TEST EXAMPLE of Final Exam
Introduction to Algorithms/Algorithms 1
600.363/463

Friday, December 21st, 9am-noon

Ethics Statement

I agree to complete this exam without unauthorized assistance from any person, materials, or device.

Name *Answer Key*

Signature Date

1 Problem #1: (100 points) YES/NO QUESTIONS (10 questions, 10 points per question)

For every question answer yes or no.

1. Any recurrence can be solved with Master Theorem

   yes [✓ no]

   Reason: does not apply, for instance, when $f(n) = \mathrm{O}(n \log n)$.

2. $\sum_{i=1}^{n} i^2 = O(n^2)$

   yes [✓ no]

   Reason: $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

3. In a Red-Black Tree the number of black nodes is always smaller than the number of red nodes

   yes [✓ no]

   Counterexample: consider the tree with only one node. It must be black.

4. Let A be a randomized algorithm that sorts an array of size n. If the expected running time of A is $O(n)$ then the worst-case running time of A is $O(n \log(n))$

   yes [✓ no]

   Counterexample: consider an algorithm which has $q > n$ possible inputs of length n, and label those inputs $\{a_1, \ldots, a_q\}$. Suppose the algorithm runs in O(n) time on inputs $a_1, \ldots, a_{q-1}$ and O(n^2) on a_q. Since $q > n$, the expected running time will be O(n), but the worst-case running time will be O(n^2) on a_q.

5. There exists a polynomial time algorithm for finding a shortest cycle in an undirected weighted graph

   [✓ yes] no

   Algorithm: given a graph $G = (V, E)$, we check for each node $v \in V$ whether there is a cycle. We do this by creating a new graph $G' = (V', E')$ where v is split into two nodes, v_{in} and v_{out}. That is, $V' = (V - \{v\}) \cup \{v_{\text{in}}, v_{\text{out}}\}$. E' is created as follows: for all $e \in E$, if $v \notin e$, then $e \in E'$. Otherwise, if $e = (u, v)$ for some $u \in V$, then $(u, v_{\text{in}}) \in E'$. Similarly, if $e = (v, u)$ for some $u \in V$, then $(v_{\text{out}}, u) \in E'$. Then we run breadth-first search from v_{out} to check whether there is a shortest path from v_{out} to v_{in}. It follows that the length of the shortest path from v_{out} to v_{in} is equal to the length of a shortest cycle that contains v. We will repeat the above algorithm for all nodes $v \in V$ and maintain the minimum value of all shortest cycles. The algorithm clearly works in polynomial time.

6. For any optimization variant of an NP-complete problem there exists a P.T.A.S.

   yes [✓ no]

   Reason: As we discussed in class, it is not known if TSP without triangle inequality accepts any constant approximation. (Similar statement is true for a maximum clique). See also the discussion on page 1107.

7. For any two functions f, g either $f = o(g)$ or $g = o(f)$

   yes [✓ no]

   Counterexample: when $f = g$.

8. Merge Sort runs in $O(n^9)$ time on arrays of size n.

   [✓ yes] no

Reason: mergesort is $O(n \log n) = O(n^9)$.

9. If the amortized running time is $\Omega(n)$ then the worst-case running time is $\Omega(n)$.

   [✓ yes] no

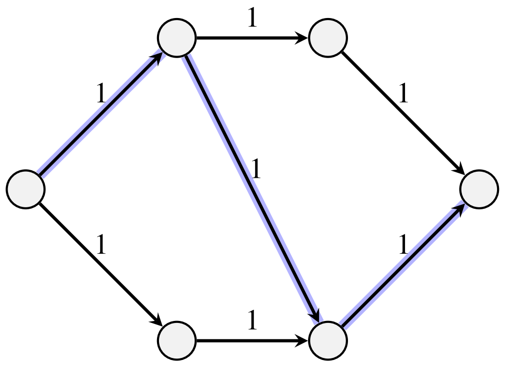
   The above statement is imprecise. We should add: Consider a data-structure operation α and consider the sequence of n operations of type α. If the amortized running time is $\Omega(n)$ then the worst-case running time of α is $\Omega(n)$.

   Reason: Consider an operation $\alpha(D)$ on a data structure D and consider a sequence of n operations on $\alpha(D_1), \dots, \alpha(D_n)$, where D_i is the result of applying α on D_i. Then we have that the amortized running time is $\frac{1}{n}\sum_{i=1}^{n} \alpha(D_i) \leq \max_{i=1,\dots,n} \alpha(D_i)$. Thus the worst-case running time cannot be smaller than the average.

