

COMS 311 EXAM 2 REVIEW

Have fun everyone. No grieving! When in doubt DNG... or write it out

Can't edit? Come back with an ISU email! Also, try clicking on updated link on piazza... I believe I updated it correctly...

Practice Exam 2

Information to (probably) be given on the exam

$O(m \log n)$ for Prim's algorithm, using heap-based priority queue.

$O(m \log n)$ for Kruskal's algorithm, using a pointer-based Union-Find structure (which we have not covered).

$O(m \log n)$ for Dijkstra's shortest path

Sorting - $O(n \log n)$

BFS/DFS - $O(|V| + |E|)$

Helpful pseudocode:

Dijkstra's:

Initially $\text{Dist}[v] = \text{infinity}$, $\text{Pred}[v] = \text{null}$ for all v ,

$S = \{s\}$

$Q = V - S$

$\text{Dist}[s] = 0$, $\text{Dist}[v] = c(s, v)$ for v adjacent to s

$\text{Pred}[v] = s$ for v adjacent to s

while S does not equal V

 remove u from Q with minimum $\text{Dist}[u]$

 put u in S

 for each neighbor v of u

 if v is not in S

 let $\text{alt} = \text{Dist}(u) + c(u, v)$ // see if there is a shorter route to v through u

 if $\text{alt} < \text{Dist}(v)$

 update $\text{Dist}(v) = \text{alt}$ and set $\text{Pred}[w] = v$

Prim's:

Initially $A[v] = \text{infinity}$, $\text{Pred}[v] = \text{null}$ for all v

Choose any starting vertex s

$S = \{s\}$

$Q = V - S$ (*)

$A[s] = 0$, $A[v] = c(s, v)$ for v adjacent to s

$\text{Pred}[v] = s$ for v adjacent to s

```
while S does not equal V
    remove u from Q with minimum A[u]
    put u in S (i.e., edge (Pred[u], u) becomes part of tree)
    for each neighbor v of u
        if v is not in S
            if  $c(u, v) < A(v)$ 
                update  $A(v) = c(u, v)$  and set  $Pred[v] = u$ 
```

(*) This corresponds to the idea that Q is a heap-based priority queue ordered by minimum $A[v]$. Initially it contains every node with $A[v] = \text{"infinity"}$. As we update $A[v]$, we perform a decreaseKey operation to maintain the heap ordering.

The "open set", or set of candidates to be added in each iteration, consists of the elements v of Q that have $A[v] < \text{infinity}$

Problem 1

1. Figure 1 shows a DAG. Write all possible topological orderings of this graph.

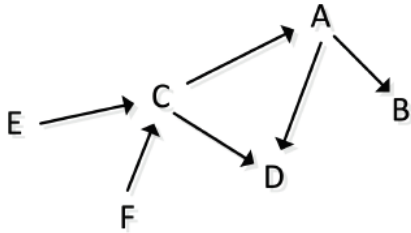


Figure 1: A DAG.

A: [Link to topological sorting geeks for geeks](#)

EF | C | A | BD

EF | C | A | DB

FE | C | A | BD

FE | C | A | DB

-Or-

EFCABD

EFCADB

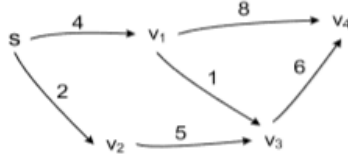
FECABD

FECADB

Why: You have two choices for the first node, E and F (**Because they have no incoming edges**). Which one dictates your next “choice”. From these you have only one option until the end, at which point you can choose between B and D (**Because they have no outgoing edges. They’re sinks!**). All told, that’s two start paths, and two end options, for four total paths.

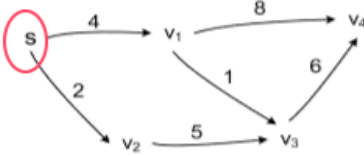
Problem 2

Execute Dijkstra's algorithm on the directed graph below. At each step **circle** the elements of the set S (nodes for which a shortest path is known), and show the predecessors and distances. (Distances that are not filled in are assumed to be "infinity".)



