CS2 Algorithms and Data Structures Note 10

Depth-First Search and Topological Sorting

In this lecture, we will analyse the running time of DFS and discuss a few applications.

10.1 A recursive implementation of DFS

In the last lecture, we have seen an iterative implementation of DFS that used a stack to store the vertices that have been visited, but not yet fully processed. Algorithms 10.1 and 10.2 show a recursive implementation of DFS. It is somewhat closer to our intuitive understanding of depth-first search: To visit all vertices reachable from the start vertex v, it visits v, then the first neighbour of v and all vertices reachable from this first neighbour — remember that the search is *depth-first*. Then it visits the next neighbour and all vertices reachable from it, except those that have already been visited before, et cetera. If we write this down in pseudo-code, we end up with Algorithms 10.2. It is not really surprising that we can replace the recursive implementation by one using a stack, because after all stacks are used by the compiler to implement recursion.

Algorithm dfs(G)

1. Initialise Boolean array $visited$ by setting all entries to FALSE
2. **for all** $v \in V$ **do**
3. **if** $visited[v] =$ FALSE **then**
4. dfsFromVertex(G, v)

Algorithm 10.1

Algorithm dfsFromVertex(G, v)

1. $visited[v] \leftarrow$ TRUE
2. **for all** w adjacent to v **do**
3. **if** $visited[w] =$ FALSE **then**
4. dfsFromVertex(G, w)

Algorithm 10.2

Let us analyse the running time of dfs. At first sight, it seems quite complicated because of the recursive calls inside the loops. Rather than writing down a recurrence for the running time of dfs, let us try to analyze it directly. Let n be the number of

vertices of the input graph G and m the number of edges. Then dfs(G) requires time $\Theta(n)$ for initialisation. Moreover, dfsFromVertex(G, v) requires time $\Theta(\text{out-degree}(v))$, because the loop is iterated out-degree(v) times.

Now the crucial observation is that dfsFromVertex(G, v) is invoked exactly once for every vertex v. To see that it is invoked *at least* once, note that $visited[v]$ is set to TRUE only if dfsFromVertex(G, v) is invoked. So if the method was never invoked, then $visited[v]$ would remain FALSE. But this cannot happen, because for all v with $visited[v] =$ FALSE, dfsFromVertex(G, v) is invoked in Line 4 of dfs(G) (in the v-execution of the loop). To see that dfsFromVertex(G, v) is invoked *at most* once, note that it is only invoked if $visited[v] =$ FALSE. However, after its first execution $visited[v]$ is TRUE and can never become FALSE again. Thus indeed dfsFromVertex(G, v) is invoked exactly once for every vertex v.

Therefore, we get the following expression for the running time of dfs(G):

$$\Theta(n) + \sum_{v \in V} \Theta(\text{out-degree}(v)) = \Theta\Big(n + \sum_{v \in V} \text{out-degree}(v)\Big)$$

Let m be the number of edges of G. Then $\sum_{v \in V} \text{out-degree}(v) = m$, and we get

$$T_{\text{dfs}}(n, m) = \Theta(n + m).$$

Note that we count an undirected edge as two edges, one in each direction. We then get $\sum_{v \in V} \text{degree}(v) = 2m$ for undirected graphs, but since $2m \in \Theta(m)$, this does not really make a difference.

10.2 DFS Forests

Note that a DFS starting at some vertex v explores the graph by building up a *tree* that contains all vertices that are reachable from v and all edges that are used to reach these vertices. We call this tree a *DFS tree*. A complete DFS exploring the full graph (and not only the part reachable from a given vertex v) builds up a collection of trees, or forest, called a *DFS forest*.

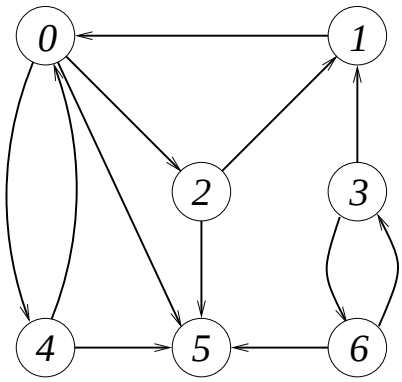

Figure 10.3.

