CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Final Exam

Due: Saturday, May 07, 12:00noon EST

Name: Ambika Yavatkar *NetID: aay22*

Instructions

1. Do not forget to write your name and NetID above, and to sign Rutgers honor pledge below.
2. The exam contains 5 problems worth 100 points in total *plus* one extra credit problem worth 10 points.
3. The exam should be done **individually** and you are not allowed to discuss these questions with anyone else. This includes asking any questions or clarifications regarding the exam from other students or posting them publicly on Piazza (textbfany inquiry should be posted privately on Piazza). You may however consult all the materials used in this course (video lectures, notes, textbook, etc.) while writing your solution, but **no other resources are allowed**.
4. Remember that you can leave a problem (or parts of it) entirely blank and receive 25% of the grade for that problem (or part). However, this should not discourage you from attempting a problem if you think you know how to approach it as you will receive partial credit more than 25% if you are on the right track. But keep in mind that if you simply do not know the answer, writing a very wrong answer may lead to 0% credit. The only **exception** to this rule is the extra credit problem: you do not get any credit for leaving the extra credit problem blank, and there is almost no partial credit on that problem.
5. **You should always prove the correctness of your algorithm and analyze its runtime.** Also, as a general rule, avoid using complicated pseudo-code and instead explain your algorithm in English.
6. You may use any algorithm presented in the class or homeworks as a building block for your solutions.

Rutgers honor pledge:

On my honor, I have neither received nor given any unauthorized assistance on this examination.

Signature: Ambika Yavatkar

Problem. #	Points	Score
1	20	
2	20	
3	20	
4	20	
5	20	
6	+10	
Total	100 + 10	

Problem 1.

(a) Mark each of the assertions below as True or False and provide a short justification for your answer.

(i) If $f(n) = 2^{\sqrt{\log n}}$ and $g(n) = n$, then $f(n) = \ Omega(g(n))$. **(2.5 points)**

Solution.

False.

Justification: $f(n) = 2^{\sqrt{\log n}}$, $g(n) = n$

$\lim_{n\to+\infty} \frac{2^{\sqrt{\log n}}}{n} = 0$ so therefore, so f(n) $\neq\Omega(g(n))$

(ii) If $T(n) = T(n/2) + T(n/6) + O(n)$, then $T(n) = O(n)$. grade2.5

Solution. True

Consider the recursion tree. At the root, we have $C * n$ time. In the next level, we have $(\frac{1}{2} + \frac{1}{6})C * n = (\frac{8}{12})C * n$. After that level, we have $(\frac{1}{2} + \frac{1}{6})^2 C * n = (\frac{8}{12})^2 C * n$. Generally, at level i, we have $(\frac{1}{2} + \frac{1}{6})^{i-1} C * n = (\frac{8}{12})^{i-1} C * n$ time.

The total runtime for this is upper bounded by $\sum_{i=1}^{\infty} C * n * (\frac{8}{12})^i = C * n * \frac{1}{1-\frac{8}{12}} = O(n)$.

Therefore, T(n) = O(n).

(iii) If P = NP, then all NP-complete problems can be solved in polynomial time. **(2.5 points)**

Solution. True. All NP-complete problems are also in NP by definition and so if P= NP, they can be solved in polynomial time also.

(iv) If P $\neq$ NP, then no problem in NP can be solved in polynomial time. **(2.5 points)**

Solution. False.

Based on this, we can denote that P is a subset of NP, this means that any problem that is solvable in polynomial time is also able to be verified in NP. $P \neq NP$ basically is saying that the decision version cannot be solved in polynomial time, though it is actually verifiable in poly-time. Furthermore, a problem that is in P would both be solvable and verifiable in poly-time, so we can say that there are some problems in NP that could possibly be solved in polynomial time.

(b) Prove the following statement: Consider a flow network G and a flow f in G. Suppose there is a path from the source to sink such that $f(e) < c_e$ for all edges of the path, i.e., the flow on each edge is strictly less than its capacity. Then, f is *not* a maximum flow in G. **(10 points)**

Solution. We can prove this by contradiction.

Let's assume that f is the maximum flow in the graph. So, with this, let us suppose that there are five edges from source to sink. and the capacities for each edge, respectively, is, a, b, c, d, e. With this, the max flow will be equal to min(a, b, c, d, e). We add the min in front because we want to prevent it from overflowing.

Let's suppose that c is the min, so f_{max} = d. This violates (contradicts) the condition given in the problem, $f(e) \leq c_e$ because the flow on each edge is less than its capacity with no leeway. So, with this, our assumption that we made in the first sentence, is incorrect, $f \leq f_{max}$. f is not the maximum flow in G. proved.

