CS2040 2020/2021 Semester 2 Final Assessment

MCQ

This section has 10 questions and is worth 30 marks. 3 marks per question.

Do all questions in this section.

1. You are given a connected, undirected graph, where all edges have the same weight. You are asked to find a valid MST of this graph, by printing out the edges belonging to this MST. If there are multiple valid MSTs, you only need to output any one of them. The best algorithm to do so runs in worst case

a. O(1) time
b. O(V) time
c. O(V + E) time
d. O(E log V) time

Details:

Since all edges have the same weight, any spanning tree of this graph is also an MST of this graph. As such, we can simply get a spanning tree of this graph in O(V + E) time via BFS or DFS.

2. Which of the following operations will run in worst case O(log n) time or better? Assume that the following data structures only contain **n** distinct keys.

i. Removing the smallest element in a binary max heap
ii. Removing the smallest element in a BST
iii. Removing the smallest element in an AVL tree

a. (i) and (iii)
b. (ii) only
c. (ii) and (iii)
d. (iii) only

Details:

Removing the smallest element in a max heap takes O(n) time, since the smallest element can be any of the n/2 children in the max heap. There is no other way to determine the smallest element apart from iterating over these children.

Removing the smallest element in a BST requires O(h) time to find the smallest element, followed by O(h) time to shift references (finding successor etc) after deleting the element, such that the BST remains valid. However, h may be up to n-1, and thus may take up to O(n) time.

Removing the smallest element in an AVL tree requires O(log n) time to find the smallest element, followed by O(log n) time (finding successor etc) to shift references after deleting the element, followed by up to O(log n) rebalancing operations. However, each rebalancing operation runs in O(1) time, so the total time taken is O(log n).

3. There are 24 different ways to insert each of the integers 1, 2, 3, 4 one at a time into an initially empty AVL tree. Of these, how many of them will result in at least one rebalancing operation?

a. 8
b. 12
c. 16
d. 20

Details:

There is only one possible case where a rebalancing operation will occur. This is when the second and third integers to be inserted into the AVL tree are both smaller or both larger than the first integer (causing the tree to be left heavy and right heavy respectively). If this is not the case, then the resulting tree after the first 3 insertions will be perfectly balanced (all nodes have balance factor 0), and adding a fourth integer will not be able to cause a rebalancing operation to occur. As such, the possible ways are as follows:

1,2,3,4
1,2,4,3
1,3,2,4
1,3,4,2
1,4,2,3
1,4,3,2
2,3,4,1
2,4,3,1
3,1,2,4
3,2,1,4
4,1,2,3
4,1,3,2
4,2,1,3
4,2,3,1
4,3,1,2
4,3,2,1

For a total of 16 ways.

4. You are given a special undirected graph with **V** vertices (**V** is guaranteed to be a multiple of 4), and every vertex has exactly one edge adjacent to it. What is the minimum number of connected components in this graph?

a. 1
b. 2
c. V/4
d. V/2

Details:

In this graph, for every vertex u with an edge to another vertex v, vertices u and v form a single component. It is impossible to have more than 2 vertices in a single component (since each vertex has only one incoming/outgoing edge, it is impossible for either u or v to connect to any other vertices). Thus, the minimum number of components is V/2.

5. In any valid binary max heap, we can obtain a list of the elements in sorted order by doing:

a. Preorder traversal
b. Inorder traversal
c. Postorder traversal
d. None of the traversal algorithms work

Details:

A binary heap (unlike a BST) does not have its elements stored in an order that allows for any traversal algorithm to consistently give a sorted order of its elements. Heap sort taking O(n log n) time, despite buildHeap() running in O(n) time, and the above traversal algorithms also running in O(n) time, should also offer a hint as to the correct answer.

6. You are asked to draw a simple directed graph with 5 vertices, which form a total of 2 strongly connected components. The maximum number of edges in this graph is:

a. 10
b. 12
c. 14
d. 16

Details:

Consider the 2 SCCs to be labelled A and B respectively, and the number of vertices in each SCC to be |A| and |B|. The maximum number of edges that will be present are:

(|A|)*(|A|-1) (draw edges between all vertices in A, in both directions), +
(|B|)*(|B|-1) (draw edges between all vertices in B, in both directions), +
(|A|)*(|B|) (draw edges from all vertices in A to all vertices in B, in one direction only)

Since |A| and |B| must both be at least one, we have the following possible values for (|A|,|B|): (1, 4), (2,3), (3, 2), (4, 1). The pairs (3, 2) and (4, 1) can effectively be ignored, since they will have the same result as (2, 3) and (1, 4) respectively. Among these 2 options, (1, 4) gives the best result of 16 edges, as compared to 14 edges for (2, 3).

7. You are given an undirected graph G, in a graph data structure of your choosing. Which of these DSes will allow you to check which vertices are reachable from a vertex X in worst case < O(V^2) time if E= O(V)?

i. Adjacency Matrix
ii. Adjacency List
iii. Edge List

The following options are independent (eg. picking (i) and (iii) means that it is possible to achieve this with either an adjacency matrix only, or an edge list only for storing the graph. However, you can use additional auxiliary data structures for your algorithm)

a. (ii) only
b. (i) and (ii)
c. (ii) and (iii)
d. (i), (ii), and (iii)

Details:

Checking for reachability with an adjacency list can easily be done via BFS or DFS. Checking for reachability with an edge list can be done using UFDS (union two sets whenever an edge is encountered, then find all sets in the same set as X).

