*
- *Since graph search algorithms follow paths, they are helpful in computing many interesting properties of graphs.*
- *The BFS algorithm is a graph-search algorithm that visits vertices level by level, in increasing hop-distances from a specified source. It specializes the generic graph search algorithm by visiting all vertices in the frontier in parallel. It can be extended (used) to compute various properties of a graph such as reachability, shortest distances etc.*

2 Depth-First Search

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

Question 2.1. *Do you think that the BFS algorithms sufficient to compute all properties of graphs?*

BFS is indeed effective for many problems but another algorithm called *depth-first seach* 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.

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 the task of wearing a harness and tying into the rope.

Figure 3 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.

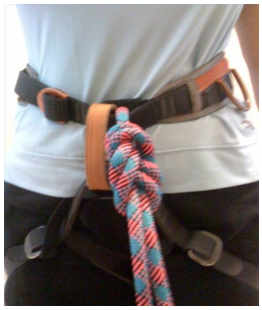


Figure 3: A simplified DAG for tying into a rope with a harness.

Question 2.2. *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.

Definition 2.3 (Directed Acylic Graph (DAG))**.** A directed acylic 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 2.4 (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 2.5. *Can you come up with a topological ordering of the climbing DAG shown in Figure 3.*

There are many possible topological orderings. 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.

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 4.

Question 2.6. *Would this change the set of valid topological sortings for the DAG?*

Since all new edges originate at the start vertex, any valid topological ordering of the original DAG can be converted into a valid topological ordering of the new dag by preceeding it with the start vertex.

Question 2.7. *Can you think of an algorithm for finding such a topological sort by using the vertex-hopping algorithm?*

