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

Homework #5

Deadline: Monday, May 03, 11:59 PM

Name: Ryan Coslove *Extension: No*

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.

We do not want paths from a highlighted cell to intersect with another path. For the purpose of this problem, we can assume the graph/matrix has intersections as nodes and their connections as edges. We can assume all highlighted cells are at the corners of the intesections.

We will model using max flow. The edges will hold a capacity value of 1. Source node has edges to each highlighted cell/node with capacity 1. Edges from boundary nodes to the sink will hold capacity 1. Set an array $C[1:n]$ that will contain all existing paths.

We have a solution if and only if the max flow equals the number of highlighted cells.

- Can assume flow is $\{0, 1\}$ on each edge by Ford-Fulkerson
- Flow on each edge (s, u) gives a path from u to the boundary

  - flow into u on (s, u) leaves on some edge (u, v)
  - flow into v on (u, v) leaves on some edge (v, w)
  - can only stop with an edge (z, t) and z us a boundary node by construction
- two paths using the same edge would violate edge capacity

Because two paths could use the same intersection at its current state, we will implement node capacity constraints.

So consider any node:

- split it into two parts
- one part for incoming edges and one for outgoing edges
- add edge between them with capacity for the node C_n

All flow through the node now goes through this internal edge. That allows us to limit the total flow using the node because node balance constraint is preserved.

Once a successful path is created we are to store is in the array $C[1 : n]$. If a path is created that is duplicated/already exists in C will we remove the path we are checking.

So we have a solution, can the cells escape the matrix or not, if and only if the max flow equals the number of highlighted cells/nodes.

If this statement is true and contains a solution, we output true or "yes". Otherwise we output false or "no".

Proof of Correctness

We already proved the paths must be edge-disjoint and the node capacity means paths must be node-disjoint as well, so we should never output a solution to be true that contains overlapping paths. We only output yes if and only if the max flow equals the number of highlighted cells. AKA every solution corresponds to a flow of the number of highlighted cells. Also every 0/1 flow of the number of highlighted cells encodes escape paths for all highlighted cells.

Runtime

We know that Ford Fulkerson runs in $O(nm)$ time, or in this case $O(n * n)$ time. Running the for loop to check for duplicate paths in array C will take $O(n)$ time. Together, we have a total runtime of $O(n^3)$.

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.

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.

Let's use BFS to check if the given graph is Bipartite or not. Knowing if the graph is biparitite we can know if the graph will be 2-color based on its edges and connections.

1. Assign RED color to the source vertex (putting into set U)
2. Color all the neighbors with BLUE color (putting into set V).
3. Color all neighbor's neighbor with RED color (putting into set U).
4. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where $m = 2$.
5. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

If we find a neighbor with the same color as the current vertex, output NO. If conditions are satsified and we find a 2-coloring graph, output YES.

Proof of Correctness

In the above, we start with source 0 and assume that all vertices are visited from it. In the event the current vertex has the same color as one of its neighbors, we know the graph is not 2-color or bipartite. We also output YES or NO appropriately so it answers the polytime algorithm decision aspect for this problem.

Runtime

Time Complexity of the above approach is same as that Breadth First Search. In above implementation is $O(V^2)$ where V is number of vertices. If graph is represented using adjacency list, then the complexity becomes $O(V + E)$. If Graph is represented using Adjacency List, time complexity will be $O(V + E)$. Works for connected as well as disconnected graph.

(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.

We can use part (a) as the algorithm that finds a 2-coloring graph or not to output yes or no. We can use this is a verifier:

If the graph is 2-colorable, we can simply use any 2-coloring of the graph as a proof. So the input to our verifier is the input graph G and a supposed 2-coloring of G (here G is the input x and the 2-coloring is the proof y). The verifier then goes over the edges of G one by one to ensure that no edge is monochromatic.

If the verifier returns that x = y, proving that no edge is monochromatic, then we return YES as G is k-coloring (where $k \geq 2$). Otherwise, return NO.

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.

Our Problem I is finding a path in G that passes through at least one fourth of the vertices. We can prove this is NP-Hard through the implication that if Problem II has a poly-time algorithm we can obtain a poly-time algorithm for all problems in NP. We will wanna use the Hamiltonian Cycle Problem as our problem II. Through this problem we can find a path or cycle that goes through evey vertex in G and we already know this problem to be NP-Hard. So in the event our Problem II outputs a path n that reaches all vertices, we can do $n/4$ and find the value/path that is (roughly) equal to $1/4$ of all vertices in the graph. This gives us at least 1 path in G that is $\geq 1/4$, which is the answer we sought for a *yes*, and because Problem II was NP-Hard and is verified to be in poly-time we can infer that Problem I was NP-Hard and runs in poly-time.

This is also a deicision problem, "does G contain a path...". We know it is NP-Hard and fits within the NP class. Therefore it is NP-Complete.

(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.

Our Problem I is finding an assignment to the variables that satisfies at least $2/3$ of the clauses. We can prove this is NP-Hard through the implication that if Problem II has a poly-time algorithm we can obtain a poly-time algorithm for all problems in NP. We will wanna use the 3-SAT Problem as our Problem II. Through this problem we can find that finds an a satisfying assignment y such that $\phi(y) = 1$ or not. This is proven in lecture to work and be NP-Hard. So in the event our Problem II outputs a satisfied ϕ we can determine that Problem I should produce a satisfied ϕ where at least $2/3$ of the clauses are satisfied. This gives us a satisfied ϕ which is the answer we sought for a *yes*, and because Problem II was NP-Hard and is verified to be in poly-time we can infer that Problem I was NP-Hard and runs in poly-time.

This is also a deicision problem, "is there an assignment...". We know it is NP-Hard and fits within the NP class. Therefore it is NP-Complete.

(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.

Our Problem I is finding the weight of the shortest path from s to t in G. We can prove this is NP-Hard through the implication that if Problem II has a poly-time algorithm we can obtain a poly-time algorithm for all problems in NP. We will wanna use the Hamiltonian Path Problem as our problem II. Through this problem we can find a path or cycle that goes through evey vertex in G and we already know this problem to be NP-Hard. Let's add a weight of -1 to all edges. If we find a path from s to t

that has a weight of $n-1$, then we know it includes all nodes. Otherwise, there is no path because we would have found it given the negative weights. So in the event our Problem II outputs a path with weight $n-1$, we can find the minimum weight path in G that is the shortest path, which is the answer we sought for. Because Problem II was NP-Hard and is verified to be in poly-time we can infer that Problem I was NP-Hard and runs in poly-time.

This is NOT a decision problem as it asks for a weight as the output, so it is NOT NP-Complete.

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).

Suppose 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)**