8. Below is the p array of a UFDS. Union-by-rank and path compression are not used in this UFDS.

Index	0	1	2	3	4	5	6	7
p	2	1	2	1	3	4	4	5

What are the sizes of each set in the UFDS? Eg. an answer of 3,5,5 means that there is 1 set with size 3, and 2 sets with size 5.

a. 1, 1, 2, 2, 2
b. 2, 2, 4
c. 2, 6
d. At least one set in this array is invalid

Details:

Sets 0 and 2 are in the set with representative item 2.

The remaining sets are in the set with representative item 1.

9. The following is an adjacency list of a directed, unweighted graph G. The value of some fields are unknown, however it is known that the list of neighbours for each vertex is always in ascending order.

0	2	
1	?	4
2	3	
3	1	
4	?	
5		

Given this information, deduce if the graph may have the following properties. The options are independent (ie. picking (i) and (iii) means that it is possible for the graph to be acyclic (by filling in the unknown cells in a certain manner), or consist of more than one SCC (by filling in the unknown cells with a possibly different set of values from (i))).

i. G may be acyclic.
ii. G may be cyclic.
iii. G may consist of two SCCs or less.

a. (i) only
b. (ii) only
c. (ii) and (iii)
d. (i), (ii) and (iii)

Details:

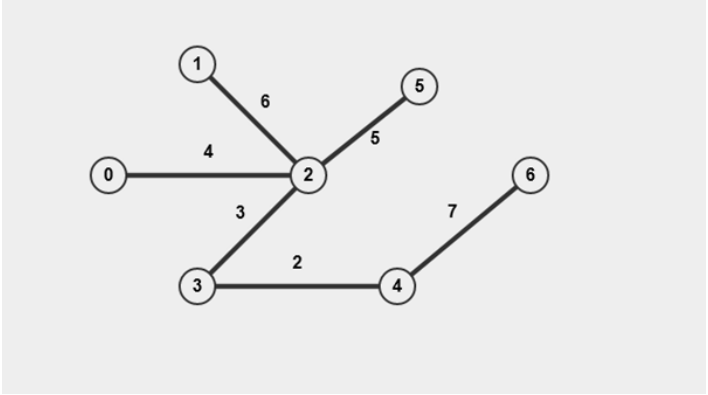
It is impossible for the graph to be acyclic. The first neighbour of vertex 1 must be 0, 2, or 3 since the list is sorted, and an edge from 1 -> 1 is invalid. All 3 vertices can reach vertex 1 via 0 -> 2 -> 3 -> 1, thereby always forming a cycle.

Since it is impossible for the graph to be acyclic, it must be cyclic.

It is impossible for the graph to contain only 1 SCC, since vertex 5 has no outgoing edges. As vertex 5 is unable to reach any other vertices, vertex 5 forms a single SCC on its own. As such, the goal is to determine if vertices (0, 1, 2, 3, 4) can form a single SCC.

We already know that (0, 1, 2, 3) can be an SCC if the unknown neighbour of vertex 1 is 0, since this forms the cycle 1 -> 0 -> 2 -> 3 -> 1. Therefore, to extend this SCC to include vertex 4, a vertex in this cycle must be able to reach vertex 4 (easily done via the edge 1 -> 4), and vertex 4 must be able to reach a vertex in this cycle (which can be done by setting the unknown neighbour of 4 to any vertex in the cycle). As such, it is possible for the graph to consist of two SCCs.

10. The following is one possible valid minimum spanning tree of a graph with 7 vertices:



It is known that the original graph consists of 9 edges. Determine the smallest possible total cost of the 3 edges which are not in the MST.

a. 9
b. 10
c. 11
d. 12

Details:

An edge e is not picked to be in the MST if it forms a cycle. Therefore, to answer this question, we look at this MST as it is constructed via Kruskal's, and determine if there are any edges that could possibly have been picked, but would form a cycle.

After picking the edge (3-4) with weight 2, there are not enough vertices to cause a cycle to be formed yet.

After picking the edge (2-3) with weight 3, an edge from (2-4) would form a cycle. This edge must have a weight of at least 3, otherwise the edge (2-4) would be preferred over (2-3).

After picking the edge (0-2) with weight 4, edges (0-3) and (0-4) would form a cycle. These edges must have a weight of at least 4, otherwise they would be preferred over (0-2).

Therefore, the smallest total cost is (3+4+4) = 11

(others)

With a UFDS using union-by-rank and path compression, it is never the case that we will put a set with a larger height under a set with a smaller height. (false)

After running Kahn's algorithm on a directed graph (with cycles), any vertices that are unprocessed must be part of some cycle. (false)

Given only the parent array of a UFDS, output the size of each disjoint set in this UFDS in O(n) time, where n is the size of the array.

Analysis

This section has 4 questions and is worth 12 marks. 3 marks per question.

Please select True or False and then type in your reasons for your answer.

Correct answer (true/false) is worth 1 marks.

Correct explanation is worth 2 marks. Partially correct explanation worth 1 marks.

Do all questions in this section.

11. For fast heap create, we can still get a valid max heap of size N in $O(NlogN)$ time if we go from index N back to index 1 but call shiftUp instead of shiftDown.

    False. Consider a heap that is initialized as [1,20,40,30], the above algo will result in [40,1,30,20] which is not a valid max heap.