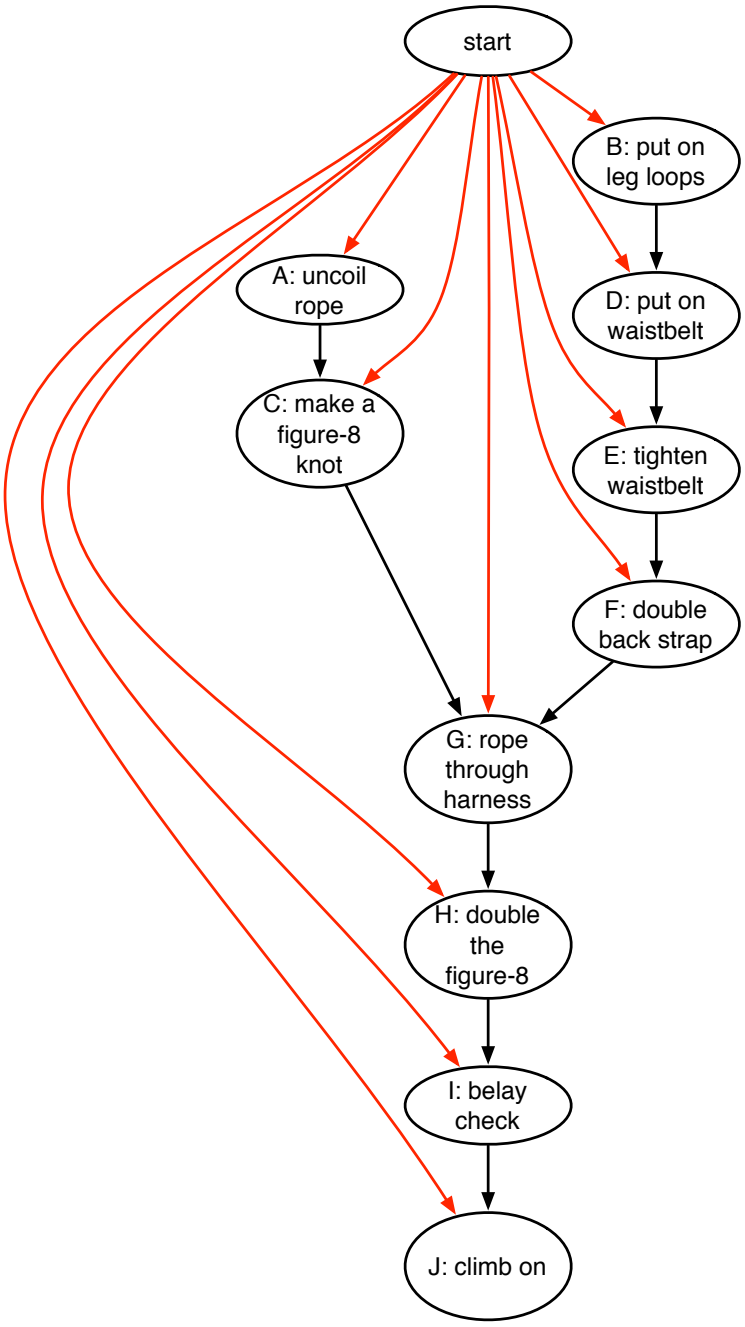


Figure 4: Climber's DAG with a start node.

We just have to make sure that we don't visit a vertex before we visit all its parents.

Question 2.8. *Can you think of an algorithm for selecting the next vertex to visit?*

There are several possibilities. 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.

Before we move on the DFS, one last question:

Question 2.9. *Can you see why the breadth-first search is not as natural for topological sorting?*

Recall that in BFS, we visit a vertex in the quickest possible way, because BFS visits vertices in level (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.

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).

Question 2.10. *Can you come up with an algorithm for determining the next vertex to visit?*

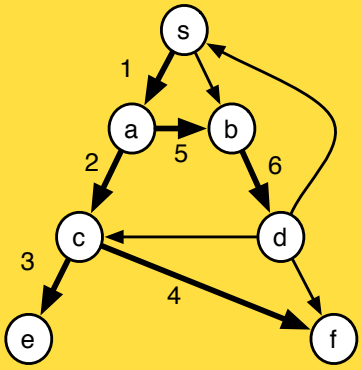
One straightfarward way is to simply time-stamp the vertices as we add to the frontier and simply remove the most recently inserted vertex. Maintaining the frontier as a sequence data structure that maintains the vertices in the frontier in an stack order (LIFO: last in first out), achieves the same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

Algorithm 2.11 (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 botton 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 4. *Convince yourself that the pebbling-based description and the DFS algorithm described above are identical.*

Example 2.12. *An example DFS.*



Question 2.13. *How does the order in which DFS visits vertices in this example differ from that of BFS?*

DFS does not visit vertices in order of their levels.

Question 2.14. *Can you think of why DFS might be more effective in solving some problems that BFS cannot.*

Since in DFS we traverse the graph sequentially, we can collect more information about the graph: at any point, you know everything about the vertices that you have visited thus far and the edges that you have taken.

DFS Numberings. 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 numberings cumulatively as *DFS numberings*. Figure 5 shows an example.

DFS numberings have interesting properties.

Exercise 5. *Can you determine by just using the DFS numbering for a graph whether a vertex is an ancestor or a descendant of another vertex?*

Question 2.15. *Can you solve give a topological sort of the graph using the DFS numberings?*

In a DAG, the sort of the vertices from the larges to the smallest finishing time yield a topological ordering of the DAG.

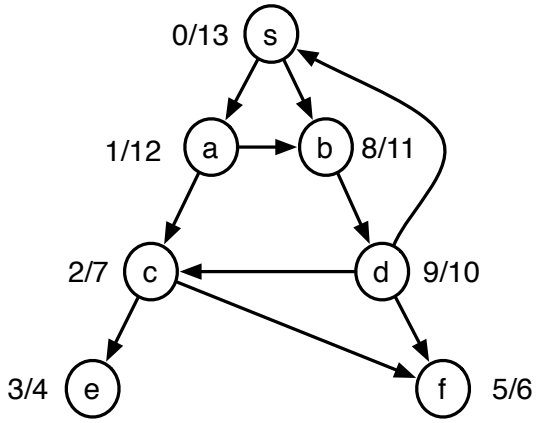


Figure 5: A graph and its DFS numbering; t_1/t_2 denotes the timestamps showing when the vertex gets its white (discovered) and red pebble (finished) respectively.