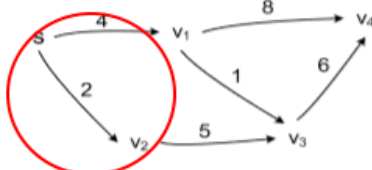
pred	
v1	
v2	
v3	
v4	

dist	
s	0
v1	∞
v2	∞
v3	∞
v4	∞



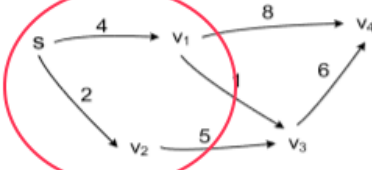
pred	
v1	s
v2	s
v3	
v4	

dist	
s	0
v1	4
v2	2
v3	∞
v4	∞



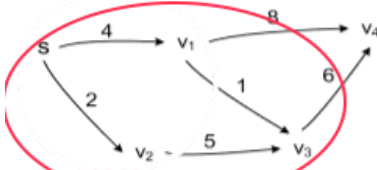
pred	
v1	s
v2	s
v3	v2
v4	

dist	
s	0
v1	4
v2	2
v3	7
v4	∞



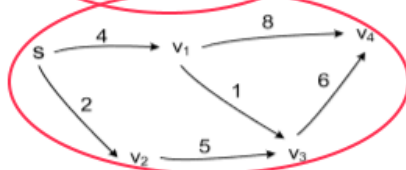
pred	
v1	s
v2	s
v3	v1
v4	v1

dist	
s	0
v1	4
v2	2
v3	5
v4	12



pred	
v1	s
v2	s
v3	v1
v4	v3

dist	
s	0
v1	4
v2	2
v3	5
v4	11



pred	
v1	s
v2	s
v3	v1
v4	v3

dist	
s	0
v1	4
v2	2
v3	5
v4	11

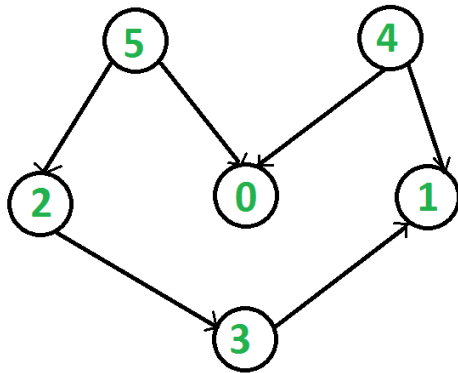
(From piazza confirmed by Steve) -

<https://piazza.com/class/jzpxqej11zo633?cid=296>

Problem 3

Q:

3. Give an algorithm that gets a DAG G as input and outputs the number of possible topological orderings.



A:

Per TA: Brute Force is the best algorithm

From Ling Tang's Recitation 11/6/19: "Brute Force: Find all topological orderings.

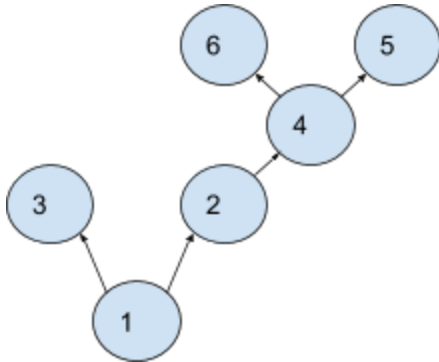
```
FindNumRec(G)
    If G has only one node
        Return 1;
    Count = 0
    Source[] = all nodes with 0 indegree in G
    For each S in source
        Count += FindNumRec(G - {S})
    Return count;
```

Time Complexity = # of topological orderings

EX: If you have a list of nodes with NO edges, then the # of topological sorts is $n!$. So, $O(n!)$ is the WORST CASE runtime for brute force."

```
count = 1
while (G is not empty) {
    set S = all nodes with no incoming edges
    count = count * |S|
    remove all nodes in S (and their edges) from G
}
return count
```

// Above algorithm doesn't work. Suppose graph:



Algorithm:

$S=\{1\} \rightarrow c=1*1=1$

$S=\{2,3\} \rightarrow c=1*2=2$

$S=\{4\} \rightarrow c=2*1=2$

$S=\{5,6\} \rightarrow c=2*2=4$

Done

$\#=4$

But, actual possible topological orderings (by hand):

123456 124365

132456 124635

124563 132465

124536 123465

124356

$\# = 10$

Topological graph means each node comes after any parents. 3's only parent is 1, so it just has to come after 1

See this example:

<https://www.geeksforgeeks.org/all-topological-sorts-of-a-directed-acyclic-graph/>

(spoiler warning that has a correct algorithm, we just have to count them at the end)

Problem: 3 only depends on 1, so it must go after 1, but can go anywhere before or after 2, 4, 5, and 6. The algorithm assumes each level away from s is connected to later layers, but this doesn't have to be the case. Pavan in class said the solution is closer to

performing DFS on each node in S as you're going through and deleting them. I think there's a little more to it since nodes like 3 in the above example can be slotted anywhere along the 2 possible DFS solutions of node 2. So maybe something like...

DFS(2) → 45, 46

3 can go anywhere so → 345, 435, 453, 346, 436, 463

→ above orderings are the ones missing from the original algorithm

So: → How do we detect nodes like 3 that can "go anywhere"?

→ How do we count the ordering produced by them?

A brute force attempt at #3

```
num_ways(Graph G){
    Int count = 0;
    For all nodes v with no incoming edges
        Count += solve(G, count,v);
    Return count;
}

Solve(graph G, int count, node v){
    Remove v and edges of v;
    If G is empty -> return count + 1;
    For all nodes w with no incoming edges in G
        Count += Solve(G,Count,w);
    Return count;
}
```


Problem 4

Q:

4. Write a pseudocode implementation of Dijkstra's algorithm using a priority queue that does *not* have a `ChangeKey` operation as described in the text. Can you still do this in $O(m \log n)$?

A:

Initialize S (Will be the SPT)

Initially $Q = \{s\}$

While Q is not empty

 Get u from Q with minimum $d[u]$

 If $u \notin S$ //what is S exactly??? <SPT Graph

 Put u in S

 For each neighbor v of u

 If v is not in S

 Let $alt = d[u] + c(u, v)$

 If $alt < d[v]$

 Put $(v, d[v])$ in Q

-what Steve wrote in class

Yes, we can still do this in $O(m \log n)$ time. At most, we can have m elements in Q. This gives us a runtime of $O(m \log m)$. We know that: $m \leq n^2$. Substituting this in we get $O(m \log n^2)$. From the properties of log, we can simplify this to $O(m 2 \log n)$. Thus we essentially get $O(m \log n)$.

And $d[u]$ is distance from s to u?

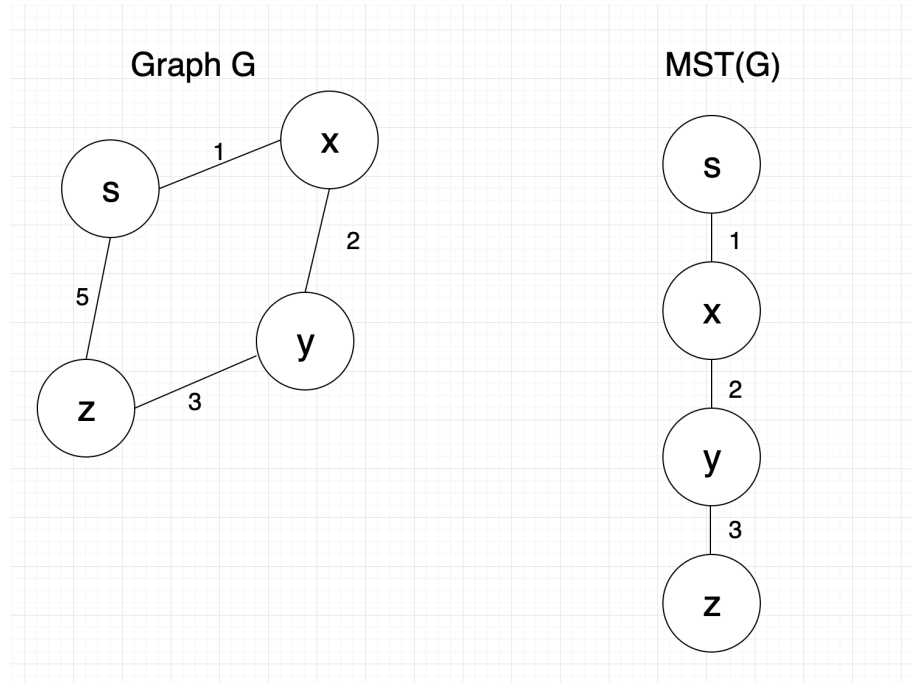
$D[u]$ is the current distance value of u (the distance from the starting no