    Grading scheme:

    Providing a counterexample where the algorithm does not produce a valid max heap will get 2 marks for explanation. Arguing that the time complexity is O(N log N), while true, will not be awarded any marks, since this does not address the validity of the resulting max heap.

12. Given an array of N integer, constructing a PQ implemented using an AVL tree can achieve the same time complexity as a PQ implemented using a min heap in the worst case.

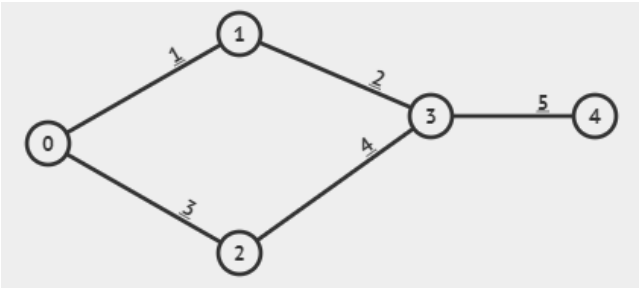
    False. For a min heap, fast create can insert all N integers in O(N) time, while for an AVL tree, it will take O(NlogN) time in general

    Grading scheme:

    Showing the correct time complexities of constructing a PQ using an AVL tree (O(N log N)), and of using a min heap (O(N)) will get the 2 marks. Giving only one of them correctly will give 1 mark, unless both time complexities given are the same, at which point 0 marks are awarded (because then you'd be arguing that the statement is true)

13. In a weighted undirected graph **G** where the edge weights are all distinct, the minimax path between a given pair of vertices **A** and **B** must only be the path from **A** to **B** in the MST of **G**.

    False. In the example given below, the minimax path from 0 to 4 in the MST of G is 0-1-3-4. However, 0-2-3-4 is also a minimax path from 0 to 4.



Grading scheme:

Showing a counterexample where there are multiple valid minimax paths from the same vertices A and B in a graph is sufficient to get the 2 marks. However, the following mistakes will not be awarded full marks:

1. Assuming that minimax path is SSSP path

2. Showing a counterexample where edge weights are not distinct

3. Arguing that there are multiple valid MSTs in the graph, and claiming that a single MST cannot cover all possible minimax paths as a result

14. Given the keys of a BST in in-order sequence and no other information, we can always reconstruct the exact same BST from the given sequence.

False. For example, if we have the sequence 0,1,2, there are actually many possible valid BST.

```
0          0          1          2
 \          \        / \        /
  1    or    2   or  0   2  or  0        among others
   \        /                  \
    2      1                    1
```

and there is no way to reconstruct the exact same BST that gave the in-order sequence given only the in-order sequence.

Grading scheme:

Showing an example of an inorder sequence with multiple valid BSTs will be awarded the 2 marks. Incorrectly deducing the number of valid BSTs (eg. claiming there are only 3 possible BSTs for the sequence (1, 2, 3)) is not penalised, as long as you show that more than 1 valid BST can be produced from the inorder sequence.

Owing to a suspected issue with LumiNUS Quiz not properly saving consecutive whitespaces, students who attempted to draw BSTs using ASCII characters may find that their trees are not represented correctly in the final answer. This is not penalised.

Structured Questions

This section has 6 questions and is worth 58 marks.

Write in **pseudo-code**.

Any algorithm/data structure/data structure operation not taught in CS2040 must be described, there must be no black boxes.

Partial marks will be awarded for correct answers not meeting the time complexity required.

15. [8 marks] Given an AVL tree **A** of size N containing unique integer keys, describe an algorithm to perform the following operation:

    GetAll(left,right): store all keys in **A** where **left** ≤ key ≤ **right** in an arrayList **B** and return **B**

    This operation should take in the worst case $O(logN + K)$ time where K is the number of nodes where left ≤ key of node ≤ right

    Ans:

    This is similar to problem 2 in tutorial 7.

    0. Modify successor so that we do not start from the root each time but start from the current node to search for the successor.
    1. Search for left in A. If found, add left to B. Otherwise insert left in A. <- O(logN) time
    2. Now starting from node containing left, keep calling successor until we hit a node >= right. For all nodes encountered along the way add their key to B. If the last node is == right add it to B and stop, otherwise just stop <- O(K) time
    3. Remove left if it is inserted in A in step 1. <- O(logN) time

    Total time complexity is O(logN+K).

    Grading Scheme:

    8 marks: O(logN+K)
    5 marks: O(K log N)
    3 marks: O(N + K log N) / O(N)
    2 marks: O(N log N)

Major mistakes:
3 marks: If algorithm is O(logN=K) but misses out quite a number of nodes
1 mark: If other major mistake(s) in algo

-1 mark each for the following mistakes:
1. pred/succ(node) (not just value) but did not explicitly mention that it is modified
2. each no explanation of how floor()/ceil() works
3. each if rank(left/right) which doesn't work with non-existent element
4. if add past last/first element

16. [6 marks] Given a min heap **A** of size N containing integer keys, describe an algorithm to perform the following operation:

deleteSubTree(val) - delete the entire subtree where the node with key == val is the root. If there is no node with key == val in **A**, nothing is done. After deletion the resultant tree should still be a min heap (meaning that extractMin and insertion operations should still work without any changes).

This operation should take in the worst case $O(N)$ time.

Ans:

1. Perform pre-order traversal of A. If current node is val, do not traverse it's left and right child.

2. For each node visited in step 1, add to another 1-based array B.

3. Perform fast heap creation on B and copy back to A.

Step 1 and 2 is basically pre-order traversal of A which can be done in O(N) time. Step 3 can be done in O(N) too. Thus total time taken is O(N).

