CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2021**

Final Exam

Due: Monday, May 10, 1:00pm EST

Name: _____ *Yinfeng CONG* *NetID:* _____ *yc957*

Instructions

1. Do not forget to write your name and NetID above, and to sign Rutgers honor pledge below.
2. The exam contains 4 problems worth 100 points in total *plus* one extra credit problem worth 10 points.
3. This is a take-home exam. You have exactly 48 hours to finish the exam.
4. 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 (any inquiry should be sent directly to the Instructor or 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**.
5. 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 it is harder to get partial credit on that problem.
6. **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.
7. 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:_____

Problem. #	Points	Score
1	25	
2	25	
3	25	
4	25	
5	+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
g(n) = n = 2_{logn}
Also, we know that $\sqrt{log\,(n)} = o\,(logn)\,, 2^{\sqrt{logn}} = o\left(2^{logn}\right)$.
So we get that f(n) = o(g(n)), so f(n) can not be $\Omega(g(n))$. Hence, it is false.

(ii) If $T(n) = T(n/3) + T(n/4) + O(n)$, then $T(n) = O(n)$. **(2.5 points)**

Solution. True
Consider the recursion tree. At the root, we have C*n times. In the next level, we have C*(n/3) + C*(n/4) = (7/12) Cn. In the level after that, we have $\left(\frac{1}{3} + \frac{1}{4}\right)^2 \cdot Cn$. In general, at level i, we have $\left(\frac{1}{3} + \frac{1}{4}\right)^i \cdot Cn$ time.
so the total runtime is upper bounded by $\sum_{i=0}^{\infty} C \cdot n \cdot \left(\frac{7}{12}\right)^i$ = Cn, and this means that 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 all 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
If P != NP, it means that P belongs to NP, and this means that there are problems that belong to P, and the problems belong to P also belong to NP(by definition). They can be solved in polynomial time, so this is false.

(b) Prove the following statements.

(i) Suppose G is a directed acyclic graph (DAG) with a unique source s. Then, there is a path from s to v for any vertex v in G. **(7.5 points)**

Solution. We can prove this by a contradiction. We suppose that there is not a path from s to any arbitrary vertex v. This means that from the source s, we can not get to the vertex v. This is a DAG here, so this should be a connected graph. Yet, for a DAG with unique source s, it should pass all the vertices to make this graph connected. However, if there is no path from s to v, then v is not connected to the graph. This is a contradiction, so there must be a path from s to v to make this graph a connected graph.

(ii) 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. **(7.5 points)**

Solution. We first need to get to know the definition of the maximum flow. A maximum flow of a graph is a flow that makes there is no more path from s can be arrived to t. (because the capacity of some edges is equal to the flow of it) This means that for a maximum flow, in every path from s to t, there should be an edge in this path that the capacity of this path is equal to the flow of this path. If there is a path that all the edges have f(e) $< c_e$, this means that this path must be added in again until we can make sure that there is no more paths can be added in again (until there is some edge in this path makes f(e) $= c_e$). This means that —- if there is a path that f(e) $< c_e$ for all the edges of the path. Then, this is not a maximum flow of G.Also, (the part below is another thought, I am not sure if it is correct, if it is incorrect, please ignore it) based on the preservation of flow, the flow into the vertex is equal to the flow out of, as the flow into the vertex decrease, the flow out of the vertex decrease as well, this would result to decrease on the edges that we pass, which make that we can not get a maximum flow.

Problem 2. We consider a different variant of the Knapsack problem in this question. You are given n items with integer weights $w_1, \ldots, w_n$ and integer values $v_1, \ldots, v_n$ and a target value V. Your goal is to determine the *smallest* knapsack size needed so that you can fit a set items in the knapsack with total value at least V. In other words, you want to *minimize* $\sum_{i \in S} w_i$ subject to $\sum_{i \in S} v_i \geq V$ (over the choice of S from n items).