Problem 5

Q: Is T a shortest-path tree?

5. Suppose we run Prim's algorithm on a graph G with s as starting node and let T be the MST constructed. Is T a shortest path tree? I.e., is the length of the path from x to any vertex v in T equal to the length of the shortest path from x to v in G ?

A: no



<https://piazza.com/class/jzpxqej11zo633?cid=310>

Explanation:

Prim's algorithm finds the MST of a graph taking advantage of the cut property, the fact that for any subset S of V such $S \neq V$, the edge from S to $V-S$ ("the outside world") with the lowest cost is part of the MST. It adds the node of that edge to S , then repeats the pattern. This creates a fully spanning tree, as every node at some point gets bumped by an edge. But, that doesn't mean the path from a node s to every other node is minimal. That's because Prim's is greedy in selecting the edge from S to $V-S$ with the lowest cost. Suppose s and p are elements in S and q is not. If there is a long path in S from s to p , a short path from p to q , and a medium-length path from s to q , Prim's algorithm will select the short path from p to q as this edge is part of the MST. But, the total distance from s to q , in this case, is $(s \text{ to } p) + (p \text{ to } q)$, which is a long path. The path directly from s to q is medium length and shorter than the path from s to p to q . So, the graph made by Prim's algorithm is not necessarily the shortest distance graph.

Problem 6

Q:

6. Section 3.3 (p. 94) from the text provides an overview of an algorithm to compute all connected components of an undirected graph and claims that the algorithm can be made to run in $O(m + n)$ time. However, the algorithm has the following step: “find a node that has not been visited by the search from s ”. At first glance, it seems that this step will take $O(n)$ time and thus the algorithm should be $O(n(m + n))$. Provide details showing how the algorithm can be made to run in $O(m + n)$ time.

A: In BFS we visit every node v once which takes $O(V)$ time. We also look at each adjacent edge of V , but we are only looking at every adjacency once. This becomes the sum of all the degrees of all nodes $\sum_u n_v = 2m$ by the handshaking lemma (for undirected graphs. Directed graphs = m). Add these two together and you get $O(n + 2m) = O(n + m)$ (I'm not sure how this answers the question....just use bfs?)

BFS uses a ‘visited’ array, where each vertex is set to 0 or 1. This is what is actually used in the algorithm to lower the time complexity.

If you use just a visited array wouldn't it be $O(n)$ time to find an unvisited node (i.e., scan through visited array, return first node that is unvisited)? NVM, bfs or dfs work fine.

- 1) Initialize all vertices as not visited.
- 2) Do following for every vertex 'v'.
 - (a) If 'v' is not visited before, call DFSUtil(v)
 - (b) Print new line character

DFSUtil(v)

- 1) Mark 'v' as visited.
- 2) Print 'v'
- 3) Do following for every adjacent 'u' of 'v'.
 - If 'u' is not visited, then recursively call DFSUtil(u)

Problem 7

Q:


7. Solve the *single-destination shortest path* problem: Given a directed graph G with positive edge weights, and a vertex s , find the shortest path from every vertex u to s .

