CS 170 Homework 13 (Optional)

1 Study Group

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

2 Multiway Cut

In the multiway cut problem, we are given a graph $G = (V, E)$ with k special vertices $s_1, s_2, \ldots, s_k$. Our goal is to find the smallest set of edges F which, when removed from the graph, disconnect the graph into at least k components, where each s_i is in a different component. When $k = 2$, this is exactly the min s-t cut problem, but if $k \geq 3$ the problem becomes NP-hard.

Consider the following algorithm: Let F_i be the set of edges in the minimum cut with s_i on one side and all other special vertices on the other side. Output F, the union of all F_i. Note that this is a multiway cut because removing F_i from G isolates s_i in its own component.

(a) Explain how each F_i can be found in polynomial time.

(b) Let F^* be the smallest multiway cut. Consider the components that removing F^* disconnects G into, and let C_i be the set of vertices in the component with s_i. Let F_i^* be the set of edges in F^* with exactly one endpoint in C_i. How many different F_i^* does each edge in F^* appear in? Which is larger: F_i and F_i^*?

(c) Using your answer to the previous part, show that $|F| \leq 2|F^*|$.

(d) **Extra Credit:** how could you modify this algorithm to output F such that $|F| \leq (2 - \frac{2}{k})|F^*|$?

Solution:

(a) Consider adding a vertex t to the graph and connecting t to all special vertices except s_i with infinite capacity edges. Then F_i is the minimum s_i-t cut, which we know how to find in polynomial time.

(b) Each edge in F^* appears in exactly two of the sets F_i^*.

Note that F_i^* is the set of edges in a cut which disconnects s_i from the other special vertices. Then by definition F_i has fewer edges than F_i^* since F_i is the minimum cut disconnecting s_i from all other special vertices.

(c) We combine the answers to the previous part and note that F's size is at most the total size of all F_i to get:

$$|F| \leq \sum_i |F_i| \leq \sum_i |F_i^*| = 2|F^*|$$

(d) To get the $(2 - \frac{2}{k})$-approximation, after computing all F_i, we instead output F as the union of all F_i except for the one with the most edges. Let this be F_j. This is still a

multiway cut because each s_j is still disconnected from all other s_i. Then:

$$|F| \leq \sum_{i \neq j} |F_i| \leq \left(1 - \frac{1}{k}\right) \sum_i |F_i| \leq \left(1 - \frac{1}{k}\right) \sum_i |F_i^*| = \left(2 - \frac{2}{k}\right) |F^*|$$

3 Relaxing Integer Linear Programs

As discussed in lecture, Integer Linear Programming (ILP) is NP-complete. In this problem, we discuss attempts to approximate ILPs with Linear Programs and the potential shortcomings of doing so.

Throughout this problem, you may use the fact that the ellipsoid algorithm finds an optimal vertex (and corresponding optimal value) of a linear program in polynomial time.

(a) Suppose that $\vec{x}_0$ is an optimal point for the following arbitrary LP:

$$\begin{aligned} \text{maximize } & c^\top x \\ \text{subject to: } & Ax \leq b \\ & x \geq 0 \end{aligned}$$

Show through examples (i.e. by providing specific canonical-form LPs and optimal points) why we cannot simply (1) round all of the element in $\vec{x}_0$, or (2) take the floor of every element of $\vec{x}_0$ to get good integer approximations.

(b) The MATCHING problem is defined as follows: given a graph G, determine the size of the largest subset of disjoint edges of the graph (i.e. edges without repeating incident vertices).

Find a function f such that:

$$\begin{aligned} \text{maximize } & f \\ \text{subject to: } & \sum_{e \in E, v \in e} x_e \leq 1 \qquad \forall v \in V \\ & 0 \leq x_e \leq 1 \qquad \forall e \in E \end{aligned}$$

is an LP relaxation of the MATCHING problem. Note that the ILP version (which directly solves MATCHING) simply replaces the last constraint with $x_e \in \{0, 1\}$.