Design an $O(n \cdot V)$ time dynamic programming algorithm for this problem.

(a) *Specification of recursive formula for the problem (in plain English)*: **(5 points)**

Solution. (First, we need to make sure that we can reach the amount value of V) For any integers $0 \leq i \leq n$, $0 \leq j \leq V$, define:
K(i,j):The minimum weight we can obtain by picking a subset of the first i items, when we have a knapsack that get the value to be equal or more than j.
We can solve this problem by returning K(n,V), because K(n,V) is the minimum weight we can obtain by picking a subset of first n items, when have a constraint that value is equal of bigger than V.

(b) *Recursive solution for the formula:* **(7.5 points)**

Solution. K(i,j) =
undefined (if $\sum j_i < V$)
0 (if i=0 or j=0)
min[k(i-1,j-V_i) + W_i, k(i-1, j)] (Otherwise)

(c) *Proof of correctness of the recursive formula:* **(7.5 points)**

Solution. First, we consider the base case of this problem: either i=0 or j = 0. In both cases, we have K(i,j) = 0 which is also the value we can achieve by using the first 0 items or when the value is 0.So the base case matches the specification.
Then,for larger values of i and j. Suppose the total value is smaller than the value constraint V. In this case, it does not obey the information provided in the question, so this part is undefined. Otherwise,we have two options for us to minimizing the value:
1. we either pick item i in our solution which leaves us with the first i-1 items remaining to choose from next and a knapsack of the value $j - V_i$ but we also collect the weight w_i. Hence, in this case, we can obtain the weight of k(i-1,j-V_i) + w_i
2. The other option would be to not pick i in our solution which leaves us with the first i-1 items remaining to choose from and a knapsack of the value j. This is captured by the weight k(i-1,j)
By picking the minimum of these two options in the formula, we obtain the best solution, proving corretness.

(d) *Runtime analysis:* **(5 points)**

Solution. We have n*V subproblems and each one takes O(1) time to compute, hence the total runtime of the dynamic programming algorithm obtained from this formula is O(n*V) (It can also be seen as two for-loops with outer-loop iterating n times and inner-loop iteration V times and we spend O(1) time each iteration)

Problem 3. You are given a directed graph $G = (V, E)$ such that every *edge* is colored red, yellow, or green, and two vertices s and t. We say that a path from s to t is a *good* path if (1) it has *at least one* edge of each color, and (2) all the red edges in the path appear before all the yellow edges, and all the yellow edges appear before the green edges. For instance, a $(red, red, yelow, green, green, green)$ path is a good path but neither a $(red, green, green)$ path nor a $(red, green, yellow)$ path are good.

Design and analyze an $O((m + n) \cdot n)$ time algorithm that outputs the size of the *largest* collection of *edge-disjoint good* paths from s to t in a given directed graph $G = (V, E)$ with n vertices and m edges.

(25 points)