A:

```
// i.e. Dijkstra's Algorithm

for each u in v:
    Dist[u] = infinity
    Pred[u] = null

let H be a heap
put s in H, Dist[s] = 0
while H is not empty:
    v = delete min of H
    for u in adjacent edges of v:
        if Dist[u] > Dist[v] + c(v, u):
            Dist[u] = Dist[v] + c(v, u)
            Pred[u] = v
            H.DecreaseKey(u)
```

 the instructors' answer, where instructors collectively construct a single answer


Notice that the problem is for paths **to** some sink s . Dijkstra's algorithm gives paths **from** a source s .


thanks! | 1

Updated 2 hours ago by Donald Stull

followup discussions for lingering questions and comments

☒ Resolved ☐ Unresolved

 **Anonymous** 1 hour ago
Ah! So in this case, we would just reverse the edges and then do Dijkstra's. Correct?

 **Steve Kautz** 56 minutes ago Yep.

Reply to this followup discussion

Another Approach:

$$\text{SDSP}(G = (V, E), s) \{$$
$$E' = \{ \}$$
$$\forall \langle v, u \rangle \in E \{$$

```

        Add  $\langle u, v \rangle, \text{dist}(\langle v, u \rangle)$  to  $E'$ 
    }
     $G' = (V, E')$  (construct inverse graph)
     $T^R = \text{Dijkstra}(G', s)$ 
    Compute  $T$  where  $\forall \langle u, v \rangle \in T^R, \langle v, u \rangle \in T$ 
}

```

Problem 8

Q:

8. What is the Big-O complexity of determining whether a directed graph has a cycle? What about an undirected graph?

A: Directed graph run time: $O(m+n)$
 Undirected graph runtime: $O(m+n)$

Ling Tang's recitation notes 11/6/19: "(For directed graph) It is only a cycle when we revisit a node in the same path. We need to maintain another stack when calling DFS that stores all nodes in current path. Maintain a Hash Table that keeps track of nodes in stack so search time is $O(1)$. Overall Big-O for a directed graph is $O(n + m)$. For undirected graph, we run a DFS or BFS and if we encounter a node we've already seen then we have a cycle."

Idea 1: Use Tarjan's Algorithm as inspiration.

Idea 2: Combine DFS, BFS with Tarjan's Algorithm

//I'm pretty sure you can just use DFS to detect cycles (correct me if I'm wrong) (You can)

Steve address this problem in class briefly by saying we can just check for a topological sort. If there isn't one then we can assume that there is a cycle. This should take $O(m+n)$ time. (But how do we check for if a top sort exists? -> Run the topsort algorithm seen at the bottom of the question)

For a directed graph: we can use DFS and upon exploring every node v , we can check if it appears in our stack. If node v is in our stack then there must be a cycle. You can also use topological sort I believe, if the algorithm doesn't terminate then the graph is not a DAG (i.e. it has a cycle)

For undirected graphs (specifically simple, undirected graphs - which are the only kind we've been dealing with in this class), we can just see if the number of edges is n or greater. (Think about how a tree has $n - 1$ edges. Any more than that, and it cannot be a tree - i.e. there must be a cycle somewhere). <= What about if the graph is not connected. Then this would not work because if there are 2 unconnected components, then the graph can have $n-1$ nodes and still contain a cycle.

If you're worried about unconnected graphs, run BFS on any node in G . As you run it, count the number of edges (and nodes) for that component. If after a component's BFS has finished the number of edges is the same or greater than the number of nodes in that component, you've found a cycle. Can also run DFS. Run DFS as far as possible, mark nodes as visited as you go. If you get to a node that is visited and the parent of the current node isn't the cycle inducing node, you have a cycle.

Alternatively, and probably more simply, is the union-find algorithm. Outlined better in the book, but basically, you start with all nodes in their own sets, then add 1 edge at a time. You look at the sets of u and v that e connects, and if they're in different sets, then you union their sets together to create 1 set. If they're in the same set, it means you're connecting u to v when there's already some other path from u to v , since that's the only way sets are unioned. Then, there's a cycle and you're done. Repeat for every edge. Runtime is $O(m+n)$ with a union find structure as defined in the book.

