Depth/Breadth first search with applications

Saad Mneimneh

1 Introduction

Given a graph $G = (V, E)$ and a vertex $u \in V$, we would like to explore every vertex (and every edge) reachable from u. Let $n = |V|$ and $m = |E|$. The graph may be directed or undirected. We will assume that the graph is given in its adjacency list representation (rather than adjacency matrix), i.e. for each vertex u, we have a linked list $adj[u]$ of the neighbors of u (all vertices v such that $(u, v) \in E$. The advantages of such representation are:

- It requires $\Theta(n + m)$ space to store the vertices and their corresponding lists, as opposed to $\Theta(n^2)$ for the adjacency matrix. Recall the handshake Lemma: $\sum_u |adj[u]| = \Theta(m)$ (m if directed, $2m$ if undirected). This is particularly good for sparse graphs.
- It makes it possible to go through the neighbors of a vertex u in $O(|adj[u]|)$ time, i.e. linear in the number of neighbors. This turns out to be important for graph traversal (see handshake Lemma above).

2 Depth first search, DFS

Consider the following algorithm that explore the graph in depth (recursively), starting at a vertex u:

$\underline{DFS - VISIT(u)}$
$d[u] \leftarrow time$
for each $v \in adj[u]$
    do if $d[u] = 0$
        then $DFS - VISIT(v)$
$time \leftarrow time + 1$
$f[u] \leftarrow time$

To be correct, the above algorithm requires that initially $d[u] = 0$ for all vertices and $time > 0$. This requires $O(n)$ time to set $d[u]$ to false for every vertex u. However, with proper implementation, this initialization can be avoided, and the above algorithm runs in $O(\sum_u |adj[u]|) = O(m)$ time and visits all vertices

(and edges) reachable from u. $d[u]$ and $f[u]$ represent the discovery time and the finish time of vertex u, respectively. The discovery time of u is the time when u is first discovered by the search. The finish time of u is the time when $DFS-VISIT(u)$ terminates (all vertices and edges reachable from u have been processed by the above algorithm). $d[u]$ and $f[u]$ will become useful later on.

If the entire graph is to be explored, one could repeat $DFS-VISIT$ starting from an unvisited vertex. The following algorithm runs in $O(n+m)$ time.

$\underline{DFS(G)}$
for each vertex $u \in V$
    do $d[u] \leftarrow 0$
$time \leftarrow 0$
for each vertex $u \in V$
    do if $d[u] = 0$
        then $DFS - VISIT(u)$

3 Classifying edges

An important property of DFS is that it can be used to classify edges of the graph $G = (V, E)$. We can define four types of edges:

- Tree edges: edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v). The tree edges form a spanning forest of G (no cycles).
- Back edges: an edge (u, v) is a back edge if it connects vertex u to an ancestor v in a DFS tree.
- Forward edges: an edge (u, v) is a forward edge if it connects a vertex u to a descendant v in a DFS tree.
- Cross edges: all other edges.

It is easy to show that:

- (u, v) is a tree edge or a froward edge $\Leftrightarrow d[u] < d[v] < f[v] < f[u]$
- (u, v) is back edge $\Leftrightarrow d[v] < d[u] < f[u] < f[v]$
- (u, v) is a cross edge $\Leftrightarrow d[v] < f[v] < d[u] < f[u]$

Figure 1 illustrates the DFS algorithm with nodes labeled by their discovery and finish times, and edges labeled by their types.

An important fact about undirected graphs is that forward and cross edges never occur.

Theorem: DFS on an undirected graph yields only tree edges and back edges.

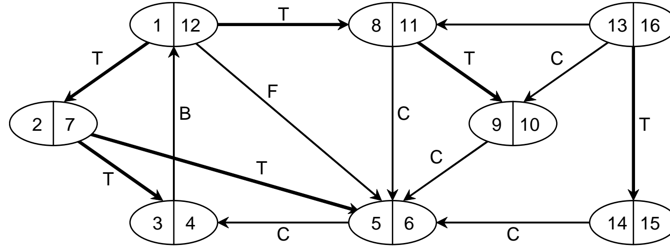


Figure 1: DFS run, with tree edges emphasized

Proof: Let $(u, v) \in E$ and assume without loss of generality that $d[u] < d[v]$. If edge (u, v) is explored first in the direction from u to v, v is discovered then, and (u, v) becomes a tree edge. If edge (u, v) is explored in the direction from v to u, the (u, v) is obviously a back edge.

