Graphs and DFS

In this note, we turn to an alternative to BFS, which is Depth-First Search (DFS).

10.1 Depth-first search

A DFS starting at a vertex \( v \) first visits \( v \), then some neighbour \( w \) of \( v \), then some neighbour \( x \) of \( w \) that has not been visited before, etc. When it gets stuck, the DFS backtracks until it finds the first vertex that still has a neighbour that has not been visited before. It continues with this neighbour until it has to backtrack again. Eventually, it will visit all vertices reachable from \( v \). Then a new DFS is started at some vertex that is not reachable from \( v \), until all vertices have been visited.

Reconsider the graph of Lecture Note 13, shown above in Figure 10.1. A DFS on the graph of Figure 10.1 starting at 0 might visit the vertices in the order 0, 2, 1, 5, 4.

DFS can be implemented in a similar way as BFS using a Stack instead of a Queue, see Algorithms 10.2 and 10.3.

10.2 A recursive implementation of DFS

A second way to implementing DFS is via a recursive algorithm (essentially we rely on the stack used to implement recursion rather than maintaining our own).

To visit all vertices reachable from the start vertex \( v \), we visit \( v \), then the first neighbour of \( v \) and all vertices reachable from this first neighbour—remember that the search is \textit{depth-first}. Then it visits the next neighbour and all new
vertices reachable from it, etc. Algorithms 10.4 and 10.5 give pseudocode for a
recursive implementation of DFS.

The analysis of algorithm dfs might seem complicated because of the recursive
calls inside the loops. Rather than writing down a recurrence for dfs, let us
analyze the running-time directly. Let \( n \) be the number of vertices of the input
graph \( G \) and let \( m = |E| \). Then \( \text{dfs}(G) \) requires time \( \Theta(n) \) for initialisation. Moreover,
\( \text{dfsFromVertex}(G, v) \) requires time \( \Theta(1 + \text{out-degree}(v)) \), because the loop is
iterated out-degree\((v) \) times (as usual we add 1 in case out-degree\((v) = 0 \)).

Now the crucial observation is that \( \text{dfsFromVertex}(G, v) \) is invoked exactly once
for every vertex \( v \). To see that it is invoked \( \text{at least once} \), note that \( \text{visited}(v) \) is
set to \( \text{true} \) only if \( \text{dfsFromVertex}(G, v) \) is invoked. So if the method was never
invoked, then \( \text{visited}(v) \) would remain \( \text{false} \). But this cannot happen, because
for all \( v \) with \( \text{visited}(v) = \text{false} \), \( \text{dfsFromVertex}(G, v) \) is invoked in Line 4 of \( \text{dfs}(G) \)
(in the \( v \)-execution of the loop). To see that \( \text{dfsFromVertex}(G, v) \) is invoked \( \text{at most once} \), note
that it is only invoked if \( \text{visited}(v) = \text{false} \). However, after its first execution
\( \text{visited}(v) = \text{true} \) and can never become \( \text{false} \) again. So \( \text{dfsFromVer-
tex}(G, v) \) is invoked exactly once for every vertex \( v \).

Therefore, we get the following expression for the running time of \( \text{dfs}(G) \):
\[
\Theta(n) + \sum_{v \in V} \Theta(1 + \text{out-degree}(v)) = \Theta(n + \sum_{v \in V} (1 + \text{out-degree}(v)))
\]
\[
= \Theta(n + n + \sum_{v \in V} \text{out-degree}(v))
\]
\[
= \Theta(n + \sum_{v \in V} \text{out-degree}(v))
\]
Let \( m \) be the number of edges of \( G \). Then \( \sum_{v \in V} \text{out-degree}(v) = m \), and we get
\[
T_{\text{dn}}(n, m) = \Theta(n + m).
\]
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 \( 2m = \Theta(m) \), this makes no
difference.

10.3 DFS Forests

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.

Suppose that we explore the graph in Figure 10.1 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 10.6.

Note that a DFS forest is not unique. Figure 10.7 shows another DFS forest
for the graph in Figure 10.1.

10.4 Connected components

As a first application of DFS, we compute the connected components of an undi-
rected graph \( G = (V, E) \). Recall that we say that a subset \( C \) of \( V \) is connected
if for all \( v, w \in C \) there is a path from \( v \) to \( w \) if \( v = w \) there is always a “path”
of length 0 from \( v \) to \( w \). A connected component of an undirected graph \( G \) is
a maximal connected subset \( C \) of \( V \). Here “maximal connected subset” means
that there is no connected subset \( C' \) of \( V \) that strictly contains \( C \). An undirected
graph is connected if it has just one connected component (for all vertices \( v, w \)
there is a path from \( v \) to \( w \)).

Our algorithm is based on the following two simple observations, which do
not necessarily hold for directed graphs:

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 \( v \).

Hence \( \text{dfsFromVertex}(G, v) \) visits the set of vertices in the connected component
of \( v \).

We modify dfs as follows: We add a statement \textbf{print} \( v \) after line 1 of \( \text{dfsFromVer-
tex}(G, v) \). After this modification, \( \text{dfsFromVertex}(G, v) \) prints the vertices in the
connected component of \( v \). Then we add a statement \textbf{print} “New Component”
before each call of \( \text{dfsFromVertex}(G, v) \) in line 4 of dfs. If the set of connected
components is needed as input for another algorithm, we can instead get dfs to
return the connected components in some convenient format.
The asymptotic running time of this algorithm for computing the connected components is clearly the same as the running time of \texttt{dfs}.