Here's some pseudo-ish code that uses colors for determining a cycle in directed graphs:

```
DFS_cycle(current, parent):
```

```
    If color[current] = black:      //Completely visited vertex
        Return
    If color[current] = gray:       //We've seen the vertex before,
    but not
                                    //finished, cycle detected
        Return that there's been a cycle detected
```

```
parent[current] = parent
color[current] = gray
```

```
For each edge (current, next) incident to current:
```

```
    If (next == parent[current]):  //We don't want to go back where
                                    //we just came from
        Do nothing
    Else:
        DFS_cycle(next, current)
```

```
color[current] = black            //Now completely visited
Return
```

Alternatively, we can check to see if there's a topological sort in a directed graph. (If there isn't a topological sort, then there must be a cycle).

To check if there's a topological sort:

Let H be a copy of G

While H is not empty:

 Find a node that has no incoming edges (alternatively, you could also look for a node with no outgoing edges)

 If there is a node with no incoming edges:

 Delete the node and its edges from H

 Else:

 We've found that a topological sort is not possible, thus there is a cycle

This can have some modifications done to it to make it a better runtime. For instance, when we delete a node that has no incoming edges, we can look right then and there at the nodes it is adjacent to, to see if those nodes will become nodes with no incoming edges after the current node is deleted.

For that method, if H is a copy of G and iterates over each node, the "While H is not empty" loop is $O(n)$. How are we finding nodes with no incoming edges in constant time? Can use a min priority queue and every time you find an edge with lower cost you decreaseKey. Very similar to dijkstras. (<https://www.cse.ust.hk/~dekai/271/notes/L07/L07.pdf>) — decreaseKey has runtime $O(\log n)$ because you have to reorder e once you've changed it. This is noted in the source. So wouldn't the algorithm be $O(n \log n)$?

Queue Q

Initialize an array counts of the incoming edges $//O(m + n)$

Find a node with no incoming edges, put into Q $//O(n)$

While Q is not empty: $//O(m)$ - we only look at each edge once

 Remove node u from Q

 Mark it as deleted from the graph

 For each outgoing edge (u, v):

 Decrement count for v $//$ this is $O(\log n)$ since v needs to be reordered in Q

 If count is 0, put it into Q


```

Prim( $G, w, r$ )
{
  for each  $u \in V$                                 initialize
  {
     $key[u] = +\infty$ ;
     $color[u] = W$ ;
  }
   $key[r] = 0$ ;                                       start at root
   $pred[r] = NIL$ ;
   $Q = \text{new PriQueue}(V)$ ;                         put vertices in  $Q$ 
  while( $Q$  is nonempty)                             until all vertices in MST
  {
     $u = Q.\text{extraxtMin}()$ ;                         lightest edge
    for each ( $v \in adj[u]$ )
    {
      if ( $(color[v] == W) \&\& (w[u, v] < key[v])$ )
         $key[v] = w[u, v]$ ;                          new lightest edge
         $Q.\text{decreaseKey}(v, key[v])$ ;
         $pred[v] = u$ ;
      }
    }
  }
}

```

Problem 9

Q:

Let $G = (V, E)$ be an undirected graph in which the edge weights are not necessarily distinct, and let S be any subset of V such that S is nonempty and $S \neq V$. Let $e(x, y)$ be an edge of minimal weight such that $x \in S$ and $y \in V - S$. Prove that there exists a MST that contains edge e .

A: *From lecture 11/5/2019: “Claim: If S & $V-S$ are nonempty and e is an edge of minimum cost with x in S , y not in S , then there exists a MST containing e . Proof: Let T be any MST. There is some path from x to y in T . This path must include some edge $e' = (x', y')$ with x' in S and y' not in S . Consider $T' = T - \{e'\} \cup \{e\}$. T' is still a spanning tree for G , and since e has minimal weight, we know $\text{cost}(e) \leq \text{cost}(e')$. Hence the cost T' is less than or equal to the cost of T , i.e., T is a minimum spanning tree.”*

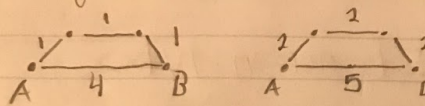
Problem 10

Q:

Let G be an undirected weighted graph. Let T be its MST and Q be its shortest path tree (from a node s). Suppose we increase the weight of every edge by 1. Is T still an MST for the new graph? Is Q still the shortest path tree for the new graph? Prove or give a counterexample.

