Recitation 8 — Dijkstra's Algorithm and DFS Numberings

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2013)

October 16, 2013

1 Topics

- Homework 6 is out. You will be required to make some modifications to Dijkstra's Algorithm and also use DFS to find bridges in graphs. We'll be reviewing Dijkstra and DFS today.
- The first midterm has been graded. If we have time, we will be covering common mistakes.
- Questions? Comments?

2 Dijkstra's Algorithm

2.1 Bingle® Maps

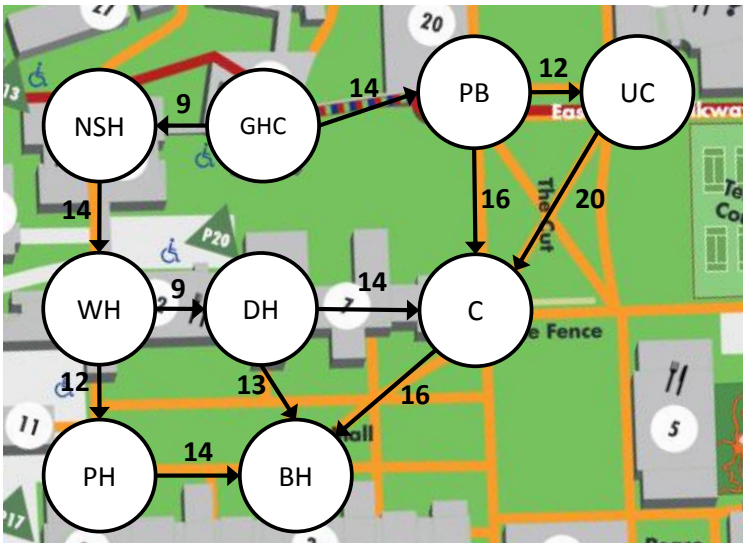
We will now review Dijkstra's single-source shortest path algorithm by way of an example. It's approaching noon on a Tuesday morning, and Professors Acar and Sleator need to get from their offices in GHC to 15-210 lecture in Baker. There are many ways to get there: they could take the Pausch Bridge to the cut, walk straight across to the Doherty entrance and then diagonally across to Baker, or they could take the bridges to Newell Simon and Wean, and then walk across the mall to Baker. Other routes are possible as well. However, they would like to find the fastest route. You decide to extend the Bingle search engine you founded in Lecture 8 to solve this problem. You realize that by representing the paths around CMU's campus as a graph, you can use Dijkstra's Algorithm to make sure the professors get to lecture on time.

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 $v \neq s$. We repeat the following steps until all nodes are explored:

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

Note that $D(v)$ is only ever updated to a smaller value, since we want to track the shortest path we've seen so far. $D(v)$ will always be greater than or equal to the length of the actual shortest path.

Performing Dijkstra's Algorithm efficiently requires an efficient way of finding the u with the minimum $D(u)$ in Step 1. This can be done by using a priority queue instead of a table to implement $D(u)$. We insert elements $(v, D(v))$ into the priority queue. Priority queues efficiently implement the `deleteMin` operation, which returns and removes the element (u, d) with the minimum d. We use this in Step 1 above. If the minimum element is a vertex that's already been visited, we throw it away and do another `deleteMin`. Now here's the tricky part: to update $D(v)$ for neighbors v, we just insert $(v, d + w_{uv})$ into the priority queue. Why do we not need to check if this is less than the existing value for v? Because the first time we visit v, if there are multiple pairs (v, d_1), ..., (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'll effectively overwrite the existing value. Cool!



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 each timestep, the table shows the node we visit at that timestep, the queue operations we perform during the step, and the values in the priority queue after the step. For simplicity, we only show the lowest value in the priority queue for each node, and don't visit already visited nodes.

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

All nodes have now been visited, so we can stop here. 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. Indeed, this is the shortest path to Baker: the path of distance 45 consisting of $\langle GHC, NSH, WH, DH, BH\rangle$. [1]

2.2 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. Dijkstra's Algorithm must visit the UC because, for all it knows, there's an edge of weight 1 directly from the UC to Baker, which would make this the shortest path. Applying your knowledge of geography, however, you realize that, barring some kind of wormhole[2], this would be impossible. You've used a *heuristic function*: a function that gives a problem-specific estimate of the shortest path from each node to the destination. In this case, your heuristic function is likely the Euclidean distance from each

[1] Walking directions are in Beta. Use caution - This route may contain stairs or construction. Bingle® Maps and the 15-210 course staff assume no responsibility if following these directions makes you late to class.