Solution. Algorithm(Reduction):
Simply turn G into a network G'(V',E') by assigning capacity 1 to every edge (and if G is originally undirected, add both direction of edges to make it directed).
For any edge e, we connect an edge of capacity 1 in G'. If there is any edge e_r belongs to E' where e_r is red, and the next neighbor edge of it is also red or yellow (the next neighbors are the edges have capacity 1 from them), we connect them together. Similarly, if there is any edge e_y belongs to E' where e_y is yellow, and the next neighbor edge of it is also yellow or it is green, we connect them together. Again, if there is any edge e_g belongs to E' where e_g is green, and the next neighbor edge of it is also green, we connect them together. Finally, we record all the edges from s that has a capacity of 1(also make sure the edge is red) and we record all the edges to t that has a capacity of 1(also make sure that the edge is green)
Find the maximum flow in this network from s to t and return the value of maximum flow as the answer of the problem.
A note here: We can use the method of BFS here, and run BFS on the graph to get all the edges can be got from s (We pause BFS and change another path once we find edges of other color in the path). Then we continue our BFS again to make sure we record the yellow edges (we pause again and change path once we find edges of the other color in the path)[Then we run the same method for green edges] and then we build our graph, and we change the graph to network and try finding the max flow.
The note above is my thought about specific method for solving this question, it might not have connection with this problem
Proof of correctness:
We prove that there is a flow of value k in G' if and only if there is a collection of k colorful paths in G. This implies that the maximum value of flow in G' is equal to the size of largest collection of edge-disjoint good paths in G.
1. Suppose there are k colorful paths in G, create a flow f of value k as follows:
For any colorful path P with edges e_r, e_y, e_g. Send one unit of flow from s to e_r and then to another e_r or e_y, then from e_y to another e_y or e_g, again from e_g to another e_g finally to t. Since in the collection of edge-disjoint good paths, all the kinds of edges would appear at least once, and the order of them is red, yellow and green, and we have make sure the order of the edges are correct thus this would be a valid flow with the same value as number of paths in the collection.
2. Suppose now there is a flow of value k in G', create a collection of k edge-disjoint good paths in G as follows:
Any flow path in G' is of the form:
from s to e_r and then to another e_r or e_y, from e_y to another e_y or e_g, from e_g to another e_g finally to t
Moreover, since capacity of every edge is only 1 in G', and all kinds of edges would appear at least once in one path. Thus, the flow of paths give us k edge-disjoint good paths in G.
Runtime analysis:
Create the network spends O(m+n) time and computing Ford-Fulkerson takes O(m*F) time, where F is the maximum value of flow from s to t. Since the value of such flow is at most n. Thus, by running Ford-Fulkerson algorithm for max-flow, the running time of this algorithm is O((m+n)*F) = O((m+n)*n) as desired.

Problem 4. 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). **(12.5 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. I have two kinds of reductions for solving this question, so I write both of them, and I am not sure if both of them are correct.
Reduction(The first method):
We have an algorithm A for 3-coloring problem, and given a 4-corloring problem graph G(V,E), we simply run A on G to find if we can find a way to solve this question just by 3-coloring method. If we find that we do not need to paint the forth color, then we are done with this (because a graph of 3-coloring can also be a kind of graph of 4-coloring). However, if we find that there are some vertices that can not be painted, we can just paint the forth color on it and check if the neighbor vertice of it is also the forth color (If the neighbor is the forth color, then it is not a 4-coloring problem, if the neighbor is not the forth color[which means we have different color on each side] then this is a 4-coloring problem) And this shows the answer to the 4-coloring problem. (which means that we used the method of 3-coloring solving the 4-coloring problem)
Reduction(the second method):
Given an instance G = (V,E) of the 3-coloring problem, and we create an instance G' of the 4-coloring problem as follows.
Add n/3 new vertices to G (make sure these n/3 vertices not be the neighbor of one another) and connect them to the vertices in G to obtain G'.
We run the algorithm for 4-coloring problem on G' to obtain its answer on G', and if we get the output that we can have 4-coloring problem correct on G', which means that output G can have 3-coloring problem correct.
Proof of correctness:
(i) If G can be solved by 3-coloring problem, then G' can be solved by 4-coloring
Once we can solve the graph G in 3-coloring method, and we added n/3 vertices to the G to get a graph G'. The only difference is that G' has the space for the left forth color, so we add n/3 places prepared for the 4th color,and if G can be solved by the 3-coloring method. We can add the forth color on the new added spaces in G'. So that it means G' can be solved by 4-coloring method.
(ii) If G' can be solved by 4-coloring problem, then G can be solved by 3-coloring
Since the n/3 vertices can be only connected to the original vertices, and none of them can be the neighbor of one another. Removing them from G' won't make the graph of G change. Hence, it does not affect the correctness of the 3-coloring method on G. Hence, G still can be solved by 3-coloring.
Runtime analysis:
The reduction can be implemented O(m+n) time and hence a poly-time algorithm for the 4-coloring problem implies a poly-time algorithm for 3-coloring problem which in turn implies P = NP. Thus a poly-time algorithm for 4-coloring problem also implies P= NP, making this problem NP-hard.