Grading scheme:

Correct Solutions:
1. O(n) -> 6 marks
2. O(n log n) -> 4 marks

Deduction for the following mistakes:
1. Correct idea, but traversal fails to "collect" all required nodes/indices -> -2 marks
2. Half an algorithm (e.g., algorithms recursively finds all nodes that need to be removed but does not anything further) -> -2 marks (depending on how much is there)
3. The result heap is not a valid heap, i.e., it fails to fulfil the heap properties -> -2 marks
4. The code or description is vague or ambiguous -1 to -2 marks

17. [15 marks] Country X has N states (labeled 0 to $N-1$). When citizens of country X are born, they will register with the state they are born in and be given a unique integer as id.

During the COVID19 pandemic, zoning by state was implemented, and citizens can only move about within the state they are registered in and cannot cross state lines. However as the pandemic becomes less severe, adjacent states are merged into larger zones where the citizens within can freely move. Ultimately once the pandemic is over, citizens can once again freely move throughout country X.

In order to track citizen movement during the pandemic, your job as a government employee is to come up with appropriate data structure(s) and algorithms to implement the following operations as efficiently as possible.

1. **Initialization()** - initialize your DS(es) with a list of size M which contains the id and state number of each citizen. There should be N zones with each state being a zone

2. **RegisterBirth(v, s)** - register a new birth (using **v** as the unique integer id) with state **s**.

3. **Merge(s1,s2)** - merge the 2 zones that state **s1** and state **s2** belong to, if they do not already belong to the same zone.

4. **LegalMovement(v, s)** - check if a citizen with id **v** can legally move about in state **s**. If possible return true else return false. If there is no citizen with id **v**, return false too.

Please analyze and state the time complexity for the operations you have implemented in terms of N and M.

Marks distribution:

Initialization(): 4 marks

RegisterBirth(v, s): 3 marks

Merge(s1,s2): 3 marks

LegalMovement(v, s): 5 marks

Ans:

Initalization():

1. Create a UFDS U of size N. Each disjoint set represents a state. <- O(N) time
2. Create a hashmap H of <key,value> pairs where key is the citizen id and value is the state number of the state the citizen belongs to.
3. Go through the list of citizens and insert a <citizen id, state number> pair into H. <- O(M) time

Total time complexity is O(N+M)

RegisterBirth(v,s):

1. insert <v,s> into H. <- O(1) time

Merge(s1,s2):

1. Perform U.unionSet(s1,s2) <- O(1)/O(α(N)) time

LegalMovement(v,s):

1. if H.find(v) == null return false <- O(1) time

2. s1 = H.find(v) <- O(1) time

3. if (U.findSet(s) == U.findSet(s1)) return true else return false <- O(1)/O(α(N)) time

Total time complexity is O(1)/O(α(N)) time

Grading Scheme:

Poor efficiency (assuming totally correct solution):
1. For each operation less efficient than model answer, deduct all marks leaving 1 mark for that operation
2. If multiple operations have time complexity T where O(M) <= T < O(M^2) or O(N) <= T < O(N^2): cap deduction at 5 marks
3. If multiple operations have time complexity >= O(M^2) or >= O(N^2): cap deduction at 10 marks

Deductions due to mistakes (correctness problems)

Init:
-1 mark for wrong implementation of hashmap affecting LegalMovement
-2 marks if use hashmap of <citizen v, state> only and no UFDS

RegisterBirth:
-1 mark if no clear expansion policy if using UFDS of citizens

Merge:
-2 marks if manually implementing unionSet() without representative item, union by rank or path compression (also penalize poorer efficiency in other operations if that occurs)

LegalMovement:
-1 mark if did not check for whether v exist or not
-2 marks if did not use Hashmap to find state efficiently (and penalize in init)

Deductions for other minor mistakes:
1. -1 mark for each time complexity of algo is missing / wrong analysis
2. -1 mark for each operation affected by poor efficiency due to poor manipulation of hashmap that can be easily corrected
3. -2 marks if state and citizens are in same hashmap (state number and citizen id may clash)
4. -1 mark (if not already -2 above) if assuming citizen ids are running numbers (might not be true)

18. [7 marks] There is a game where you have N dots numbered from 1 to N, and for some pairs of dots **A**,**B** there is an arrow going from **A** to **B** (if there is an arrow going from **A** to **B** then there will not be an arrow going from **B** to **A**) . There are M arrows in total.

The objective of the game is to place the pencil at some starting dot and trace the arrows in such a way as to go through as many dots as possible without lifting the pencil (you may traverse some arrows or dots multiple times to achieve this).

Given there is at least 1 starting dot **A'** which allows you to trace the arrows in the above stated way and go through all the other dots, any tracing that will start from **A'** and go through all vertices will stop at some vertex **B'** (note that not all vertices can be a **B'**).

Model the game as a graph and give the most efficient algorithm in terms of worst case time complexity you can think of to return one possible **B'** that you will stop at after having gone through all vertices from some starting **A'** (**A'** is not given to you).

State the time complexity of your algorithm in terms of N and M.