Problem 2. We have an array $A[1:n]$ consisting of n positive integers in $\{1,\ ldots, M\}$. Our goal is to check whether it is possible to partition the indices $[1:n] = \{1, \ldots, n\}$ of the array into two sets S and T ($S \cup T = [1:n]$ and $S \cap T = \emptyset$) such that the sum of elements in both sets is the same, i.e.,

$$\sum_{s \in S} A[s] = \sum_{t \in T} A[t].$$

Design a dynamic programming algorithm with worst-case runtime of $O(n^2\ cdotM)$ to determine whether or not such a partition exists for a given array $A[1:n]$. The algorithm only needs to output *Yes* or *No*, depending on whether such a partition exists or not. **(20 points) Example.** A couple of examples for this problem:

- Suppose $n = 6$ and $M = 3$ and $A = [3, 1, 1, 2, 2, 1]$. In this case, the answer is *Yes*: we can let $S = \{2, 3, 4, 6\}$ and $T = \{1, 5\}$ and have that:

$$\sum_{s \in S} A[s] = A[2] + A[3] + A[4] + A[6] = 1 + 1 + 2 + 1 = 5,$$
$$\sum_{t \in T} A[t] = A[1] + A[5] = 3 + 2 = 5.$$

- Suppose $n = 4$ and $M = 8$ and $A = [1, 2, 3, 8]$. In this case, the answer is *No* as there is no way we can partition $set1, 2, 3, 4$ for this problem (to see this, note that $8 > 1 + 2 + 3$ and so the set containing 8 (index 4) is always going to have a larger sum).

Remember to *separately* write your specification of recursive formula in plain English (**7.5 points**), your recursive solution for the formula and its proof of correctness (**7.5 points**), and runtime analysis (**5 points**).

Solution.

Algorithm: For this problem, we can create a two-dimensional array with boolean elements. We use boolean elements so there is either a true or false option which is similar to Yes/No and what we are looking for.

- For every element i in the array and sum value s (this will be incremented until it reaches value S/2)
  - Within this loop, check if you can exclude element i and form a subset with the sum equal to s.
  - Next, test the condition, value of element is less than s
    - if true, then continue and check if you can include i and form a subset with sum equal to s
  - While checking, if any conditions are true, then store true into array at a value at ith row and sth comlumn
    - this means that we can form the subset of elements with sum which is equal to s.
- return an output of yes if we can form a subset of elements with sum equal to s.
- else return no.

Proof of Correctness:

Runtime Analysis:

Problem 3. You are given a directed graph $G = (V, E)$ with two designated vertices $s, t \in V$, and some of the vertices in G are colored *blue*. Design and analyze an $O(n + m \log m)$ time algorithm that finds a path from s to t that uses the *minimum* number of blue vertices (note that the path can use any number of non-blue vertices). **(20 points)** Remember to *separately* write your algorithm (**7.5 points**), the proof of correctness (**7.5 points**), and runtime analysis (**5 points**).

Solution. *Algorithm:* Objective is to find the shortest path from s to t that uses the minimum number of blue vertices.

- Create a weighted graph, G' = (V,E) by copying graph G.
- weight of the directed edge, (u,v) = 1 if color[v] = 1, otherwise, the weight of (u,v) will be 0.
- Starting from s, perform Dijkstra algorithm on G' to get the shortest path from s to all vertices.
- Return the shortest path from s to t, which is the solution we are looking for.

Proof of Correctness:

For any 2 s-t paths P... we will prove two statements

- If P is the shortest path in G', then the corresponding path in G uses the least amount of blue vertices.
  - Let us assume that P is the shortest path in G' and the corresponding path in G does not use the least amount of blue vertices. So, in the graph that would make the most sense, the shortest path in G' would be at worst be equal to $nb_1 - 1$, the number of edges connecting non-blue vertices to each other where $s, t \in nb$. because of this, it is not possible for the corresponding path in G to not use the least amount of blue vertices because G' uses no blue vertices and the weight of one blue vertex in G' would be n+1. It is not possible for P to be the shortest in G' without using the lease amount of vertices in G because $nb_1 - 1 < n + 1$. So, therefore if P is the shortest path in G', then the corresponding path in G uses the least amount of blue vertices.