(b) **Hamiltonian Path Problem:** Given an undirected graph $G = (V, E)$, does G contain a path that goes through all vertices, i.e., a Hamiltonian path? **(12.5 points)**

For this problem, use a reduction from the *s-t Hamiltonian Path problem*. Recall that in the *s-t* Hamiltonian Path problem, you are given a graph $G = (V, E)$ and two vertices s, t and the goal is to decide whether there is a *s-t* path in G that passes through all other vertices.

(Note that the difference between Hamiltonian Path problem and *s-t* Hamiltonian Path problem is that in the former problem, the path can start from any vertex and end in any vertex as long as it goes through all vertices, while in the latter it should start from s and ends at t.)

Solution. Reduction:
Given an instance G(V,E) of the undirected s-t Hamiltonian path problem, we create an instance G' of the Hamiltonian Path problem as follows.
For any vertex v and z, add them to G and connect one of them before s the other one after t to obtain G'.
We run the algorithm for the Hamiltonian Path Problem on G' and output G has a s-t Hamiltonian Path if and only if the algorithm outputs G' has Hamiltonian Path.
Proof of correctness:
(i) If G has a s-t Hamiltonian path, then G' has Hamiltonian path.
Let P be a s-t Hamiltonian path from s to t in G. This path has passed all the vertices in the G. If there are no vertices disconnect, we add the two vertices to G' and we just added two vertices (one in the front the other one at the end) It does not affect the result of Hamiltonian path. Hence, G' has Hamiltonian path. (ii) If G' has Hamiltonian path , then G has a s-t Hamiltonian path.
Since the two vertices are added in the front of s and at the back of t, so when we delete them, they do not affect the result (if there is a hamiltonian path in the graph). The graph still has the s-t paths, and if it used to have a paths that passed all vertices in G', after deleting the two vertices, it would still have a hamiltonian path of s-t.
Runtime analysis:
The reduction can be implemented O(m+n) time and hence a poly-time algorithm for the Hamiltonian path problem implies a poly-time algorithm for s-t Hamiltonian path problem which in turn implies P = NP. Thus a poly-time algorithm for Hamiltonian path problem also implies P= NP, making this problem NP-hard.

Problem 5. [**Extra credit**] Alice wants to throw a party and is deciding who to invite. She has n people to choose from and she has made up a list of which pairs of these people know each other. She wants to pick as many people as possible, subject to the constraint that at the party, each person should know at least five other people.

Give a polynomial time algorithm that takes as input a list of n people and the list of pairs who know each other and outputs the maximum number of guests that Alice can invite.

(+10 points)

Hint: Get creative and design an algorithm for this problem from scratch; this problem is *not* about using reductions to the problems you have already seen in this course.

Solution. Algorithm:
We build a graph G(V,E), and set V as people, and E as the relationship between them.
First, initial the weight of all edges to 0.
Check the list of pairs and change the weight of edge to 1 (The edge here is the edge that mentioned in the pairs)
For i = 1 to n,
Then we run DFS on G, and set a count[], record the indegree of each vertices in count[i]
Sort count[] in decreasing order, and invite the first n people appear.
Then we can return the maximum number of guest.
Proof of correctness:
we have known the correctness of DFS, and we want to get to know the maximum number of people to invite and make sure that the people have the largest number of people they know, so we would have a list of how many people they know, and sort them. That is how the idea of this problem come out. Thus, the correctness is proved.
Runtime analysis:
Create the graph spends (m+n) time, and it uses (m+n) time for the DFS, and spend O(nlogn) for sorting. So the total runtime is O(nlogn), and it is a polynomial time.

Extra Workspace

Extra Workspace

Extra Workspace