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

Homework #5

Deadline: Monday, May 03, 11:59 PM

Name: Yinfeng Cong *Extension: Yes*

Homework Policy

- If you leave a question completely blank, you will receive 25% of the grade for that question. This however does not apply to the extra credit questions.
- You are allowed to discuss the homework problems with other students in the class. **But you must write your solutions independently.** You may also consult all the materials used in this course (video recordings, notes, textbook, etc.) while writing your solution, but no other resources are allowed.
- Do not forget to write down your name and whether or not you are using one of your two extensions. Submit your homework on Canvas.
- Unless specified otherwise, you may use any algorithm covered in class as a "black box" – for example you can simply write "use Ford-Fulkerson's algorithm to find a maximum flow of the input network in $O(m \cdot F)$ time". You are **strongly encouraged to use graph reductions** instead of designing an algorithm from scratch whenever possible (even when the question does not ask you to do so explicitly).
- Remember to always **prove the correctness** of your algorithms and **analyze their running time** (or any other efficiency measure asked in the question).
- The "Challenge yourself" and "Fun with algorithms" are both extra credit. These problems are significantly more challenging than the standard problems you see in this course (including lectures, homeworks, and exams). As a general rule, only attempt to solve these problems if you enjoy them.

Problem 1. You are given an $n \times n$ matrix and a set of k cells $(i_1, j_1), \ldots, (i_k, j_k)$ on this matrix. We say that this set of cells can **escape** the matrix if: (1) we can find a path from each cell to any arbitrary *boundary cell* of the matrix (a path is a sequence of *neighboring* cells, namely, top, bottom, left, and right), (2) these paths are all *disjoint*, namely, no cell is used in more than one of these paths.

Design an $O(n^3)$ time algorithm that given the matrix and the input cells, determines whether these cells can escape the matrix (together) or not. **(25 points)**

Solution. Solution to Problem 1 goes here.
First, we need to make sure that there are more white corner cells compared to the green cells.
If the amount of green cells are much more than corner cells, then it can not escape.
Reduction:
First we build edges among the cells and assign their capicity as 1 to each of these edges, and we turn the original cell v to two vertices v_{in} and v_{out}, and turn any edge (u,v) in the original graph to an edge (u_{out}, v_{in}); then also add the edge (v_{in}, v_{out} with capicity one to the graph.
Then find the path that make every green cells can be escaped to the corner cell, and there is no cell that can be visited twice.
(We can do this by marking the cells as True/False, if it has been visited, then mark it as true, if it is not, mark it as false)
This question is similar to the vertex-disjoint path problem mentioned in the lecture note, but this time for the s-t flow problem. All the green cells would be thought as start point s, and the white corner cells would be thought as the end point t.

Proof of correctness:
Firstly, any s-t flow of value k in this network corresponds to a collection of vertex-disjoint paths. This is because each vertex only carry one unit of flow in the network and so any of the k units of flow starting from s and ending at t should follow a path, vertex-disjoint from the rest, from s to t. This proves that the solution is feasible.
Secondly, any collection of vertex-disjoint paths in the graph implies a flow of value l in the network, simply route one unit of flow across each path and since they are vertex-disjoint, no vertex of the network carry more than 1 unit of flow which is its capacity. This means that there is a one-to-one mapping between flows and vertex-disjoint paths and thus by returning if a vertex is visted more than once. We get that if the graph can be escaped. This proves the solution is optimal.
Runtime analysis:
Creating the network takes O(n+n) time and run the Ford-Fulkerson takes O(m*F), here we do not need to calculate the maximum value of flow from s to t, so we can ignore F or set is as 0. Here, in the s-t flow, we have less than 4n kinds of s to choose from (the green cells, and the amount of it is smaller than 4n as mentioned above), and we have about 4n kinds of t to choose from (the amount of the corners is at most 4n). So the total runtime is O(n+n + n*4n*4n) = O(n^3)

Problem 2. 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.