- if P is the path in G that uses the least amount of blue vertices, then the corresponding path in G' is the path that has minimum weight.
  - Let us assume that P is the path in G that uses the least amount of blue vertices and that the corresponding path P' in G' is not path that has minimum weight. Since P is the path with the least amount of blue vertices, this means that it will only use blue vertices if necessary, like if cutting was taking place in the graph. So, with this, if it was possible, P would only use non-blue vertices, making its corresponding weight in G' to be equal to the $nb - 1$. If that same path in G' does not have the minimum weight, that means it must have gone through a blue vertex which weighs n+1 which is more than the maximum weight obtainable in a graph with 0 blue vertices. This is impossible because G used the least amount of blue vertices, so if P is the path in G that uses the least amount of blue vertices, then the corresponding path in G' is the path that has minimum weight.

Runtime Analysis:

The runtime is O(n + m*logm). This is because we ran Dijkstra's algorithm on G' and copied the graph g (which takes O(n+m)). So, the entire algorithm's runtime will take O(n+m+logm).

Problem 4. You are given an undirected graph $G = (V, E)$ such that every *vertex* is colored red, blue, or green. We say that a collection of paths $P_1, \ldots, P_k$ is **colorful** if:

(i) each path has only three vertices and these vertices are colored red, blue, and green *in this order*,

(ii) each vertex of the graph appears in *at most* one of these paths in total.

Design an $O((m + n) \cdot n)$ time algorithm that outputs the size of the emphlargest collection of colorful paths in a given undirected graph $G = (V, E)$. grade20 Remember to *separately* write your algorithm (**7.5 points**), the proof of correctness (**7.5 points**), and runtime analysis (**5 points**).

Solution.

Algorithm:

→ Create graph G' = (V,E) where every vertex in G is in G' as well. However, there is an exception of blue vertices that will split into v^{in} and v^{out} in order to enforce vertex capacity of 1. Blue vertices v in G will have an edge with a capacity of 1 between v^{in} and v^{out} in G'. For every edge e(u,v) ∈ G (where u is a red vertex), we will add a directed edge e(u, v^{in}) with a capacity of 1 to G' if and only if v is a blue vertex. For every edge (u,v) ∈ G where u is a blue vertex, add a directed edge $e(u^{out}, v)$ with a capacity of 1 to G' if and only if v is a green vertex.

- First, add a source s such that s has an edge capacity of 1 connected to every red vertex.
- Then Add a sink t, where t has an edge capacity of 1 connected to every green vertex.
- Find the Ford-Fulkerson max flow algorithm on G' which will allow us to find the max number of colorful paths in graph G.

Proof of Correctness:

Assigning a capacity of 1 to all blue edges, connecting all red vertices to s and all green vertices to t, you compute the max flow in G'. This computation is also the same as finding the maximum number of vertex disjoint paths. Furthermore, in this we know that the red vertices are only directed towards the blue vertices and the blue vertices only directed towards the green ones. Because of this, we can ensure that all paths will follow a red, blue, and green path and also simultaneously only have 3 vertices per path.

- If the number of vertex disjoint paths in G' is k, then k is also the size of the largest collection of colorful paths in G.
  - Since the paths are vertex disjoint, we use the paths found by the algorithm to get max flow and the trimming that occurred from graph G also make sures that every path will only have 3 vertices while also following an order of red, blue, green.
- If the k is the size of the largest collection of colorful paths in G, then k is also the number of vertex disjoint paths in G'.
  - We find vertex disjoint paths in G' using the max flow algorithm. Since k is the number of colorful paths we can make in G, then by definition of colorful path, there are also j vertex disjoint paths in G'.

Runtime Analysis:

This algorithm's runtime is $O((m + n) * n)$. This is true because the network takes a runtime of O(n+m) to create. Also, the Ford-Fulkerson algorithm has a runtime of O(m'*F), where m is the number of edges in G'. Because m' = m+n and the max flow is n/3 or n, the runtime ends up being $O((m + n) * n)$.

Problem 5. Prove that the following problems are NP-hard. For each problem, you are only allowed to use a reduction from the problem specified.

(a) **4-Coloring Problem:** Given an undirected graph $G = (V, E)$, is there a 4-coloring of vertices of G? A 4-coloring is an assignment of colors $\{1, 2, 3, 4\}$ to vertices so that no edge gets the same color on both its endpoints. **(10 points)** For this problem, use a reduction from the 3-*Coloring problem.* Recall that in the 3-Coloring problem, you are given a graph $G = (V, E)$ and the goal is to find whether there is a 3-coloring of G or not. A 3-coloring is an assignment of colors $\{1, 2, 3\}$ to vertices so that no edge gets the same color on both its endpoints

Solution. Reduction from the 3 coloring problem:

Reduction Steps:

- We are given an instance, $G = (v, E)$ of the 3-coloring problem, we can create an instance of G' of the 4-coloring problem
  - Add a new vertex to graph G.
  - Draw an edge from each of the 3-colored vertices to the new vertex added in the step above this one.
  - note, this vertex has to be assigned to a different color because if the 3-colored graph is not colored properly, then it will mess up the 4-color graph, G'.
  - Run algorithm for 4-color problem on G'.
  - Give the produced output, G'.

