CS264: Homework #3

Due by midnight on Thursday, February 2, 2017

Instructions:

(1) Form a group of 1-3 students. You should turn in only one write-up for your entire group. See the course site for submission instructions.

(2) Please type your solutions if possible and feel free to use the LaTeX template provided on the course home page.

(3) **Students taking the course pass-fail should complete 5 of the Exercises and can skip all of the Problems. Students taking the course for a letter grade should complete 6 of the Exercises, Problem 8, and as many other Problems as they choose.** We'll grade the Problems out of a total of 40 points, according to the formula

$$\text{score on Problem } 8 + \min\{20, \text{total points earned on other problems}\}.$$

Meanwhile, you'll also receive $\max\{0, \text{total points earned on other problems} - 20\}$ extra credit points.

(4) Write convincingly but not excessively. Exercise solutions rarely need to be more than 1-2 paragraphs. Problem solutions rarely need to be more than a half-page (per part), and can often be shorter.

(5) You may refer to your course notes, and to the textbooks and research papers listed on the course Web page *only*. You cannot refer to textbooks, handouts, or research papers that are not listed on the course home page. (Exception: feel free to use your undergraduate algorithms textbook.) Cite any sources that you use, and make sure that all your words are your own.

(6) If you discuss solution approaches with anyone outside of your team, you must list their names on the front page of your write-up.

(7) Exercises are worth 5 points each. Problem parts are labeled with point values.

(8) No late assignments will be accepted.

Lecture 5 Exercises

Exercise 11

The point of this exercise is to give a parameterized analysis of the the greedy algorithm for the Knapsack problem discussed in lecture. Recall that in the Knapsack problem, you are given n items with nonnegative values $v_1, \ldots, v_n$ and sizes $s_1, \ldots, s_n$. There is also a knapsack capacity C. The goal is to compute a subset $S \subseteq \{1, 2, \ldots, n\}$ of items that fits in the knapsack (i.e., $\sum_{i \in S} s_i \leq C$) and, subject to this, has the maximum total value $\sum_{i \in S} v_i$.

Consider the following greedy algorithm:

1. Order items by "density," meaning reindex them so that $\frac{v_1}{s_1} \geq \frac{v_2}{s_2} \geq \cdots \geq \frac{v_n}{s_n}$.

2. Return the largest prefix $\{1, 2, \ldots, j\}$ that fits in the knapsack (i.e., with $\sum_{i=1}^{j} s_i \leq C$).

Suppose that all items are *small* in the sense that, for a parameter $\alpha \in (0,1)$, $s_i \leq \alpha C$ for every i. Prove that the solution returned by the greedy algorithm has total value at least $(1-\alpha)$ times that of an optimal solution.

[Hint: how much of the knapsack can be unused at termination? What can you say about the part of the knapsack that did get used?]

Exercise 12

In lecture we gave two different proofs that the Vertex Cover problem is fixed-parameter tractable. Recall that, in every graph $G = (V,E)$, a subset $S \subseteq V$ of vertices is a vertex cover if and only if its complement $V - S$ is an independent set (i.e., if no two vertices of $V - S$ are adjacent in G). Does this correspondence imply that the independent set problem is also fixed-parameter tractable, with respect to the target solution size k? Explain your answer.

Exercise 13

The point of this exercise is to show that, if we don't care about having a kernel of polynomial size, then the existence of a kernel characterizes fixed-parameter tractability.

Formally, a *parameterized (decision) problem* is a language $L \subseteq \{0,1\}^* \times \mathbb{N}$. We denote a generic instance by (I,k).[1] A parameterized problem L is *fixed-parameter tractable* if there is an algorithm for deciding membership of an instance (I,k) in L that runs in time at most $f(k) \cdot \text{poly}(n,k)$, where n denotes the description length of I (in bits), f is an arbitrary (computable) function of k only, and $\text{poly}(n,k)$ is some polynomial in n and k (with exponent independent of n and k).