(c) It turns out that the polytope of the linear program from part (b) has vertices whose coordinates are all in $\{0, \frac{1}{2}, 1\}$. Using this information, describe an algorithm that approximates MATCHING and give an approximation ratio with proof.

Hint: round up, then fix constraint violations.

(d) There is a class of linear program constraints whose polytopes have only integral coordinates. Let $\mathcal{P}_{>2,\text{odd}}(V)$ be the set of subsets of the vertices with size that is odd and greater than 2. It turns out that, if we simply add to the LP from part (b) the following constraints:

$$\sum_{e \in E(S)} x_e \leq \frac{|S| - 1}{2} \qquad \forall S \in \mathcal{P}_{>2,\text{odd}}(V),$$

then all vertices of the new polytope are integral. First, interpret this constraint in words and explain why it still describes the MATCHING problem. Then, explain what this result implies about approximating ILPs with (special) LPs.

(e) Why doesn't the observation in part (d) imply that MATCHING $\in$ P?

Solution:

(a) Neither of these work because our rounding/truncating could leave us with points outside of the feasible region. For example, if the constraints of the LP are:

$$\begin{aligned} \text{Maximize } & x + y \\ \text{subject to: } & 3x \leq 2 \\ & 3y \leq 2 \\ & -3x \leq -1 \\ & -3y \leq -1 \\ & x, y \geq 0 \end{aligned}$$

Then, the optimal point is at $\left(\frac{2}{3}, \frac{2}{3}\right)$, which rounds to $(1, 1)$ outside the feasible region. Then, rounding down would leave us with $(0, 0)$, which is not in the feasible region.

(b) Each x_e variable represents whether we include edge e in our matching. In our relaxation, we can allow for partially choosing edges. So, we want $f = \sum_e x_e$, meaning the maximum number of edges chosen in our matching.

(c) If we simply run the ellipsoid algorithm, we'll get an optimal vertex in polynomial time whose coordinates are within $\{0, \frac{1}{2}, 1\}$. Let's call this solution LP. We round the $\frac{1}{2}$ values to 0 or 1 in the following way:

while there still exists $e = (u, v)$ s.t. $x_e = \frac{1}{2}$ **do**
  $x_e \leftarrow 1$
  for all edges $e' \neq e$ incident to one of u or v **do**
    $x_{e'} = 0$

Let's call this rounded solution $\widetilde{LP}$.

We first bound $\widetilde{LP}$ from above. Since $\widetilde{LP}$ only contains the values $\{0, 1\}$ and satisfies the other constraints, it is contained in the ILP's feasible region. Thus, $\widetilde{LP} \leq OPT$.

Then, we bound $\widetilde{LP}$ from below. During each while loop iteration of the rounding process, we will modify up to 3 edges (e and 2 edges e'). In the worst case, the sum of the values of the three modified edges goes from $\frac{3}{2}$ to 1, and each edge is modified at most once. This gives us $\widetilde{LP} \geq \frac{2}{3} LP$. Finally, since the LP formulation is a relaxed version of ILP, we have

$$\widetilde{LP} \geq \frac{2}{3} LP \geq \frac{2}{3} OPT.$$

Therefore, we have $\frac{2}{3} OPT \leq \widetilde{LP} \leq OPT$, which yields an approximation ratio of $\frac{3}{2}$.

(d) This constraint states that for all odd-sized subsets S of the vertices, the number of edges in the matching is at most $\frac{1}{2}(|S| - 1)$. For example, out of any three vertices, at most one whole edge can be used in our matching, which disallows solutions where three adjacent edges are assigned to $\frac{1}{2}$. This property holds for every valid (integer)

matching since breaking this property means that at least one vertex in S is incident to 2 edges (one way we can see this is the Pigeonhole Principle).

Given this new LP, we can run the ellipsoid algorithm to get an optimal vertex in polynomial time with respect to n and m (number of variables and constraints). Since all vertices are integral, then an optimal vertex for the linear program is also an optimal vertex for the integer linear program. Therefore, we can simply take an output to the LP to solve the ILP, yielding an approximation ratio of 1. We can always solve the MATCHING ILP by converting it to an LP!

