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

Practice Final Exam Solutions

Problem 1.

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

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

Solution. *False.*
$f(n) = n^{\log n} = (2^{\log n})^{\log n} = 2^{\log^2 n}$.
Since $\log^2 n = o(\sqrt{n})$, $2^{\log^2 n} = o(2^{\sqrt{n}})$ also. So $f(n) = o(g(n))$ and hence $f(n) \neq \Omega(g(n))$.

(ii) If $T(n) = 2T(n/2) + O(n^2)$, then $T(n) = O(n^2)$. **(2.5 points)**

Solution. *True.*
Consider the recursion tree. At the root, we have $C \cdot n^2$ time. In the next level, we have $2 \cdot C \cdot (n/2)^2 = C \cdot n^2/2$. In the level after that, we have $4 \cdot C \cdot (n/4)^2 = C \cdot n^2/4$. In general, at level i, we have $2^{i-1} \cdot C \cdot (n/2^{i-1})^2 = C \cdot n^2/2^{i-1}$ time.
So the total runtime is upper bounded by $\sum_{i=1}^{\infty} C \cdot n^2/2^{i-1} \leq 2 \cdot C \cdot n^2$ as the series $\sum_{i=1}^{\infty} 1/2^{i-1}$ converges to 2.
This means $T(n) = O(n^2)$.

(iii) If a problem in NP can be solved in polynomial time, then all problems in NP can be solved in polynomial time. **(2.5 points)**

Solution. *False.*
This statement is true only about NP-complete problems not all problems in NP. In particular, P $\subseteq$ NP, so there are already many problems in NP that can be solved in polynomial time.

(iv) 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.

(b) Prove the following statement: 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., there is at least one edge that carries the same amount of flow as its capacity. **(10 points)**

Solution. Suppose towards a contradiction that this is not true. Let P be a path from s to t in the network G. Let e^* be the edge on this path with the minimum value of $c_{e^*} - f(e^*)$. Since $f(e) \neq c_e$ for all edges $e \in E$ in the network by our assumption, we know that $c_{e^*} - f(e^*)$ is strictly larger than zero. Consider the new flow function f' which is obtained from f by adding another flow of value $c_{e^*} - f(e^*)$ to all edges of the path P.

Firstly, this flow still satisfies the capacity constraint: because on every edges out of P, $f'(e) = f(e)$ on those edges (and so satisfy the constraint because f did), and for any edge in P,

$$f'(e) = f(e) + c_{e^*} - f(e^*) \leq f(e) + c_e - f(e) = c_e$$

because $c_{e^*} - f(e^*)$ was the minimum value by definition.

Secondly, this flow also still satisfies the preservation of flow, because the new flow-in and flow-out of vertices is equal as we added a flow along a path from s to t.

Finally, since f' is a new valid flow with value strictly more than f, we get a contradiction that f was a maximum flow. Thus, our original assumption was false, proving the statement.

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). **(20 points)**

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

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 and its proof of correctness:*

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

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

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.

(c) *Runtime analysis:*

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)**

Remember to *separately* write your algorithm (**7.5 points**), the proof of correctness (**7.5 points**), and runtime analysis (**5 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. As an example, notice that a *matching* introduced in the class is a $(1,1)$-factor.

Design an $O((m+n) \cdot n \cdot (p+q))$ time algorithm that given a graph G and two integers p and q, outputs the size of the *largest* (p,q)-factor of G. **(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 (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 $F \leq \min\{n \cdot p, n \cdot q\} \leq n \cdot (p+q)$, 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. For each problem, you are only allowed to use a reduction from the problem specified.

(a) **Two-Third 3-SAT Problem:** Given a 3-CNF formula Φ (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)**

For this problem, use a reduction from the *3-SAT problem*. Recall that in the 3-SAT problem, we are given a 3-CNF formula Φ and the goal is to output whether there exist an assignment that satisfies *all* clauses.

Solution. We show that any instance Φ 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 Φ of 3-SAT problem):

(1) Let m denote the number of clauses in Φ.

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

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

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

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

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

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

This implies the correctness of the reduction.

Runtime analysis: Constructing the new formula takes polynomial time in size of Φ as we are simply copying Φ 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) **Maximum Pyramid Problem:** Given an undirected graph $G = (V, E)$, what is the size of the largest **pyramid** in G? A pyramid is a set of vertices $v_1, \ldots, v_k$ plus another vertex u such that there are no edges between $v_1, \ldots, v_k$ in G and every vertex v_i is neighbor to u. **(10 points)**

For this problem, use a reduction from the *maximum independent set problem.* Recall that in the maximum independent set problem, you are given an undirected graph $G = (V, E)$ and the goal is to find the largest independent set in G, namely, the largest set of vertices with no edges between them.

Solution. *Reduction (from Maximum Independent Set):*

(1) Given an instance $G = (V, E)$ of the maximum independent set problem, we create an instance G' of the maximum pyramid problem as follows.

(2) Add a new vertex z to G and connect it to all vertices of G to obtain G'.

(3) We run the algorithm for the maximum pyramid problem on G' to obtain its answer on G', denoted by k; we then output $k - 1$ as the answer to maximum independent set problem in G.

Proof of correctness. We show that for any integer t, G' has a pyramid of size t if and only if G' has an independent set of size $t - 1$. This then implies that the maximum pyramid size in G' is always exactly one larger than the maximum independent set size in G, thus proving the correctness of the reduction.

(i) If G has a maximum independent set of size t, then G' has a pyramid of size $t + 1$:
Let $v_1, \ldots, v_t$ be the independent set in G. Consider the pyramid consisting of vertices $v_1, \ldots, v_t$ in G' plus the vertex z. There are no edges between $v_1, \ldots, v_t$ in G' and they are all neighbor to z; so this is indeed a valid pyramid and its size is $t + 1$ as desired.

(ii) If G' has a pyramid of size $t + 1$, then G has a maximum independent set of size t:
Let $v_1, \ldots, v_t$ and u be the pyramid in G'. Note that size z has an edge to every vertex of the graph, none of v_i's can be equal to z and thus those vertices are all originally in G. Since there are no edges between $v_1, \ldots, v_t$, they form an independent set of size t in G as desired. (We emphasize that the pyramid found here may or may not have $u = z$ unlike part 1 where the top vertex of the pyramid was always z. But this is not important for the correctness of the reduction).

Runtime analysis: This reduction can be implemented in $O(n + m)$ time and hence a poly-time algorithm for the Maximum Pyramid Problem implies a poly-time algorithm for Maximum Independent Set which in turn implies P = NP (by definition of Maximum Independent Set being NP-hard). Thus a poly-time algorithm for Maximum Pyramid also implies P = NP, making this problem 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.

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.