(25 points)

Solution. Solution to Problem 2 goes here.
Reducion:
We have known a bipartite graph G = (V,E) in the maximum edge problem (p,q). Construct the following flow network G' = (V',E') with source s and sink t and capicity c_e on each edge e belongs to E':
We set V'=V and s, t for two new vertices s and t. We add a directed edge (u, v) to E' whenever a vertex u belongs to L of G has an edge to v belongs to R of G.
Moreover, for any vertex u belongs to L we add an edge (s,u) to E' and for any vertex v belongs to R, we add an edge (v,t)
We set the capacity of any edge e= (s, u) for u belongs to L to be p. Similarly, we set the capacity of any edge e'= (v, t) for v belongs to R to q. We set the capacity of all remaining edges to be infinity.
We then find the maximum flow f from s to t in the network G'. We create a sum S in the original graph G by picking any edgeu, v for u belongs to L and v belongs to R to be added to M if and only if f(u, v) = 1 in this flow
Proof of correctness:
S is the sum of edges based on the condition of p and q:
Different from matching in lecture note, we do not need to pick a matching this time. So the capacity from s to u and v to t is p this time. (because there is a request that only p edges can go from one vertice from u). So we change the capacity from 1 to p this time, so it won't be overloaded. Also, only q edges can be got from for one vertice v. So we change the capacity for v-t to q this time.
S is a maximum sum of edges based on the condition of p and q:
We prove that any flow f in G' corresponds to a sum of edges S in G with size of S equal to the value of flow f and vice versa. This then implies that maximum flow corresponds to a maximum sum of edges and thus S is a maximum sum of edges. By the above part, we know that for any flow f in G' there is a sum S of the same size in G. We now prove the other direction.
Fix any sum S in G and consider the flow f where for any edge u,v in S, we add a flow path f(s,u) = p, f(u,v)

= 1, and f(v,t) = q to our flow f. This definition of f clearly satisfies the preservation flow. Here, the vertice u appears p times and v appears q times, so f also satisfies the capacity constraints and hence is indeed a flow. The value of this flow is equal to the size of S by definition.
Runtime analysis:
We can createthe network above in O(m+n) time by traversing the edges of graph G directly.The runtime of reduction we run Ford-Fulkerson algorithm again and find its maximum flow in O([m+n]*F) where F is the value of maximum flow. F is not just the simple n here (this is not a matching question) In the matching question there is only one path from u to v, but now there can be p ways from u and q ways to v. So here F is n*(p+q). So the runtime in total is O([m+n] * n * [p+q])

Problem 3. Given an undirected graph $G = (V, E)$ and an integer $k \geq 2$, a k-coloring of G is an assignment of k colors to the vertices of V such that no edge in E has the same color on both its endpoints.

(a) Design a poly-time *algorithm for solving* the decision version of the 2-coloring problem: Given a graph $G = (V, E)$ output *Yes* if G has a 2-coloring and *No* if it does not. **(15 points)**

Solution. Solution to Problem 3, part (a) goes here.
Algorithm:
To get if the graph is 2-coloring or not, we need to know if the graph has a cycle first, and we use the method of DFS to solve this problem here.
We run DFS on the graph, and set all the vertices to be a start point s for one time. Create a array Count[] of size n to be the sum of the vertices in a cyle
For i from 1 to n, Start from s, we run DFS on it, and mark all the passed vertices as true, until we find that we come back to the start point s again. if it does not come back and all the vertices are marked true, we delete the mark of the newest mark and return to the last vertice, do this again and again until we find that we can get to the start point s. We record the number of vertices into the Count[i] and then we change a start point and do DFS on it again until we have tried all the vertices.
If we do not get a cycle in the graph, we can output Yes.
If we get a cycle or many cycles, we need to check the amount of vertices in them to get the answer. If Count = odd, then output No. Else if Count = even, output Yes.
Proof of correctness:
We have known the correctness of DFS, so we just need to prove this algorithm is correct.
If there is no cycle, then can put the two colors and let them not be next to each other, then the 2-coloring will always work.
If there is a cycle, and the Count of it is even, we can put the vertices not next to each other again, and we can do this because the number is even. (the remainder of 2n/2 = 0, but the remainder of 1/2 and (2n-1)/2 are all 1, so they can be different from each other.
However, if there is a cycle, and the Count of it is odd, we can not put all the vertices not next to each other because the number if odd. (the remainder of (2n+1) / 2 = 1, and the remainder of 2n/2 = 0, the remainder of 1/2 = 1. So we can not make they two both unequal to the 2n+1, which means that one of them must have the same color with the 2n+1 vertice)
Runtime analysis:
Runtime of build a array of Count[] is n, and the runtime of DFS is O(m+n), and we run DFS for n times (because we run it on each vertice for one time), so the total runtime would be O(n + n* (m+n)) = O(n*(m+n)), and it is a poly-time algorithm.

