Contents

1 Data Structures for Searching

Queue: First in, first out. Used for BFS!

Stack: Last in, first out. Used for DFS!

Cool thing: You can implement BFS and DFS in the same code just by changing which vertex comes out of your data structure.

2 More on Edge Classification

How do we tell which edge is which?

Recall:

1. Tree edge: actually traversed in search
2. Back edge: goes from descendent to ancestor
3. Forward edge: non-tree edge from ancestor to descendent
4. Cross edge: all other edges

How do we tell which edges are which?

Discovery and finish times tell us:

- If $[d[u], f[u]] \subset [d[v], f[v]]$ then v is an ancestor of u. So an edge from v to u is either a tree or a forward edge. An edge from u to v is a back edge.
- If $[d[u], f[u]]$ is disjoint from $[d[v], f[v]]$ then an edge from u to v is a cross edge

We can also classify them during search by the color of the terminating vertex when exploring an edge (u, v) (so v is terminating vertex). For how colors are assigned, see last week's recitation notes or CLRS:

- If v is White, (u, v) is a tree edge since we will explore it in search.

- If v is Gray, v must currently be being processed. Therefore, v is an ancestor of u and (u, v) is a back edge.
- If v is black, we have already finished v. So (u, v) is either a cross or a forward edge. Looking at starting and finishing times would tell us for sure.

3 Connected Components for Undirected Graphs

Two vertices in an undirected graph are *connected* if there is a path between them. Connectivity (i.e. path from u to v) forms an **equivalence relation**:

Reflexive: There is trivially a path from v to itself.

Symmetric: If there is a path from u to v then the same set of edges is also a path from v to u because the graph is undirected. This requirement is why connectivity is not an equivalence relation for directed graphs!

Transitive: If there is a path from u to v and a path from v to w then there is a path from u to w.

A *class* of an equivalence relation is a set of elements R such that each element of R is related to all other elements of R and each element of R is related to no elements outside R. For the connectivity relation, each class is a *connected component*.

In BFS, we find *one* connected component if we're given starting square.

To find connected components: run BFS or DFS from *any* starting square. You will find the connected component that square belongs to. Start from another square not in the component.

To do this: modify DFS-Visit to take *owner* array which marks which connected component each vertex corresponds to. Can also store in the vertex which component they belong to.

```
ConnectedComponents(V, E)
    // Input: A graph with vertices V and edges E
    // Output: The connected components of the graph
    // Running Time: O(|V| + |E|)
1   for v in V
2       color[v] ← White
3       C[v] ← −1 // C[v] is connected component of v
4   connected-component ← 0
5   for u ∈ V
6       if color[u] = White
7           DFS-Visit(u, C, connnected-component)
8           connected-component ←connected component + 1
```

```
ConnectedComponent-Visit(u, C, connected-component)
1   color[u] ← Gray
2   C[u] ← connected-component
3   for v in Adj(u)
4       if color[v] = White
5           π[v] ← u
6           ConnectedComponent-Visit(v)
7   color[u] ← Black
```

Running Time: We've added a little more work per vertex to DFS but it's still constant per vertex so still $O(|V| + |E|)$.

Correctness: We see every vertex by DFS proof. Assume there is a path from u to v. Let u be in the connected component C and let the vertex from which we started DFS when discovering u be s. Then there is also a path from s to v and so, by the proof of DFS correctness, we find v during the search from s. Thus, we also mark v as being in the connected component C.

Now assume u there is no path from u to v. Let the connected component containing u be C and the starting vertex of the DFS run to discover C be s. Then there must be a path from s to u and therefore from u to s (remember: undirected graph). Thus there can be no path from s to v and v will be placed in a different connected component.

In case of DFS**, CLRS also refers to connected components as *trees* and the full graph a *forest of trees*.**

4 Topological Sort

Idea: Order vertices of a directed graph one before another so that if v precedes u there is no path from u to v.

$\Rightarrow$ This is only possible if the graph contains no cycles!

DAG: Directed acyclic graph. Directed graph with no cycles. Just what the name says!

Source Theorem: Every DAG has a *source*, which is a vertex with no incoming edges.

Proof: We proceed by contradiction.. Start at any vertex. Traverse edges backwards (against their direction). Now assume we never "get stuck", that we can continue the traversal forever. Then, since there are only $|V|$ vertices, we must eventually repeat a vertex v. So there is a cycle originating at v. But we stipulated that G is a DAG so it cannot hav cycles! Therefore, we must stop the traversal at some vertex s. This vertex has no incoming edges (or the traversal could continue) so s is a source.

Algorithm: Find source (can use DFS on *transpose graph*!). Remove source and find another. Etc.

Running Time: $O(|V|^2 + |V||E|) = O(|V||E|)$. That's bad...

Correctness: When we assign vertex v a schedule number, it is a source of the graph consisting of all vertices that will receive a later schedule time. Therefore, no vertices with a later schedule can possibly reach v.

Better Algorithm: The *reverse* of the finish times is a valid schedule!

Running Time: We can assign finishing times during DFS as we saw last recitation. So $O(|V|+|E|)$. Yay!

Correctness: We show that if an edge (u, v) exists then $f[v] < f[u]$ so u will be ordered before v. We consider the color of v when the edge (u, v) is first explored. If v is White, then v becomes a descendent of u and must have an earlier finish time since $[d[v], f[v]] \subset [d[u], f[u]]$. If v is Gray, then (u, v) is a back edge, which cannot exist because the graph is a DAG. So v is not gray. If v is black then its finish time has already been assigned and must be smaller than the finish time of u, which has not yet been assigned.

A simple induction proof would show that this implies that if u can reach v then $f[v] < f[u]$. Do that on your own.

Transpose Graph: Traversing edges "backwards" or ordering finish times in reverse can be confusing. So we talk about the transpose graph $G^T = (V, E^T)$ where E^T is E with the direction of each edge flipped.