CSCI 570 - Spring 2021 - HW 1 Rubric

Due Feb 07

1

$2^{\sqrt{2logn}}$, $(\sqrt{2})^{logn}$, 2^{logn}, $nlogn$, $n(logn)^3$, 2^{n^2}, 2^{2^n}

[Rubic (10 points) The grader will try to find the longest common subsequence between the student's solution and our solution. The partial credit will be given according to the length of the longest common subsequence, L, with the following rule:

- 10 points if $L = 7$: the student's answer is completely correct;
- 10 - $2 \times (7 - L)$ points if $3 \leq L \leq 6$.
- 1 point if $L = 2$, which means the grader can only find one pair of functions in the student's solution is correct.
- 0 point if $L = 1$.

Notice that if the grader notices that the student's solution is arranging the functions by the decreasing order of growth rate, the grader should reverse the student's answer, and give points by the following:

- 7 points if the reversed sequence is exactly the same as our solution.
- 4 points if the longest common subsequence between the reversed sequence and our solution is 6.
- zero points otherwise

]

2

The BFS algorithm outputs the list of edges of the BFS-tree. Let's call the BFS-tree as T. For now, let's assume G is a connected graph and as a result T is a connected graph as well - this can be easily be extended to the general case. We claim that G has cycle if and only if there exists an edge (u, v) in edges of G which is not in edges of T. The proof is in two folds:

(1) *If the algorithm outputs YES, it means G has a cycle.* If there is an edge (u, v) outside T, since T is a connected graph, then there is a (u, v)-path between u and v using edges of T only. Then adding the (u, v)-edge to the (u, v)-path makes a cycle in G.
(2) *If there is a cycle in G, then the algorithm outputs YES.* This could be done by contradiction: Let's assume the algorithm outputs NO (i.e. no cycle) while it should output YES because G has a cycle. Then it means that all the edges of G are covered by the edges of T. However, we know that T is a tree and tree does not have any cycles. This is a contradiction and algorithm should output YES.
For finding the list of vertices in the cycle, we need to perform another BFS tree. Let's assume the edge inside G which is not in T is (u, v). We know if we remove (u, v) edge from graph G then there exists another path between u and v (because (u, v) edges shapes a cycle). So, we just need to remove (u, v) edge from G and perform another BFS for finding a path between u and v in the remaining graph by calling a BFS algorithm with root u and reporting a path from v to u.
We know the time complexity of the BFS algorithm is $O(V + E)$. For checking whether all the edges of G are presented in T, takes $O(V + E)$ time (using adjacency list). The second BFS for fiding the list of edges in the cycle takes $O(V + E)$ time. so, the overall time complexity of the algorithm is $O(V + E)$.

[Rubic (15 points):

- [5 points] for explaining the one direction of if and only if.
- [5 points] for explaining the other direction of if and only if.
- [3 points] for reporting the list of vertices in the cycle.
- [2 points] for time complexity.
- No reduction of credit if they did not mention the assumption of connectivity.
- if they propose the algorithm by removing each edge and checking the connectivity of the remaining graph, decrease -5 points. The time complexity of this algorithm is $O(E \times (V + E)) = O(E^2)$ and it is not linear.

]

3

Claim: in the nonempty binary tree with n leaves, the number of nodes with two children is $n - 1$. We prove this by induction, suppose there is a tree with k leaves:

- When $k = 1$, a path-tree with single leaf (and root with degree 1) has 1 leaf and 0 nodes with two children. So, the hypothesis is true for $k = 1$.

- Suppose the claim stands when $n = k$, which means for the tree with k leaves there are $k - 1$ nodes with two children. We want to prove that it also stands when $n = k + 1$. Select a leaf from this tree, and remove it. The remaining part of the tree could have two cases:
  - 1) The removed node was connected to an existing node that had two children and now it has one child. In this case, the number of leaves decreases by one and becomes k. Based on induction hypothesis, the number of nodes with two children in the induced tree is $k-1$. Since the orignal tree has one extra node with two children in comparison with the induced tree, then the original tree has $k + 1$ leaves and k nodes with two children.
  - 2) The removed node was connected to an existing node that had one child and now it is a leaf. In this case the number of leaves of the graph remains the same $(k + 1)$. Doing the leaf-removal until we remove all the leafs connected to nodes with one children, eventually the leaf-removal falls into case 1. In case 1 we showed the proof for the induction hypothesis.

Therefore, the proof is complete.

[Rubic (10 points):

- [2 points] for listing the case when there is one leaf node.
- [8 points] for explaining two cases ([4 points] for each case).
- No reduction of point if induction was on number of nodes.

]

4

Assume K_5 is a planar graph. Based on the 4 color Theorem, Any simple planar graph can be colored with less than or equal to 4 colors. Let's assume there exist a correct 4-coloring for nodes of graph K_5. Let's assume vertex 1 has color c_1. Since vertex 1 is adjacent to vertices 2, it cannot have color c_1. So, let's assume vertex 2 has color c_2. Since vertices 3 is adjacent to vertices 1 and 2, it cannot have color c_1 and c_2. Following the same logical steps, vertex 3 and vertex 4 should have different colors c_3 and c_4. Since we assumed there exists a 4-coloring of K_5, vertex 5 should have one of the colors $\{c_1, c_2, c_3, c_4\}$. However, vertex 5 is adjacent to all of vertices $1 - 4$ (having distinct colors), so vertex 5 should have the same color with one of the adjacent vertices to it. This is contradiction: the 4-coloring of K_5 could not be a proper vertex coloring and as a result K_5 is not a planar graph.