Suppose, for example, that we explore the graph in Figure 10.3 by a DFS starting at vertex 0 that visits the vertices in the following order: $0, 2, 1, 5, 4, 3, 6$. The corresponding DFS forest is shown in Figure 10.4.

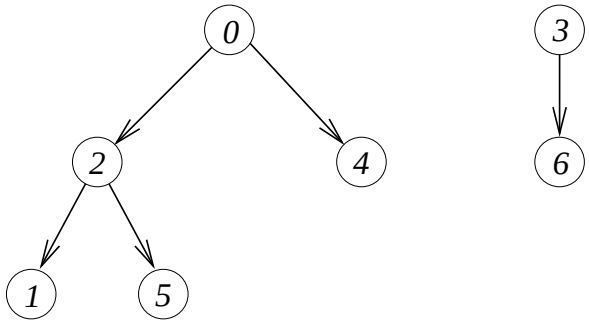

Figure 10.4.

Note that, just like the order in which the vertices are visited during a DFS, a DFS forest is not unique. Figure 10.5 shows another DFS forest for the graph in Figure 10.3.

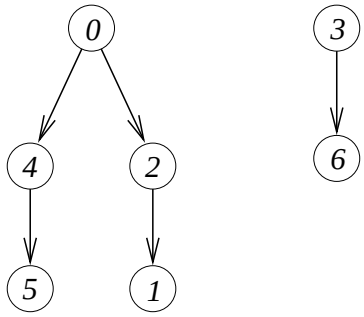

Figure 10.5.

10.3 Connected components

As a first application of DFS, we want to compute the connected components of an undirected graph. Recall the definition of connected components first. Let $G = (V, E)$ be an undirected graph. A subset C of V is *connected* if for all $v, w \in C$ there is a path from v to w. (This includes the case that $v = w$ because in this case there's a path of length 0 from v to w.) A *connected component* of an undirected graph G is a maximum connected subset C of V. Here "maximum connected subset" means that there is no connected subset C' of V that strictly contains C, and not that C is a connected subset of V with the maximum number of vertices. An undirected graph is *connected* if it only has one connected component, that is, if for all vertices v, w there is a path from v to w.

Our algorithm is based on the following two simple observations:

(1) Each vertex of an undirected graph is contained in exactly one connected component.

(2) For each vertex v of an undirected graph, the connected component that contains v is precisely the set of all vertices that are reachable from v.

Thus, by (2), dfsFromVertex(G, v) visits exactly the vertices in the connected component of v.

We modify dfs as follows: We add a statement **print** v after line 1 of dfsFromVertex(G, v). After this modification, dfsFromVertex(G, v) prints exactly the vertices in the connected component of v. Then we add a statement **print** "New Component" before each call of dfsFromVertex(G, v) in line 4 of dfs. The modified algorithm will print "New Component", followed by the vertices of a connected component, and repeat that until all components have been printed.

Of course, depending on what we want to do with the components, we may modify the algorithm in such a way that it returns the connected components in some convenient format. For example, it may put the vertices of each component into a linked list and return a linked list of these linked lists.

The asymptotic running time of this algorithm for computing the connected components is clearly the same as the running time of dfs, i.e., $\Theta(n + m)$.

Components of directed graphs

It is not so clear what connectivity means in directed graphs. For example, is the graph in Figure 10.3 connected? It looks "joined up", but then vertex 6 is not reachable from vertex 0. We say that a directed graph G is *weakly connected* if the undirected graph we obtain from G by disregarding the direction of the edges is connected. A directed graph is *strongly connected* if for all vertices v, w there is a path from v to w and a path from w to v. Strong connectivity is the more important notion. The graph in Figure 10.3 is weakly connected, but not strongly connected; for example, there is no path from 0 to 6.

Derived from the notions of weak and strong connectivity, we have *weakly connected components* and *strongly connected components*. For example, the digraph in Figure 10.3 only has one weakly connected component (containing all vertices), and it has three strongly connected components:

$$0, 1, 2, 4$$
$$3, 6$$
$$5.$$

Computing the weakly connected components of a directed graph is easy: writing an algorithm that does this is a good exercise. Computing the strongly connected components is much harder. It can also be done by an algorithm based on DFS, but this application of DFS is much more sophisticated than those discussed in CS2.

