**CS 344: Design and Analysis of Computer Algorithms** **Rutgers: Fall 2019**

# Practice Final Exam Solutions

December 10, 2019

*Name:* ______________________________ *NetID:* ______________

## Instructions

1. Do not forget to write your name and NetID above.
2. The exam contains 5 problems worth 100 points in total *plus* one extra credit problem worth 10 points. You have 180 minutes to finish the exam. The exam is closed-book and closed notes.
3. **Note that problems appear on both odd and even numbered pages.** There should be more than enough space to write down your solution for each problem below the problem itself. But if you ran out of space, you can also use the extra sheet at the end of the exam; if you do so, be clear about which problem you are solving.
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 it is harder to get 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 as a building block for your solutions.

**Suggestion:** Leave the extra credit problem for last as it worths fewer points.

| 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 (you do *not* need a proof for this part).

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

**Solution.** *False.*

($f(n) = 2^{\log^2 n}$, $g(n) = 2^{\sqrt{n}}$, and $\log^2(n) = o(\sqrt{n})$, thus $f(n) = o(g(n))$.)

(ii) If $T(n) = 2T(n/2) + O(n^2)$, then $T(n) = O(n^2)$. **(2 points)**

**Solution.** *True.*

(Write the recursion tree for this recurrence.)

(iii) If a problem in NP can be solved in polynomial time, then all problems in NP can be solved in polynomial time. **(2 points)**

**Solution.** *False.*

(P $\subseteq$ NP so there are already many problems in NP that can be solved in polynomial time, yet we do not know P = NP or not.)

(iv) If P = NP, then all NP-complete problems can be solved in polynomial time. **(2 points)**

**Solution.** *True.*

(All NP-complete problems belong to NP and so if P = NP, then all NP-complete problems can be solved in polynomial time.)

(b) Prove or disprove the following assertions (you do need a proof or a counter example for this part).

(i) Consider a flow network $G$ and a maximum flow $f$ in $G$. There is always an edge $e$ with $f(e) = c_e$, i.e., the flow passing the edge is equal to the capacity of the edge, such that increasing the capacity of edge $e$ in $G$ increases the maximum flow of the network. **(6 points)**

**Solution.** *False.*

Consider the network consisting of only three vertices $s, v$ and $t$, and two edges $(s, u)$ and $(u, t)$ both with capacity 1. The only maximum flow in this network is to send one unit of flow over both edges, but increasing the capacity of either edge does not increase the maximum flow because the other edge is still a bottleneck.

(ii) Suppose $G(V, E)$ is an undirected connected graph such that for every cut $(S, V - S)$, there are at least two cut edges in $G$. Then, every vertex $v$ of $G$ belongs to some cycle. **(6 points)**

**Solution.** *True.*

Consider any vertex $v$ of $G$. Since $G$ is connected, degree of $v$ is at least one and hence there exists an edge $\{v, u\}$ in the graph. Since every cut $(S, V - S)$ has at least two cut edges, after removing the edge $\{v, u\}$, every cut of the graph $G - \{v, u\}$ has at least one cut edge and hence this graph is still connected; in particular, this means that there is a path $P$ from $v$ to $u$ in $G - \{v, u\}$. Combining this path $P$ with the edge $\{v, u\}$ gives us a cycle that contains vertex $v$.

**Problem 2.** You are given a two arrays $score[1:n]$ and $wait[1:n]$, each consisting of $n$ positive integers. You are playing a game with the following rules:

- For any integer $1 \leq i \leq n$, you are allowed to pick number $i$ and obtain $score[i]$ points;
- Whenever you pick a number $i$, you are no longer allowed to pick the previous $wait[i]$ numbers, i.e., any of the numbers $i-1, \ldots, i-wait[i]$.

Design an $O(n)$ time dynamic programming algorithm to compute the maximum number of points you can obtain in this game. It is sufficient to write your specification and the recursive formula for computing the solution (i.e., you do *not* need to write the final algorithm using either memoization or bottom-up dynamic programming).

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

**Solution.** For any integer $1 \leq i \leq n$, we define:

- $S(i)$: the maximum number of points we can get by picking a subset of numbers $\{1, \ldots, i\}$.

The final solution to the problem can then be obtained by returning $S(n)$.