[2] The entrance to it would be in Entropy, of course.

place to Baker. What if, instead of ordering the nodes in the priority queue based on $D(v)$, we order them based on $D(v)$ plus the value of our heuristic function? This way, even though we have a short path to the UC, we wouldn't explore it because the heuristic tells us that this gets us farther from Baker, not closer. The A^* algorithm is a variant of Dijkstra that uses this technique. In Homework 6, you will implement A^*. The lab writeup gives a more precise formulation of the algorithm, and the requirements for the heuristic function. Get started early and let us know if you need more help understanding A^*.

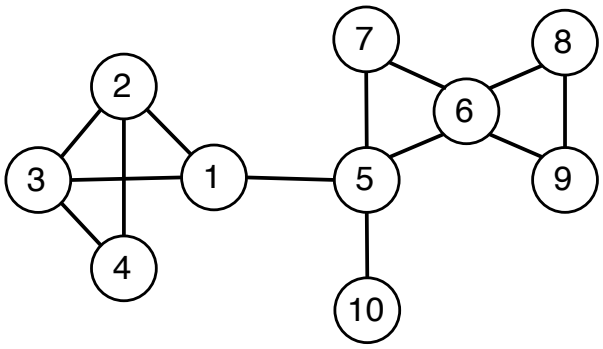
3 DFS

Recall the following definitions relating to trees:

- A *tree* is an undirected acyclic graph[3]
- A *root* is a distinguished node in a tree. Usually, it's drawn at the top of the tree.
- u is a *descendant* of v if the unique path from the root to u passes through v.
- u is an *ancestor* of v if v is a descendant of u.

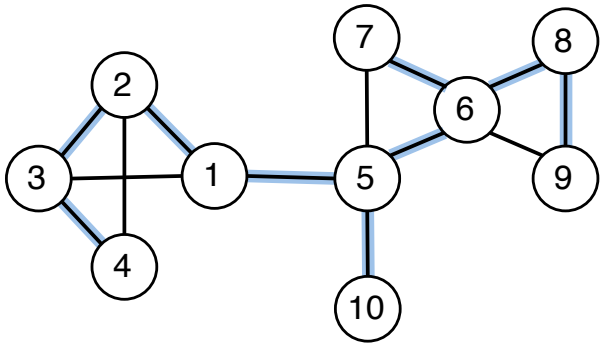
When you apply DFS to a graph, it implicitly defines a spanning tree rooted at the start vertex. If more than one tree is needed to span all vertices in the graph (when will this happen?), then we call it a *DFS forest*, although typically we just refer to it as the DFS tree. Edges in the graph that correspond to edges in the DFS tree are called *tree edges*. Edges in the graph that go from a vertex v to an ancestor u in the DFS tree are called *back edges*. Edges that go from a vertex v to a descendant u in the DFS tree are called *forward edges*. All other edges are called *cross edges*, since they cross from one subtree of the DFS tree to another.

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.) Think about why there can't be any cross edges.

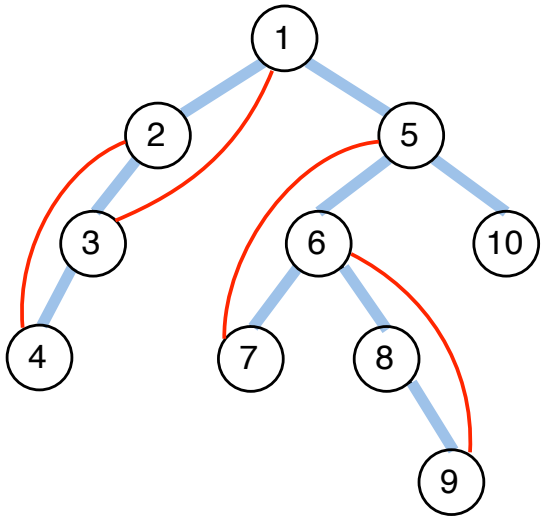


Show the DFS tree on the above graph. Use the vertices in increasing order.