4 Detecting cycles

DFS can be used to detect whether a graph has cycles or not. Consider first the case of an undirected graph G. We already know that all edges of G will be classified as either tree edges or back edges. In addition, if G is acyclic, then only tree edges will occur. This is because if G is acyclic, then it is a forest (disconnected trees). Therefore, we have the following result: An undirected graph is acyclic iff DFS yields no back edges.

Now let us consider the case of a directed graph. A directed acyclic graph is called a DAG. The same result above extends to DAGs.

Theorem: A graph G is a DAG iff DFS yields no back edges.

Proof: $\Rightarrow$: if there is a back edge (u, v), then by the definition of a back edge, clearly G has a cycle. $\Leftarrow$: if G has a cycle, let v be the first vertex to be discovered on the cycle, and let (u, v) be the preceding edge. Since $d[v]$ is smallest, all vertices (including u) will be descendants of v in a DFS tree. Therefore, (u, v) is a back edge.

Therefore, detecting cycles can be done in $O(n + m)$ time by running DFS and observing the types of edges it yields in the graph. For undirected graphs, this can be done in $O(n)$ time only: if at any point in time, at least as many edges as vertices have been explored, we can stop the DFS because a forest must have $m \leq n - 1$.

5 Topological sorting

A topological sorting of a DAG $G = (V, E)$ is a total order relation $<$ on the vertices of V such that for every edge $(u, v) \in E$, $u < v$. DFS can be used to obtain a topological sorting and, hence, also as a proof that topological sorting of a DAG is always possible. An important application of topological sorting is the executing of tasks with dependencies. For instance, each task is represented by a vertex. An directed edge (u, v) indicates that task v depends on task u and, hence, task u must be performed first. If the resulting graph is a DAG, a topological sorting gives an order by which we can execute the tasks. Here's an algorithm for topological sorting based on DFS.

Topological sorting

1. Run DFS to compute the finish times $f[u]$ for every vertex u
2. Sort the vertices in order of decreasing $f[u]$

While sorting can be done in $O(n)$ time using counting sort or radix sort, it can be avoided by inserting a vertex onto the front of a linked list once the vertex is finished. The proof that this algorithm results in a topological order lies in Section 3. Since back edges cannot occur, all remaining edges (u, v) satisfy $f[v] < f[u]$.

Another observation about topological sorting is that a DAG has at least one source (a vertex with no incoming edges) and at least one sink (a vertex with no outgoing edges). The first vertex in the topological order is a source. Similarly, the last vertex in the topological order is a sink.

6 Connectedness

A connected component of an undirected graph G is a maximal subgraph of G that is connected. With a small modification, DFS can be used to identify the connected components of an undirected graph. Observe that if DFS-VISIT is started on a vertex u, all the vertices in the connected component containing u will be visited. Therefore, we can assign each vertex u, a component number $cc[u]$, such that $cc[u] = cc[v]$ iff u and v are in the same connected component. The component number starts at 0 and is incremented by one every time we start DFS-VISIT.

$\underline{DFS - VISIT(u)}$
$d[u] \leftarrow time$
$cc[u] \leftarrow cc$
for each $v \in adj[u]$
    do if $d[u] = 0$
        then $DFS - VISIT(v)$
$time \leftarrow time + 1$
$f[u] \leftarrow time$

```
DFS(G)
for each vertex u ∈ V
    do d[u] ← 0
time ← 0
cc ← 0
for each vertex u ∈ V
    do if d[u] = 0
          then cc ← cc + 1
               DFS − VISIT(u)
```

