Lecture 11 — Depth-First Search and Applications

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

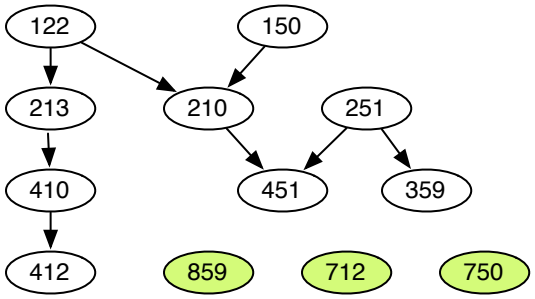
Lectured by Guy Blelloch — October 2, 2012

What was covered in this lecture:

- Depth-first Search
- Using DFS for Cycle Detection (undirected and directed graphs) and Topological Sort

1 A Toy Example

Alice is an ambitious freshman who wants to be well-versed in both theory and systems. She plans to take the following classes during her undergraduate career: 15-122, 15-150, 15-210, 15-213, 15-251, 15-359, 15-451, 15-410, 15-412, 15-750, 15-859, and 15-712. But she knows that she should only take one CS class per semester—and most classes have prerequisites that prevent her from getting in right away. According to her research, the course catalog indicates the following prerequisite structure, depicted as a directed graph:



For example, she cannot take 15-451 Algorithms Design and Analysis until she is done with 15-210 and 15-251—although she could take 15-859 Advanced Algorithms or 15-712 Advanced and Distributed Operating Systems in her first semester (after all, most graduate classes don't have any a formal prerequisite list).

We would like to help her *construct a schedule to take exactly one CS class per semester*. This is known more formally as the *topological sort* problem or simply TopSort. But before we try to generate a schedule for her, we might be interested in finding out whether the graph has a cycle. In a more general context, what we have is a dependency graph and a cycle in a dependency graph indicates a deadlock. For Alice, this would mean such a schedule doesn't exist and she will never graduate unless she drops some of the classes. In this lecture, we will also look at the cycle detection problem for both directed and undirected graphs.

For both problems, we will develop an algorithm using a graph traversal idea called *depth-first search* that looks at an edge at most twice!

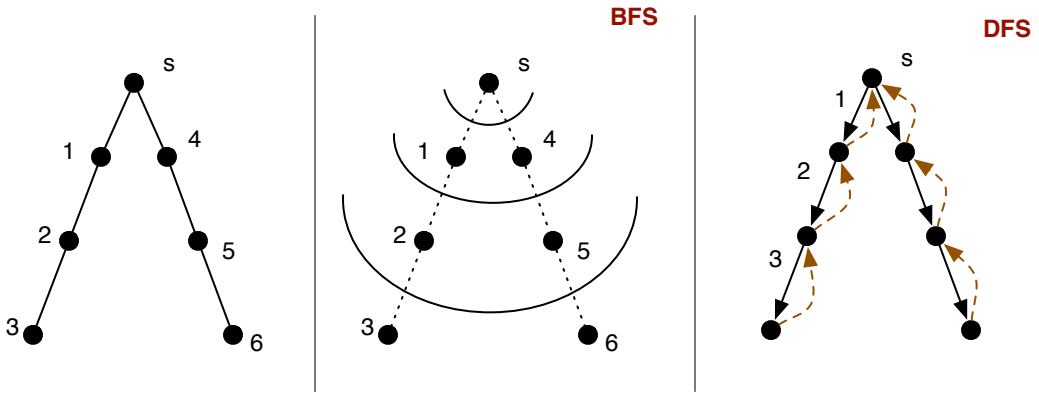
†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

2 DFS: Depth-First Search

Last time, we looked at breadth-first search (BFS), a graph search technique which, as the name suggests, explores a graph in the increasing order of hop count from a source node. We'll spend the bulk of this lecture discussing another equally-common graph search technique, known as depth-first search (DFS). Unlike BFS which explores vertices one level at a time in a breadth first manner, the depth-first search approach proceeds by going as deep as it can until it runs out of unvisited vertices, at which point it backs out until it finds a node with an unvisited neighbor and goes there in the same manner.

