Recitation 8 – *Shortest Paths and DFS Numberings*

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2014)

March 4th, 2014

1 Announcements

- How's *ThesaurusLab*?
- *AbridgedLab* has been released!
- Spring break plans?

2 That's Some Deep Searching, Yo.

2.1 DFS Basics

When you apply DFS to a graph, it implicitly defines a spanning tree rooted at the start vertex. If more than one tree is needed to span all vertices in the graph (when will this happen?), then we call it a *DFS forest*.

Q: What are edges that correspond to edges in the DFS tree called?

A: Tree edges.

Q: What are edges that go from a vertex v to an ancestor u in the DFS tree called?

A: Back edges.

Q: What are edges that go from a vertex v to an descendant u in the DFS tree called?

A: Forward edges.

Q: What are the remaining edges called?

A: Cross edges, since they cross from one subtree of the DFS tree to another.

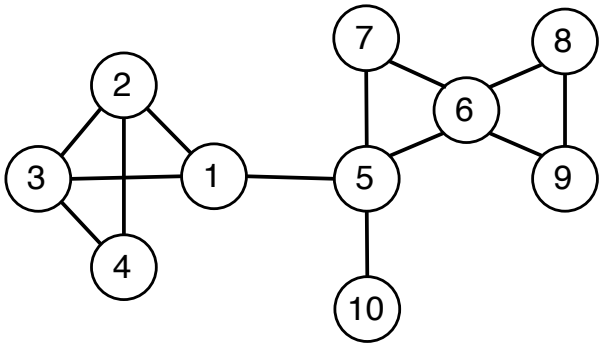
Q: In an *undirected* graph, if DFS finds a vertex that it has visited before, it has found a cycle. Is this sufficient?

A: There is a cycle in the undirected graph iff there is a back edge.

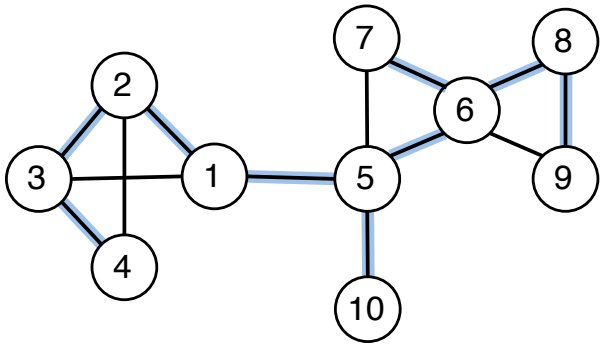
In an undirected graph, such as the one we will consider now, all edges are either tree edges or back edges (equivalently, we could say *forward edges* instead of *back edges*, since there's no notion of direction). Why can't there be any cross edges?

2.2 Drawing DFS Trees

What's the DFS tree of this graph? Use the vertices in increasing order.

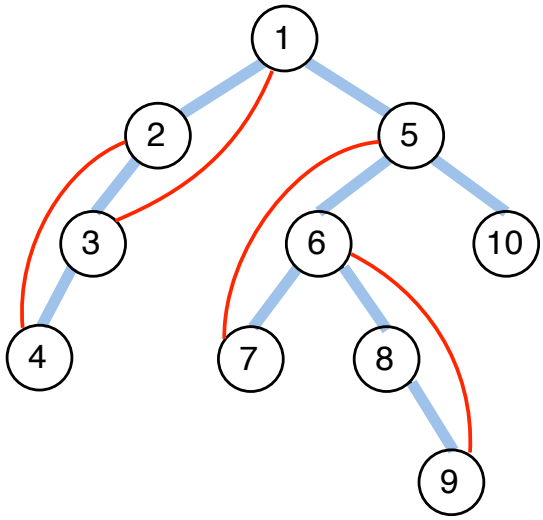


Solution 2.0



Redraw the resulting DFS tree in a more familiar tree-like format.

