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

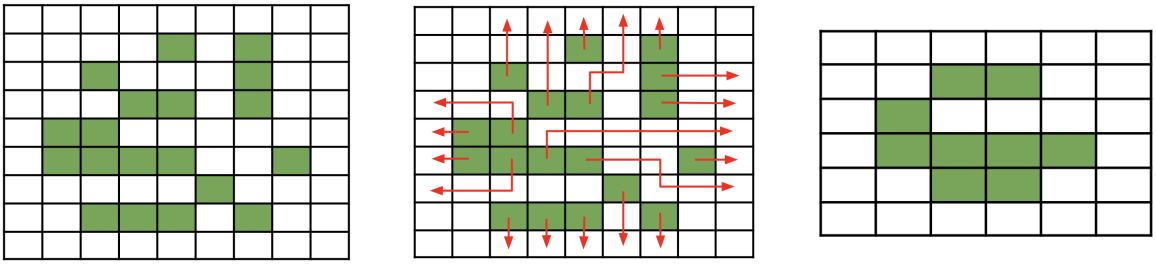
Homework #5

Deadline: Monday, May 03, 11:59 PM

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. See Figure 1:



(a) An "escapable" input (b) A way of escaping the matrix (c) A "non-escapable" input

Figure 1: The green cells correspond to the input k cells of the matrix.

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[1]. **(25 points)**

[1]This (type of) problem is used typically to connect multiple components of a circuit to the power sources that can only be placed at the boundary of the circuit – disjointness of the paths ensures that the components of the circuit do not interfere with each other.

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)

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)**

(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)**

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)**

(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)**

(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)**

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)**