CS 170 HW 12

Due **2020-11-23, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

In addition, we would like to share correct student solutions that are well-written with the class after each homework. Are you okay with your correct solutions being used for this purpose? Answer "Yes", "Yes but anonymously", or "No"

2 Survivable Network Design

Survivable Network Design is the following problem: We are given two $n \times n$ matrices: a cost matrix d_{ij} and a (symmetric) connectivity requirement matrix r_{ij}. We are also given a budget b. We want to find a undirected graph $G = (\{1, ..., n\}, E)$ such that the total cost of all edges (i.e. $\sum_{(i,j)\in E} d_{ij}$) is at most b and there are exactly r_{ij} edge-disjoint paths between any two distinct vertices i and j, or if no such G exists, output "None". (A set of paths is edge-disjoint if no edge appears in more than one of them)

Show that Survivable Network Design is NP-Complete. (Hint: Reduce from a NP-Hard problem in Section 8 of the textbook.)

Solution: To show Survivable Network Design is NP. Given a solution to Survivable Network Design (i.e. a graph G), we can check the budget constraint in linear time easily. To check the connectivity requirements, we can check if the max flow between every pair i, j using unit capacities on the edges is at least r_{ij}. Any max-flow of value F can be decomposed into F edge-disjoint i-j paths, since all capacities are unit.

We reduce from Rudrata Cycle to Survivable Network Design: Given $G = (V, E)$, take $b = n$ (recall $|V| = n$), $d_{ij} = 1\ \forall (i, j) \in E$ and $d_{ij} > n$ otherwise; set $r_{ij} = 2\ \forall (i, j)$.

Clearly any Rudrata Cycle is a solution to this Survivable Network Design instance. So we just need to show any solution in this instance is a Rudrata Cycle.

Any solution uses edges such that the sum of the weights of all the edges is at most n, i.e. uses at most n edges since all usable edges have weight 1. In order to have two edge-disjoint paths between any two distinct vertices, every vertex must have degree at least 2. But since we can only use n edges, the total degree of all vertices can't be more than $2n$. So every vertex has degree exactly 2. This means that the edges must form a set of disjoint cycles. But every pair of vertices needs to be connected by the edges, so this set of disjoint cycles must actually be a single cycle visiting every vertex, i.e. a Rudrata Cycle.

3 (3,3)-SAT

Consider the (3,3)-SAT problem, which is the same as 3-SAT except each literal or its negation appears *at most* 3 times across the entire formula. Notice that (3,3)-SAT is reducible to 3-SAT because every formula that satisfies the (3,3)-SAT constraints satisfies those for

3-SAT. We are interested in the other direction.

Show that (3,3)-SAT is NP-Complete via reduction from 3-SAT. By doing so, we will have eliminated the notion that the "hardness" of 3-SAT was in the repetition of variables across the formula. (Hint: If variable x_i appears in k clauses, your reduction should replace x_i with k variables $x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(k)}$. How can we enforce that the copies of x_i all have the same value?).

Give a precise description of the reduction and prove its correctness.

Solution:

(3, 3)-SAT is clearly in NP, since we can verify all clauses are satisfied in linear time.

To reduce 3-SAT to (3, 3)-SAT: Assume that the variable x_i or $\neg x_i$ appears $k > 3$ times. Let us replace the k occurrences of x_i with $x_i^{(j)}$ for $j = 1, \ldots, k$. We now have used each literal $x_i^{(j)}$ exactly once. We additionally add the clauses $\neg x_i^{(j)} \vee x_i^{(j+1)}$ and $\neg x_i^{(k)} \vee x_i^{(1)}$ for each $j = 1, \ldots, k-1$. Recall that $\neg x_i^{(j)} \vee x_i^{(j+1)}$ is equivalent to $x_i^{(j)} \Rightarrow x_i^{(j+1)}$ and $\neg x_i^{(j+1)} \Rightarrow \neg x_i^{(j)}$, so the collective chain enforces that $x_i^{(1)} = x_i^{(2)} = \ldots = x_i^{(k)}$.

Make the replacement for each literal that occurs more than 3 times ensures that all literals in the new formula occur at most 3 times. This is only a polynomial increase in the size of the problem description.

A satisfying assignment $\{x_i = \alpha_i\}$ for the original formula can be translated to an assignment for the new formula by setting $x_i^{(j)} = \alpha_i$ by the previous argument.

For the other direction, a satisfying assignment for the new formula must have $x_i^{(1)} = x_i^{(2)} = \ldots = x_i^{(k)} = \alpha_i$, so we can set $x_i = \alpha_i$ in the original formula.

4 Randomization for Approximation

This is a solo question.

Oftentimes, extremely simple randomized algorithms can achieve reasonably good approximation factors.