Solution 2.0 To see the tree edges and back edges clearly, it's helpful to redraw the DFS tree in a more familiar tree-like format:



2.3 Bridges

In *AbridgedLab*, your task is to find all bridges in an undirected graph.

Q: What is a bridge?

A: A bridge is an edge whose removal disconnects the graph. In other words, a bridge is not on a cycle.

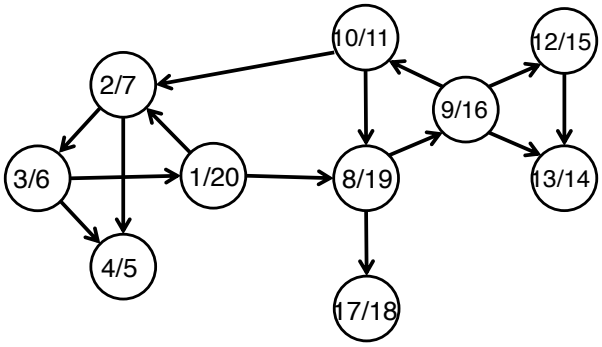
Q: Which edges are bridges in the above graph? And in the DFS tree?

A: $(1, 5)$ and $(5, 10)$.

2.4 DFS Numberings

DFS can easily be modified to record the *discovery time* $d(u)$ and *finishing time* $f(u)$ for every vertex u. The discovery time corresponds to the time when a vertex is first visited and the finishing time corresponds to the time when the vertex is last visited.

Here are the DFS numbers for an example directed graph:



The following is a theorem that relates DFS numberings to the structure of the DFS tree. Fill in the following 3 cases for the relationships between d and f numbers, for any two vertices u and v:

1. If the intervals $d(u) \rightarrow f(u)$ and $d(v) \rightarrow f(v)$ are entirely disjoint, ...

2. If the interval $d(u) \rightarrow f(u)$ is contained entirely within the interval $d(v) \rightarrow f(v)$, ...

3. If the interval $d(v) \rightarrow f(v)$ is contained entirely within the interval $d(u) \rightarrow f(u)$, ...

Solution 2.0

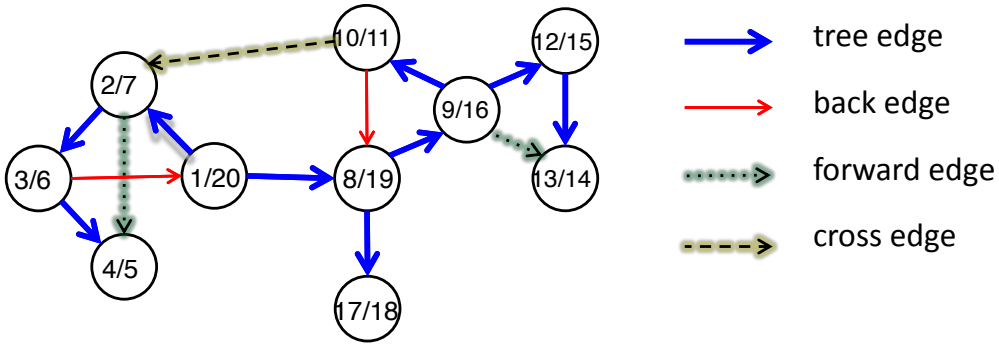
1. If the intervals $d(u) \rightarrow f(u)$ and $d(v) \rightarrow f(v)$ are entirely disjoint, neither u nor v is a descendant of the other in the DFS forest.
2. If the interval $d(u) \rightarrow f(u)$ is contained entirely within the interval $d(v) \rightarrow f(v)$, u is a descendant of v in a DFS tree.
3. If the interval $d(v) \rightarrow f(v)$ is contained entirely within the interval $d(u) \rightarrow f(u)$, v is a descendant of u in a DFS tree.

Q: Based on this theorem, how would you use the discovery times and finishing times to classify edges into tree, forward, back or cross edges?

