CS4102 - ALGORITHMS - SPRING 2022

DUE FRIDAY, MARCH 18, 2022 AT 11:30 PM VIA GRADESCOPE

B2BASIC(SSB3VK)

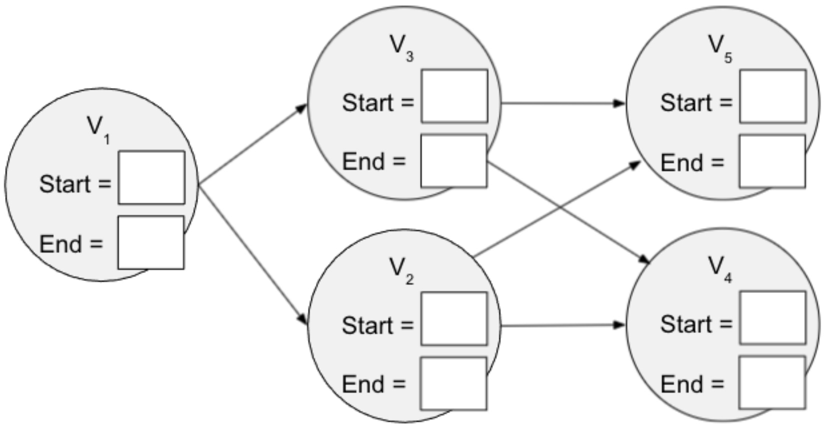
Collaboration Policy: You are encouraged to collaborate with up to 3 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not share written notes, documents (including Google docs, Overleaf docs, discussion notes, PDFs), or code. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite by naming the book etc. or listing a website's URL. Do not submit a solution that you are unable to explain orally to a member of the course staff. Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.

Collaborators: sjb9qvc, hk2ku, bnh5fh

Sources: Cormen, et al, Introduction to Algorithms. *(add others here)*

PROBLEM 1 *DFS and Topological Sort*

1. Run DFS on the following graph. List start and finish times (beginning at $t = 1$) for each node in the table shown below the image of the graph. Note: For this problem, by "start" we mean the discovery time, and by "end" we mean finish times.) Use V_1 as your start node. *To help us grade this more easily, when multiple nodes can be searched, always search neighboring nodes in increasing order (e.g., if V_2 and V_3 are both adjacent to the current node, search V_2 first).*



Vertex	V_1	V_2	V_3	V_4	V_5
Start	1	2	8	3	5
End	10	7	9	4	6

Solution: Solution in table

2. Using your answer above, give the specific *Topological Ordering* that would be produced by the *DFS-based* algorithm we discussed in class.

Solution: V_1, V_3, V_2, V_5, V_4

PROBLEM 2 *True or False. (You don't have to explain this in your submission, but you should understand the reason behind your answer.)*

A. If you use DFS to find a topological sorting on a directed graph, the last vertex discovered in the search could legally be the last vertex in the sorted ordering of the vertices.
Solution: Actually false. V3 is the last vertex discovered. But wouldn't necessarily be the last vertex in the topological sort.

B. For the disjoint set data structure we studied, if we had a $\Theta(\log n)$ implementation of *find-set()*, then the order class for the time-complexity of *union(i,j)* would be improved (i.e., better than the result we learned).
Solution: True. This is because the speed of our implementation of union depends on the speed of find-set(). And our implementation initially had a linear runtime for find-set() so log(n) is faster.

C. Both path-compression and union-by-rank try to improve the cost of future calls to *find-set()* by making the trees representing a set shorter without changing the set membership for the items in that set.
Solution: True.

PROBLEM 3 *Kruskal's Runtime*

What is the runtime of Kruskal's algorithm if find() and union() are $\Theta(1)$ time?
Solution: In the worst case scenario, we iterate over each edge perform two find operations and potentially a union operation. This yields a runtime of E * (2 * $\Theta(1)$ + $\Theta(1)$) which we can then compress to E * 3 which yields $\Theta(3E)$ which yields $\Theta(E)$ or $\Theta(V^2)$. Instead of the $\Theta(E * V)$ found originally.

But note that we have not included the cost of the initial edge-queue insert. Factoring that in, we would get a runtime of: $\Theta(E * log(V) + E)$. Or to put it another way, we would have a $\Theta(E * log(V))$ runtime.

