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

Final Exam

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

Name: *Ryan Coslove* *NetID:* *rmc326*

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: Ryan Coslove

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

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

(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 $C*n$ time. In the next level, we have $(\frac{1}{4}+\frac{1}{3})C*n = \frac{7}{12}C*n$. In the level after that, we have $(\frac{1}{4}+\frac{1}{3})^2C*n = (\frac{7}{12})^2C*n$. In general, at level i, we have $(\frac{1}{4}+\frac{1}{3})^{i-1}*C*n = (\frac{7}{12})^{i-1}C*n$ time.

So the total runtime is upper bounded by $\sum_{i=1}^{\infty} C*n*(\frac{7}{12})^i = C*n*\frac{1}{1-\frac{7}{12}} = O(n)$

This means $T(n) = O(n)$

(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

We know that P is a subset of NP. That is, any problem that can be solved in polynomial time can also be verified in NP. P $\neq$ NP would imply that the decision version of the problem cannot be solved in poly time, despite being verifiable in poly time. On the other hand, a problem already known to be in P would continue to both be solvable and verifiable in poly time. So at least some problems in NP, not none, can be solved in polynomial time.

(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.

It is important to prove that G contains no cycle. So we will topologically sort G. The algorithm for topological sort:

i. Let D[1:t] be an array initilized to 0 and O be an empty linked-list for the output ordering.
ii. Go over all vertices v and for any $u \exists N(v)$, increase D[u] by one. (At the end of this step, D[v] denotes the in-degree of v for all $v \exists V$).
iii. Insert every vertex v with $D[v] = 0$ into a queue Q.
iv. While Q is not empty:
   - Let v be the first vertex of Q and dequeue (remove) this vertex from Q.
   - Add v to the end of the linked-list O.
   - For $u \exists N(v)$: Reduce D[u] by one. If D[u] = 0 insert u to Q
v. If $|O| < n$, output as not a DAG path; otherwise output O as a topological ordering of G

Knowing G is topologically sorted, we prove that it is acyclic so v never comes before s, (v, s). So we prove that by contradiction that there will always be a path from s to v, (s, v).

(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 define a $flow$ in a given network $G = (V, E)$ as any function $f : VxV \rightarrow R^+$ that satisifies the following capacity constraint:

- For any edge $e = (u, v) \in E, f(u, v) \leq c_e$ and if there is no edge from u to v, then $f(u, v) = 0$ (a flow over each edge can't be larger than the capacity of the edge).

If you ran a ford-folkerson algorithm on the above condition where $f(u, v) \leq c_e$ you would find a maximum flow.

With the condition we have in the problem where the flow edge is strictly less than the capacity of the edge, $f(e) < c_e$, we know that if we run the same algorithm it's max flow would typically, if not always, be less than that of a normal flow where the edge can equal its capacity value (like above). Therefore our f in the problem is not a max flow in G.

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.

For any integers $0 \leq i \leq n$ and $0 \leq j \leq V$, we define:

- $K(i, j)$: the minimum value we can obtain by picking a subset of first i items, i.e., items 1. ..., i, when we have a knapsack of size j.

By returning $K(n, V)$, we can solve the original problem. This is because $K(n, V)$ is the minimum value we can obtain by picking a subset of the first n items, when we have a knapsack whose target value is V.

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

Solution.

We write a recursive formula for $K(i, j)$ as follows:

$$K(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ K(i-1, j) & \text{if } v_i > j \\ min\{K(i-1, j-v_i) + w_i, K(i-1, j)\} & \text{otherwise} \end{cases}$$

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

Solution.

Let us consider the base case of this function first: either when $i = 0$ 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 (i.e., no item at all or $i = 0$) or when the knapsack has no size (i.e., $j = 0$). So the base case of this function matches the specification.

We now consider the larger values of i and j. Suppose first $v_i > j$. In this case, $K(i, j) = K(i - 1, j)$. This is correct because we cannot fit item i in a knapsack of size j and thus the best value we can achieve is by picking the best combination of items from the first $i - 1$ items, which is captured by $K(i - 1, j)$. Thus, whenever $v_i > j$, $K(i, j) = K(i - 1, j)$ precisely captures the specification we had.

Finally, we have the case when $v_i \leq j$ (corresponding to the last line of the recursive formula). At this point, we have two options in front of us for minimizing the value of items:

(a) 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 size $j - v_i$ but we also collected the value w_i. Hence, in this case, we can obtain the value of $K(i - 1, j - v_i) + w_i$ (remember that $K(i - 1, j - v_i)$ is the smallest value we can get by picking a subset of the first $i - 1$ items in a knapsack of size $j - v_i$ which is precisely the size of our knapsack after we pick item i in the solution).

(b) 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 same size j (we do not collect any value in this case). This is captured by the value $K(i - 1, j)$.

But which of options (1) or (2) we should choose? Since our goal is to pick the minimum value we can get by picking a subset of first i items in a knapsack of size j (the definition of $K(i, j)$), we should also pick the option between (1) and (2) which gives us the minimum value; hence,$K(i, j) = min\{K(i - 1, j - v_i) + w_i, K(i - 1, j)\}$ as computed by the function as well.