A *kernelization algorithm* (or simply *kernel*) for a parameterized problem L accepts as input an instance (I,k) of the problem, runs in time polynomial in n (the description length of I in bits) and k, and outputs an instance (I',k') of the problem such that: (i) k' and the description length of I' are bounded by some (computable) function $g(k)$ of k only;[2] and (ii) $(I',k') \in L$ if and only if $(I,k) \in L$.

Consider a parameterized problem L for which membership can be decided in finite time. Prove that L is fixed-parameter tractable if and only if it admits a kernelization algorithm.

[Hint: for the "only if" direction, suppose there is an algorithm $\mathcal{A}$ that decides membership of (I,k) in time at most $f(k) \cdot (n+k)^d$ for constant d (where n is the length of I). Given an instance (I,k), run $\mathcal{A}$ for $(n+k)^{d+1}$ time steps. If the algorithm doesn't halt within the allotted time, conclude that (I,k) itself can be used as the kernel's output.]

Exercise 14

Let P be a path with k vertices in the graph $G = (V,E)$. Prove that, if every vertex $v \in V$ is given a color uniformly at random from $\{1,2,\ldots,k\}$, then the probability that every vertex of P gets a different color is at least e^{-k}.

[Hint: use Stirling's approximation to approximate $k!$.]

Exercise 15

Let $G = (V,E)$ be a graph with vertices colored as in the previous exercise. A *panchromatic k-path* is a path P with k vertices, all of which have distinct colors. Give an algorithm that decides whether or not G has at least one panchromatic k-path in time $O(2^k \cdot n^d)$, where d is a constant (independent of k and n).

[Hint: dynamic programming.]

[1] For example, I could encode a Knapsack instance and k the number of bits needed to describe one of the input numbers; or I could encode an undirected graph and k the target size for a vertex cover.

[2] $g(k)$ is the *size* of the kernel.

Exercise 16

Extend the color-coding algorithm from lecture (and Exercises 14 and 15) to the following problem: given a graph $G = (V, E)$ and a tree $T = (W, F)$ with k vertices, does G contain a copy of T?[3] What is the running time of your algorithm?

Lecture 6 Exercises

Exercise 17

Recall the algorithm from lecture that recovers the optimal solution in 2-perturbation stable k-median instances. Give an explicit example of a k-median instance where this algorithm does not return an optimal solution. Can you devise γ-perturbation-stable such instances, with γ arbitrarily close to 2?

Exercise 18

Recall that an α-approximation algorithm for an optimization problem always outputs a feasible solution with objective function value within an α factor of the optimal value. When can we be sure that the output of an approximation algorithm is "meaningful?"

Call a k-median instance (X, d) γ-*approximation stable* if the output of a γ-approximation algorithm is guaranteed to be the optimal k-clustering (i.e., the clustering that minimizes the k-median objective function). This definition makes explicit the usual implicit assumption that "the output of a good-enough approximation algorithm should be meaningful."

For every $\gamma \geq 1$, prove that an γ-approximation-stable k-median instance is also γ-perturbation-stable.[4] Prove that the converse is false.

Problems

Problem 5

In this problem you will prove a parameterized guarantee for a classic machine learning algorithm, the *perceptron algorithm* for classification. The input to the algorithm is n points in $\mathbb{R}^d$, with a label $b_i \in \{-1, +1\}$ for each point $\mathbf{x}_i$. The goal is to compute a *separating hyperplane*: a hyperplane with all of the positive $\mathbf{x}_i$'s (i.e., with $b_i = +1$) on one side, and all of the negative $\mathbf{x}_i$'s on the other. For simplicity, in this problem we assume that there exists a separating hyperplane, and moreover that some such hyperplane passes through the origin.[5] We are then free to scale each data point $\mathbf{x}_i$ so that it lies on the sphere (i.e., so that $\|\mathbf{x}_i\|_2 = 1$)—such scaling does not change which side of a hyperplane $\mathbf{x}_i$ is on.