(e) Note that $\mathcal{P}_{>2,\text{odd}}$ has size $O\left(2^{|V|-1}\right) = O\left(2^{|V|}\right)$, meaning that there are $O\left(2^{|V|}\right)$ constraints. On the other hand, the size of G is $O\left(|V|^2\right)$. Therefore, we are passing an instance to the ellipsoid algorithm that is *exponential* in the size of the instance to MATCHING. Even though running the ellipsoid algorithm (or any other polynomial-time algorithm for solving LPs) is polynomial in the size of its input, it is actually exponential in the input to MATCHING.

4 Randomization for Approximation

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

(a) Consider Max 3-SAT: given an instance with m clauses each containing exactly 3 distinct literals, 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 c clauses, your algorithm should produce an assignment that satisfies at least $\frac{7c}{8}$ clauses in expectation.

Hint: use linearity of expectation!

(b) Given a Max 3-SAT instance I, let OPT_I denote the maximum fraction of clauses in I satisfied by any variable assignment. What is the smallest value of OPT_I over all instances I? In other words, what is $\min_I \mathsf{OPT}_I$?

Hint: use part (a), and note that a random variable must sometimes be at least its mean.

(c) **(Challenge:)** Derandomize your algorithm from part (a), i.e give a deterministic algorithm that achieves the same approximation factor. Justify the correctness of your algorithm.

Disclaimer: this subpart is particularly difficult, so please only attempt this after completing the rest of the homework and if you want extra practice.

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 $\frac{7c}{8}$.

(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).

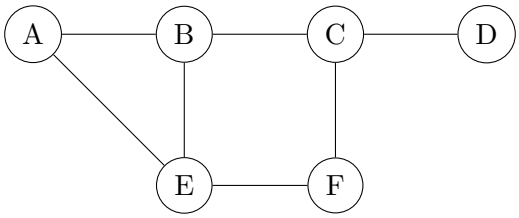
This lower bound is tight: Consider an instance with 3 variables and all 8 possible clauses including these variables. Then any solution satisfies exactly 7 clauses.

(c) `https://cs.stackexchange.com/questions/80887/randomized-algorithm-for-3sat`.

5 Vertex Cover to Set Cover

To help jog your memory, here are some definitions:

> **Vertex Cover:** given an undirected unweighted graph $G = (V, E)$, a vertex cover C_V of G is a subset of vertices such that for every edge $e = (u, v) \in E$, at least one of u or v must be in the vertex cover C_V.
>
> **Set Cover:** given a universe of elements U and a collection of sets $\mathcal{S} = \{S_1, \ldots, S_m\}$, a set cover is any (sub)collection C_S whose union equals U.
>
> In the *minimum vertex cover problem*, we are given an undirected unweighted graph $G = (V, E)$, and are asked to find the smallest vertex cover. For example, in the following graph, $\{A, E, C, D\}$ is a vertex cover, but not a minimum vertex cover. The minimum vertex covers are $\{B, E, C\}$ and $\{A, E, C\}$.



> Then, recall in the *minimum set cover problem*, we are given a set U and a collection $\mathcal{S} = \{S_1, \ldots, S_m\}$ of subsets of U, and are asked to find the smallest set cover. For example, given $U := \{a, b, c, d\}$, $S_1 := \{a, b, c\}$, $S_2 := \{b, c\}$, and $S_3 := \{c, d\}$, a solution to the problem is $C_S = \{S_1, S_3\}$.

Give an efficient reduction from the minimum vertex cover problem to the minimum set cover problem. Briefly justify the correctness of your reduction (i.e. 1-2 sentences).

Solution:

Algorithm Description: Let $G = (V, E)$ be an instance of the minimum vertex cover (MVC) problem. Create an instance of the minimum set cover (MSC) problem where $U = E$ and for each $u \in V$, the set S_u contains all edges incident to u.

Let $C_S = \{S_{u_1}, S_{u_2}, \ldots, S_{u_k}\}$ be a set cover, where $k = |C_S|$. Then our corresponding vertex cover will be $C_V = u_1, u_2, \ldots, u_k$.