For directed graphs, we define the strongly connected components. A strongly connected component of a directed graph G is a maximal set of vertices C such that for every pair of vertices u and v in C, u and v are reachable from each other. DFS can also be used to identify the strongly connected components of a directed graph. Here's an important observation: the strongly connected components of a directed graph form a DAG. In other words, given the directed graph $G = (V, E)$, define the graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ where $\mathbb{V}$ is the set of strongly connected components of G, and $(C, C') \in \mathbb{E}$ iff there exist u and v such that $u \in C$ and $v \in C'$ and $(u, v) \in E$. $\mathbb{G}$ is a DAG. This is an immediate consequence of strongly connected components. If $\mathbb{G}$ has a cycle, then some strongly connected components of G will collapse into one, contradicting their maximality. Since $\mathbb{G}$ is a DAG, it must have a sink. We are not going to explicitly construct $\mathbb{G}$, but if we start DFS at a vertex $u \in C$, such that C is a **sink** in $\mathbb{G}$, then DFS-VISIT will terminate when all vertices in C (and only those) have been visited, thus identifying that component. However, it is easier to identify a vertex that belongs to a **source** component, by the following theorem.

Theorem: Let C and C' be two strongly connected components of a directed graph G. If there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$, then $\max_{u \in C} f[u] > \max_{v \in C'} f[v]$.

Proof: There are two cases to consider. If DFS-VISIT visits C (a vertex in) before C', then clearly all of C and C' will be explored before DFS-VISIT terminates. Therefore, the first vertex visited in C will have a larger finish time than any other vertices in C'. On the other hand, if C' is visited first, then DFS-VISIT will terminate after exploring all of C' but before exploring any vertex in C, in which case the property also follows.

The theorem implies that a vertex with a largest finish time must be in a source component. Note that the theorem does **not** imply that a vertex with a smallest finish time must be in a sink component (consider the graph given by $V = \{a, b, c\}$ and $E = \{(a, b), (b, a), (a, c)\}$, where b can finish first). To work around this, let G^T be the graph obtained by reversing all edges in G (which can

be done in linear time given the adjacency list representation). The strongly connected components of G^T are exactly those of G. Furthermore, any sink component of $\mathbb{G}$ is now a source component of $\mathbb{G}^T$. This leads to the following $O(n+m)$ time algorithm for strongly connected components.

$\underline{SCC(G)}$

1. Compute G^T
2. Run $DFS(G^T)$ to compute the finish times $f[u]$
3. Run $DFS(G)$ with the $cc[u]$ addition, but in the main loop of DFS, consider vertices in order of decreasing $f[u]$ as obtained in 2

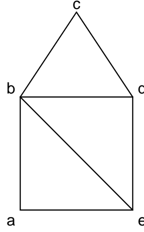
The proof that this works correctly is by induction: we always start DFS-VISIT in a sink component, once the identified components have been removed.

7 Lexicographic BFS

Let G be an undirected and connected graph (if not, work individually on each connected component). Lexicographic BFS uses an auxiliary labeling of the vertices to impose an order on how they are explored. Initially, all vertices start with an empty label ϕ.

Lex $\underline{BFS(G)}$
for each vertex $u \in V$
    do $l[u] \leftarrow \phi$
        $f[u] \leftarrow 0$
for $i \leftarrow n$ downto 1
    do let u be such that $f[u] = 0$ and $l[u]$ is largest
        $f[u] \leftarrow i$
        for each vertex $v \in adj[u]$ such that $f[v] = 0$
            do add i to $l[v]$

The algorithm simply picks a vertex u that is unvisited ($f[u] = 0$) and has the largest label, assigns i to $f[u]$, and adds i (counts down) to the label of all of its neighbors. This guarantees that, lexicographically, the neighbors of u have now the largest labels (hence the name lexicographic BFS). Here's an example run:



u	$l[u]$	$f[u]$	$l[u]$	$f[u]$	$l[u]$	$f[u]$	$l[u]$	$f[u]$	$l[u]$	$f[u]$	$l[u]$	$f[u]$
a	ϕ	0	ϕ	5	ϕ	5	ϕ	5	ϕ	5	ϕ	5
b	ϕ	0	$\{5\}$	0	$\{5\}$	4	$\{5\}$	4	$\{5\}$	4	$\{5\}$	4
c	ϕ	0	ϕ	0	$\{4\}$	0	$\{4\}$	0	$\{4,2\}$	0	$\{4,2\}$	1
d	ϕ	0	ϕ	0	$\{4\}$	0	$\{4,3\}$	0	$\{4,3\}$	2	$\{4,3\}$	2
e	ϕ	0	$\{5\}$	0	$\{5,4\}$	0	$\{5,4\}$	3	$\{5,4\}$	3	$\{5,4\}$	3