[Rubic (10 points)

- [2 points] Using the 4-coloring theorem.

- [6 points] Using contradiction for proof (i.e. having assumption, following logical steps and showing the contradiction ...)

- [2 point]: having step by step logical sentences for the proof (e.g. explaining why two nodes cannot have same color, by using the named colors or clique ...).

- In case the student used Euler formula for the proof ($e \leq 3v - 6$) ([2 points]) and have a good explanation ([1 point]) and used contradiction ([2 points]) give the full credit. In this question $v = 5$ and $e = 10$, and the inequality does not hold.

]

5

In aggregate analysis, the amortized cost is the total cost divided by number of operations n. The cost of operation i is 1 if i is not power of 2, and it is i otherwise. So:

$$\sum_{\substack{1\leq i\leq n \\ i\notin\{2^x, x\in\mathbb{N}\}}} 1 + \sum_{\substack{1\leq i\leq n \\ i\in\{2^x, x\in\mathbb{N}\}}} i = \sum_{1\leq i\leq n} 1 + \sum_{\substack{1\leq i\leq n \\ i\in\{2^x, x\in\mathbb{N}\}}} (i-1) = n + \sum_{\substack{1\leq i\leq n \\ i\in\{2^x, x\in\mathbb{N}\}}} (i-1)$$

$$= n + \sum_{x\in\{0,1,..,\lfloor log(n)\rfloor\}} 2^x - 1$$

We know geometric series $2^0 + 2^1 + ... + 2^k = 2^{k+1} - 1$, so:

$$= n + \sum_{x\in\{0,1,..,\lfloor log(n)\rfloor\}} 2^x - 1 \leq n + 2^{\lfloor log(n)\rfloor+1} - 1 \leq n + 2n = 3n$$

So, the amortize cost would be $\frac{3n}{n} = O(constant)$.

[Rubic (15 points):

- [3 points] Writing the correct total cost equation (i for power of 2 and 1 for the rest).

- [4 points]Using geometric series to simplify the sum.

- [3 points] Dividing total cost by number of operations.

- [5 points] Correct final value.

- No point reduction if they used O(3) vs O(constant).

]

Insertions	1	2	3	4	5	6	7	8	9	10
Old Size	1	1	3	3	5	5	7	7	9	9
New Size	-	3	-	5	-	7	-	9	-	11
# of copy	-	1	-	3	-	5	-	7	-	9

6

In the following table we have the numer of copy for the first 10 insertions: The amortize cost of insertion is total cost divided by number of insertions. The total cost is equal to:

$$\sum_{i \in 1,3,\ldots,2\times\lceil\frac{n}{2}\rceil-1} i = (\lceil\frac{n}{2}\rceil)^2$$

So, the amortize cost is equal to $\frac{\lceil\frac{n}{2}\rceil^2}{n} = O(n)$. In comparison with "double up" resize policy which has constant amortize cost, this method is less efficient.

[Rubic (17 points):

- [3 points]: understating the question ([1 point] increasing the size after 2 insertions [1 point] summing the number of copy operation [1 point] summing only odd insertions. [1 point])
- [3 points]: writing the correct total cost equation.
- [4 point]: simplify the series.
- [3 points]: dividing the total cost by number of insertions.
- [4 points]: correct final answer.

]

7

The optimal path from s to t in graph G is the path where the minimum edge weight is maximized. This means that the optimal path is constructed by edges with larger weights: in the other words we should eliminate edges with small weight. The algorithm is in three steps: (1) sort edges in a descending order. (2) Starting from n disconnected node and starting from the largest edge, add edges one by one until the two vertices s and t are in the same component. Let's define graph H as a subgraph of G until this step (i.e. s and t are in the same component). (3) Perform a BFS/DFS algorithm on graph H to find a path from s to t.

Time Complexity: (1) The sorting of the edges would take $O(E \times log(E))$ time. (2) The removal of edges and checking for connectivity would take $O(E \times (V + E))$ and the last step (3) would take $O(V + E)$ time. So, the overall complexity of the algorithm is $O(E \times (V + E + log(E)))$.

- The (2) part of the algorithm could be done more efficiently by using disjoint-set data structure. The disjoint-set data structure keeps information about which vertices are in which connected component in $O(log^*(n))$ which is smaller than $O(V+E)$. Using disjoint-set data structure the time complexity of part (2) would be $O(E \times log^*(n))$ and overall complexity would be $O(E \times log(E))$.

- Another solution to this problem is to find the maximum spanning tree using the Kruskal algorithm and then perform BFS/DFS for finding the path. The time complexity of the Kruskal algorithm is $O(E \times log(E))$.

[Rubic (23 points):

- [5 points]: explaining why removal of edges with small weight is enough.

- [6 points]: explaining the three steps ([2 point] for each step).

- [6 points]: correct time complexity of each step ([2 point] for each step).

- [6 point]: reporting the final time complexity and connecting steps and overall answer.

- If Maximum Spanning Tree algorithms + BFS/DFS algorithms have been used and the time complexity explained and reported correctly; give the full credit.

- Using disjoint-set data structure or BFS/DFS algorithm in part (2) does not matter. No credit reduction.

- For part (2) if instead of adding large edges, they removed small edges until the two components of s and t are disconnected; give the full credit.

]