10. Let G be a flow network with capacities c and let f be a flow on G such that for any path P from s to t there exists an edge $(a, b) \in P$ such that $c(a, b) = f(a, b)$. Then f is a maximum flow.

    yes [✓ no]

    Counterexample: The blue path indicates the flow. All capacities are one


1
1
1
1
1
1
1

Problem #2: (50 points)

Let A and B be two arrays of integers, both of size n. Design an algorithm that counts the number of integers that appear in A but not in B. Full credit will be given if your algorithm is correct, works in $O(n \log(n))$ time and you clearly explain why the algorithm works correctly.

Solution

You can assume that all elements in A are distinct and all elements in B are distinct. Intuition: if $\mathcal{A}$ represents the set of elements in A and $\mathcal{B}$ represents the set of elements in B then we need to compute $|\mathcal{A} \setminus \mathcal{B}| = |\mathcal{A}| - |\mathcal{A} \cap \mathcal{B}|$. First, sort the two arrays. Then, iterate through them, counting up how many integers appear in both arrays, and storing this count in the variable c. At the end, $n - c$ will be the number of integers unique to A.

```
$A \leftarrow$ mergesort($A$)
$B \leftarrow$ mergesort($B$)
$i \leftarrow 0$
$j \leftarrow 0$
$c \leftarrow 0$
while $(i < n) \wedge (j < n)$ do
    if $A[i] < B[j]$ then
        $i \leftarrow i + 1$
    else if $A[i] > B[j]$ then
        $j \leftarrow j + 1$
    else
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
        $c \leftarrow c + 1$
    end if
end while
return $n - c$
```

This procedure is very similar to the merge step of mergesort; by extension we can conclude that this procedure visits the elements of A and B in increasing order. This assures that if $\exists\, a, b$ such that $A[a] = B[b]$, then there will be an iteration of the while loop where $i = a$ and $j = b$, so c will get incremented appropriately

This algorithm runs in O($n \log(n)$) time because each mergesort takes O($n \log(n)$) time and the while loop runs in O(n). This can be seen by analogy to the merge step of mergesort, or by observing that each iteration of the while loop increments at least one of i and j. i and j can increment a total of $2n$ times, and therefore the while loop can iterate at most $2n = \mathrm{O}(n)$ times.

Problem #3: (50 points)

Let G be a directed weighted graph and let $a, b, c, d \in V$ be distinct vertices in V. Define $z \in V$ to be an **intersection point** if z belongs to a shortest path from a to b and to a shortest path from c to d. Design an efficient algorithm to find if there is an intersection point. Full credit will be given if your algorithm is correct, works in $O(EV)$ time and you clearly explain why the algorithm works correctly.

Solution

Denote by $\delta(x, y)$ the length of the shortest path from x to y. Run Bellman-Ford from a and c respectively to get the shortest-path distances $\delta(a, v)$ and $\delta(c, v)$ for every node $v \in V$. Then reverse the edge- directions in G and run Bellman-Ford from b and d respectively to get the shortest path distances $\delta(v, b)$ and $\delta(v, d)$ for every node $v \in V$.

Then iterate through V checking to see if there is some $z \in V$ such that $\delta(a, z) + \delta(z, b) = \delta(a, b)$ and $\delta(c, z) + \delta(z, d) = \delta(c, d)$. If such a node exists it is an intersection point.

Justification: From the properties of shortest paths, we know that if a node z is on the shortest path p from a to b, then p can be decomposed into two halves, the shortest path from a to z followed by the shortest path from z to b. At the same time if $\delta(a, b) = \delta(a, z) + \delta(z, b)$ then z belongs to some shortest path from a to b. Thus, z is an intersection point if and only if:

$$\delta(a, b) = \delta(a, z) + \delta(z, b), \qquad \delta(c, d) = \delta(c, z) + \delta(z, d).$$

Also, to find all shortest paths from all $z \in V$ to s it is sufficient to reverse directions of all edges and find all shortest paths from s to all $z \in V$ in the "reversed" graph. Running time is computed as follows: $O(EV)$ steps are needed for all Bellman-Ford computations, $O(V + E)$ time is needed to reverse the graph and $O(V)$ steps are needed to check the above equations for all $z \in V$.

Problem #4: (50 points)

Let $G = (V, E)$ be a weighted, directed graph with exactly one negative-weight edge and no negative-weight cycles. Let s be a vertex. Give an algorithm to find the shortest distance from s to all other vertices in V that has the same running time as Dijkstra. Describe the running time of your algorithm. Prove formally that your algorithm is correct.