**Components of directed graphs**

It is not so clear what connectivity means in directed graphs. For example, is the graph in Figure 10.1 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 \textit{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 a more important notion. The graph in Figure 10.1 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 (i.e., directed graph) in Figure 10.1 only has one weakly connected component (containing all vertices), and it has three strongly connected components:

\[
\begin{align*}
0, 1, 2, 4 \\
3, 6 \\
5.
\end{align*}
\]

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 is outside the scope of this course.

**10.5 Classifying vertices during a DFS**

Let \( G \) be a graph. Recall that during an execution of \texttt{dfs}(\( G \)), the subroutine \texttt{dfsFromVertex}(\( G,v \)) is invoked exactly once for each vertex \( v \). Let us call vertex \( v \) \textit{finished} after \texttt{dfsFromVertex}(\( G,v \)) is completed. During the execution of \texttt{dfs}(\( G \)), a vertex can be in three states:

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

We can modify our recursive DFS algorithm so that it keeps track of the states of the vertices, see Algorithms 10.8 and 10.9.

The following property will be important in the next section:

**Algorithm** \texttt{dfs}(\( G \))

1. Initialise array \textit{state} by setting all entries to \textit{white}.
2. for all \( v \in V \) do
3. \hspace{1em} if \textit{state}[\( v \)] = \textit{white} then
4. \hspace{2em} \texttt{dfsFromVertex}(\( G,v \))

**Algorithm 10.8**

**Algorithm** \texttt{dfsFromVertex}(\( G,v \))

1. \textit{state}[\( v \)] = \textit{grey}
2. for all \( w \) adjacent to \( v \) do
3. \hspace{1em} if \textit{state}[\( w \)] = \textit{white} then
4. \hspace{2em} \texttt{dfsFromVertex}(\( G, w \))
5. \textit{state}[\( v \)] = \textit{black}

**Algorithm 10.9**

**Lemma 10.10.** Let \( G = (V,E) \) be a graph, \( v \in V \). Consider the time during execution of \texttt{dfs}(\( G \)) when \texttt{dfsFromVertex}(\( G,v \)) is called. Then for all vertices \( w \) we have:

1. If \( w \) is \textit{white} and reachable from \( v \), then \( w \) will be \textit{black} before \( v \).
2. If \( w \) is \textit{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 \texttt{dfsFromVertex}(\( G,v \)) is started (and thus black before \( v \)) or they will be visited during the execution of \texttt{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, \texttt{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 \( w \).

**10.6 Topological Sorting**

Suppose we 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 is started.
- Task 1 and Task 2 must be completed before Task 3 is started.
Theorem 10.13. (directed cycle then there is no topological order.
In fact, there are many digraphs that do not have a topological order: It is not obvious how to find a topological order of a digraph efficiently (assuming a total order of the vertex set \( V \)).

Definition 10.12. 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, one topological order of the digraph in Figure 10.11 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 (assuming it has one). In fact, there are many 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 (we cannot take either \( v \prec w \) nor \( w \prec v \), so any ordering is not total). In general, if the graph has a directed cycle then there is no topological order.

A directed graph that does not have a cycle is called a directed acyclic graph (DAG). The following theorem holds for DAGs.

Theorem 10.13. A directed graph has a topological order if, and only if, it is a DAG.

Our algorithm for finding the topological order of a DAG is based on dfs. Consider the execution of dfs(G) on a directed graph \( G \). We define the order \( \prec \) on \( V \) by setting \( v \prec w \) if \( w \) becomes black before \( v \) (later finishers are smaller).

We now show that if \( G \) is a DAG, then \( \prec \) is a topological order of \( G \). 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 dfsStartG(\( G, v \)) is called.

- If \( w \) is already black at this moment, there is nothing to prove.
- If \( w \) is white, then by Lemma 10.10(1), \( w \) will be black before \( v \).
- If \( w \) is grey, then by Lemma 10.10(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 dfsSortG(\( G, v \)) for some vertex \( v \), an edge from \( v \) to a grey vertex \( w \), then we know that \( G \) contains a cycle.

We can modify our basic DFS algorithm in order to get an algorithm for computing the order \( \prec \) and printing the vertices in this order. If \( G \) is not a DAG, our algorithm will simply print “\( G \) has a cycle”. Otherwise the algorithm adds all vertices to the front of a linked list when they become black. Vertices becoming black earlier appear later in the list, which means that the list is in order \( \prec \). If during the execution of dfsSortG(\( G, v \)) for some vertex \( v \), an edge from \( v \) to a grey vertex \( w \) is found, then there must be a cycle, and the algorithm 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 10.14

The running time of topSort is the same as that of dfs, \( O(n + m) \).
Algorithm sortFromVertex\(G, v\)

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

Algorithm 10.15

Exercises

1. Give orders in which a BFS and DFS starting at vertex \(n\) may traverse the graph displayed in Figure 10.16.

![Diagram](image)

Figure 10.16.

2. Give 3 different DFS forests for the graph in Figure 10.16.

3. Let \(G\) be a graph with \(n\) vertices and \(m\) edges. Explain why \(O(\lg m)\) is \(O(\lg n)\).

4. 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.