The perceptron algorithm is extremely simple.

Input: n points on the sphere $\mathbf{x}_1, \ldots, \mathbf{x}_n$ with labels $b_1, \ldots, b_n \in \{-1, +1\}$.

1. Initialize t to 1 and $\mathbf{w}_1$ to the all-zero vector.
2. While there is a point $\mathbf{x}_i$ such that $\text{sgn}(\mathbf{w}_t \cdot \mathbf{x}_i) \neq b_i$, set $\mathbf{w}_{t+1} = \mathbf{w}_t + b_i \mathbf{x}_i$, and then increment t.

Figure 1: The Perceptron Algorithm.

---

[3]Formally, by a "copy," we mean there is an injective function $\sigma : W \to V$ such that $(\sigma(u), \sigma(v)) \in E$ whenever $(u, v) \in F$.

[4]Thus, positive results for perturbation-stable instances hold also for approximation-stable instances.

[5]The second assumption is without loss of generality, since one can use an extra "dummy coordinate" (with all $\mathbf{x}_i$'s having value 1 in this coordinate) to simulate an intercept. The first assumption is not without loss of generality, but it can be encouraged through the addition of supplementary dimensions.

Intuitively, the update step makes the candidate solution "more correct" on $\mathbf{x}_i$, by increasing $\mathbf{w} \cdot \mathbf{x}_i$ by $b_i \cdot (\mathbf{x}_i \cdot \mathbf{x}_i) = b_i \cdot \|\mathbf{x}_i\|_2^2 = b_i$. Of course, this update could screw up the classification of other $\mathbf{x}_j$'s, so our job is to prove that the procedure eventually terminates.

The parameter μ that we use is called the *margin*, and it is defined as

$$\mu = \min_{i=1}^{n} |\mathbf{w}^* \cdot \mathbf{x}_i|,$$

where $\mathbf{w}^*$ is the unit normal vector of some separating hyperplane (i.e., with points $\mathbf{x}$ with $\mathbf{w}^* \cdot \mathbf{x} > 0$ on one side of the hyperplane, and points with $\mathbf{w}^* \cdot \mathbf{x} < 0$ on the other side). Geometrically, the parameter μ is the cosine of the smallest angle that a point $\mathbf{x}_i$ makes with the separating hyperplane defined by $\mathbf{w}^*$.

(a) (5 points) Prove that in every iteration t,

$$\|\mathbf{w}_{t+1}\|^2 \leq \|\mathbf{w}_t\|^2 + 1.$$

That is, the norm-squared of $\mathbf{w}$ does not grow very quickly with the number of iterations.

(b) (5 points) Prove that in every iteration t,

$$\mathbf{w}_{t+1} \cdot \mathbf{w}^* \geq \mathbf{w}_t \cdot \mathbf{w}^* + \mu.$$

That is, the projection of $\mathbf{w}$ onto $\mathbf{w}^*$ grows with each iteration.

(c) (5 points) Conclude that the iteration count t never exceeds $1/\mu^2$.

Problem 6

Recall that in the Vertex Cover problem, the input is an undirected graph $G = (V, E)$, and the goal is to compute a minimum-cardinality subset $S \subseteq V$ such that, for every edge $e \in E$, at least one of e's endpoints is in S. In Lecture #5 we gave an algorithm that checks for the existence of a vertex cover of size at most k in time $O(2^k m)$, where m is the number of edges. This bound follows from the fact that the algorithm makes $O(2^k)$ different recursive calls, and linear work is done in each.

(a) (10 points) Modify the algorithm so that the number of recursive calls is $O(\phi^k)$, where ϕ is the largest root of the equation $x^2 = x + 1$ (also known as the golden ratio, $\frac{1+\sqrt{5}}{2} \approx 1.618$), thus resulting in an algorithm that runs in $O(\phi^k m)$ time.