(b) Design a poly-time *verifier* for the decision version of the k-coloring problem for any $k \geq 2$: Given a graph $G = (V, E)$ and k as input, output *Yes* if G has a k-coloring and *No* if it does not. Remember to specify exactly what type of a proof you need for your verifier. **(10 points)**

Solution. Solution to Problem 3, part (b) goes here.
Verifier:
If the graph is k-colorable (YES), we can simply use any k-coloring of the graph as a proof. So the input to our verifier is the input of graph G and a supposed k-coloring of G. The verifier then goes over the edges one by one to ensure that no edge is monochromatic.
If we can not prove the part above, then it is NO, and it is not k-colorable.
Runtime analysis:
The graph has m edges, so the runtime of of the verfier is O(m), so it is a poly-time verfier.
But if we want to solve this question, I do not have a clear mind about the algorithm of this problem.

Problem 4. Prove that each of the following problems is NP-hard and for each problem determine whether it is also NP-complete or not.

(a) **One-Fourth-Path Problem:** Given an undirected graph $G = (V, E)$, does G contain a path that passes through *at least one forth* of the vertices in G? **(8 points)**

Solution. Solution to Problem 4, part (a) goes here.
One-forth path problem is NP-hard. Nevertheless, we prove that One-forth path problem is NP-hard, meaning that a poly-time algorithm for One-forth path problem implies P = NP. We do this by a reduction from Undirected s-t Hamiltonian Path.
Reduction:
Suppose we have an algorithm A for One-forth path problem. Then, given any instance G(V,E) of the Undirected s-t Hamiltonian path, we simply run A on G to find a path that covers the one-forth vertices of the graph from s, and then we delete the vertices we have got by the algorithm A, and run the algorithm on G again, and then delete the new path we find again (repeat this step until we delete all the vertices in the graph. And this shows the answer to the Undirected s-t Hamiltonian path.
Proof of correctness:
To prove the ccorrectness of this problem, it is similar to other reduction problems.
(i) The passed vertices of four parts of One-forth path —- the passed vertices in Undirected s-t Hamiltonian path: Let G be an undirected graph (V,E) that can be divided to four parts of One-forth paths. Each one of these four parts is an undirected Hamiltonian path, and one of them starts from a start point s, one of them ends at t. After combining the four parts, we get an undirected s-t Hamiltonian path, and we have known that it passed all the vertices. This means we get an undirected s-t Hamiltonian path.
(ii) The passed vertices in Undirected s-t Hamiltonian path —- The passed vertices of four parts of One-forth path: Let G be an undireted graph (V,E) that it is an Undirected s-t Hamiltonian path. We can devide this graph into four parts, and each of them passes one-forth of the total graph. As a result, this means that we get four One-forth paths, and each of them obeys the contraints metioned in the question.
Runtime analysis:
We have obtained a poly-time algorithm for Undirected s-t Hamiltonian path, and so we can obtain a poly-time algorithm for One-forth path as well. (because Undirected s-t Hamiltonian path is just four one-forth paths)
As a result, this question is a NP-hard problem.
One-forth path is a transformation of Undirected s-t Hamiltonian path, so it is an NP problem.(We know that Undirected s-t Hamiltonian Path is a NP problem) In conclusion, this problem is an NP-Complete problem.

