CIS 121

Fei Liu

# Homework 07

Recitation 201

Collaborators:

---

**Q1.** This problem can be solved with a digraph, the vertices representing the houses and the directed edges representing the one-way streets. My algorithm takes in a graph, the source vertex, and the target vertex, and outputs the number of paths from the source to the target vertex. Here's what my algorithm does:

1. Initialize an empty set $S$ of vertices, initialize an empty stack of vertices, and initialize an integer variable called numOfPaths and set it equal to zero.
2. Mark all vertices white. Make all parents null. Add the target vertex, $C$ (Caroline's house), to the set.
3. Call DFS-visit on $H$, the source vertex.
4. Return numOfPaths.

DFS-visit:

1. Change $v$'s color to black. Push $v$ into the stack.
2. For each vertex $u$ in $v$'s adjacency list:

2.1 If $u$ is white, then set $u$'s parent to $v$, and call DFS-visit on $u$.

2.2 If $S$ contains $u$, then while the stack is not empty, pop all elements from the stack and add those elements to $S$. Then increment numOfPaths by 1. Else, pop from the stack.

P.O.C. (general explanation): My algorithm is based off DFS, but with a few changes. The set $S$ represents a growing cloud of vertices that have a path to the target vertex. The algorithm only calls DFS-visit on the source vertex because we're interested in how many paths there are from the source vertex to the target vertex, not from any other vertex to the target vertex. As the algorithm traverses each vertex in the graph, it will push each vertex into the stack along the way, and if it reaches a vertex that's in $S$, then it will pop everything from the stack, add it to $S$, and the total number of paths to the target vertex will have increased by one. If this does not happen, then the algorithm backtracks through the parent vertex, popping the vertices from the stack along the way. Because it pops vertices while it backtracks, this makes sure that vertices without a path to the target vertex are not mistakenly added to the set.

P.O.C. (formal): I will prove my algorithm's correctness by induction the number of vertices. The base case for one or two vertices is trivial. Let's consider a base case of three vertices, $v_1$, $v_2$, $v_3$. Let $v_1$ be our source and $v_3$ be our target. Do all possible cases on paper, and my algorithm returns the correct number of paths from $v_1$ to $v_3$ every time. The induction hypothesis is that for $n = k$, where $k$ is an arbitrary integer greater than zero, my algorithm correctly returns the correct number of paths from a given source vertex to a given target vertex in all possible cases of a graph of size $k$. For the induction step, I have a graph of $k + 1$ vertices. Now, I can split this step into two cases. The new vertex, denoted $v_{k+1}$, can either create an additional path to the target vertex, denoted $v_t$, or it will not. Let's look at the case where $v_{k+1}$ does create an additional path to $v_t$:

1. $v_{k+1}$ has an incoming edge from a vertex that is on a path from the source vertex $v_s$ to the $v_t$, and has an outgoing edge to a vertex that is a the path from $v_s$ to $v_t$. In this case, when the algorithm traverses $v_{k+1}$, it will see that the next vertex is already in the set, so $v_{k+1}$ will be added to the set, and numOfPaths will be increased by 1. $v_{k+1}$ is never traversed again after it backtracks to its parent, and the algorithm will return one extra path than before $v_{k+1}$ existed.

Now, let's look at the cases where $v_{k+1}$ does not create a new path:

1. $v_{k+1}$ can be placed in the middle of a path, thereby extending the path from $v_s$ to $v_t$. In this case, when the algorithm traverses past $v_{k+1}$, it will simply push it into the stack, and the vertex will be popped when the algorithm eventually

hits another path that leads to $v_t$, or $v_t$ itself. In this case, the algorithm does not increment numOfPaths any additional times from the presence of $v_{k+1}$.
2. $v_{k+1}$ is a sink. In this case, if the algorithm hits $v_{k+1}$ in its traversal, it will simply backtrack, popping vertices that were on the path to this dead end from the stack, until it backtracks to the last vertex with more than one outgoing edges. In this case, the algorithm did not increment numOfPaths when it reaches $v_{k+1}$, so the correct number of paths was still returned.
3. There is no path from $v_s$ to $v_{k+1}$. In this case, the algorithm never traverses to $v_{k+1}$, due to the nature of DFS-visit, and the algorithm does not increment numOfPaths an additional time.
These are all of the cases of what $v_{k+1}$ can be in relation to the other vertices of the graph, and in each case, my algorithm still returns the correct number of paths from $v_s$ to $v_t$.