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

Solution.

There are n choices for i and V choices for j so there are in total $n * V$ subproblems. Each subproblem also, ignoring the time it takes to do the inner recursions, takes $O(1)$ time. Hence, the runtime of the algorithm is $O(nV)$.

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 to network flow finding edge-disjoint problem):

Create a network $G' = (V', E')$ where $V' = \{s, t\} \cup U_R \cup W_R \cup U_Y \cup W_Y \cup U_G \cup W_G$: starting from s on our way to t, for any red edge $e \in E$, we create two edges $u_e \in U_R$ and $w_e \in W_R$; similarly, for any yellow edge in $u_e \in E$ we create two edges $u_e \in U_Y$ and $w_e \in W_Y$; and finally for any green edge in $u_e \in E$ we create two edges $u_e \in U_G$ and $w_e \in W_G$.

We set the capacity of any edge $e = (s, u)$ for $u \in L$ to be $c_e = 1$. Similarly, we set the capacity of any edge $e' = (v, t)$ for $v \in R$ to one also, i.e., $c_{e'} = 1$. This is construced from u_e to w_e in G'. Moreover, if there is an edge $(e, e') \in E$, where e is red and e' is yellow, we connect $w_e \in W_R$ to $u'_e \in U_Y$ with edge capacity $c_e = 1$. Similarly, if there is an edge $(e, e') \in E$, where e is yellow and e' is green, we connect $w_e \in W_Y$ to $u'_e \in U_G$ with edge capacity $c_e = 1$. Finally, we connect s to all vertices in U_R with edges of capaicty 1 and connect all vertices in W_G to t with edges of capacity 1.

Find a maximum flow k from s to t and return the value of the maximum flow as the answer to the probelm.

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 the largest collection of good/colorful paths in G.

1. Suppose there are k colorful paths in G, create a flow f of value k as follows: For any good path P with edges $e_{red}, e_{yellow}, e_{green}$, send 1 unit of flow from s to $u_{e_{red}} \in U_R$ to $w_{e_{red}} \in W_R$, then from $w_{e_{red}} \in W_R$ to $u_{e_{yellow}} \in U_Y$ and next from $u_{e_{yellow}} \in U_Y$ to $w_{e_{yellow}} \in W_Y$, then from $w_{e_{yellow}} \in W_Y$ to $u_{e_{green}} \in U_G$ and next from $u_{e_{green}} \in U_G$ to $w_{e_{green}} \in W_G$, and finally from $w_{e_{green}} \in W_G$ to t. Since in the collection of good paths no vertex is used more than once, we will not use any the edges above more than once also and thus this is a valid flow with the same value asnumber of paths in the collection.
2. Suppose now there is a flow of value k in G', create a collection of k good paths in G as follows: Any flow path in G' is of the form:

$$s \to u_{e_1} \in U_R \to w_{e_1} \in W_R \to u_{e_2} \in U_Y \to w_{e_2} \in W_Y \to u_{e_3} \in U_G \to w_{e_3} \in W_G \to t.$$

This translates to a path $e_1 \to e_2 \to e_3$ in G where e_1 is red, e_2 is yellow, and e_3 is green. Moreover, since capacity of every edge (u_e, w_e) is only 1 in G', no vertex or edge can appear in more than once of these paths. Thus, the flow paths give us k good paths in G.

Runtime

Network G' has $O(n)$ vertices and $O(m + n)$ edges, and maximum flow F in the network is at most n (as there are at most n edges of capacity 1 going out of s). Thus, by running Ford-Fulkerson algorithm for max-flow, the running of 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. Reduction (from 3-coloring problem)

(a) Given an instance $G = (V, E)$ of the 3-coloring problem, we create an instance G' of the 4-coloring problem as follows.

(b) Add a new vertex to graph G.

(c) Draw an edge from each of the 3-colored vertices to the new vertex. This vertex is assigned a new color not previously used. (This 4 color graph G' can only be colored correctly if the original 3 colored graph G is colored correctly).

(d) We run the algorithm for 4-color problem on G' and output G'.

Proof of correctness

Since 3-coloring problems is NP-complete, all NP problems can be reduced to 3-coloring, so we can use this strategy to reduce them all to 4-coloring. We show that G has a solution G' that outputs a 4-coloring graph.

(a) If G has a working 3-color graph of size k vertices, then G' has a working 4-color graph with $k + 1$ vertices. Let the vertices be labeled $v_1, ..., v_k$ in G with edges $e_1, ..., e_k$. Now we've added a vertex, v_{k+1}. All k vertices will add an edge to the new vertex v_{k+1}. Now we have our desired size of $k + 1$ vertices. Our new vertex will be given the 4th color and be connected to vertices that are 1 of 3 colors.