[3] We sometimes draw trees as directed graphs, generally with all the edges facing away from the root. In this case, to satisfy the condition, the graph must remain acyclic if the edges were made undirected.



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



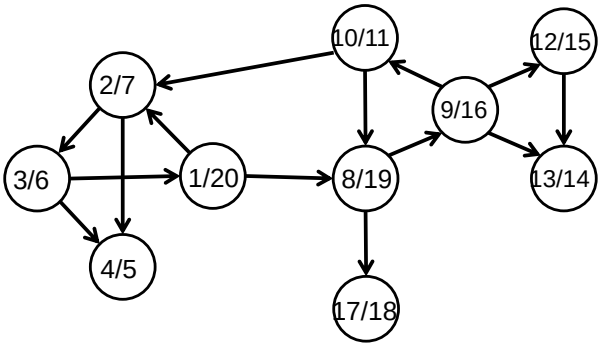
Recall the cycle detection algorithm from the lecture notes: if DFS finds an edge that we have visited before, we have found a cycle. Why is this sufficient? Does it find all cycles?

In the homework, you are to find all bridges in an undirected graph. A *bridge* of a graph 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?

3.1 DFS Numbering

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 discovery time/finishing time numbers for an example directed graph.



The following theorem relates DFS numberings to the structure of the DFS tree.

Theorem 3.1 (Theorem 22.7 in Cormen, Leiserson, Rivest and Stein, Introduction to Algorithms)**.** *For any two vertices u and v, exactly one of the three conditions holds:*

1. *the intervals* $[d[u], f[u]]$ *and* $[d[v], f[v]]$ *are entirely disjoint, and neither u nor v is a descendant of the other in the DFS forest.*
2. *the interval* $[d[u], f[u]]$ *is contained entirely within the interval* $[d[v], f[v]]$*, and u is a descendant of v in a DFS tree.*
3. *the interval* $[d[v], f[v]]$ *is contained entirely within the interval* $[d[u], f[u]]$*, and v is a descendant of u in a DFS tree.*

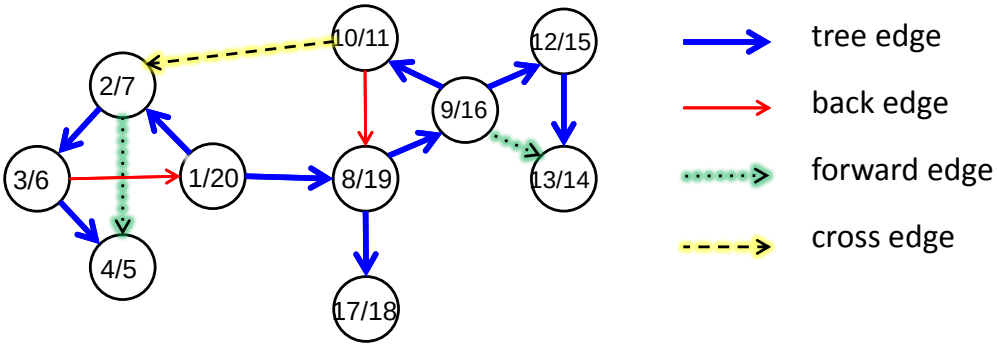
Q: Based on this theorem, how would you use the discovery times and finishing times to classify edges into tree, forward, back or cross edges?

A: An edge (u, v) is

1. a tree/forward edge if and only if $d[u] < d[v] < f[v] < f[u]$,
2. a back edge if and only if $d[v] < d[u] < f[u] < f[v]$,
3. a cross edge if and only if $d[v] < f[v] < d[u] < f[u]$.

We now classify the edges of the above graph into tree, forward, back or cross edges. The tree edges can be found by simply doing DFS. The non-tree edges can be classified using the hint above.

The tree edges of the example graph are shown in blue, forward edges in green, back edges in red and cross edges in yellow.



Q: How do you know if there is a cycle in a graph by using DFS numberings?

3.2 Homework 6 — Bridges

Q: Find a way to use the DFS numbering to identify bridges in a graph. Remember that we are working with undirected graphs for this problem.

4 Exam feedback

The average and median on the first midterm were 58.7 and 60.0, respectively, out of 100. As many of you have guessed, this exam was pretty tough. But keep in mind that it was tough for everyone.