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 2.16. 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.

Question 2.17. *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.

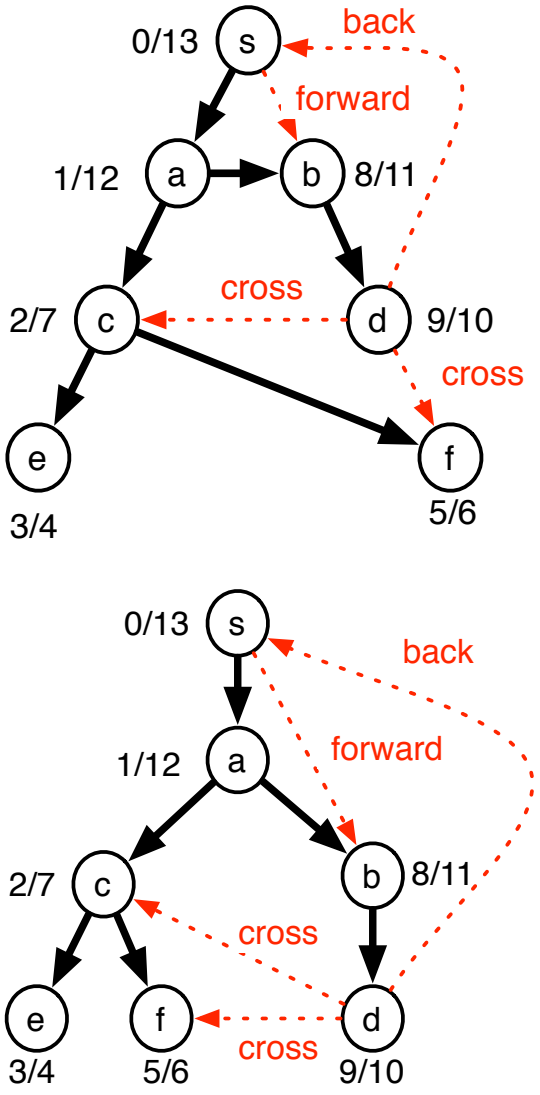


Figure 6: Tree edges (black), and non-tree edges (red, dashed) illustrated.

Pseudo Code 2.18 (DFS). *Here is the pseudo-code of DFS.*

```
             :  fun DFS(G, s) = let
             :    fun DFS'(X, v) =
             :      if (v ∈ X)
  TOUCH v    :      then X
             :      else let
  ENTER v    :          X' = X ∪ {v}
             :          X'' = iter DFS' X' (N_G(v))
    EXIT v   :      in X'' end
             :  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. Otherwise it visits the vertex v by adding it to X (line ENTER v), iterating itself recursively on all neighbors, and finally returning the updated set of visited vertices (line EXIT v).

Recall that (`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 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.

Question 2.19. *Can we use BFS to find the shortest paths?*

Unlike BFS, 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 6. *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 2.20. *Is* DFS *a parallel algarithm? 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 Figure 5, if we search from the outedges on s in parallel, we would visit the vertex b twice. More generally we would visit the vertices b, c, f multiple times.

Remark 2.21. 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 2.22. *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 2.23 (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 $O(n + m)$.*

3 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

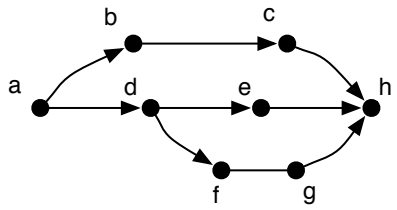
1. reflexivity — $a \leq_p a$,
2. antisymmetry — if $a \leq_p b$ and $b \leq_p a$, then $b = a$, and
3. transitivity — if $a \leq_p b$ and $b \leq_p c$ then $a \leq_p c$.

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

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:

Problem 3.1 (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:

```
                :  fun topSort(G = (V, E)) = let
                :      s = a new vertex
                :      G' = (V ∪ {s}, E ∪ {(s, v) : v ∈ V})
                :
                :    fun DFS'((X, L), v) =
                :      if (v ∈ X)
TOUCH v         :      then (X, L)
                :      else let
ENTER v         :          X' = X ∪ {v}
                :          (X'', L') = iter DFS' (X', L) (N_G'(v))
EXIT v          :      in (X'', v :: L') end
                :
                :  in DFS'(({}, []), s) end
```

The significant changes from the generic version 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.

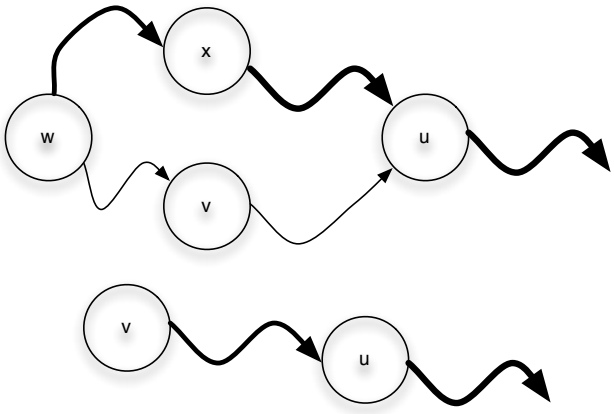


Figure 7: The two cases in the Proof of Theorem 3.2

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 3.2. *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. This is shown in the top case in Figure 7.[2]
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 case is show at the bottom of Figure 7. □

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.

4 Cycle Detection: 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

[2]Bold lines indicate the path for DFS.

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.

```
        : fun undirectedCycle(G, s) = let
        :   fun DFS' p ((X, C), v) =
        :     if (v ∈ X)
TOUCH v :     then (X, true)
        :     else let
ENTER v :       X' = X ∪ {v}
        :       (X'', C') = iter (DFS' v) (X', C) (N_G(v) \ {p})
 EXIT v :     in (X'', C') end
        : in DFS' s ((∅, 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 7. *In the final line of the code the initial "parent" is the source s itself. Why is this OK for correctness?*

5 Cycle Detection: 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:

```
:  fun directedCycle(G = (V, E)) = let
:      s = a new vertex
:      G' = (V ∪ {s}, E ∪ {(s, v) : v ∈ V})
:    fun DFS'((X, Y, C), v) =
:      if (v ∈ X)
TOUCH v :      then (X, Y, (v ∈? Y or C))
:      else let
ENTER v :        X' = X ∪ {v}
:        Y' = Y ∪ {v}
:        (X'', Y'', C') = iter DFS' (X', Y', C) (N_G'(v))
EXIT v :      in (X'', Y'' \ {v}, C') end
:    (_, _, C) = DFS'(({}, {}, false), s)
: 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 5.1. *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 8. *Prove this theorem.*

5.1 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 three functions:

```
Σ0    : α
touch : α × vertex × vertex → α
enter : α × vertex × vertex → α
exit  : α × vertex × vertex → α
```

where each function takes the state, the current vertex v, and the parent vertex p in the DFS tree, and returns an updated state. The code can then be written as:

```
1  fun DFS(G, Σ0, s) = let
2    fun DFS' p ((X, Σ), v) =
3      if (v ∈ X) then (X, touch(Σ, v, p))
4      else let
5          Σ' = enter(Σ, v, p)
6          (X', Σ'') = iter (DFS'p) (X ∪ {v}, Σ') N_G^+(v)
7          Σ''' = exit(Σ'', v, p)
8      in (X', Σ''') end
9  in DFS' s (({}, Σ0), s) end
```

At the end, DFS returns an ordered pair $(X, \Sigma) : \texttt{Set} \times \alpha$, which represents the set of vertices visited and the final state Σ.

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

$\Sigma_0 = ([s],\ \texttt{false}) : \texttt{vertex list} \times \texttt{bool}$
fun touch$((h :: T,\ \texttt{fl}),\ v,\ p) = (L,\ h \neq p)$
fun enter$((L,\ \texttt{fl}),\ v,\ p) = (v :: L,\ \texttt{fl})$
fun exit$((h :: T,\ \texttt{fl}),\ v,\ p) = (T,\ \texttt{fl})$

For topological sort we have.

$\Sigma_0 = [\,] : \texttt{vertex list}$
fun touch$(L,\ v,\ p) = L$
fun enter$(L,\ v,\ p) = L$
fun exit$(L,\ v,\ p) = v :: L$

For directed cycle detection we have.

$\Sigma_0 = (\{\},\ \texttt{false}) : \texttt{Set} \times \texttt{bool}$
fun touch$((S,\ \texttt{fl}),\ v,\ p) = (S,\ v \in^? S)$
fun enter$((S,\ \texttt{fl}),\ v,\ p) = (S \cup \{v\},\ \texttt{fl})$
fun exit$((S,\ \texttt{fl}),\ v,\ p) = (S \setminus \{v\},\ \texttt{fl})$

For these last two cases we need to also augment the graph with the vertex s and add the the edges to each vertex $v \in V$.

6 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.

```
 1  fun DFS(G : (int seq) seq, s : int) =
 2  let
 3    fun DFS′ p ((X : bool stseq, Σ), v : int) =
 4      if (X[v]) then (X, touch(Σ, v, p))
 5      else let
 6          X′ = update(v, true, X)
 7          Σ′ = enter(Σ, v, p)
 8          (X″, Σ″) = iter (DFS′v) (X′, Σ′) (G[v])
 9      in (X″, exit(Σ″, v, p))
10     X_init = stSeq.fromSeq(⟨false : v ∈ ⟨0, ..., |G|−1⟩⟩)
11  in
12    stSeq.toSeq(DFS′((X_init, Σ_0), s))
13  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.

7 SML Code

Here we give the SML code for the generic version of `DFS` along with the implementation of directed cycle detection and topological sort.

```
signature DFSops =
sig
  type vertex
  type state
  val start : state
  val enter : state * vertex * vertex -> state
  val exit : state * vertex * vertex -> state
  val touch : state * vertex * vertex -> state
end

functor DFS(structure Table  : TABLE
            structure Ops : DFSops
            sharing type Ops.vertex = Table.Key.t) =
struct
  open Ops
  open Table
  type graph = set table

  fun N(G : graph, v : vertex) =
    case (Table.find G v) of
        NONE => Set.empty
      | SOME(ngh) => ngh

  fun dfs (G : graph, s : vertex) =
    let
      fun dfs' p ((X : set, S : state), v : vertex) =
```

```
        if (Set.find X v) then (X, touch(S,p,v))
        else
          let
            val X' = Set.insert v X
            val S' = enter(S, p, v)
            val (X'',S'') = Set.iter (dfs' p) (X',S') (N(G,v))
            val S''' = exit(S'', p, v)
          in (X'',S''')
          end
    in
       dfs' s ((Set.empty, start), s)
    end
end


functor CycleCheck(Table : TABLE) = DFS(
  structure Table = Table
  structure Ops =
  struct
    structure Set = Table.Set
    type vertex = Table.key
    type state = Set.set * bool
    val start = (Set.empty,false)
    fun enter((S,fl),p,v) = (Set.insert v S, fl)
    fun exit((S,fl),p,v) = (Set.delete v S, fl)
    fun touch((S,fl),p,v) = if (Set.find S v)
                            then (S,true)
                            else (S,fl)
  end)


functor TopSort(Table : TABLE) = DFS(
  structure Table = Table
  structure Ops =
  struct
    type vertex = Table.key
    type state = vertex list
    val start = []
    fun enter(L,_,_) = L
    fun exit(L,p,v) = v::L
    fun touch(L,_,_) = L
  end)
```