(b) *Recursive solution for the formula:* **(5 points)**

**Solution.** We write the following recursive formula for $S(i)$ for any integer $1 \leq i \leq n$:

$$S(i) = \begin{cases} 0 & \text{if } i \leq 0 \\ \max\{S(i-1), S(i - wait[i] - 1) + score[i]\} & \text{otherwise} \end{cases}.$$

(c) *Proof of correctness of the recursive formula:* **(5 points)**

**Solution.** The base case when $i \leq 0$ is true because we cannot collect any points from numbers less than 1.

For larger values of $i$, to obtain the best solution from the set $\{1, \ldots, i\}$, we only have two options: (1) we either do not pick the number $i$ and instead pick the best solution from $\{1, \ldots, i-1\}$ which gives us the value $S(i-1)$, or (2) we pick number $i$ and hence we are no longer allowed to pick any of the numbers $i-1, \ldots, i-wait[i]$ and thus should pick the best solution from $\{1, \ldots, i-wait[i]-1\}$ for the remainder of the game; this gives us the value $S(i-wait[i]-1)+score[i]$. By picking the maximum of these two options in the formula, we obtain the best solution for $\{1, \ldots, i\}$, proving correctness.

We should note that since $S(i) = 0$ for all $i \leq 0$ (and not only $i = 0$), we do not need to worry when $i - wait[i] - 1 < 0$ as the value of $S$ on this entry is anyway 0.

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

**Solution.** We have $n$ subproblems and each one takes $O(1)$ time to compute, hence the total runtime of the dynamic programming algorithm obtained from this formula is $O(n)$.

**Problem 3.** You are given a directed graph $G(V, E)$ such that every *edge* is colored red, blue, or green. We say that a path $P_1$ is *lighter* than a path $P_2$ if one of the conditions below holds:

- $P_1$ has fewer number of red edges;
- $P_1, P_2$ have the same number of red edges and $P_1$ has fewer blue edges;
- $P_1, P_2$ have the same number of red and blue edges and $P_1$ has no more green edges than $P_2$.

Design an $O(n + m \log m)$ time algorithm that given $G$ and vertices $s, t$, finds one the lightest paths from $s$ to $t$ in $G$, namely, a path which is lighter than any other $s$-$t$ path. **(20 points)**

**Solution.** *Algorithm (reduction from the shortest path problem):*

1. Assign a weight of $(n + 1)^2$ to every red edge, $(n + 1)$ to every blue edge, and 1 to every green edge.
2. Run Dijsktra's shortest path algorithm on the graph $G$ starting from vertex $s$ with the given weights, and return the $s$-$t$ path found by the algorithm as the solution.

*Proof of Correctness:* We prove that for any two $s$-$t$ paths $P_1, P_2$:

$(i)$ If $P_1$ is lighter than $P_2$ in the original graph, then weight of $P_1$ in the new graph is smaller than or equal to weight of $P_2$.

Let $r_1, b_1, g_1$ denote the number of red, blue, and green edges in $P_1$. Weight of $P_1$ in the new graph is $r_1 \cdot (n+1)^2 + b_1 \cdot (n+1) + g_1$. Moreover, note that both $b_1 + g_1 < n$ as a path has at most $n-1$ edges. As such, weight of $P_1$ is smaller than $(r_1 + 1) \cdot (n+1)^2$ and also smaller than $r_1 \cdot (n+1)^2 + (b_1 + 1) \cdot (n+1)$. This ensures that if $P_1$ is lighter than $P_2$ then weight of $P_1$ would be smaller than or equal to weight of $P_2$ as well.

$(ii)$ If weight of $P_1$ is smaller than weight of $P_2$ in the new graph, then $P_1$ is lighter than $P_2$ in the original graph.

By the above part, weight of $P_1$ can only be smaller than weight of $P_2$ if either $P_1$ has fewer red edges (otherwise we violate the first equation above), or $P_1$ and $P_2$ has the same number of red edges and $P_1$ has fewer blue edges (otherwise we violate the second equation above), or $P_1, P_2$ has the same number of red and blue edges, and $P_1$ has at most as many green edges as $P_2$. But this is precisely the definition of $P_1$ being lighter than $P_2$.

The above implies that the shortest $s$-$t$ path in the new graph is precisely the same as the lightest path in the original graph, proving the correctness of the algorithm.

*Runtime analysis:* Constructing the weights on the edges takes $O(m)$ time by simply going over each edge once and running Dijkstra's algorithm takes $O(n+m \log m)$ time. Hence, the total runtime is $O(n+m \log m)$.