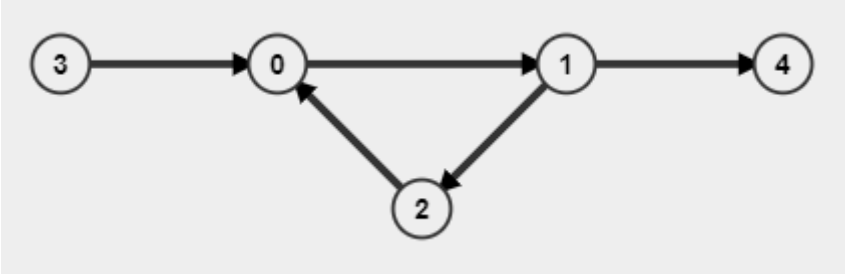
Ans:

Key idea: If there is a way to start from some A' and go through all vertices and end at some vertex B', we can draw out this path (might not be a simple path). Now if we reverse all the edges along the path it means there is a path from B' that goes through all the vertices and end at A'.

1. Model each dot as a vertex and each arrow as a directed edge that connects two vertices. Let this graph be G.
2. Create the transpose graph G' of G. This takes O(N+M) time
3. Run DFS topological sort on G'. Without reversing the toposort array, the last vertex in that array is B'. This takes O(N+M) time

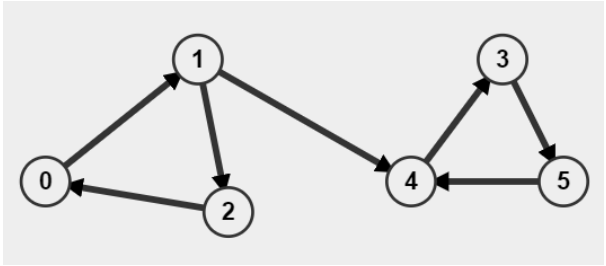
Time complexity is O(N+M).

Wrong answer:
Toposort G and then picking the 1st vertex in the topoosort array (before you reverse it) is not correct. The following graph is a counter example. Toposort array will give [2,4,1,0,3] (if we visit neighbors in increasing vertex number), but 2 is not a possible B'. 4 is the only possible B' here.



Another wrong answer:

Pick any vertex with 0 outgoing edges if there is 1, else just pick any vertex. In the following graph there is no vertex with 0 outgoing edges, but you can only pick 3,4,5 as possible ending vertices and not 0,1,2.



Grading Scheme (from least to most marks, before deductions due to mistakes) is as follows:

1 mark: algorithm checks for vertices with out-degree 0 only

2 marks: the algorithm generally doesn't use Kosaraju's eg:
uses BFS/DFS from random vertices to determine B', or
incorrectly identifies the problem (MST, SSSP, SSLP, APSP, APLP), or
uses Kahn's algorithm to determine toposort, or
runs BFS/DFS from A' by assuming A' is given already

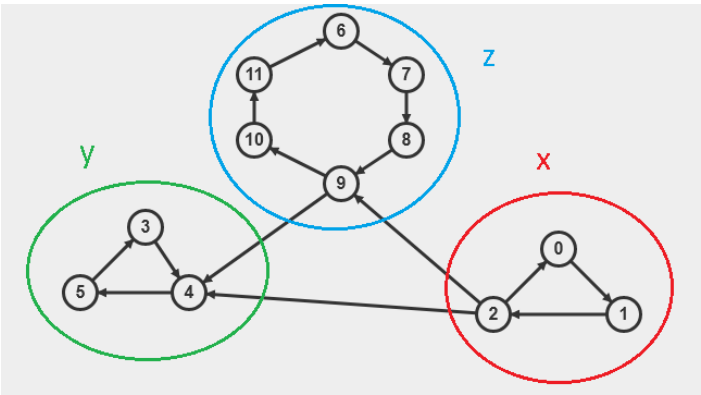
3 marks: algorithm uses DFS toposort/Kosaraju's in some form

4 marks: algorithm returns first element in K from Kosaraju's, or performs BFS/DFS from A' (after being determined from Kosaraju's) to find B'

5 marks: algorithm always returns correct B' in all valid graphs, but runs slower than the expected O(M+N) time

7 marks: algorithm always returns correct B' in all valid graphs, in O(M+N) time

Some common wrong answers have been posted above. Here's another discussion as to why some algorithms (from the 1-4 marks range) may return incorrect results:



Suppose this is the original graph of dots and arrows. All vertices in SCC x (in red) are possible A', and all vertices in SCC y (in green) are possible B'. Vertices in SCC z (in blue) are neither possible A' nor B'.

Algorithm: find the longest SSSP path in the graph. The endpoint must be B'.

The longest path here is from vertex 0 to vertex 8 (8 edges). Vertex 8 is not in SCC y, and is therefore not a possible B'.

Algorithm: run BFS/DFS from A' until all vertices are visited. The last vertex visited must be B'.

Starting from A', running BFS will likely lead to vertex 8 being the last vertex to be visited. Running DFS may cause vertex 4 to be visited before vertex 9, thereby causing SCC z to be visited before SCC y. This results in a vertex in SCC z to be the last vertex visited, which is not a possible B'.

Answers which attempt to find the longest SSSP path from A' will run into the same incorrect result as BFS from A'.

Algorithm: run Kosaraju's to get K, and return the first element from K. This must be B'.

DFS toposort only adds a vertex to K once all neighbours of that vertex have been processed already (or in the case of a cyclic graph, if these neighbours are being processed from an earlier call to DFSrec(u)). As such, it is possible for a call to DFSrec(9) to explore the edges 10->11->6->7->8 before 4->5->3, and thereby end up reaching vertex 8 first. Since 8's only neighbour (9) is still in the middle of being processed, 8 will be added onto K as the first vertex. Vertex 8 is not a possible B'.