10.4 Classifying vertices during a DFS

Let G be a graph. Recall that during an execution of dfs(G), the subroutine dfsFromVertex(G, v) is invoked exactly once for each vertex v. Let us call vertex v *finished* after

dfsFromVertex(G, v) is completed. During the execution of dfs(G), a vertex can be in three states:

- not yet visited (let us call a vertex in this state *white*),
- visited, but not yet finished (*grey*).
- finished (*black*).

We can modify our DFS algorithm so that it keeps track of the states of the vertices, see Algorithms 10.6 and 10.7.

Algorithm dfs(G)

1. Initialise array $state$ by setting all entries to $white$.
2. **for all** $v \in V$ **do**
3. **if** $state[v] = white$ **then**
4. dfsFromVertex(G, v)

Algorithm 10.6

Algorithm dfsFromVertex(G, v)

1. $state[v] \leftarrow grey$
2. **for all** w adjacent to v **do**
3. **if** $state[w] = white$ **then**
4. dfsFromVertex(G, w)
5. $state[v] \leftarrow black$

Algorithm 10.7

The following property will be important in the next section:

Lemma 10.8. *Let G be a graph and v be a vertex of G. Consider the moment during the execution of* dfs(G) *when* dfsFromVertex(G, v) *is started. Then for all vertices w we have:*

(1) If w is white and reachable from v, then w will be black before v.

(2) If w is grey, then v is reachable from w.

We will not give a formal proof here. Intuitively, (1) follows from the fact that all vertices w that are reachable from v are either black before dfsFromVertex(G, v) is started (and thus black before v) or they will be visited during the execution of dfsFromVertex(G, v), because a DFS starting at v visits all vertices reachable from v that have not been visited earlier. Thus they will become black while v is still grey. (2) follows from the fact if w is still grey, dfsFromVertex(G, w) is not yet completed. However, a DFS starting at w only visits vertices reachable from w. Thus v must be reachable from w.

10.5 Topological Sorting

Suppose you have a list of tasks to do, some of which depend on others to be completed first. For example, a practical exercise may involve 10 tasks, numbered 0–9. Task 0 must be completed before Task 1 can be started. Task 1 and Task 2 must be completed before Task 3 can be started. Task 4 must be completed before Task 0 or Task 2 can be started. Task 5 must be completed before Task 0 or Task 4 can be started. Task 6 must be completed before Task 4, Task 5 or Task 7 can be started. Task 7 must be complete before Task 0 or Task 9 can be started. Task 8 must be completed before Task 7 or Task 9 can be started. Task 9 must be completed before Task 2 or Task 3 can be started. A good way to arrange all this information is in a *dependency graph*. The vertices of this directed graph are the tasks to be performed, and there is an edge from task v to task w if v must be completed before w can be started. Figure 10.9 shows the dependency graph of our example. (Fortunately, the dependency graphs of CS2 practical exercises are simpler!)

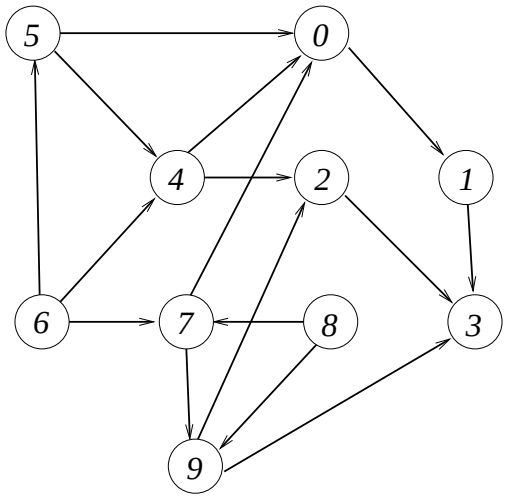

Figure 10.9.

Before we can carry out the tasks we have to arrange them in an order that respects all the dependencies. This is what we call a *topological order* of the dependency graph.

Definition 10.10. Let $G = (V, E)$ be a directed graph. A *topological order* of G is a total order $\prec$ of the vertex set V such that for all edges $(v, w) \in E$ we have $v \prec w$.