**Problem 4.** You are given an undirected *bipartite* graph $G$ where $V$ can be partitioned into $L \cup R$ and every edge in $G$ is between a vertex in $L$ and a vertex in $R$. For any integers $p, q \geq 1$, a *$(p,q)$-factor* in $G$ is any subset of edges $M \subseteq E$ such that no vertex in $L$ is shared in more than $p$ edges of $M$ and no vertex in $R$ is shared in more than $q$ edges of $M$.

Design an $O((m+n) \cdot n \cdot (p+q))$ time algorithm for outputting the size of the *largest* $(p,q)$-factor of any given bipartite graph.

**(20 points)**

*Hint:* This problem is quite similar to the maximum matching problem we studied in the lectures.

**Solution.** *Algorithm (reduction from the network flow problem):*

1. Create a network $G'(V', E')$ where $V' = L \cup R \cup \{s,t\}$. Connect $u \in L$ to $v \in R$ with an edge of capacity 1 whenever $\{u,v\}$ is an edge in the original graph. Moreover, connect $s$ to every vertex in $L$ with an edge of capacity $p$ and every vertex in $R$ to $t$ with an edge of capacity $q$.
2. Compute a maximum flow $f$ from $s$ to $t$ in the network $G'$ and return its value as the answer.

*Proof of Correctness:* We prove that the size of maximum flow in $G'$ is equal to the size of the largest $(p,q)$-factor in $G$. This is done by proving the following two assertions:

$(i)$ If the maximum flow in $G'$ is at least $k$, then there is a $(p,q)$-factor of size $k$.

Fix a maximum flow $f$ in $G'$. Let $M$ be the set of edges $\{u,v\} \in E$ with $u \in L$ and $v \in R$ such that $f(u,v) = 1$. We claim that $M$ is a $(p,q)$-factor of size $k$. Firstly, since capacity of the incoming edge from $s$ to any vertex $u \in L$ is exactly $p$, the flow incoming to $u$ can be at most $p$ and hence the flow going out of $u$ can be at most $p$ also: this means that the degree of $u$ in $M$ can be at most $p$. Similarly, using the fact that the capacity of the edge connecting each $v \in R$ to $t$ is $q$, the flow going out of $v$, and hence the flow coming into $v$ can be at most $q$ also and hence the degree of $v$ in $M$ can be at most $q$. This means that $M$ is a $(p,q)$-factor. Now since the value of flow $f$ is $k$ and all this flow needs to use the edges $(u,v)$ with $u \in L$ and $v \in R$ to go from $s$ to $t$, size of $M$ is $k$ also.

$(ii)$ If the size of largest $(p,q)$-factor of $G$ is $k$, then there is a flow of value of $k$ in $G'$.

Fix a maximum size $(p,q)$-factor $M$ in $G$. Define the flow $f$ as follows: (1) for any $\{u,v\} \in M$ (with $u \in L$ and $v \in R$), let $f(u,v) = 1$, (2) for any $u \in L$, let $f(s,u)$ be equal to the number of edges incident on $u$ in $M$, and for any $v \in R$, let $f(v,t)$ be equal to the number of edges incident on $v$ in $M$. This is a feasible flow as the flow entering each vertex is equal to the flow out from that vertex, and since $M$ is a $(p,q)$-factor, $f(s,u) \leq p$ for all $u \in L$ and $(v,t) \leq q$ for all $v \in R$, and thus capacity constraints are also satisfied. Moreover, the value of this flow is equal to the flow out of $s$ which is equal to the degree of vertices in $L$ inside $M$ which is equal to the number of edges, i.e., $k$.

This implies that the maximum flow in $G'$ is equal to the size of largest $(p,q)$-factor in $G$, concluding the proof.

*Runtime analysis:* Constructing the network takes $O(n+m)$ time and running Ford-Fulkerson algorithm for maximum flow, takes $O(m' \cdot F)$ time where $m'$ is the number of edges in $G'$ and $F$ is the value of maximum flow. Since $m' = m + 2n$ and $F \leq \max\{n \cdot p, n \cdot q\} \leq n \cdot (p+q)$, we obtain that the runtime of the algorithm is $O((n+m) \cdot n \cdot (p+q))$.