Proof of Correctness: To see that C_V is a vertex cover, take any $(u, v) \in E$. Since $(u, v) \in U$, there is some set S_{u_i} containing (u, v), so u_i equals u or v and (u, v) is covered in the vertex cover. Thus, every vertex cover in G corresponds to a set cover in $U, \mathcal{S}$.

Now take any vertex cover $u_1, \ldots, u_k$. To see that $S_{u_1}, \ldots, S_{u_k}$ is a set cover, take any $(u, v) \in E$. By the definition of vertex cover, there is an i such that either $u = u_i$ or $v = u_i$. So $(u, v) \in S_{u_i}$, so $S_{u_1}, \ldots, S_{u_k}$ is a set cover. Thus, every set cover in $U, \mathcal{S}$ corresponds to a vertex cover in G.

Since every vertex cover has a corresponding set cover (and vice-versa) and minimizing set cover minimizes the corresponding vertex cover, the reduction holds.

6 Dijsktra's Sort

Show how to use Dijsktra's algorithm to sort n real numbers (not necessarily non-negative or integral) in ascending order in $O(n \log n)$ time. You may use the intermediate outputs of Dijsktra's to do the sorting (as opposed to using it as a blackbox).

Argue that this means that improving upon the $O(m + n \log n)$ run-time of Dijsktra's algorithm (with Fibonacci heap) would lead to a faster sorting algorithm than merge and quick sort.
Hint: Given the numbers, construct a graph such that the order of vertices being extracted from priority queue by Dijsktra's algorithm corresponds to the right sorting order.

Solution: Consider the star graph with one center vertex s connected to n other vertices. Let the input numbers be $\{t_1, \cdots, t_n\}$. Let edge (s, i) have weight t_i. We consider each non-center vertex i as corresponding to the number t_i We claim that the order of vertices being extracted from the priority queue by the Dijsktra's algorithm is precisely the ascending order of the inputs. In the first iteration, s is extracted, and we can ignore that, since it doesn't correspond to any input number. Observe that then the Dijsktra's algorithm updates the vertex labels `dist` to be exactly the input numebers. Hence, the next iteration will extract exactly the vertex corresponding to the smallest number t_i. But since a non-center vertex is not adajacent to anything except the center, we do not update `dist`, so we move on. Then the algorithm proceeds to extract the vertex corresponding to the second smallest number t_i in the next iteration, and so on.

If there's any data structure that enables faster run-time for Dijsktra's, then we can always apply the reduction above to sort numbers faster. Essentially, we are just using the fact that Dijsktra's uses priority queue and priority queue can be used for sorting.

7 Orthogonal Vectors

In the 3-SAT problem, we have n variables and m clauses, where each clause is the OR of (at most) three of these variables or their negations. The goal of the problem is to find an assignment of variables that satisfies all the clauses, or correctly declare that none exists.

In the orthogonal vectors problem, we have two sets of vectors A, B. All vectors are in $\{0,1\}^m$, and $|A| = |B| = n$. The goal of the problem is to find two vectors $a \in A, b \in B$ whose dot product is 0, or correctly declare that none exists. The brute-force solution to this problem takes $O(n^2 m)$ time: compute all $|A||B| = n^2$ dot products between two vectors in A, B, and each dot product takes $O(m)$ time.

Show that if there is a $O(n^c m)$-time algorithm for the orthogonal vectors problem for some $c \in [1, 2)$, then there is a $O(2^{cn/2} m)$-time algorithm for the 3-SAT problem. For simplicity, you may assume in 3-SAT that the number of variables must be even.

Hint: Try splitting the variables in the 3-SAT problem into two groups.

Solution: We use an $O(2^{n/2} m)$-time reduction from 3-SAT to orthogonal vectors. We split the variables into two groups of size $n/2$, V_1 and V_2. For each group, we enumerate all $2^{n/2}$ possible assignments of these variables. For each assignment x of the variables in V_1, let v_x be the vector where the ith entry is 0 if the ith clause is satisfied by one of the variables in this assignment, and 1 otherwise. We ignore the variables in the clause that are in V_2. For example, if clause i only contains variables in V_2, then $v_x(i) = 0$ for all x. Let A be the $2^{n/2}$ vectors produced this way.