(b) **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? **(8 points)**

Solution. Solution to Problem 4, part (b) goes here.
Two-Third 3-SAT Problem is NP-hard. Nevertheless, we prove that Two-Third 3-SAT Problem is NP-hard, meaning that a poly-time algorithm for Two-Third 3-SAT Problem implies P = NP. We do this by a reduction from 3-SAT Problem.
Reduction:
Suppose we have an algorithm A for Two-Third 3-SAT Problem. Then, given any 3-CNF formula Φ, we simply run A on Φ to find a Two-Third 3-SAT (Run this algorithm on the first 2/3 parts of the Φ) that satisfies 2/3 of the total clauses. Then, we run A again on Φ to find a Two-Third 3-SAT (Run this algorithm on the last 2/3 parts this time) that satisfies 2/3 of the total clauses. Then, we delete the repeat part of the two times of algorithm, and combine them together. We get the formula Φ that satisfies all the clauses.
Proof of Correctness:
To prove the correctness of this problem, it is similar to other reduction problems.
(i) The combination of the two parts of Two-Third 3-SAT problem —- the clauses of 3-SAT problem: Let Φ be a 3-CNF formula, and divide it into two parts: The first 2/3 of it and the last 2/3 of it. Each one of these two parts is a 3-SAT problem itself. After deleting the repeated part and combine them, we get a compelete 3-SAT problem, and we know that all the clauses in it satisfies. This means that we get a 3-SAT problem.
(ii) The clauses of 3-SAT problem —- The combination of the two parts of Two-Third 3-SAT problem: Let Φ be a 3-CNF formula, and it is a 3-SAT. We can divide this 3-SAT into two parts, and each of them satisfies 2/3 of the total clauses, and the combination of them includes all the clauses inside the 3-SAT. This means that we get two Two-Third 3-SAT problem.
Runtime analysis:
We have obtained a poly-time algorithm for 3-SAT problem, and so we can obtain a poly-time algorithm for Two-Third 3-SAT problem. (because 3-SAT Problem is just 2 Two-Third 3-SAT problems)
As a result, this problem is a NP-hard problem.
Two-Third 3-SAT problem is a transformation of 3-SAT problem, so it is an NP problem. (We know that 3-SAT is a NP problem)
So this problem is an NP-Complete problem.

(c) **Negative-Weight Shortest Path Problem:** Given an undirected 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? **(9 points)**

Solution. Solution to Problem 4, part (c) goes here.
Negative-Weight Shortest Path Problem is not a decision problem and hence cannot be in NP.
So this problem also cannot be an NP-complete problem.
Negative-Weight Shortest Path Problem is NP-hard. Nevertheless, we prove that Negative-Weight Shortest Path Problem, meaning that a poly-time algorithm for Negative-Weight Shortest Path Problem implies P = NP. We do this by a reduction from Undirected s-t Hamiltonian Path problem.
Reduction:
Suppose we have an algorithm A for Negative-weight shorest path problem. We use the decision version of TSP, which is 'is there a TSP of cost at most k'.
We reduce Hamiltonian Path to TSP. Given G(V,E) with —V— = n we define c(e) = -1 for all e belongs to E. Then we add edges E' to G to make G a complete graph and assign c(e) = -2 for all e belongs to E'. We do this in polynomial time. Now given this cost, if the answer to is there is a TSP of cost at most n is "yes", then we know there is a path that visits all nodes. The edges it uses are from E, hence we can say for sure that there is Hamilitonian path in G.
Proof of correctness:
(i) The shortest path of at most k —- An undirected s-t Hamiltonian path problem: We have known that we have a shortest path of at most k, and we know that we have passed all the vertices by this