Algorithm: find a vertex with out-degree 0. Else, find a vertex with the smallest out-degree. This vertex must be B'.

No vertices with out-degree 0 exist in this graph. The smallest out-degree of 1 can be found among vertices in SCC x, y, and z, and is thereby unable to guarantee that the result is a possible B'.

Algorithm: convert all SCCs into individual vertices, then find the SCC furthest away from the starting point (ie. the SCC containing A', which is x in this case). That SCC must contain B'.

When converting this graph into a graph of SCCs only, SCCs y and z are both 1 edge away from SCC x. Picking a vertex from SCC z as the answer would be incorrect in this case.

Algorithm: use Kahn's algorithm to perform toposort, and then (does something with toposort result).

No vertices with in-degree 0 exist here, so Kahn's algorithm cannot work.

19. [12 marks] In country Z where the roads are painted red or blue, it will be the inauguration of the new president in a few days. On that day, he will leave his current abode and travel with a motorcade to his new presidential abode.

In order to prevent any unexpected situation from happening enroute to the new abode, the shortest route in terms of distance is preferred by the security team. However, the president is a very superstitious man, and he believes it is auspicious to take only a route that will cross exactly 12 red roads.

To accommodate the president's requirement, the security team has determined there is at least 1 route that will cross exactly 12 red roads. However, they will still want the shortest route that crosses exactly 12 red roads.

Now as a person in charge of decorating both sides of the roads that will be used in the route from the president's old abode to the new abode, you need to determine the total length of decorations to use (in terms of meters). The problem is that the actual route is a secret and is not revealed to you.

You are given a graph **G** that contains the N junctions in country Z as vertices and the M one-directional roads as edges linking junctions/vertices together. For each road you are also given the color of the road and the length in meters of the road. You may assume the graph is stored in an adjacency list.

Now given **G**, **s** (the vertex representing the president's old abode) and **s'** (the vertex representing the president's new abode), write the most efficient algorithm you can think of in terms of worst case time complexity to do 1 of the following:

1. Find the total length of decorations to use along all roads on one potential valid shortest route. **6 marks if you get this correct**
2. Find the total length of decorations to use along all roads on all possible valid shortest routes. **12 marks if you get this correct**

If you need to solve the problem by transforming **G,** please describe the transformation before giving the algorithm. Also state the time complexity of your algorithm in terms of N and M

Note that you will need to decorate both sides of the road.

Ans:

For 1.

Transform the graph G in a new graph G' as follows:

1. For each vertex v in G, create 13 corresponding vertices (v,0),(v2,)…(v,12) in G'. For a vertex (v,c), c will indicate the number of red roads used to reach v.

2. For each edge (u,v) in G, if color of edge is red, link (u,c) to (v,c+1) for 0 <= c < 12 in G' with an edge of the same weight. Otherwise link (u,c) to (v,c) for 0 <= c <= 12 in G' with an edge of the same weight.

Store G' in an adjacency list.

Now simply run modified Dijkstra from vertex (s,0) and return 2*cost of SP to vertex (s',12).The time complexity is just O((N+M)logN) since the transformation only increases the number of vertices and edges in G' by a constant factor of 12.

This is equivalent to modifying the distance array into a distance matrix D[N][13], and storing the SP of 0 red edge, 1 red edge, … 12 red edges from source vertex to each vertex v.

For 2:

Perform the same transformation as in 1 and get G'. Here we want all roads that lie on all possible shortest valid paths, thus instead of just storing a predecessor along one SP to each vertex, we need to store a list of predecessors along all SPs to each vertex.

Make the following modification to modified Djikstra.

1. Instead of a predecessor array p of integers, create a predecessor array p of integer lists. Thus the predecessor array is now like an adjacency list!

2. During relaxation of edge (u,v) while running modified Djikstra:

   i. if D[v] < D[u]+weight of (u,v) perform the relaxation and create a new integer list for p[v].

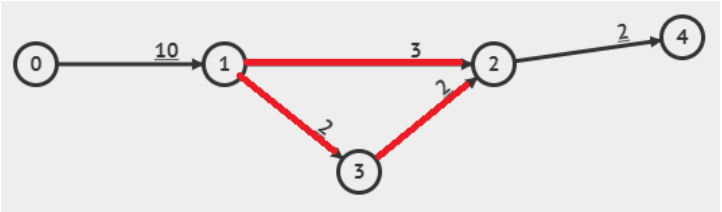
   ii. if D[v] == D[u]+weight of (u,v) add u to the integer list for p[v].

After running modified Djikstra, use p like an adjacency list and run DFS from vertex (s',12) and sum up all the weight of the edges traversed. Return that sum*2.

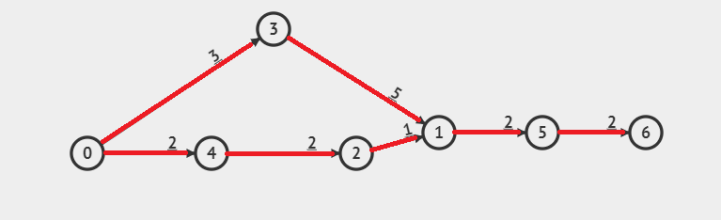
Common wrong answers:

1. Simply modify Dijkstra's to include an extra parameter to count the number of red edges along the SP to the item stored in the PQ (making it a triplet) but without transforming G. Variation to this wrong algo include i.) using red edges only if count is < 12, ii.) not visiting s' if count != 11 etc…

An example is given below. If the required number of red roads is 2, s = 0 and s' = 4. D[2] = 13 with SP 0->1->2 and not 0->1->3->2 since both have number of red roads <= 2 but 0->1->2 is the better path. However, 0->1->2->4 is not a valid SP from 0 to 4 that uses 2 red edges. Thus we need to track all SP with 0 red edges up to X red edges from source to each vertex where X is the # of red edges required.