We construct B containing $2^{n/2}$ vectors in a similar manner, except using V_2 instead of V_1.

We claim that the 3-SAT instance is satisfiable if and only if there is an orthogonal vector pair in $A \times B$. Given this claim, we can solve 3-SAT by making the orthogonal vectors instance in $O(2^{n/2} m)$ time, and then solving the instance in $O((2^{n/2})^c m) = O(2^{cn/2} m)$ time.

Suppose there is a satisfying assignment to 3-SAT. Let x_1 be the assignment of variables in V_1, and x_2 be the assignment of variables in V_2. Let v_1, v_2 be the vectors in A, B corresponding to x_1, x_2. Since every clause is satisfied, one of $v_1(i)$ and $v_2(i)$ must be 0 for every i, and so $v_1 \cdot v_2 = 0$. So there is also a pair of orthogonal vectors in the orthogonal vectors instance.

Suppose there is a pair of orthogonal vectors v_1, v_2 in the orthogonal vectors instance. Then for every i, either $v_1(i)$ or $v_2(i)$ is 0. In turn, for the corresponding assignment of variables in V_1, V_2, the combination of these assignments must satisfy every clause. Hence, the combination of these assignments is a satisfying assignment for 3-SAT.

Comment: It is widely believed that SAT has no $O(2^{.999n} m)$-time algorithm - this is called the Strong Exponential Time Hypothesis (SETH). So it is also widely believed that orthogonal vectors has no $O(n^{1.99} m)$-time algorithm, since otherwise SETH would be violated. It turns out that we can reduce orthogonal vectors to string problems such as edit distance and longest common subsequence, and so if we belive SETH then we also believe those problems also don't have $O(n^{1.99})$-time algorithms. The field of research studying reductions between problems with polynomial-time algorithms such as these is known as fine-grained complexity, and orthogonal vectors is one of the central problems in this field.

8 Local Search for Max Cut

Sometimes, local search algorithms can give good approximations to NP-hard problems. Recall that in the Max-Cut problem, we have an unweighted graph $G = (V, E)$ and we want to find a cut (S, T) that maximizes the number of edges "crossing" the cut (i.e. with one endpoint in each of S, T). Consider the following local search algorithm:

1. Start with any cut (e.g. $(S, T) = (V, \emptyset)$).
2. While there is some vertex $v \in S$ such that more edges cross $(S \setminus \{v\}, T \cup \{v\})$ than (S, T) (or some $v \in T$ such that more edges cross $(S \cup \{v\}, T \setminus \{v\})$ than (S, T)), move v to the other side of the cut.

Now, let us prove a couple of guarantees that this algorithm achieves.

(a) Give an upper bound on the number of iterations this algorithm can run for (i.e. the total number of times we move a vertex).

(b) Show that when this algorithm terminates, it finds a cut where at least half the edges in the graph cross the cut.

Hint: when we move v from S to T, v must have more neighbors in S than T. What does this observation suggest about the neighbors of each vertex once the algorithm terminates? Then, what can we say about the number of edges crossing the cut vs. the number of edges within each side of the cut?

Solution:

(a) $|E|$ iterations. Each iteration increases the number of edges crossing the cut by at least 1. The number of edges crossing the cut is between 0 and $|E|$, so there must be at most $|E|$ iterations.

(b) $\delta_{in}(v)$ be the number of edges from v to other vertices on the same side of the cut, and $\delta_{out}(v)$ be the number of edges from v to vertices on the opposite side of the cut. The total number of edges crossing the cut the algorithm finds is $\frac{1}{2}\sum_{v \in V} \delta_{out}(v)$, and the total number of edges in the graph is $\frac{1}{2}\sum_{v \in V}(\delta_{in}(v) + \delta_{out}(v))$. We know that $\delta_{out}(v) \geq \delta_{in}(v)$ for all vertices when the algorithm terminates (otherwise, the algorithm would move v across the cut), so the former is at least half as large as the latter.