Running time analysis: Step 1 of my algorithm is initializing objects, that takes constant time. Step 2 iterates through every vertex, that takes $\theta(n)$ time, and then adding a vertex to the set takes constant time. Step 3 is DFS-visit, and from class, we know that DFS-visit runs in $O(n + m)$ time, in the worst case where it traverses to every vertex in the graph. In step 1 of DFS-visit, I also add $v$ to the stack, which takes constant time, so this does not increase the asymptotic running time. In step 2.2 of DFS-visit, I add a new conditional to the DFS-visit from class. As the algorithm traverses the graph, the current vertex is pushed to the stack, and everything is popped when the algorithm meets a vertex from the set. Since DFS-visit only traverses each vertex reachable from $v_s$ exactly once, over the course of the entire algorithm, at most, only $n$ vertices will be pushed into the stack and at most, only $n$ vertices will be popped from the stack. Thus, the addition of the conditional in 2.2 does not increase the asymptotic running time of the DFS-visit. In step 4 of the algorithm, I simply return the correct answer, which takes constant time, so the running time of my algorithm is $O(n + m)$.

**Q2.** The rooms and boardwalks can be represented with a graph, the rooms being vertices and the boardwalks are edges. A crucial assumption in this problem is that there is a way to remove $k$ vertices such that the remaining $n - k$ vertices are still connected. This algorithm takes in a graph and a constant $k$ and outputs a length $k$ list of vertices that can be removed while keeping the graph connected. Here's what my algorithm does:
1. Initialize an empty list $L$ of vertices. Initialize an integer variable called count, set to zero.
2. Mark all vertices white.
3. Run DFS-visit on an arbitrary vertex.
4. Return $L$.
Let the current vertex that DFS-visit is on be $v$. Here's what DFS-visit does:
1. Mark $v$ black.
2. Initialize a boolean variable called isExpendableVertex, set to true.
3. For each vertex $u$ in $v$'s adjacency list, if $u$'s color is white, set isExpendableVertex to false, then call DFS-visit on $u$.
4. If isExpendableVertex is true and count is less than $k$, then add $v$ to $L$.

P.O.C.: The crux of this P.O.C. lies in the boolean variable isExpendableVertex. In DFS-visit, if all of the vertices in $v$'s adjacency list is not white, then this means that all of them has been reached via some other path, so the current vertex $v$ can be removed, and the graph will stay connected. If this is the case, then isExpendableVertex stays true throughout the for loop in step 3 of DFS-visit, and step 4 of DFS-visit will enter the conditional where it will add $v$ to $L$, which is to be returned by my algorithm. Step 4 of DFS-visit also increments the count variable whenever a vertex is added to $L$, and stops adding when the count variable reaches $k$, which means that $k$ vertices have been added to $L$. This ensures that the returned list does not return more than $k$ vertices.
Also, in step 2, I can start DFS-visit on an arbitrary vertex because the graph is connected.

Running time analysis: Steps 1, 2, and 4 of my algorithm take constant time. Step 3 is DFS-visit, which normally runs in $O(m+n)$ time. Every other step that I added/modified to DFS-visit takes constant time, so it does not affect DFS-visit's asymptotic running time. Thus, the running time of my algorithm, is $O(m+n)$.

**Q3.** This problem can be solved using a digraph, the vertices representing the buildings, and the directed edges representing the one-way streets. The graph is connected because it is given that there is a path from a given vertex to any other vertex. Furthermore, a path from any source vertex $v_s$ to any other target vertex $v_t$ must go through a central vertex (representing the High Rise Field) $v_c$. My algorithm takes in a graph, the central vertex $v_c$, and it outputs a size $n$ map with vertices being the key and a distance being the value. Here's what my algorithm does:
1. Initialize an empty map $M$ with vertices being the key and an integer as the value representing distance from the key vertex to $v_c$.
2. Using $v_c$ as the source vertex, run Dijkstra's algorithm on the graph. As vertices are added to the set of completed vertices, add an entry to $M$ with the newly added vertex as the key and the distance of that vertex (from $v_c$) as the value.
3. Return $M$.
If Jong Min wants to look up the shortest distance from $v_s$ to $v_t$, all he has to do is get the distance values of $v_s$ and $v_t$ from the map and add them together. The get operation in a map takes constant time, and adding the distances also takes constant time, so it takes constant time for Jong Min to look up the shortest distance from $v_s$ to $v_t$.

P.O.C.: I will prove my algorithm via contradiction. I assume that my algorithm does not return a map with correct distances from a given vertex to $v_c$. My algorithm uses Dijkstra's algorithm to find these distances, the only modification being that vertex and distance pairs are added to a map $M$. This means that Dijkstra's algorithm is returning the incorrect shortest paths from a source node to all other nodes, and that is a contradiction since Dijkstra's algorithm has been proven correct.