(b) If G' has a graph of size $k+1$ and a working 4-color graph , then G has a working 3-color graph of size k. Let the vertices be labeled $v_1, ...v_{k+1}$ and the added edges e_{added} be the graph of G'. Note that the vertex v_{k+1} has an edge to every vertex, meaning the color of the vertex is not equal to 1 of the 3 colors of the vertices that are originally in G. Since there are extra edges and one extra vertex, they can be removed $(k + 1 - 1)$ to form the 3-color graph of G containing k vertices.

Runtime

The runtime of the algorithm is the same as the one for 3-coloring problem, which is polynomial time, so $O(n + m)$.

(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 from $s - t$ Hamiltonian Path

(a) Given an instance $G(V, E)$ of the undirected $s - t$ Hamiltonian path problem, we create a G' where a path goes through all vertices as follows.

(b) Obtain G' by adding n vertices to G and connecting them to s and adding one additional dummy vertex and connecting it to t.

(c) We then run the algorithm for Hamiltonian Path Problem on G' and output G has $s-t$ hamiltonian path if and only if the algorithm outputs G' has a "full" hamiltonian path. ("full" meaning path goes through all vertices).

Proof of correctness

We show that G has an $s - t$ Hamiltonian path if and only if G' has a hamiltonian path that goes through all vertices. Note that the number of vertices in G' is $n + 1$, hence any valid path for the full Hamiltonian path problem in G' has to be of length at least $n + 1$.

(a) If G has a $s - t$ Hamiltonian path P, then G' has a full Hamiltonian Path that goes through all vertices. The path $P' = d_s \to s \rightharpoonup pt \to d_t$, where d_s is one of dummy vertices connected to s and d_t is the dummy vertex connected to t, is a path in G' that passes through exactly $n + 1$ vertices. Thus, it is a valid answer for the full Hamiltonian Path problem.

(b) If G' has a Hamiltonian Path, then G has a $s - t$ Hamiltonian path. Note that any path length more than 2 cannot contain a dummy vertex connected to s and hence the maximum length of a path in G' is $n + 1$. Moreover, any path of length $n + 1$ in G' must contain a dummy vertex connected to s, all original n vertices, and the dummy vertex connected to t and consequently has to start from and end in a dummy vertex. Hence if G' has a path of length $n + 1$, then removing the first and last (dummy) vertices results in a Hamiltonian path from s to t in G.

This implies the correction of the reduction.

Runtime

The reduction can be implemented in $O(n+m)$ time and hence a poly-time algorithm for Hamiltonian Path implies a poly-time algorithm for undirected $s-t$ Hamiltonian path which in turn implies $P = NP$, which also implies 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*

1. Create a network $G = (V, E)$ where $V = \{s, t\} \cup V_P \cup V_S$. For each person $p_i \in P$, add a vertex $u_i \in V_P$, and for each set S_j, add a vertex $v_j \in V_S$. Connect u_i to v_j with an edge $e = (u_i, v_j)$ of capacity 1 if and only if the person $p_i \in S_j$. Connect s to each u_i with an edge of capacity 1 and connect each v_j to t with an edge of capacity r_j.
2. Find the maximum flow in this graph from s to t. Return the person can be invited if and only if the max flow value is ≥ 5 or $\sum_{j=1}^{n} r_j \geq 5$.

Proof

We prove that the maximum flow in G is equal to $\sum_{j=1}^{n} r_j \geq 5$ if and only if the person can be invited because they know at least 5 other people.

1. Suppose a person can be invited to the party; create the flow f as follows. Send 1 unit of flow from each vertex $u_i \in V_P$ to vertex $v_j \in V_S$ if the person exists within the subset/[pairs of people who know each oher to satisfy the requirement of the set S_j. Since capacity of incoming edge of u_i from s is 1 this is always possible (we assume that the person does not over satisfy a requirement). Since the student can be invited and consequently every requirement is satisfied, the incoming flow of every vertex $v_j \in V_S$ is equal to r_j. Since v_j is connected to t by an edge of capacity r_j the maximum flow is ≥ 5 or $\sum_{j=1}^{n} r_j \geq 5$.
2. Suppose now the maximum flow is ≥ 5 or $\sum_{j=1}^{n} r_j \geq 5$. For each vertex $v_j \in V_S$, we have that the incoming flow to this vertex is equal ≥ 5 or r_j. Since capacity of every incoming edge of v_j is 1, there must be r_j vertices in V_P that provide this flow. Moreover, since these vertices can only transfer 1unit of flow, it means that the outgoing flow of all these vertices is going only to v_j. Hence, we can use the people corresponding to these vertices to satisfy the requirement for S_j, while ensuring that no person is being used towards satisfying multiple requirements. Consequently, every requirement can be satisfied.

Runtime

By running Ford-Fulkerson algorithm for max-flow, the running of time of this algorithm is $O(m * F)$ where $m = O(nq)$ and $F = O(\sum_{j=1}^{n} r_j \geq 5) = O(n)$. Hence, this algorithm is in polynomial time.

Extra Workspace

Extra Workspace

Extra Workspace