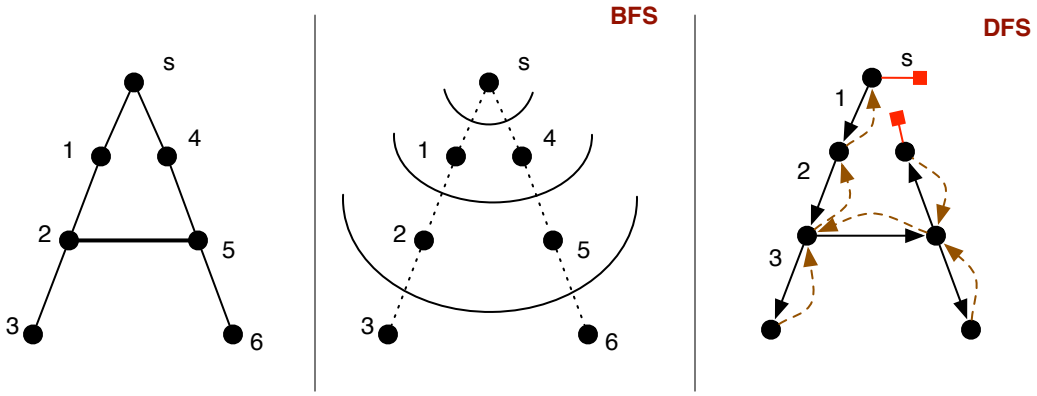
As with BFS, DFS can be used to find all vertices reachable from a start vertex v, to determine if a graph is connected, or to generate a spanning tree. Unlike BFS, it cannot be used to find shortest unweighted paths. But, instead, it is useful in some other applications such as topologically sorting 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.

Example 2.1. *To contrast BFS with DFS, let's consider the following simple V-shaped graph:*



In this example, a BFS starting from s will first visit s, then visit vertices at distance 1 *from s (*1 *and* 4*), then vertices at distance* 2 *from s (*2 *and* 5*), then vertices at distance* 3 *away (*3 *and* 6*)–and we're done. Whereas, a DFS starting form s will go as deep as it can: at s, we could go to* 1 *or* 4 *as the first vertex. Suppose we choose* 1*, then we will proceed to visit* 2 *and* 3*, then after hitting a dead end, we back out to* 2 *then back to* 1*, and proceed down* 4 *to* 5 *to* 6 *and back out.*

Example 2.2. *As another example, we'll look at a slightly more complicated version of the graph above, where we join the nodes* 2 *and* 5 *together with an edge.*



Nothing changes when we run BFS despite that extra edge between 2 *and* 5*; however, the order in which DFS visits vertices changes drastically. Suppose we start at s and choose to go down node* 1 *first. We will visit* 2*,* 3*, then back out to* 2*, then since* 2 *has* 5 *as its neighbor, we'll visit* 5*, which in turn, will take us to* 4 *and to s, but s has been visited before, so we won't visit that—we back out. We'll then go to* 5*,*6*, back to* 5*, to* 2*, to* 1*, to s. We will try to visit* 4 *but quickly realize that* 4 *has been visited, so again, we back out.*