Running time analysis: Steps 1 and 3 of my algorithm take constant time to execute. Step 2 is simply Dijkstra's algorithm which runs in $\theta(m * lgn)$ time, which is $O(n^2 + mlgn)$. The extra step I added to Dijkstra is a put operation to a map, which takes constant time, so Dijkstra's running time does not change. Thus, the running time of my algorithm meets the constraint given by the problem.

**Q4.** This question is asking: if the outgoing edges from the source vertex are of negative weights, will Dijkstra's still return the shortest path from the source vertex to all other vertices? The answer is yes. The reason why Dijkstra works on a graph with only non-negative weights is because it's a greedy algorithm. At each step of edge relaxation, it looks for the next closest vertex, denote this $v$, to our "cloud" of vertices, while ignoring the possibility that there may be a shorter path to $v$ later down the road via a negative edge.
However, in this question, all negative-weighted edges occur only in the first iteration of the while loop in Dijkstra's algorithm (in the adjacency list of the source vertex). In the first iteration of the while loop in the algorithm, Dijkstra will find the shortest path from the source to each of its neighbors correctly. Then, in all future iterations of the while loop, there are no more negative-weighted edges, so all distances found from the source vertex as the "cloud" expands will be guaranteed to be minimal - if there were negative-weighted edges, there is a possibility that for a vertex in the "cloud" of vertices, a negative-weighted edge could make a new shortest path to that vertex - throughout the duration of the algorithm.

**Q5.**

(a) Assume towards a contradiction that there does not exist an $e' \in T_1$ that is of the same weight as $e$. Let's take a look at the unique cycle $C$ in $T_1 \cup \{e\}$. $e'$ is guaranteed to be in the cycle created by the addition of $e$ to $T_1$. If you sort the edges of $T_1$ and $T_2$ in ascending weight, and go down both lists of edges, if they are the same in both lists, they represent the same edges in the graph. At the first point where the two lists show a difference in edge weight, this is where $T_1$ and $T_2$ use up different edges from the graph. Since there are two different MSTs, there must be a cycle in the graph (otherwise, $T_1$ and $T_2$ are the same), so all edges after the point where the edge weights of $T_1$ and $T_2$ diverge, their weights must even out at some point later down the list, and this should be the point where the edges converge to the same vertex again on the graph.

W.L.O.G., $e'$ and $e$ are edges in $T_1$ and $T_2$ with different weights, respectively. In the previous paragraph, I proved that $e$ and $e'$ are part of the same cycle in $T_1 \cup \{e\}$. By the cycle property, neither $e$ nor $e'$ can be the edge with the maximum weight in $C$. If $w(e') < w(e)$, then $T_2$ would have $e'$ instead of $e$, and if $w(e') > w(e)$, then $T_1$ would have $e$ instead of $e'$, and this contradicts our assumption that there exists an edge $e' \in T_1$ and an edge $e \in T_2$ such that $w(e) \neq w(e')$ so it must be true that there exists an $e$ and $e'$ such that $w(e) = w(e')$.

(b) Every spanning tree during a transformation from one MST to another is also an MST because we can always edge flip edges of the same weight. I proved from part $a$ that between two different MSTs of the same graph, there exists an edge from each MST that have the same weight (and belong in the same cycle, so edge flipping is possible). This is true for a graph of ANY SIZE.

I will now prove the claim in the problem by induction. The base case for a graph of one or two vertices is trivial. In a graph of three vertices, it can obviously be seen that any MST can be transformed into another MST with just one edge flip, which is less than $n-1$. The induction hypothesis is that for a graph with $n = k$ vertices, where $k$ is an arbitrary positive integer, any MST of this graph can be transformed into another MST of the same graph with at most $k-1$ edge flips.

For the induction step, I want to show that for a graph with $k+1$ vertices, any MST of this graph can be transformed into another MST of the same graph with at most $k$ edge flips. Let's take one vertex away and apply the induction hypothesis. Now, add the vertex back to the graph, denote this new vertex $v$. Any MST of this graph will have $v$ connected to one other vertex, let's denote the edge connecting these two vertices $e$. If the MST that we want to transform into requires $v$ to be connected to some other vertex, denote this edge $e'$, then all we have to do is add $e'$ to the starting MST, then remove $e$, and you will end up with the target MST. We did this in one edge flip, so in the worst case scenario, if $k-1$ edge flips was already done before the vertex was added, then we were able to transform any MST to another MST in $k-1+1 = k$ flips, which was our limit.

With the induction step completed, I have proved that for a graph with $n$ vertices, where $n$ is an arbitrary positive number, it is possible to transform any MST of the graph to another MST of the graph in at most $n-1$ edge flips.