Solution

Let (x, y) be the negative-weight edge. Let $v \in V$ be a vertex. The shortest path from s to v either goes through (x, y) or it doesn't. Thus, we will compute the shortest path from s to v which uses (x, y), as well as the shortest path from s to v that doesn't.

We do this as follows. Since Dijkstra's algorithm cannot handle negative weight edges, we remove (x, y) from the graph. Then we run Dijkstra twice: once with s as the source, and the other with y as the source. This gives us shortest path distances $\delta'(s, v)$ and $\delta'(y, v)$ in the modified graph for each $v \in V$.

To find the shortest paths in the original graph, we compute

$$\delta(s, v) = \min(\delta'(s, v), \delta'(s, x) + w(x, y) + \delta'(y, v))$$

As in the previous problem, the correctness of this solution relies on the fact that if a node z is on the shortest path p from a to b, then p can be decomposed into two halves, the shortest path from a to z followed by the shortest path from z to b.

Problem #5: (Dynamic Programming) (50 points)

Let $G = (V, E)$ be a weighted, directed graph. Define $G^* = (V, E^*)$ to be the following graph. G^* has the same set of vertices as G and $(a, b) \in G^*$ if and only if there is a path from a to b in G. Design an efficient algorithm to construct G^*. Full credit will be given if your algorithm is correct, works in $O(|V|^3)$ time and you clearly explain why the algorithm works correctly.

Solution

The solution to this problem is a slight variant of the Floyd-Warshall algorithm, where the table stores booleans instead of path lengths. We will fill out a three-dimensional table T such that $T[i, j, k]$ is true iff there exists a path p between v_i and v_j whose intermediate vertices are all in the set $\{v_1, \ldots, v_k\}$.

As in the Floyd-Warshall algorithm, there are two possibilities: either p includes vertex v_k or it doesn't. Thus $T[i, j, k] = \big(T[i, k, k-1] \wedge T[k, j, k-1]\big) \vee T[i, j, k-1]$. Also, $T[i, j, 0]$ should be true if and only if there is an edge between i and j.

We can fill out the table by iterating over i, j, and k in a triply-nested for loop. Since each variable ranges over $|V|$ vertices and it takes O(1) time to fill in each table entry, this will give us an O($|V|^3$) algorithm. T is computed, the edges of G^* defined as follows: there is an edge (i, j) if and only if $T[i, j, n]$ is true. For more details, see Chapter 25.2.

Problem #6 : Formal proofs (50 points)

Let G be a connected undirected weighted graph with a weighting function w. A bridge is an edge whose deletion makes G disconnected. Let x be an edge such that x is not a bridge and $w(x) > w(y)$ for all other edges $y \in E$. Prove formally that x does not belong to any MST. Full credit will be given for a full and formal proof that does not skip any step. You may use any claim that we proved in the class. Partial credit will be given to an informal explanation of the intuition.

Solution

Assume BWOC that x belongs to some MST. Let $x = (u, v)$. Remove x from the MST. BFS from u in the MST and add all reached vertices to U. BFS from v in the MST and add all reached vertices to V. No vertices can be shared between the two sets, because that would mean u would still be able to reach v (by going through the shared vertex) without x, implying the MST has a cycle, which is impossible for a tree. So $U \cap V = \emptyset$. Furthermore, there can be no edge in the MST between any vertex in V and any vertex in U, because otherwise, once again, u would be able to reach v without x, creating a cycle. That means U and V are two distinct components. Now consider the cut in G of (U, V). x cannot be the only edge that crosses the cut, because that would make x a bridge (removing x would not allow vertices on the opposite side to reach each other, since no other edge crosses the cut). If there is another edge y that crosses the cut, we know that $w(x) > w(y)$. Stick the edge y back into our MST that had deleted x. This graph is also a spanning tree, because y crosses the (U, V) cut, which means any vertex in U can now reach any vertex in V by crossing y (it must be a tree because y creating a cycle would imply that vertices in U and V could reach each other without y). Furthermore, since y is lighter than x and all other edges remained the same, the weight of the spanning tree decreased. But that implies the MST was not in fact a minimum. This is a contradiction, so x cannot belong to any MST.

Problem #7: Concepts (30 points)

In a few sentences explain what is an approximation algorithm. Full credit will be given to the right idea. No formal definition is required.

Solution

An approximation algorithm is an algorithm that returns a near-optimal solution, i.e., a solution that is not the best possible but is a close to the best possible solution. An approximation ratio is the ratio between the cost of the optimal solution and the approximation solution in the worst-case scenario.