Another example is given below. If # of red roads required is 4, s= 0 and s' = 6, then SP from 0 to 1 will be 0->4->2->1 since it is < 4 red edges instead of 0->3->1, but now we don't have a valid SP from 0 to 6 since 0->4->2->1->5->6 will use 5 red edges. The actual SP using 4 red edges should be 0->3->1->5->6.



Grading scheme:

12 marks option

Correct solution:

1. O(MlogN) solution ~ 12 marks
2. O(M^3) solution (correct modification to Floyd Warshall) ~ 8 marks
2. exponential solution ~ 4 marks – correctly find all paths from s to s' and among them, sum up all unique edges involved in all paths which cost the minimum that uses 12 red edges.

Incorrect solution to find valid path but where algo to find all roads in all valid path (assuming they are found correctly) is correct:

1. Find all paths that uses 12 red roads to get from s to s' rather than all SP that uses 12 red roads from s to s' ~ 3 marks
2. SSSP using Dijkstra with additional # of edge parameter for items stored in PQ but without transforming G ~ 6 marks
3. simple SSSP/APSP on G (with or without additional global counter for # of red edges) ~ 4 marks

6 marks option

Correct solution:

1. O(MlogN) solution ~ 6 marks
2. O(M^3) solution (correct modification to Floyd Warshall) ~ 4 marks
3. exponential solution ~ 3 marks

Incorrect solution:

1. SSSP using Dijkstra with additional # of edge parameter for items stored in PQ but without transforming G ~ 3 marks
2. simple SSSP/APSP on G (with or without additional global counter for # of red edges) ~ 2 marks

Deduction for mistakes:

1. non-terminating algo due to traversing cycles endlessly ~ -1 mark
2. unclear graph modeling: correctly transform the vertices, but not how they are linked (edges are not correct) ~ -2 marks
3. edge from (u,i) to (v,i+1) if there is edge (u,v) in original graph regardless if edge is blue or red ~ -2 marks
4. other minor mistakes ~ -1 marks each

20. [10 marks] Snags are an alien species found on planet Z. They live in subterranean burrows connected by bi-directional tunnels. Each tunnel has a given integer diameter value (same in both direction).

In order to fit into the tunnels to travel from burrow to burrow, Snags have the magical ability to change their size. To travel from a burrow **s** to another burrow **d**, a Snag will initially expend 0 energy to match the diameter of the first tunnel it takes. Subsequently, it will only need to expend energy to travel through a tunnel of diameter **x** if **x** > **p** (maximum diameter of tunnels already travelled) or **x** < **p'** (minimum diameter of tunnels already travelled). Energy expended is **x**−**p** or **p'**−**x** respectively.

In order to conserve energy, a Snag would like to expend the least total energy to get from some source burrow **s** to some destination burrow **d**.

Now given N burrows (labelled 1 to N), descriptions of all M bi-directional tunnels (starting burrow, ending burrow and diameter), a source burrow **s** and destination burrow **d**, give the most efficient algorithm in terms of worst case time complexity you can think of to output the least total energy to get from **s** to **d**.

State the time complexity of your algorithm in terms of N and M.

You may assume that there is always at least 1 path from any **s** to any **d**.

Ans:

1. Let each tunnel be an edge. Store all M edges together with their diameter (the M descriptions of the tunnels) in an edgelist EL.

2. sort EL in increasing order of diameter <- O(MlogM)

3. Let leastTotE be the least total energy expended going from s to d. Initialize to infinity

4. For i = 0 to M-1 <- O(M)

    i. run Kruskals starting from index i of EL. <- O(M)

    ii. At some index j of EL during running of Kruskal, if s and d are in the same disjoint set, leastTotE = min(leastTotE,EL[j].diameter-EL[i].diameter). Stop Kruskal's and continue to next iteration of main for loop in step 4.

    iii. if s and d is never in the same disjoint set after running Kruskal's, exit the main for loop in step 4.

5. return leastTotE.

Total time complexity = $O(MlogM+M^2) = O(M^2)$

Idea behind solution:

The total energy expended by a Snag along any path from s to d must be the difference between the largest edge weight and smallest edge weight. Thus what we want is a path where largest edge weight - smallest edge weight is minimized.

Now if there exist a path from s to d, then there exist a path where largest edge weight-smallest edge weight which we will call leastTotalEnergy is minimized. Let this largest edge be of weight M and let this smallest edge be of weight M'.

If we know M', we can easily find M, by simply running kruskal's considering only the edges >= M'. Now M must be the minimax edge among all paths from s to d that include M' as the weight of the smallest edge.

But how can we find M'?