A: An edge (u, v) is:

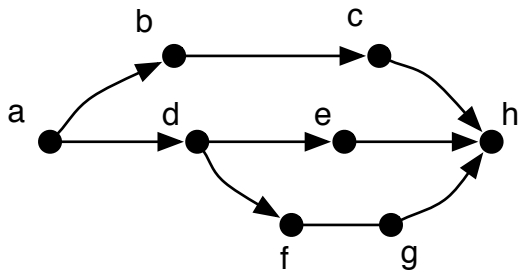
1. A tree/forward edge iff $d(u) < d(v) < f(v) < f(u)$
2. A back edge iff $d(v) < d(u) < f(u) < f(v)$
3. A cross edge iff $d(v) < f(v) < d(u) < f(u)$

Now classify the edges of the above graph into tree, forward, back or cross edges!



3 Topologicalizationalistically Sort This Graph

Let's do a topological sort on the following DAG:



Solution 3.0 A possible call order would be to start the DFS on the top path. This hits all the vertices down to `h` with no branch-offs. When we exit each vertex we add it to our list of vertices, meaning that when we get back to `a`, our list is `[b, c, h]`. Before exiting `a` we need to search each of its children, so we move down to `d`. If the next path that we search down is the middle path, only `e` will be added to our list because our DFS has already touched `h`. We do the same process for the bottom path, so right before `d` exits, our list is `[f, g, e, b, c, h]`. Next we exit `d` and add it to the list, then we exit `a` and add it to the list, too. Our final list is `[a, d, f, g, e, b, c, h]`

4 So You Think You Know Your Way Around CMU?

4.1 Dijkstra's Basics

It's just after 15–210 lecture on Wednesday, and you need to get from Rashid to your next class in Baker. There are many ways to get there! You could:

- Take the Pausch Bridge to the Cut, walk straight across to the Doherty entrance, then diagonally across to Baker.

- Take the indoor bridge to Newell-Simon then to Wean, then walk across the Mall to Baker.
- Take the exit to Forbes, walk to Craig, get a delicious serving of *froyo* from Razzy, back to Gates, then option 1 to Baker!

Of course, you want the fastest route!

For Dijkstra's algorithm, we need to maintain a table $D(v)$ which contains the shortest path from the start node s (GHC in this example) to v *following only nodes that we've already explored*. We start out with $D(s) = 0$ and $D(v) = \infty$ for all $v \neq s$. Repeat the following steps until all nodes are explored:

1. Find a vertex u such that $D(u) = \min_{v \in N \setminus X} D(v)$, where X is the set of visited vertices.
2. For all $v \in N_G(u)$, set $D(v) = \min\{D(v),\ D(u) + w(u,v)\}$, where $w(u,v)$ is the weight of the edge from u to v.

Q: What data structure do we use to find the u with minimum $D(u)$?

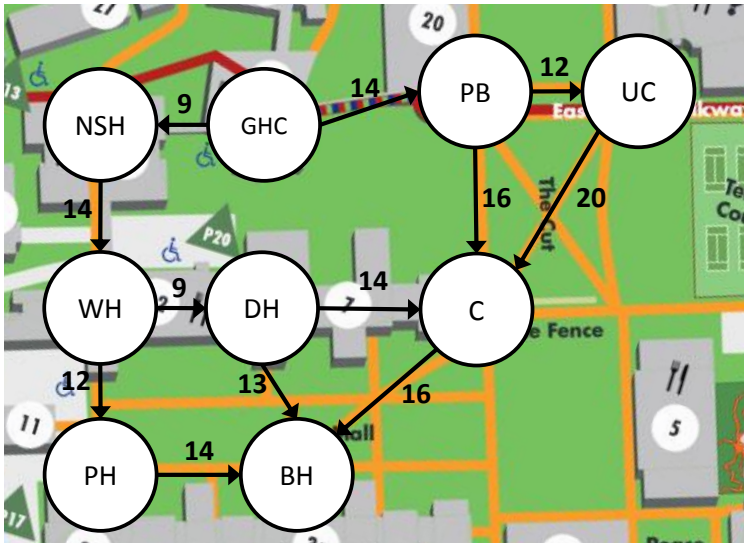
A: A priority queue! Using the `PQ.deleteMin` operation:

```
val deleteMin : 'a pq -> (key * 'a) option * 'a pq
```

Q: To update $D(v)$ for neighbors v of u, we just insert $(v,\ d + w(u,v))$ into the priority queue. Why do we not need to check if this is less than the existing value for v?