[Hints: Introduce another preprocessing step for the case where there is a degree-1 vertex in the current graph. Recall that the golden ratio is closely connected to the Fibonacci numbers.]

(b) (5 extra credit points) Can you improve your algorithm to run in time $O(\lambda^k m)$, where $\lambda \approx 1.4656$ is the largest root of the equation $\lambda^3 = \lambda^2 + 1$?

Problem 7

Recall the Vertex Cover problem from Lecture #5 and the previous problem. Given an undirected graph $G = (V, E)$, consider the following linear program, with one decision variable x_v for each vertex v:[6]

$$\min \sum_{v \in V} x_v$$

subject to

$$\begin{aligned} x_v + x_w &\geq 1 \qquad \text{for every edge } e = (v, w) \in E \\ x_v &\geq 0 \qquad \text{for every vertex } v \in V. \end{aligned}$$

(a) (2 points) Prove that G has a vertex cover of size at most k only if the optimal objective function value of this linear program is at most k.

[6]If you need to review what linear programs are, see, e.g., the notes for Lecture #7 of the instructor's CS261 course.

(b) (7 points) Let $\mathbf{x}^*$ denote an optimal solution to the above linear program, and define

$$A = \{v \in V \,:\, x_v^* > \tfrac{1}{2}\};$$

$$B = \{v \in V \,:\, x_v^* = \tfrac{1}{2}\};$$

and

$$C = \{v \in V \,:\, x_v^* < \tfrac{1}{2}\}.$$

Prove that there exists an optimal vertex cover of G that includes every vertex of A and excludes every vertex of C.

[Hints: If S^* is an optimal vertex cover, argue that $A \setminus S^*$ and $S^* \cap C$ must have the same size — otherwise obtain a contradiction with either the optimality of S^* or the optimality $\mathbf{x}^*$. Use this to obtain another optimal vertex cover that has the desired form.]

(c) (2 points) Prove that if G contains a vertex cover of size at most k, then $|B| \le 2k$.

(d) (4 points) Use (a)–(c) to give an algorithm that checks whether or not a graph G contains a vertex cover of size at most k and runs in time $\text{poly}(n) + 2^{O(k)}$. (Here $\text{poly}(n)$ denotes some polynomial function of n. You can assume that linear programs can be solved in polynomial time.)

(e) (5 extra credit points) The previous parts show that the Vertex Cover problem, parameterized by the solution size k, admits a kernel of size at most $2k$ (here "size" refers to the number of vertices). Assume for this part that, for every constant $\epsilon > 0$, there is no polynomial-time $(2-\epsilon)$-approximation algorithm for the Vertex Cover problem.[7] Discuss in detail the implications of this assumption for the prospects of a smaller kernel for Vertex Cover, meaning one with at most $(2-\epsilon)k$ vertices, where $\epsilon > 0$ is an arbitrarily small constant.

Problem 8 (Required)

(20 points) You are given as input an n-point metric space (X, d) and a spanning tree T of X. Each of the $\binom{n-1}{k-1}$ ways to remove $k-1$ edges of T induces a k-clustering (with clusters corresponding to connected components). The objective is to compute, among all such k-clusterings, the one with the minimum k-median objective function value.[8] Equivalently, compute the optimal (w.r.t. the k-median objective) partition of T into k non-empty connected subgraphs.

Give a dynamic programming algorithm that solves this problem in polynomial time (polynomial in both n and k). For this problem it is sufficient to give an algorithm that computes the value of an optimal solution (i.e., you do not need to describe the standard backtracking argument for reconstructing the optimal solution itself). Clearly state your subproblems, the order in which you solve the subproblems, and the subroutine for computing the solution to a subproblem from the solution to smaller subproblems. Give a brief proof of correctness (maximum 3 sentences) and an explicit (polynomial) bound on the asymptotic running time of your algorithm.