Proof of Correctness:

We can show that G has a solution G' that outputs a 4-coloring graph. This is possible because 3-coloring types of problems are NP-complete and therefore any NP problem can be reduced to 3-coloring and this action can also be applied to the 4-coloring problems to reduce them.

Proving two statements:

- If G has a working 3-color graph of size k vertices, then G' has a working 4-color graph with k+1 vertices.
  - Vertices can be labeled as follows: v_1,...v_{k+1} and edges can be referred to as e_1,...e_k.
  - Next, we know that we have added a vertex, v_{k+1} and all k vertices will add an corresponding edge to the new vertex v_{k+1}. This also means that there is a size of $k + 1$ vertices.
  - So, with this, the new vertex will be assigned to the 4th color and will be connected to the other vertices in the 3-colored that are all separately assigned to three separate colors.
- If G' has a graph of size $k + 1$ and a 4-color graph, then G has a working 3-color graph of size k.
  - Vertices can be labeled as follows: v_1,...v_{k+1} and edges can be referred to as $edges_e$ which are added edges and these both make up the graph, G'.
  - In graph G, there are 3 separate colors for the vertices, so, this means that the color of the vertex, v_{k+1} which has an edge to every vertex does not have any of the colors that are associated with the 3 colors in graph G.
  - Since in G' there is an extra vertex and its associated extra edges, you can remove the extras with $(k + 1 - 1)$ to go back to forming the 3-color graph of G containing the amount of k vertices.

Runtime Analysis:

The runtime of this reduction algorithm is in polynomial time which is also the same as the runtime for the 3-coloring problem. $O(n + m)$.

(b) **Two-Third "Hamiltonian" Cycle Problem:** Given an undirected graph $G = (V, E)$ is there a cycle in G that passes through at least 2/3 of vertices? **(10 points)** For this problem, use a reduction from the Hamiltonian Cycle problem. Recall that in the Hamiltonian Cycle problem, we are given an undirected graph $G = (V, E)$ and the goal is to output whether there is a cycle in G that passes through *all* vertices.

Solution. We typically say that a graph G has a (2/3)-Path if there is a path in G that passes through at least two thirds of the vertices in G. *Reduction (from Undirected s-t Hamilton Path):*

- Given an instance G = (V, E) of the undirected s-t HP problem, we can create an instance G' of Two-Third-Path as follows:
  - We can obtain G' by adding a = $\frac{1}{2}$n + 2 dummy vertices to graph G and connecting them to s and adding one additional dummy vertex and connecting it to t.
  - We can then run the algorithm for Two-Third-Path on G' and output G has s-t HP iff the algorithm outputs G' has a (2/3) path.

Proof of Correctness: We can show that G has an s-t HP path iff G' has a (2/3) path.

- if G has a s-t Hamilton Path P, then G' has a (2/3) path:
  - The path P' = d_s →s squiggly arrow p t→, $where$d$_s$ is one of the dummy vertices connected to s and d_t is the dummy vertex connected to t, is a path in G' that passes through exactly n+2 vertices. So, it is a valid answer for the (2/3) Path problem.
- If G' has a (2/3) path, then G has a s-t Hamiltonian path:
  - need to prove this one and relook at answer in hw i confused hehe

This implies the correction of the reduction.

Runtime Analysis: Creating the graph G' can be implemented in O(n+m) time and a poly-time algorithm for (2/3) Path also implies a poly-time algorithm for undirected s-t HP which also in turn implies a P= N.

Problem 6. [**Extra credit**] You are given an *undirected* graph $G = (V, E)$, and three vertices $u, v, w \in V$. Design and analyze an $O(m+n)$ time algorithm to determine whether or not G contains a (simple) emphpath from u to w that passes through v. You do *not* need to return the path. **(+10 points)** Note that by definition of a path, no vertex can be visited more than once in the path so it is *not* enough to check whether u has a path to v and v has a path to w.

Solution.

Algorithm:

- Create a flow network from G. We can do this by making all edges in G bidirected.
- Next, add a new source vertex s that is connected to u and w
- Then, run DFS algorithm on u and mark all nodes in the path as they are visited while the algorithm is running.
- While running DFS, if the nodes w is true then, we can say that there is a path between u and w
- Next, also check if path also goes through node v.
  - If true, then return Yes
- Otherwise, return No

Runtime Analysis:

We are using DFS algorithm and so therefore the runtime will be O(m+n).