A: Because the first time we visit v, if there are multiple pairs $(v,\ d_1), \ldots, (v,\ d_n)$ in the priority queue, `deleteMin` will always give us the one with the lowest d_i. Thus, if we insert an existing node with a larger distance, it'll get ignored, and if we insert it with a smaller distance, it will just overwrite the existing value!

4.2 Finding our way to Baker!



Let's work through what Dijkstra's algorithm does on the graph of CMU above, with edge weights given by arbitrarily-scaled straight-line distances. For simplicity, we don't visit already-visited nodes. The first row has been done for you.

	Node	**PQ Operations (after `deleteMin`)**	**PQ State (*lowest values only*)**
1	GHC	`insert(PB, 14)`, `insert(NSH, 9)`	$\{NSH \mapsto 9,\ PB \mapsto 14\}$
2	NSH		
3	PB		
4	WH		
5	UC		
6	C		
7	DH		
8	PH		
9	BH		

Solution 4.0

	Node	**PQ Operations (after `deleteMin`)**	**PQ State (*lowest values only*)**
1	GHC	`insert(PB, 14)`, `insert(NSH, 9)`	$\{NSH \mapsto 9,\ PB \mapsto 14\}$
2	NSH	`insert(WH, 23)`	$\{PB \mapsto 14,\ WH \mapsto 23\}$
3	PB	`insert(UC, 26)`, `insert(C, 30)`	$\{WH \mapsto 23,\ UC \mapsto 26,\ C \mapsto 30\}$
4	WH	`insert(DH, 32)`, `insert(PH, 35)`	$\{UC \mapsto 26,\ C \mapsto 30,\ DH \mapsto 32,\ PH \mapsto 35\}$
5	UC	`insert(C, 46)`	$\{C \mapsto 30,\ DH \mapsto 32,\ PH \mapsto 35\}$
6	C	`insert(BH, 46)`	$\{DH \mapsto 32,\ PH \mapsto 35,\ BH \mapsto 46\}$
7	DH	`insert(C, 46)`, `insert(BH, 45)`	$\{PH \mapsto 35,\ BH \mapsto 45\}$
8	PH	`insert(BH, 49)`	$\{BH \mapsto 45\}$
9	BH		$\{\ \}$

In steps 5, 7 and 8 we inserted values for C, C, and BH, respectively, which were greater than the existing value, so these inserts are ignored. In step 7, we inserted a value for BH which was smaller than the existing value, thus updating the shortest path.

Hey guess what? This is indeed the shortest path to Baker: the path of distance 45 consisting of $\langle \text{GHC}, \text{NSH}, \text{WH}, \text{DH}, \text{BH} \rangle$. *DISCLAIMER: Walking directions are in BETA. Use caution: this route may contain stairs. The 15–210 course staff assumes no responsibility if following these directions makes you late to class!*

4.3 A*

You may now be left with a rather sour view of Dijkstra's Algorithm. You would never go to the UC trying to find a shortest route from GHC to Baker, and yet the algorithm did just that. But why? Think about what intuition you're using that Dijkstra doesn't have.

Here's where A* saves the day.

Q: How is A* different from Dijkstra's?

A: Instead of examining just the distance $d(v)$ of a vertex v, it considers $d(v) + h(v)$, where h is a *heuristic function*. More about this in the *AbridgedLab* handout.

Q: What are the practical applications of A*?

A: Glad you asked! You're welcome :)

http://www.youtube.com/watch?v=DlkMs4ZHHr8