For example, a topological order of the digraph in Figure 10.9 is

$$8 \prec 6 \prec 7 \prec 9 \prec 5 \prec 4 \prec 2 \prec 0 \prec 1 \prec 3.$$

It is not obvious how to find a topological order of a digraph efficiently. As a matter of fact, it not even clear whether every digraph has a topological order. A moment's thought reveals that there are digraphs that do not have a topological order: If there are vertices v, w such that there is both an edge from v to w and an edge from w to v, then no topological order of the graph exists, because neither $v \prec w$ nor $w \prec v$ alone would comply with the definition, and both are not possible since $\prec$ is a *total* order. More generally, if the graph has a cycle then there is no topological order.

Let us call a directed graph that does not have a cycle a *directed acyclic graph*, or *DAG* for short. Does every DAG have a topological order? The answer to this question is 'yes'. To prove this, we give an algorithm that computes a topological order for a given DAG. Before we explain the algorithm, let us just record the result:

Theorem 10.11. *A directed graph has a topological order if, and only if, it is a DAG.*

Our algorithm is based on DFS. Let G be a directed graph. Consider the execution of dfs(G). We define an order $\prec$ of the vertices of G by saying that $v \prec w$ if w becomes black before v (i.e., vertices that finish later are smaller in the order).

I claim that if G is a DAG, then $\prec$ is a topological order of G. To prove this, let us assume that $G = (V, E)$ is a DAG. Let $(v, w) \in E$. We have to prove that $v \prec w$, i.e., that w becomes black before v. Consider the moment in the DFS when dfsFromVertex(G, v) is called.

- If w is already black at this moment, there is nothing to prove.
- If w is white, then by Lemma 10.8(1), w will be black before v.
- If w is grey, then by Lemma 10.8(2) v is reachable from w. Thus there is a path from w to v, and together with the edge (v, w), this path forms a cycle. But we assumed that G is acyclic, so this cannot happen.

Note that our argument gives us some additional information: If we find, during the execution of dfsFromVertex(G, v) for some vertex v, an edge from v to a grey vertex w, then we know that G contains a cycle.

We can now modify our basic DFS algorithm in order to get an algorithm for computing the order $\prec$ and printing the vertices in this order. If the input graph G is not a DAG, our algorithm will simply print "G has a cycle". The algorithm adds all vertices to the front of a linked list when they become black. Thus vertices becoming black earlier appear later in the list, which means that the list is in order $\prec$. If during the execution of sortFromVertex(G, v) for some vertex v, an edge from v to a grey vertex w is found, then there must be a cycle, and the algorithm reports this and stops.

Algorithm topSort(G)

1. Initialise array *state* by setting all entries to *white*.
2. Initialise linked list L
3. **for all** $v \in V$ **do**
4. **if** $state[v] = white$ **then**
5. sortFromVertex(G, v)
6. print all vertices in L in the order in which they appear

Algorithm 10.12

The running time of topSort is the same as that of dfs, $\Theta(n + m)$.

Algorithm sortFromVertex(G, v)

1. $state[v] \leftarrow grey$
2. **for all** w adjacent to v **do**
3. **if** $state[w] = white$ **then**
4. sortFromVertex(G, w)
5. **else if** $state[w] = grey$ **then**
6. **print** "G has a cycle"
7. **halt**
8. $state[v] \leftarrow black$
9. L.insertFirst(v)

Algorithm 10.13

Exercises

1. Give 3 different DFS forests for the graph in Figure 10.14.

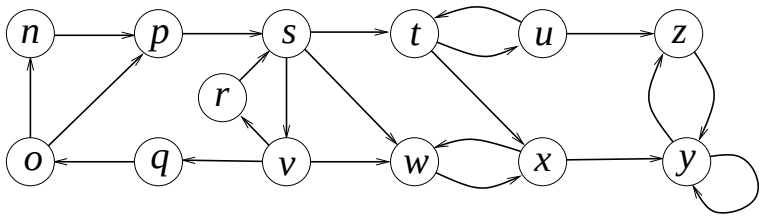

Figure 10.14.

2. The *reflexive transitive closure* of a directed graph $G = (V, E)$ is the graph G^* with the same vertex set V as G and an edge from vertex v to vertex w if there is a path (possibly of length 0) from v to w.

   Describe an algorithm that computes the reflexive transitive closure G^* of a graph G in time $O(n(n + m))$, where n is the number of vertices and m the number of edges of G. Represent the output G^* in adjacency matrix representation.

Don Sannella