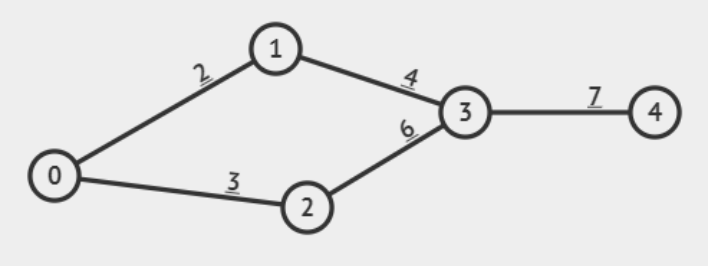
By simply using each edge as the starting edge (smallest edge) to run Kruskal's, and for each run get the smallest edge weight M' and the largest edge weight M (the edge that cause s and d to be in the same disjoint set). Update leastTotalEnergy to be min(leastTotalEnergy, M-M').

We can stop when we can no longer find a path that connects s and d (the edges remaining does not constitute any path from s to d).

Common wrong answers:

1. *Modification of SSSP/APSP where relaxation condition is to find the optimal path from s to any vertex v such the difference between the maximum and minimum edge is minimized.*

This solution with not work for the following graph among others, as optimal subpath property does not hold for this problem



The best path from 0 to 4 will involve vertex 3, but instead of the path of least total expended energy from 0 to 3, which is 0-1-3 (total energy so far is 2), it should be using 0-2-3 (total energy so far is 3), since the path 0-1-3-4 uses 5 energy while 0-2-3-4 uses 4 energy only.

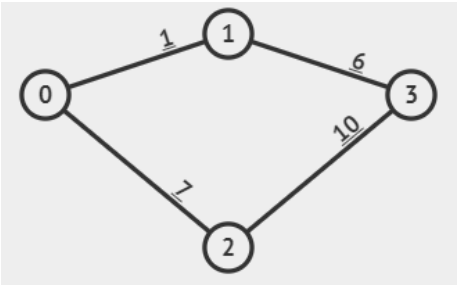
The problem here is that if v is an intermediate vertex along the path of least total energy expended from s to d, it does not mean the subpath from s to v is also a path of least total energy expended from s to v. **The optimal subpath property does not hold for this problem**.

2. *Find the minimax edge and maximin edge from s to d, and return their difference.*

The minimax and maximin edge might not lie on the same path.

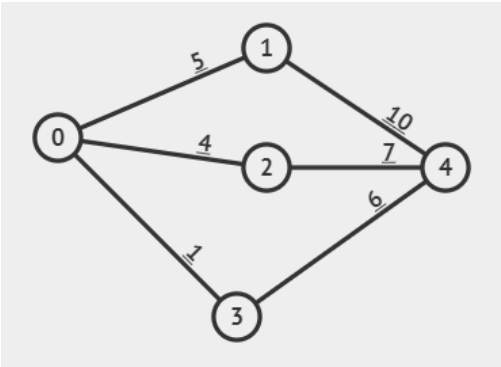
In the graph below, the minimax path from 0 to 3 is 0-1-3 (minimax edge is (1,3)) and the maximin path from 0 to 3 is 0-2-3 (maximin edge is (0,2)).

However, there is no path from 0 to 3 that contains both (1,3) and (0,2), and the best path that minimized the difference between the largest and smallest edge is 0-2-3.



3. *Find the minimax path, then the maximin path and choose the one where the largest edge - smallest edge is minimized.*

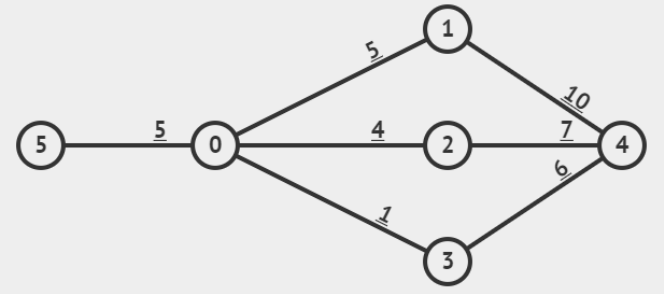
The optimal path from s to d might not have anything to do with the minimax edge and maximin edge. In the graph below, if s = 0 and d =4, the minimax path is 0-3-4 and the maximin path is 0-1-4, but the best path is 0-2-4.



4. *Multi source MST, where each neighbour v of s is a source. There will thus be v minimax path from each v to s'. Choose the one which minimizes the largest and smallest edge along the path.*

This is closer to the correct solution, since multiple minimax paths may be considered. However, since not all edges are considered for M' (the starting smallest edge), it can still lead to wrong solution.

In the example below, you can see that it is basically the previous graph with an extra vertex (vertex 5) connected to vertex 0. If s = 5, d = 4, you see that the 5-0-1-4 is the only path considered, but it is not the best path. 5-0-2-4 is the best path.



Grading Scheme:

1.) O(M^2) correct algorithm ~ 10 marks
2.) O(NM^2) correct algorithm (a modification of Prim's/Dijkstra to keep track of all paths with distinct smallest and largest edge from s to every vertex v, thus transforming D[N] to D[N][M][M]) ~ 7 marks
3.) exponential time correct algorithm (find all paths from s to d, keeping track of path where difference between largest and smallest edge is minimized) ~ 3 marks

wrong solution:
pre-req: correct graph modelling
1.) minimax or maximin in O(MlogN)/O(N^3) ~ 3/2 marks
2.) minimax edge weight - maximin edge weight in O(MlogN)/O(N^3) ~ 3/2 marks

only graph modelling: 1 mark

Deduction for errors:
1.) incorrect computation of the energy expended ~ -1 mark
2.) Forget to compute the energy expended ~ -2 marks
3.) other minor errors ~ -1 mark