**Problem 5.** Prove that the following problems are NP-hard.

(a) **Two-Third 3-SAT Problem:** Given a 3-CNF formula $\Phi$ (in which size of each clause is *at most* 3), is there an assignment to the variables that satisfies at least 2/3 of the clauses? **(10 points)**

*Hint:* Use a reduction from the *3-SAT problem*. Recall that in the 3-SAT problem, we are given a 3-CNF formula $\Phi$ and the goal is to output whether there exist an assignment that satisfies *all* clauses.

**Solution.** We show that any instance $\Phi$ of 3-SAT problem can be solved in polynomial time if we are given a poly-time algorithm for two-third 3-SAT problem.

*Reduction (given an input $\Phi$ of 3-SAT problem):*

(1) Let $m$ denote the number of clauses in $\Phi$.

(2) We create a new 3-CNF formula $\Phi'$ by adding all variables and clauses of $\Phi$, as well as defining $m$ *new* variables $z_1, \ldots, z_m$ and adding clauses $(z_1), \ldots, (z_m)$ and $(\overline{z_1}), \ldots, (\overline{z_m})$ to $\Phi$, i.e.,

$$\Phi' = \Phi \wedge (z_1) \wedge \cdots (z_m) \wedge (\overline{z_1}) \wedge \cdots (\overline{z_m}).$$

(3) We run the algorithm for two-third 3-SAT on $\Phi'$ and output $\Phi$ is satisfiable if and only if that algorithm outputs $\Phi'$ has an assignment that satisfies 2/3 of clauses.

*Proof of Correctness:* We prove that $\Phi$ is satisfiable if and only if $\Phi'$ has an assignment that satisfies 2/3 of clauses.

(1) If $\Phi$ is satisfiable then $\Phi'$ has an assignment that satisfies 2/3 of clauses.
Let $x$ be a satisfying assignment of $\Phi$. Consider the assignment of $x$ to $\Phi$-part of $\Phi'$ and assigning True to all variables $z_1, \ldots, z_m$. Clearly $\Phi$ part and clauses $(z_1), \ldots, (z_m)$ are all satisfied in $\Phi'$ which constitute $2m$ clauses. Since the total number of clauses in $\Phi'$ is $3m$, this implies this assignment satisfies 2/3 of clauses of $\Phi'$, hence $\Phi'$ has such an assignment.

(2) If $\Phi'$ has an assignment that satisfies 2/3 of clauses then $\Phi$ is satisfiable.
Let $y$ be an assignment that satisfies at least 2/3 of clauses in $\Phi'$. Note that since we have both clauses $(z_i)$ and $(\overline{z_i})$, any assignment can satisfies exactly half the new clauses of $\Phi'$. Hence, to for this assignment to satisfies 2/3 of total clauses, it should also satisfies all clauses of $\Phi$ in $\Phi'$. This means that in this case $\Phi$ is satisfiable.

This implies the correctness of the reduction.

*Runtime analysis:* Constructing the new formula takes polynomial time in size of $\Phi$ as we are simply copying $\Phi$ and adding $m$ new clauses in $O(m)$ time. Thus any polynomial time algorithm for two-third 3-SAT problem implies a polynomial time for 3-SAT which is an NP-hard problem (and thus it having a poly-time algorithm means P=NP). As such a poly-time algorithm for two-third 3-SAT problem implies P = NP and so two-third 3-SAT is also NP-hard.

(b) **Negative-Weight Shortest Path Problem:** Given a graph $G(V,E)$, two vertices $s,t$ and *negative* weights on the edges, what is the weight of the shortest path from $s$ to $t$? **(10 points)**

*Hint:* Use a reduction from the *s-t Hamiltonian Path problem.* Recall that in the $s$-$t$ Hamiltonian Path problem, we 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.

**Solution.** We show that any instance $G,s,t$ of $s$-$t$ Hamiltonian path problem can be solved in polynomial time if we are given a poly-time algorithm for negative-weight shortest path problem.

*Reduction (given an input $G,s,t$ of s-t Hamiltonian path problem):*

(1) Assign a weight of $-1$ to every edge of $G$.

(2) Run the algorithm for the negative-weight shortest path problem on $G,s,t$ with the new weights.

(3) If the weight of the path found by the algorithm is $-(n-1)$, output there is an $s$-$t$ Hamiltonian path in $G$ and otherwise output no such path.

*Proof of Correctness:* We prove that $G$ has a Hamiltonian $s$-$t$ path if and only if the shortest path from $s$ to $t$ in $G$ has weight $-(n-1)$.