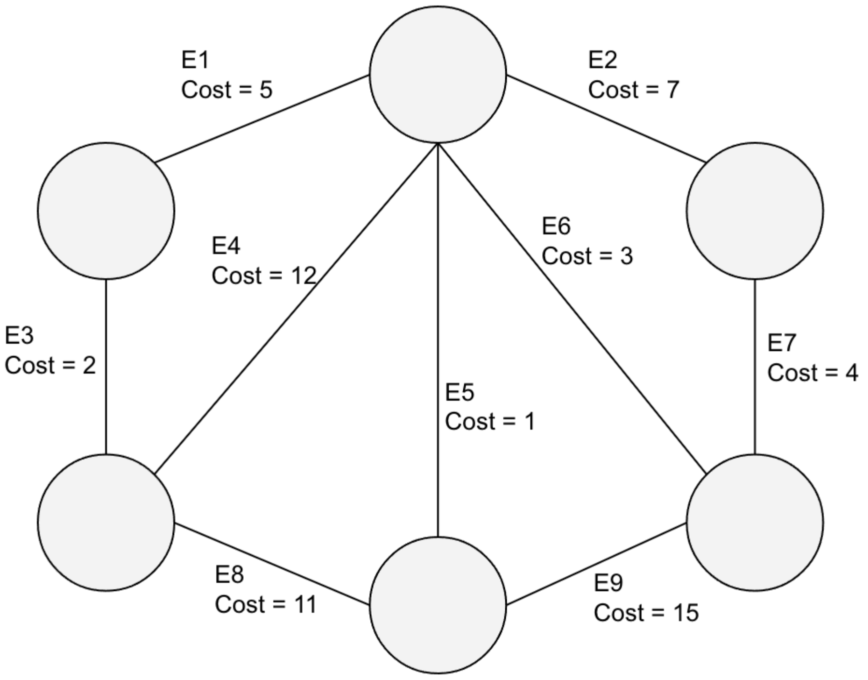
PROBLEM 4 *Strongly Connected Components*

Your friend Kai wants to find a digraph's SCCs by initially creating G_T and running DFS on that. In other words, he believes he might be able to *first* do something with the the transpose graph as the first step for finding the SCCs. (The algorithm we gave you first did something with G and not with G_T.)
Do you think it's possible for Kai to make this approach work? If not, describe a counter-example or explain why this will fail.
If it is possible, explain the steps Kai's algorithm would have to do to complete the algorithm, and briefly say why this approach can lead to a correct solution.
Solution: I don't see why this wouldn't work. Instead of running DFS on G first and then use the results of DFS(G) to influence our DFS run on G_T, we could instead just run DFS(G_T) and use the results of that run to influence DFS(G).

This would lead to the same solution (in terms of identifying sets of SCCs) because the G_T and G have the same sets of SCCs. There is no difference in SCCs between a graph and the transpose of the graph.

PROBLEM 5 *Executing Kruskal's MST Algorithm*

Run Kruskal's algorithm on the graph below. List the order in which the edges are added to the MST, referring to the edges by their provided labels.
(*Consider how your answer would change if E1 had weight 12. However, you don't need to provide an answer to us for this part.*)



Your answer (list of edges in order):
Solution: E5, E3, E6, E7, E1

PROBLEM 6 *Difference between Prim's MST and Dijkstra's SP*

In a few sentences, summarize the relatively small differences in the code for Prim's MST algorithm and Dijkstra's SP algorithm.
Solution: The differences between Dijkstra and Prim amounts to the values that they compare. Dijkstra compares the distance between the current node and the start node whereas Prim only compares the "cost" of adding the current node, the weight of the edge between the current node and the rest of the tree. This yields different final results. Dijkstra creates a tree that minimizes the distances in the graph between the start node and every other node. Whereas Prim's algorithm creates a weighted MST. So different algorithms for different purposes.

PROBLEM 7 *True or False. (You don't have to explain this in your submission, but you should understand the reason behind your answer.)*

A. An *indirect heap* makes *find()* and *decreaseKey()* faster (among others), but *insert()* becomes asymptotically slower because the indices in the indirect heap must be updated while percolating value up towards the root of the heap.
Solution: Assuming that we are comparing to the naive implementation of an array storing the values at each key position. find[min]() is faster (linear – constant) (iterate through elements vs check top for heap) but decreaseKey() goes from (constant – log(n)) (because now we have to percolate) so it actually gets slower. But insert becomes slower because before it was constant time but now it is log(n) because we have to insert and then percolate

up. So I will answer false. I talked to a TA about this and he said the real answer is true but to answer false cause of my justification.

This question is so strange. The TA and I were going back and forth on this for 30 minutes.

But I see how even if were comparing to a naive heap implementation where all we have are key, value pairs (node, weight), find would be faster (cause now we can directly index to the node we need) and decreasekey would be faster as well (same reason) but I still don't see how insert would become asymptotically slower? Aren't we still performing a linear number of operations?

B. If all edges in an undirected connected graph have the same edge-weight value k, you can use either BFS or Dijkstra's algorithm to find the shortest path from s to any other node t, but one will be more efficient than the other.
Solution: True. BFS will be more efficient because we don't have to mess with priority queues and don't have a notion of weight to contend with.

C. In the proof for the correctness of Dijkstra's algorithm, we learned that the proof fails if edges can have weight 0 because this would mean that another edge could have been chosen to another fringe vertex that has a smaller distance than the fringe vertex chosen by Dijkstra's.
Solution: False. Dijkstra only has issues with negative weights. Because Dijkstra doesn't reiterate on known values you would think it would have an issue. But, the while loop that keeps selecting the v with smallest distance helps work around the issue of magically finding a more optimal path.

Additionally, a pair of nodes connected by an edge with weight 0 can just be compressed into the same node as there is 0 cost between traversing the two vertices, enabling us to reduce them to the same vertex.

PROBLEM 8 *Gradescope Submission*

Submit a version of this `.tex` file to Gradescope with your solutions added, along with the compiled PDF. You should only submit your `.pdf` and `.tex` files.