[Hints: suppose you knew one of the optimal centers r, and root the tree T at r. Define one subproblem for each point $x \in X$ (and corresponding subtree), each possibility for the center to which x is assigned, each prefix $\{y_1, \ldots, y_i\}$ of x's children (w.r.t. some arbitrary but fixed ordering, and including $\emptyset$), and each possibility for the total number of centers that reside in the union of the i subtrees rooted at $y_1, \ldots, y_i$. Make sure that your clusters are guaranteed to be connected subgraphs of T!]

[7] This assumption is known to follow from the "Unique Games Conjecture (UGC)," a plausible strengthening of the $P \neq NP$ conjecture.

[8] Recall that, given a k-clustering $C_1, \ldots, C_k$, it's clear how to pick the centers: independently for each i, choose $c_i \in C_i$ to minimize $\sum_{x \in C_i} d(x, c_i)$.

Problem 9

(10 points) Another well-studied optimization problem over k-clusterings is the *k-means* problem, where the goal is to choose clusters $C_1, \ldots, C_k$ and centers $c_1, \ldots, c_k$ (with $c_i \in C_i$ for each i) to minimize the sum of squared distances:

$$\sum_{i=1}^{k} \sum_{x \in C_i} d(x, c_i)^2.$$

A third is the *k-center* problem, where the goal is to minimize the maximum distance:

$$\max_{i=1}^{k} \max_{x \in C_i} d(x, c_i).$$

The definition of γ-perturbation stability makes sense for both of these problems. Prove that the optimal solution of a 2-perturbation-stable instance of either problem can be recovered in polynomial time.

[Hint: just describe the modifications needed relative to the algorithm from Lecture #6 (and its proof of correctness) and the previous problem.]

Problem 10

In the *Maximum Cut* problem, the input is an undirected graph $G = (V, E)$ with a nonnegative weight $w_e \geq 0$ for each edge $e \in E$. The goal to compute a cut (A, B)—a partition of V into two sets—to maximize the total weight of the cut edges (the edges with one endpoint in each of A and B). This problem is NP-hard in general (unlike the *minimum* cut problem, which can be solved in polynomial time).

An instance of the Maximum Cut problem is *γ-perturbation-stable* if, for all choices of $w'_e \in [\frac{1}{\gamma} w_e, w_e]$ for each $e \in E$, the unique maximum cut with the edges weights $\mathbf{w}'$ is the same as that with the original edge weights $\mathbf{w}$.

In this problem, you will design a polynomial-time algorithm to recover the optimal cut in $2n$-perturbation-stable instances, where $n = |V|$.[9]

(a) (3 points) Suppose you are given a polynomial-time subroutine that, given a γ-perturbation-stable instance of the Maximum Cut problem, outputs two vertices that are on the same side of the optimal cut. Prove that you can use this subroutine to obtain a polynomial-time algorithm for solving the Maximum Cut problem in γ-perturbation stable instances.

(b) (3 points) Let $w(u)$ denote the sum of the weights of the edges incident to a vertex u. Prove that in the optimal solution of a γ-perturbation-stable instance of the Maximum Cut problem, for every vertex u, the sum of the weights of the edges incident to u that do not cross the cut (i.e., that have both endpoints on the same side) is at most $w(u)/(\gamma + 1)$.

(c) (3 points) Prove that in an n-perturbation-stable instance of the Maximum Cut problem, the maximum-weight edge must cross the optimal cut (i.e., has one endpoint on either side).

[Hint: if the edge is $e = (u, v)$, what fraction of $w(u)$ (or $w(v)$) must e account for?]

(d) (3 points) Prove that in a $2n$-perturbation-stable instance of the Maximum Cut problem, both the maximum-weight edge $e = (u, v)$ and the maximum-weight edge incident to u or v (not counting e) must cross the optimal cut.

(e) (3 points) Prove that the optimal solution can be recovered in polynomial time in $2n$-perturbation stable instances of Maximum Cut.

[9]In a future lecture, we will get a much better recovery guarantee, using a much more sophisticated algorithm.