path of k from s to t. This means that we get a Undirected s-t Hamiltonian Path.
(ii)An undirected s-t Hamiltonian path problem —- The shortest path of at most k: In another way, we have got to know that We have a undirected s-t Hamiltonian path problem with negative weights on it, and the shortest path of it from s to t would be the answer. This means that we get a Negative-Weight shortest path problem.
Runtime analysis:
If we have a poly-time algorithm for undirected s-t Hamiltonian path problem, we will obtain a poly-time algorithm for Negative-Weight shortest path problem.
This concludes the proof of NP-hardness of Negative-Weight shortest path problem since we proved a poly-tome algorithm for Negative-Weight shortest path problem implies a poly-time algorithm for an NP-hard problem, namely, undirected s-t Hamiltonian path problem.

You may assume the following problems are NP-hard for your reductions:

- **Undirected s-t Hamiltonian Path:** Given an undirected graph $G = (V, E)$ and two vertices $s, t \in V$, is there a Hamiltonian path from s to t in G? (A Hamiltonian path is a path that passes every vertex).
- **3-SAT Problem:** Given a 3-CNF formula Φ (where each clause as *at most* 3 variables), is there an assignment to Φ that makes it true?

Fun with Algorithms. You are given a puzzle consists of an $m \times n$ grid of squares, where each square can be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors.

It is easy to see that for some initial configurations of stones, reaching this goal is impossible. We define the Puzzle problem as follows. Given an initial configuration of red and blue stones on an $m \times n$ grid of squares, determine whether or not the puzzle instance has a feasible solution.

Prove that the Puzzle problem is NP-complete. **(+10 points)**

Consider solving **at most one** of the following two challenge yourself problems.

Challenge Yourself (I). The goal of this question is to give a simple proof that there are decision problems that admit *no* algorithm at all (independent of the runtime of the algorithm).

Define Σ^+ as the set of all *binary* strings, i.e., $\Sigma^+ = \{0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$. Observe that any decision problem Π can be identified by a function $f_\Pi : \Sigma^+ \to \{0, 1\}$. Moreover, observe that any algorithm can be identified with a binary string in Σ^+. Use this to argue that "number" of algorithms is "much smaller" than "number" of decision problems and hence there should be some decision problems that cannot be solved by any algorithm.

Hint: Note that in the above argument you have to be careful when comparing "number" of algorithms and decision problems: after all, they are both infinity! Use the fact that *cardinality* of the set of real numbers $\mathbb{R}$ is larger than the cardinality of integer numbers $\mathbb{N}$ (if you have never seen the notion of cardinality of an infinite set before, you may want to skip this problem). **(+10 points)**

Challenge Yourself (II). Recall that in the class, we focused on *decision* problems when defining NP. Solving a decision problem simply tells us whether a solution to our problem exists or not but it does not

provide that solution when it exists. Concretely, let us consider the 3-SAT problem on an input formula Φ. Solving 3-SAT on Φ would tell us whether Φ is satisfiable or not but will not give us a satisfying assignment when Φ is satisfiable. What if our goal is to actually find the satisfying formula when one exists? This is called a *search* problem.

It is easy to see that a search problem can only be "harder" than its decision variant, or in other words, if we have an algorithm for the search problem we will obtain an algorithm for the decision problem as well. Interestingly, the converse of this is also true for all NP problems and we will prove this in the context of the 3-SAT problem in this problem. In particular, we reduce the 3-SAT-SEARCH problem (the problem of finding a satisfying assignment to a 3-CNF formula) to the 3-SAT (decision) problem (the problem of deciding whether a 3-CNF formula has a satisfying assignment or not).

Suppos e you are given, as a black-box, an algorithm A for solving 3-SAT (decision) problem that runs in polynomial time. Use A to design a poly-time algorithm that given a 3-CNF formula Φ, either outputs Φ is not satisfiable or *finds* an assignment x such that $\Phi(x) = True$. **(+10 points)**