The running time of lexicographic BFS depends on the detail of the implementation. The bulk of the work lies in finding a vertex with the largest label and updating the labels. If not done carefully, the running time will exceed the $O(n+m)$ bound for BFS. Luckily, the labels need not be computed explicitly, but can be simulated. A sequence of sets of vertices is maintained, initially consisting of the single set of all vertices. The intention is that vertices in the same set have the same label. Repeatedly, we select and delete an element u from the last set (this is the one with the largest label). We now process the edges of u. For every edge (u,v) such that $f[v]=0$, we delete v from its set S and move it to a newly created set T, inserted into the sequence immediately after S. While processing the edges of u, each element v deleted from S will be added to this newly created T. If S becomes empty, it is deleted. It should be obvious that this technique simulates the lexicographic ordering of the labels. If the sequence of sets is implemented as a doubly linked list, and each set in the sequence also as a doubly linked list, then every operation on the sets can be done in $O(1)$ time, assuming each vertex maintains a pointer to its set. Therefore, lexicographic BFS can run in $O(n+m)$ time.

Ordering vertices by their finish times in lexicographic BFS has many important applications in graph theory. We mention here some theory about chrodal graphs. A cordal graph is a graph in which every cycle of length four or more has a chord. The graph in the above example is chordal. Define a perfect elimination scheme as a total order relation $<$ on the vertices such that for every vertex u, the set $\{v : v \in adj[u] \text{ and } u < v\}$ form a clique. Here's a theorem, stated without proof:

Theorem: A graph is chordal if and only if the order of vertices given by their finish times in lexicographic BFS is a perfect elimination scheme. [*The graph in the above example is chordal, so verify that the Lex BFS order of the vertices*

gives a perfect elimination scheme.]

Therefore, this suggests a way to recognize chordal graphs: order the vertices by the finish times and check if that order is a perfect elimination scheme. The first part can be done in $O(n+m)$ time. We show that the second part can also be done in $O(n+m)$ time.

<u>PERFECT</u>

```
for each vertex u
    do A(u) ← ∅
for each vertex u in Lex BFS order
    do X_u ← {x ∈ adj[u] | f[x] > f[u]}
        if X_u ≠ ∅
          then v ← min_{f[x]} X_u
               add X_u − {v} to A(v)
        if A(v) − adj[v] ≠ ∅     (actual checking)
          then return false
return true
```

In the above algorithm, the list $A(v)$ collects all the vertices which will eventually have to be checked for adjacency with v. The actual checking is delayed until v's iteration. This technique is used so that in u's iteration there is no search for $adj[v]$. $A(v)$ is implemented as a linked list, but may contain repetitions. The actual checking looks for a vertex w in the list $A(v)$ which is not adjacent to v, and can be done in $O(adj[v] + A[v])$ time (how?). Therefore, the entire algorithm runs in

$$O(n) + \sum_u |adj[u]| + \sum_v |A(v)|$$

where $|A(v)|$ is taken at its final value. Now $\sum_v |A(v)| \leq + \sum_u |adj[u]|$ since a given $adj[u]$ appears as part of at most one of the lists $A(v)$. Therefore, the algorithm runs in $O(n+m)$ time.

A modification of the above algorithm finds all maximal cliques of a chordal graph (this problem is NP-hard in general). The idea is that every maximal clique must correspond to a set $\{u\} \cup X_u$. But not every such set is a maximal clique, so some of these will need to be filtered out. Observe that $\{u\} \cup X_u$ is not a maximal clique iff X_u is concatenated to $A(u)$. Let $s[v]$ be the size of the largest set that is concatenated to $A[v]$. We need to check that $s[u] < |X_u|$ before we declare $\{u\} \cup X_u$ a maximal clique.

```
MAXIMAL-CLIQUES
for each vertex u
    do s[u] ← 0
for each vertex u in Lex BFS order
    do if adj[u] = ∅
          then print {u}
       X_u ← {x ∈ adj[u] | f[x] > f[u]}
       if X_u ≠ ∅
          then v ← min_{f[x]} X_u
               s[v] ← max(s[v], |X_u| − 1)
               if s[u] < |X_u|
                  then print {u} ∪ X_u
```