How do we turn this idea into code? We first consider a simple version of depth-first search that simply returns a set of reachable vertices. Notice that in this case, the algorithm returns exactly the same set as BFS—but the crucial difference is that DFS visits the vertices in a different order (depth vs. breadth). Here is the code:

```
        :  fun DFS(G, s) = let
        :    fun DFS'(X, v) =
        :      if (v ∈ X)
TOUCH v :      then X
        :      else let
ENTER v :        val X' = X ∪ {v}
        :        val 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 a single vertex 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 $(\texttt{iter}\ f\ s_0\ A)$ 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.

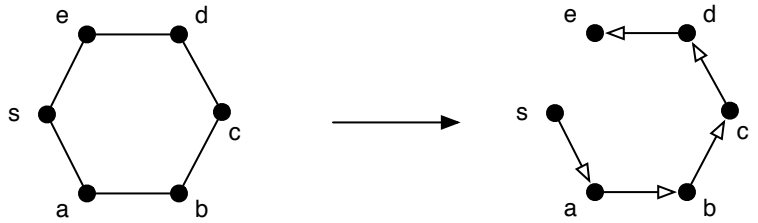
What this means for the DFS algorithm is that when the algorithm visits a vertex v (i.e., reaches the line ENTER v), it picks the first outgoing edge (v, w_1), through `iter`, calls `DFS'`$(X \cup \{v\}, w_1)$ to explore the unvisited vertices reachable from w_1. When the call `DFS'`$(X \cup \{v\}, w_1)$ returns the algorithm has fully explored the graph reachable from w_1 and the vertex set returned (call it X_1) includes all vertices in the input $X \cup v$ plus all vertices reachable from w_1. The algorithm then picks the next edge (v, w_2), again through `iter`, and fully explores the graph reachable from w_2 starting with the the vertex set X_1. The algorithm continues in this manner until it has fully explored all out-edges of v. At this point, `iter` is complete—and X'' includes everything in the original $X' = X \cup \{v\}$ as well as everything reachable from v.

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 1. *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.*

Is `DFS` parallel? At first look, we might think this approach 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.

To understand the situation better, we'll look at a simple example: a cycle graph on 6 nodes (C_6). The left figure shows a 6-cycle C_6 with nodes s, a, b, c, d, e and the right figure shows the order (as indicated by the arrows) in which the vertices are visited as a result of starting at s and first visiting a.



In this example, since the search wraps all the way around, we couldn't know until the end that e would be visited (in fact, it got visited last), so we couldn't start searching s's other neighbor e until we are done searching the graph reachable from a. More generally, in an undirected graph, if two

unvisited neighbors u and v have any reachable vertices in common, then whichever is explored first will always wrap all the way around and visit the other one.

Indeed, 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 2.3. *For a graph $G = (V, E)$ with m out 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. Every vertex will be visited at most once since we always add them to X when we enter so the the test $v \in X$ can only fail n times. Every out-edge will only be traversed once invoking a call to `DFS'` since its vertex is only visited once, so at most m calls will be made to `DFS'`. Finally we can only enter a vertex once per call to `DFS'` so the number of enters is bounded by $\min(n, m)$. □

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

Corollary 2.4. *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 $O(m)$.

3 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 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 :        val X' = X ∪ {v}
        :        val (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 2. *In the final line of the code the initial "parent" is the source s itself. Why is this OK for correctness?*

4 Topological Sorting

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

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

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

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

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

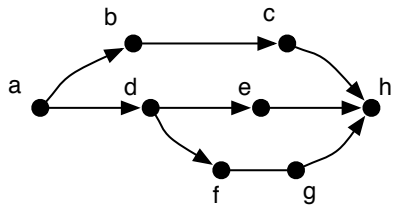
Remember that a partial order is a relation $\leq_p$ that obeys

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

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

Problem 4.1 (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
        :   val s = a new vertex
        :   val 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 :       val X' = X ∪ {v}
        :       val (X'', L') = iter DFS' (X', L) (N_G'(v))
 EXIT v :     in (X'', v :: L') end
        :
        : in DFS'(({}, []), s) end
```

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

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.

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 4.2. *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.
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.

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
          :    val s = a new vertex
          :    val G' = (V ∪ {s}, E ∪ {(s, v) : v ∈ V})
          :
          :    fun DFS'((X, Y, C), v) =
          :      if (v ∈ X)
TOUCH v   :      then (X, Y, v ∈? Y)
          :      else let
ENTER v   :        val X' = X ∪ {v}
          :        val Y' = Y ∪ {v}
          :        val (X'', Y'', C') = iter DFS' (X', Y', C) (N_G'(v))
EXIT v    :      in (X'', Y'' \ {v}, C') end
          :
          :    val (_, _, 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 3. *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:

$$
\begin{array}{ll}
\Sigma_0 & : \alpha \\
\texttt{touch} & : \alpha \times \texttt{vertex} \times \texttt{vertex} \to \alpha \\
\texttt{enter} & : \alpha \times \texttt{vertex} \times \texttt{vertex} \to \alpha \\
\texttt{exit} & : \alpha \times \texttt{vertex} \times \texttt{vertex} \to \alpha
\end{array}
$$

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        val Σ' = enter(Σ, v, p)
6        val (X', Σ'') = iter (DFS'p) (X ∪ {v}, Σ') N_G^+(v)
7        val Σ''' = exit(Σ, 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 $\texttt{touch}((h :: T,\ \texttt{fl}),\ v,\ p) = (L,\ h \neq p)$
fun $\texttt{enter}((L,\ \texttt{fl}),\ v,\ p) = (v :: L,\ \texttt{fl})$
fun $\texttt{exit}((h :: T,\ \texttt{fl}),\ v,\ p) = (T,\ \texttt{fl})$

For topological sort we have.

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

For directed cycle detection we have.

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

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        val X' = update(v, true, X)
7        val Σ' = enter(Σ, v, p)
8        val (X'', Σ'') = iter (DFS'v) (X', Σ') (G[v])
9      in (X'', exit(Σ'', v, p))
10   val 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)
```