(a) Consider Max 3-SAT (given a set of 3-clauses, find the assignment that satisfies as many of them as possible). Come up with a simple randomized algorithm that will achieve an approximation factor of $\frac{7}{8}$ in expectation. That is, if the optimal solution satisfies k clauses, your algorithm should produce an assignment that satisfies at least $\frac{7}{8} * k$ clauses in expectation. You may assume that every clause contains exactly 3 distinct variables.

(b) Given a Max 3-SAT instance I, let OPT_I denote the maximum fraction of clauses in I satisfied by any assignment. What is the smallest value of OPT_I over all instances I? In other words, what is $\min_I OPT_I$? (Again, you may assume that every clause contains exactly 3 distinct variables)

Solution:

(a) Consider randomly assigning each variable a value. Let X_i be a random variable that is 1 if clause i is satisfied and 0 otherwise. We can see that the expectation of X_i is $\frac{7}{8}$. Note that $\sum X_i$ is the total number of satisfied clauses. By linearity of expectation, the

expected number of clauses satisfied is $\frac{7}{8}$ times the total number of clauses. Since the optimal number of satisfied clauses is at most the total number of satisfied clauses, a random assignment will in expectation have value at least $\frac{7}{8} * k$.

(b) Our randomized algorithm satisfies fraction 7/8 of clauses in expectation for any instance. So any instance must have a solution that satisfies at least fraction 7/8 of clauses (a random variable must sometimes be at least its mean).

This lower bound is tight: Consider an instance with 3 variables and all 8 possible clauses including these variables. Then any solution satisfies exactly 7 clauses.

5 Approximately Coloring a 3-Colorable Graph

Recall that the 3-coloring problem is NP-hard. In this problem, we will devise a polynomial-time algorithm for $O(\sqrt{n})$-coloring a 3-colorable graph.

(a) Let G be a graph of maximum degree Δ. Show that G is $(\Delta + 1)$-colorable.

(b) Suppose G is a 3-colorable graph. Let v be any vertex in G. Show that the graph induced on the neighborhood of v is 2-colorable (the graph induced on the neighborhood of v refers to the subgraph of G obtained from the set V' of vertices adjacent to v and all edges of G with both endpoints in V').

(c) Give a polynomial time algorithm that takes in a 3-colorable n-vertex graph G as input and outputs a valid coloring of its vertices using $O(\sqrt{n})$ colors. Just an algorithm and proof of correctness are needed. You may use the polynomial-time algorithm for 2-coloring as a black box.
Hint: Use the previous two parts. Your algorithm should first color "high-degree" vertices and their neighborhoods using part b, and then color the rest of the graph using part a.

Solution:

(a) While there is a vertex with an unassigned color, give it a color that is distinct from the color assigned to any of its neighbors (such a color always exists since we have a palette of size $\Delta + 1$ but only at most Δ neighbors).

(b) In an induced graph, we only care about the direct vertices adjacent to v and v itself. In any 3-coloring of G, v must have a different color from its neighborhood and therefore the neighborhood must be 2-colorable.

(c) **Main idea:** While there is a vertex of degree-$\geq \sqrt{n}$, choose the vertex v, pick 3 colors, use one color to color v, and the remaining 2 colors to color the neighborhood of v. Never use these colors ever again and delete v and its neighborhood from the graph. When all vertices have degree less than $\sqrt{n}$, we can greedily $\sqrt{n}$-color the graph as in part a.

Proof of Correctness: Since each step in the while loop deletes at least $\sqrt{n}$ vertices, there can be at most $\sqrt{n}$ iterations. This uses only $3\sqrt{n}$ colors. After this while loop is done we will be left with a graph with max degree at most $\sqrt{n}$. We can $\sqrt{n} + 1$ color with fresh colors this using the greedy strategy from the solution to part b. The total number of colors used is $O(\sqrt{n})$.

6 Fast Modular Exponentiation

This is a solo question.

Give a polynomial time algorithm for computing $a^{b^c} \mod p$ given prime p and integers a, b, and c. Just the algorithm description is needed.

(Remember that by "polynomial time", we mean that if a, b, c, p are all at most n bits long, your algorithm should take time polynomial in n.)

Solution: Main idea: We know how to compute $x^y \mod z$ efficiently for any x, y, z: Square x and apply $\mod z$ repeatedly to compute $x, x^2, x^4 \ldots$ all $\mod z$. Then x^y can be written as some product of these (e.g. $x^5 = x * x^4$), so we can compute x^y easily. (This is given in the textbook, so you may black-box this part of the solution).

Then, we show how to reduce this problem to two instances of finding $x^y \mod z$:

- Since p is prime, by Fermat's Little Theorem, we know $a^{p-1} \mod p = 1$. So we first find $d = b^c \mod (p-1)$.
- We then note that $a^{b^c} \mod p = a^d \mod p$. Then, we just compute $a^d \mod p$.