(1) If $G$ has a Hamiltonian $s$-$t$ path then the shortest path from $s$ to $t$ in $G$ has weight $-(n-1)$.
Let $P$ be a Hamiltonian path from $s$ to $t$ in $G$. This path has weight $-(n-1)$ under new weights as it passes through every vertex and thus has $n-1$ edge of weight $-1$. Moreover, any path in $G$ has at most $n-1$ edges and thus no path can have weight less than $-(n-1)$. Hence, the shortest $s$-$t$ path has weight $-(n-1)$ in this case.

(2) If the shortest path from $s$ to $t$ in $G$ has weight $-(n-1)$ then $G$ has a Hamiltonian $s$-$t$ path.
Let $P$ be the shortest $s$-$t$ path in $G$ under new weights. Since weight of $P$ is $-(n-1)$ it should have exactly $n-1$ edges and thus $n$ vertices. But then it means that $P$ is a Hamiltonian $s$-$t$ path in $G$.

This implies the correctness of the reduction.

*Runtime analysis:* Constructing the new weights takes polynomial time in the size of the graph. Thus any polynomial time algorithm for negative-weight shortest path problem implies a polynomial time for $s$-$t$ Hamiltonian path problem which is an NP-hard problem (and thus it having a poly-time algorithm means P=NP). As such a poly-time algorithm for the negative-weight shortest path problem implies P = NP and so negative-weight shortest path is also NP-hard.

**Problem 6.** [**Extra credit**] You are given a set $C$ of $p$ courses that a student has taken. You are also given $q$ subsets $S_1, \ldots, S_q$ of $C$, and $q$ integers $r_1, \ldots, r_q$. In order to graduate, the student has to meet the following $q$ requirements:

- For every $1 \leq i \leq q$, the student needs to have taken at least $r_i$ courses from the subset $S_i$.

The goal is to determine whether or not the courses taken by the student can be used to satisfy all $q$ requirements. The tricky part is that any given course *cannot* be used towards satisfying multiple requirements.

For example, if one requirement states that the student should have taken 2 courses from $S_1 = \{A, B, C\}$ and another requirement asks for taking 2 courses from $S_2 = \{C, D, E\}$, then a student who had taken just $\{B, C, D\}$ *cannot* graduate.

Design a polynomial time algorithm that given a set $C$ of $p$ courses a student has taken, $q$ subsets $S_1, \ldots, S_q$ of $C$, and $q$ integers $r_1, \ldots, r_q$, determines whether or not the student can graduate.

*Hint:* Reduce this problem to a network flow problem on a carefully designed network.

**Solution.** *Algorithm (reduction to network flow problem):*

1. Create a network $G(V, E)$, where $V = \{s, t\} \cup V_C \cup V_S$. For each course $c_i \in C$, add a vertex $u_i \in V_C$, 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 course $c_i \in S_j$. 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 student can graduate if and only if the max flow value is $\sum_{j=1}^{q} r_j$.

*Proof of Correctness:* We prove that the maximum flow in $G$ is equal to $\sum_{j=1}^{q} r_j$ if and only if the student can graduate.

1. Suppose the student can graduate; create the flow $f$ as follows. Send 1 unit of flow from each vertex $u_i \in V_C$ to vertex $v_j \in V_S$ if the student uses the course $c_i$ 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 student does not over satisfy a requirement). Since the student can graduate 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 equal to $\sum_{j=1}^{q} r_j$.
2. Suppose now the maximum flow is equal to $\sum_{j=1}^{q} r_j$. For each vertex $v_j \in V_S$, we have that the incoming flow to this vertex is equal to $r_j$. Since capacity of every incoming edge of $v_j$ is 1, there must be $r_j$ vertices in $V_C$ that provide this flow. Moreover, since these vertices can only transfer 1 unit of flow, it means that the outgoing flow of all these vertices is going only to $v_j$. Hence, we can use the courses corresponding to these vertices to satisfy the requirement for $S_j$, while ensuring that no course is being used towards satisfying multiple requirements. Consequently, every requirement can be satisfied.

*Runtime Analysis:* By running Ford-Fulkerson algorithm for max-flow, the running of time of this algorithm is $O(m \cdot F)$ where $m = O(pq)$ and $F = O(\sum_{j=1}^{q} r_j) = O(p)$. Hence, this algorithm runs in polynomial time.

## Extra Workspace

## Extra Workspace

## Extra Workspace