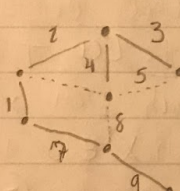
A: **From lecture 11/5/2019:** "T is still an MST for the new graph. Let T be an MST for a weighted, undirected graph G . Let G' be the graph contained by changing $\text{cost}(e)$ to $\text{cost}(e)+1$ for every e in E . Then, T is still a MST for G' . Total weight of any spanning tree T' for G is exactly $(n-1)$ more than weight of T' in G ." (Feel free to correct my notes if I wrote anything down incorrect)

10.



Let T be a MST for weighted, undirected graph G . Let G' be the graph contained by changing $c(e)$ to $c(e)+1$. For every $e \in E$. Then T is still a MST for G' .

Total weight of any spanning tree T' for G is exactly $(n-1)$ more than weight of T' in G .



Q is not the shortest path tree (starting at s) for the new graph. Counterexample:

S--4--B

| |
1 1

| |
C--1--D

For this tree it is cheaper to get to B through C and D.

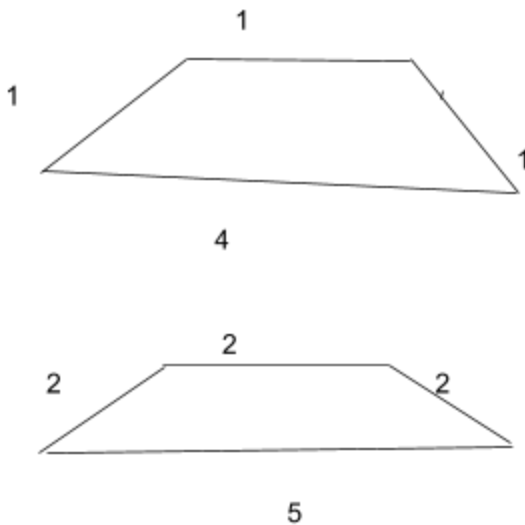
After adding 1 to each weight:

S--5--B

| |
2 2

| |
C--2--D

For this tree it is cheaper to go directly from S to B.



Q would still be valid, as an MST tries to minimize total cost, not cost between vertices. The path from S to B would be smaller taking 5 vs 2+2+2, but the MST itself, once vertices C and D are included, is much larger. ← I think you're confusing Q and T. T is the MST, and is still valid after the weight changes. Q is the shortest path tree, and is no longer valid after the weight changes (see above for the counter example). ← Ah I was thx

Problem 11

Q:

11. Prove the correctness of Kruskal's algorithm using an exchange argument.

A:

Exchange Arguments

They work by showing that you can iteratively transform any optimal solution into the solution produced by the greedy algorithm without changing the cost of the optimal solution, thereby proving that the greedy solution is optimal. Jul 29, 2013

- 1) Sort edges by cost
- 2) Insert edge and make sure that it does not create a cycle

Proof: Let T be an optimal MST. The weight ($w(T)$) is the smallest weight possible. Let e be the lowest cost edge that is in T_a that is not in T . Add e to T . There is now a cycle in T so we will remove e' , the largest weight now in T . This will create T' . Now it is evident that $w(T') \leq w(T)$ so it is optimal.

Problem 12

Q:

12. let m_1, m_2, \dots, m_n be distinct non-integers on the number line, in increasing order. Your goal is to color all of them blue. You have magical blue pens with the following property: When you place the pen at co-ordinate x , all the points in the range $[x - 5, x + 5]$ turn blue. A pen can be used only once. Give an algorithm to color all the points using as few pens as possible. Prove the correctness of the algorithm and derive the run-time.