7 Wilson's Theorem

Wilson's theorem says that a number N is prime if and only if

$$(N-1)! \equiv -1 \pmod{N}.$$

(a) If p is prime, then we know every number $1 \leq x < p$ is invertible modulo p. Which of these numbers are their own inverse?

(b) By pairing up multiplicative inverses, show that if p is prime then $(p-1)! \equiv -1 \pmod{p}$.

(c) Complete the proof of Wilson's Theorem by showing that if $(N-1)! \equiv -1 \pmod{N}$ then N is prime. (Hint: If $p > 1$ divides x, it doesn't divide $x+1$)

(d) Unlike Fermat's Little theorem, Wilson's theorem is an if-and-only-if condition for primality. Why can't we immediately base an efficient primality test on Wilson's theorem?

Solution:

(a) A number $1 \leq n < p$ is its own inverse modulo p if and only if $n^2 \equiv 1 \mod p$ as $n^2 - 1 = (n-1)(n+1) \equiv 0 \mod p$. So the answer is 1 and $p-1$.

(b) Among the $p-1$ numbers, 1 and $p-1$ are their own inverses and the rest have a (different than themselves) unique inverse $\mod p$. Since inversion is a bijective function, each number a in $\{2, \cdots, p-2\}$ is the inverse of some number in the same range not equal to a. Thus,$(p-2)(p-3)\cdots 2 \equiv 1 \mod p \Rightarrow (p-1)! \equiv (p-1) \equiv -1 \mod p$.

(c) There are multiple correct approaches using the same key ideas, we only provide one for brevity.

Suppose equivalently $(N-1)!+1 \equiv \pmod{N}$. Any divisor of N thus also divides $(N-1)!+1$. But this means $2, 3, \ldots, N-1$ can't be divisors of N because they are divisors of $(N-1)!$ and thus are not divisors of $(N-1)!+1$. So N must be prime.

(d) This rule involves calculating a factorial product which takes time exponential in the size of the input. Thus, the algorithm would not be efficient.

8 (Extra Credit) Approximation Hardness of Independent Set

Recall the maximum independent set problem: Given an undirected graph G, we want to find the largest independent set, i.e. largest set of vertices S such that no two vertices in S are adjacent. An algorithm is a α-approximation algorithm for maximum independent set if given a graph G, it outputs an independent set of size at least OPT/α, where OPT is the size of the largest independent set.

Show that if there is a polynomial-time $2^{2^{100}}$-approximation algorithm for maximum independent set, there is a polynomial-time 2-approximation algorithm for maximum independent set.

Solution: Given a graph G, we use the following "graph squaring" procedure to construct a graph G', such that if the size of the largest independent set in G is OPT, the size of the largest independent set in G' is OPT^2. G' has a vertex labelled $\{u, v\}$ for every ordered pair of vertices in G (including repeat pairs, such as (v, v)). G' has an edge between $\{u_1, v_1\}$ to $\{u_2, v_2\}$ if at least one of (u_1, u_2) and (v_1, v_2) is an edge in G.

If an independent set S of size $|S|$ exists in G, one of size $|S|^2$ exists in G'. We simply include all vertices $\{u, v\}$ in G' for which $u, v \in S$. Since S was an independent set, the set of edges we included in G' ensures this is also an independent set.

If an independent set S of size $|S|$ exists in G', one of size at least $\sqrt{|S|}$ exists in G. Let S_1 be the set of all vertices in G that are "first coordinates" of vertices in S. e.g. if $\{u_1, v_1\}$ is in S, then u_1 will be in S_1. Define S_2 similarly, but for the second coordinate. We have $|S| \leq |S_1||S_2|$, i.e. one of S_1, S_2 is size at least $\sqrt{|S|}$. Furthermore, both S_1, S_2 are independent sets in G since S is an independent set in G', which means none of the first (or second) coordinates in S are adjacent in G.

Given a $2^{2^{100}}$-approximation algorithm for maximum independent set, we can design a 2-approximation algorithm as follows: Take G with maximum independent set of size OPT, and apply graph squaring to G 100 times. This gives a graph G' such that the largest independent set in G' has size $OPT^{2^{100}}$. Furthermore, G' has size $O((|V|+|E|)^{2^{100}})$, so we can construct it in polynomial-time. Now run the $2^{2^{100}}$-approximation algorithm for maximum independent set on G' to find an independent set of size $(OPT/2)^{2^{100}}$. Using the procedure from the previous paragraph, we can retrieve from this an independent set of size at least $OPT/2$ in G.

Comment: In essence, this shows that if it is NP-Hard to 2-approximate independent set, it is NP-Hard to $2^{2^{100}}$-approximate independent set or really, α-approximate independent set for any constant α. This is a very simple example of a technique known as gap amplification,

which is fundamental in the area known as "hardness of approximation", which seeks to show that finding α-approximation algorithms for some problems is as hard as solving the problem exactly.