A:

Algorithm:

While there are still unmarked elements

Find the current lowest unmarked element

Find the floor of that element as it is a non-integer ($1.3 \rightarrow 1$) *edit

Add 5 to that element and place the pen there

Mark the numbers that are within 5 before the pen and 5 after the pen

Proof:

Assume you can remove a pen from my algorithm and still mark all the points to make a more optimal algorithm.

My algorithm will have no overlap and the first point marked by a pen will always be at 5 less than the pen's position making it marked at the very edge of where the pen marks things blue.

Removing a pen will always unmark at least that one point.

Then you need to be able to shift the current pens to the right to mark that point

That is not possible because any shift to the right will unmark the point that is at the far left edge of the pen's marking range for at least one pen.

Therefore my algorithm will provide the optimal number of pens every time.

$O(n)$

Proof by exchange argument:

Say there is an optimal solution that has pens starting at location O_1, O_2, \dots, O_n , and in our greedy algorithm the pens start at location G_1, G_2, \dots, G_m . Say the first i locations $O_1 \dots O_i$ and $G_1 \dots G_i$ are the same for both solutions, we want to prove that by changing O_{i+1} to G_{i+1} , we will not increase the number of pens needed in the optimal solution.

To prove this, we can say that because our greedy algorithm always starts at the most right location that marks the smallest number blue, any O_{i+1} would be smaller than G_{i+1} in order to still color every number blue. Hence swapping G_{i+1} with O_{i+1} will not increase the number of pens needed. Greedy algorithm proved to be optimal.

Problem 13

Q:

13. You have n tasks to complete, where each task takes exactly one unit of time. Each task i comes with a deadline time d_i , and a value r_i , such that if you complete it by d_i , you will be rewarded r_i dollars. Give a greedy algorithm that picks tasks so that it will maximize the rewards. Prove the correctness using an exchange argument.

“Sort the jobs from most to least reward, so now $r_1 \geq r_2 \geq \dots \geq r_n$.

For $i = 1$ to n :

 find latest possible open interval $[a, a+1]$ such that $a+1 \leq d_i$

 if one exists, put job i in this interval.

So, for example, this algorithm puts the first job (highest reward) at the latest possible time, just before its deadline.

Like everything else, it's helpful to go through examples. Since you "know" there's a greedy algorithm, you can come up with a couple simple ideas. Each time, see if you can find a counter example. If you find one then you get a better understanding of the problem, and come up with another idea. A couple of iterations of this should lead you to the right approach.

For the proof, you can proceed by induction on the number of jobs. The base case is easy. For the induction step, begin by showing (using an exchange argument for example), that there is an optimal solution with the first job (highest reward) scheduled at the latest possible time.”

<https://piazza.com/class/jzpxqej11zo633?cid=311>

Can anyone else try to explain this please??? This solution is very difficult to understand.

Homework 4 Canvas Solutions

Homework 5 Canvas Solutions

Frequently Asked Questions

- How to find/remove a cycle from a graph?
 - One (probably not great) solution is to run BFS, with the exception of any time you encounter an edge that leads to a node that has been seen, delete that edge.
- Difference between MST and SPT?
 - MST is the graph with the shortest tree overall connecting all of the nodes while SPT is the shortest path to each individual node from a starting point typically referred to as s .
- Is a directed graph strongly connected?
 - Do a BFS on G
 - Do a BFS on G^{reverse}
 - If all nodes are reachable in both directions, then G is strongly connected.

Links to helpful sites/piazza posts/canvas pages

<https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>

Found an awesome YouTube Channel that does a good job of explaining (exchange argument, Prim vs Dijkstra, etc):

<https://www.youtube.com/channel/UCmJz2DV1a3yfgrR7GqRtUUA/videos>

Mememes to get you through the day

You swerve to avoid a squirrel. Unknown to you, the squirrel pledges a life debt to you. In your darkest hour, the squirrel arrives.



“Where is the albino squirrel of good luck post when I need them the most”

“In order to understand recursion one must first understand recursion”

How to do good on the exam:

