1 Summary of Approximation Algorithms from Lecture

1.1 k-Approximations

Approximation algorithms help us get around the lack of efficient algorithms for NP-hard problems by taking advantage of the fact that, for many practical applications, an answer that is "close-enough" will suffice. An approximation ratio, k, gives the ratio that bounds the solutions generated by our k-approximation.

For maximization problems, we want: $\text{APP} \geq \frac{1}{k} \cdot \text{OPT}$ and for minimization problems, we want $\text{APP} \leq k \cdot \text{OPT}$, with k being as close to 1 as possible.

1.2 Examples of approximation algorithms

- **Minimum Vertex Cover**: Given a graph $G = (V, E)$, can you find the minimal cardinality set of vertices $S \subseteq V$ such that for all $(u, v) \in E$, we have at least one of $u \in S$ or $v \in S$.

  A 2-approximation algorithm for this problem is relatively straight forward. Just greedily find a $(u, v) \in E$ for which $u \notin S_i$ and $v \notin S_i$, and construct $S_{i+1} = S_i \cup \{u, v\}$. We have $S_0 = \emptyset$. When not such (u, v) exist, return S.

  Furthermore, because this is a 2-approximation, it can be shown that $|S| \leq 2|\text{OPT}|$ where OPT is a minimum set cover.

- **Maximum Cut**: Given a graph $G = (V, E)$, can you create two sets S, T such that $S \cap T = \emptyset, S \cup T = V$, and the number of $(u, v) \in E$ that cross the cut is maximal?

  For this problem, we've covered two 2-approximation algorithms. The first is to randomly assign each $u \in V$ into either S or T. The second is to deterministically and iteratively improve our maximal cut by moving edges from S to T or vice-versa.

- **Euclidean Traveling Salesman Problem**: Give a set of points (x_i, y_i) in Euclidean space with l_2 norm, find the tour of minimum length that travels through all cities.

  A 2-approximation algorithm for this problem can be achieved by short-circuiting DFS on minimum spanning tree. Furthermore, the approximation can be improved to $\frac{3}{2}$-approximation. Both require the triangle inequality to hold.

- **Max Sat**: Find the truth assignment for variables which satisfies the largest number of OR clauses.

  We actually have two approximation algorithms. The first is by random coin-flipping (does best on long clauses). The second is randomized rounding after applying linear programming relaxation (does best on short clauses). We can also combine both for an even better solution.

2 Practice Problems

Exercise 1. Minimum Set Cover
We are given inputs to the minimum set cover problem. The inputs are a finite set $X = \{x_1, x_2, ..., x_n\}$, and a collection of subsets $\mathcal{S}$ of X such that $\bigcup_{S \in \mathcal{S}} S = X$. Find the subcollection $\mathcal{T} \subseteq \mathcal{S}$ such that the sets of $\mathcal{T}$ cover X, that is:

$$\bigcup_{T \in \mathcal{T}} T = X$$

Recall *minimum set cover* seeks to minimize the size of the selected subcollection (minimize $|\mathcal{T}|$).

(a) Formulate the set cover problem as an integer linear program.

(b) Let k be the number of times the most frequent item appears in our collection of subsets, $\mathcal{S}$. Use linear program relaxation to obtain a k-approximation algorithm for minimum set cover

Solution

(a) We create decision variables z_S for each $S \in \mathcal{S}$. Then the problem can be phrased as:

$$\begin{aligned} \text{minimize} \quad & \sum_{S \in \mathcal{S}} z_S \\ \text{subject to} \quad & \sum_{S : x_j \in S, S \in \mathcal{S}} z_S \geq 1, \qquad j = 1, ..., n \\ & z_S \in \{0, 1\}, \quad S \in \mathcal{S} \end{aligned}$$

Intuitively, $z_S = 1$ if $S \in \mathcal{T}$, otherwise $z_S = 0$. Our goal is to minimize the number of selected subsets subject to the constraint that for each of the x_i items, at least one of the selected subsets must contain that item.

(b) We can then take the above ILP and relax the integer constraint to create the below LP, which is solvable in poly-time:

$$\begin{aligned} \text{minimize} \quad & \sum_{S \in \mathcal{S}} z_S \\ \text{subject to} \quad & \sum_{S : x_j \in S, S \in \mathcal{S}} z_S \geq 1, \qquad j = 1, ..., n \\ & 0 \leq z_S \leq 1, \quad S \in \mathcal{S} \end{aligned}$$

Given the above solution, we look at the achieved value of each z_S for $S \in \mathcal{S}$. If $z_S \geq \frac{1}{k}$, then $S \in \mathcal{T}$, otherwise $S \notin \mathcal{T}$.

The claim is that the above is a k-approximation. For correctness, note that the maximum number of sets S for which $x_j \in S$ is at most k. Therefore, our constraint sums over at most k sets containing item x_j, and this sum must be ≥ 1. Then there must exists at least one set containing x_j for which $z_S \geq \frac{1}{k}$. Therefore, for each item, we take at least one $S \in \mathcal{T}$ such that $x_j \in S$.

To see the k-approximation, let OPT be the optimal solution. Then our algorithm returns a set which is at most kOPT because OPT has to cover every element, and we have at most k sets covering that element.

Exercise 2. Maximum Set Packing

In the maximum set packing problem, we are given sets $S = \{S_1, S_2, \ldots, S_n\}$ each of which is a subset of a universe X. The goal is to find $T = \{S_{i_1}, S_{i_2}, \ldots, S_{i_t}\}$ such that all the S_{i_j}'s are mutually disjoint and $t = |T|$ is as large as possible.

(a) Show that maximum set packing is NP-complete by reducing from maximum independent set. In fact, show that there is a bi-directional reduction between these two problems.

(b) Suppose now that $|S_i| \leq k$ for all $i \in \{1, 2, \ldots, n\}$. A very simple greedy algorithm for this problem is to choose an arbitrary set $S_i \in S$, add it to T, remove all sets in S whose intersection with S_i is non-empty, and recurse until S is empty. Prove that this greedy algorithm achieves an approximation ratio of k.

(c) Consider a local search algorithm where we first run the greedy algorithm described in (b) to get an initial guess on T. Then, we keep on trying to increase the number of sets in T by removing one element of T and replacing it with two elements of $S - T$ such that the sets of T are still pairwise disjoint. Prove that this local search algorithm achieves a $\frac{k+1}{2}$ approximation ratio.

Solution

(a) Given an instance of the maximum set packing problem, we can create a graph $G = (V, E)$ where we have a vertex v_i for each set $S_i \in S$ and create the edge (v_i, v_j) if $S_i \cap S_j \not\emptyset$. Any independent set of G represents a valid set packing because we only drew an edge between two vertices v_i, v_j if S_i and S_j were not disjoint. A independent set of size k in G corresponds to being able to choose k disjoint sets out of S. Therefore, the size of the maximum independent set of G is also the size of the maximum set packing using sets in S.

For the other direction, given any undirected graph $G = (V, E)$ where we are asked to find the maximum independent set, we can turn this into an instance of the set packing problem as follows: Let the universe X be E. Then, for each $v \in V$, we create the set $S_v = \{e : e \text{ is incident to } v\}$. This means that S_v contains all the edges that v is an endpoint of. Now, the maximum independent set on G is the maximum set packing of the $S = \{S_v : v \in V\}$ because any independent set corresponds to S_v's that do not intersect.

(b) Suppose the greedy algorithm constructs a T_g of size g. Then, we want to show that the optimal solution T' must be of size $\leq k \cdot g$. We'll say that two sets conflict if they have a non-empty intersection. For every set $S_i \in T_g$, there are at most k sets in T' that conflict with it. This is because every S_i has at most k elements, and thus each of those elements can be in at most 1 set of T'. Therefore, there are at most $k \cdot g$ elements in T' conflict with some element in T_g. Can there be any elements in T' that conflict with *none* of the sets in T? No, because our greedy algorithm for generating T_g terminates only when there are no more sets left that do not intersect with any of the already chosen sets. Therefore, T' can only have elements that conflict with at least one set from T_g and that is bounded by $k \cdot g$.

(c) Let T_ℓ be the our final solution once the local search algorithm has terminated. Let T' be the optimal solution. We can split T' into T'_1 and T'_2. For every $S_i \in T'_1$, there exists exactly *one* set $S_j \in T_\ell$ such that S_i conflicts with S_j. For each $S_i \in T'_{2+}$, there exists at least *two* sets in T_ℓ that S_i conflicts with. There will not exist any sets in T' that do not conflict with any set of T_ℓ because then the local search algorithm should not have terminated.

Consider any two sets $S_i, S_j \in T_1'$. By the definition of T_1', S_i conflicts with exactly one set in T_ℓ and S_j conflicts with exactly one set in T_ℓ. The claim is that those two sets in T_ℓ cannot be the same set. Suppose S_i and S_j both conflicted with $S^* \in T_\ell$. Then, the local search algorithm can make an improvement to T_ℓ by removing S^* and replacing it with S_i and S_j. We know this is valid because S_i and S_j only conflict with $S^* \in T_\ell$, so removing S^* and adding S_i, S_j maintains the property that all our chosen sets in T_ℓ do not conflict with one another. This gives us the equation: $|T_1'| \leq |T_\ell|$ because each element of T_1' must correspond to a unique element in T_ℓ.

In addition, we know from (a) that every element in T_ℓ can conflict with at most k elements in T'. Therefore, the total number of conflicts between T_ℓ and T' is bounded by $k \cdot |T'|$. We also know that the total number of conflicts is at least $|T_1'| + 2|T_2'|$ because we defined that every element of T_1' conflict with 1 element in T_ℓ and every element in T_2' conflicts with at least 2 elements in T_ℓ. Therefore, we get the inequality: $|T_1'| + 2|T_2'| \leq k|T_\ell|$.

Adding up these two inequalities, we get:

$$2|T_1'| + 2|T_2'| \leq (k+1)|T_\ell|$$

Using the fact that $|T_1'| + |T_2'| = |T'|$ which we defined to be optimal, we get the desired inequality that $|T'| \leq \frac{k+1}{2} \cdot |T_\ell|$.

Exercise 3. Subset Sum

Suppose you are given a set of positive integers $A = \{a_1, a_2 ..... a_n\}$ and a positive integer b. A subset $S \subset A$ is called feasible if the sum of the numbers in S does not exceed b. You would like to select a feasible subset S of A whose total sum is as large as possible. You may assume that $a_i \leq b$ for all i.

(a) Show that the problem is NP-Hard.

(b) Consider the following greedy algorithm: Start with $S = \{\}$, which is certainly feasible. For $i = 1$ to $i = n$, if we can add a_i to S if doing so would make S still feasible. Otherwise, we do nothing with a_i. Show that this greedy algorithm can do arbitrarily poorly compared to the optimal solution. That is, show that for every $c > 0$, there exists a possible input (A, b) to this problem where the greedy algorithm achieves a sum that is c times smaller than the optimal sum.

(c) Show that if we first sort A in $O(n \log n)$ time, then the greedy algorithm from (b) achieves a 2-approximation. Give an example of an input (A, b) where this modified algorithm's solution is almost exactly a factor of 2 worse than the optimal solution.

(d) Modify the greedy algorithm from (b) in a small way to give an $O(n)$ 2-approximation to the subset sum problem. You will not be able to sort A.

(e) Describe a dynamic programming algorithm that solves this problem exactly and analyze its run-time.

Solution

(a) This problem is NP-hard via a reduction from number partition, which is defined on PA 3. Recall that in the number partition problem, we are $X = \{x_1, x_2, \ldots, x_n\}$ and asked to

partition X into two pieces such that the sum of the elements in each piece is as close to each other as possible. Let $t = \sum_{i=1}^{n} x_i$. Then, the number partition problem on X is equivalent to the subset sum problem on (A, b) where $A = X$ and $b = \lfloor \frac{t}{2} \rfloor$. We will not give a formal proof of this reduction, but it should be quite clear after working out a couple examples by hand.

(b) Consider the very simple example where $A = \{1, 10000\}$ and $b = 10000$. Then, the optimal solution would be to have $S = \{10000\}$, but our greedy solution would take $S = \{1\}$. The greedy solution does worse by a factor of 10000. Replace 10000 with c and we get the desired result, that this greedy algorithm can be made arbitrarily bad.

(c) Without loss of generality, let's assume that $\sum_{i=1}^{n} a_i > b$ or else we would simply take $S = A$. The claim now is that the greedy algorithm is a 2-approximation because the greedy algorithm will always get an S whose sum is at least $b/2$. We know that the greedy algorithm will start by taking $a_1, a_2, \ldots, a_k$ for some k, but not take a_{k+1}. Afterwards, it may choose some subset of the a_i's with $i > k + 1$, but we won't worry about those. Let $a = \sum_{i=1}^{k} a_i$.

We know that $a + a_{k+1} > b$ because that is why our greedy algorithm skipped a_{k+1}. We also know that by our ordering of the a_i's, $a_{k+1} \leq a_k \leq a_{k-1} \leq \ldots \leq a_1$. Thus, we must have $a \geq a_{k+1}$. If a and a_{k+1} sum up to more than b, and a is at least as large as a_{k+1}, then a must be at least $b/2$. Thus we have shown that a, which is the sum of the elements we added to S so far, is at least $b/2$. The optimal solution can get b at best, so this is a 2-approximation.

A simple example of when the greedy algorithm does poorly is: $b = 100$, $A = \{51, 50, 50\}$. Here, it achieves an approximation ratio of $100/51 \approx 2$.

(d) We run the greedy solution from (b), which does not involve sorting the numbers in A. Suppose that gives us a set S whose sum is a_g. Let $a_{max} = \max_{1 \leq i \leq n} a_i$. If $a_g > a_{max}$, then we output S. Otherwise, we output $S = \{a_{max}\}$.

This algorithm clearly runs in $O(n)$ time, but why is it a 2-approximation? Note that if $a_{max} \geq b/2$, then this is certainly true because the optimal solution can do no better than b and we are doing better than $b/2$. Thus, let's consider the case where $a_{max} < b/2$. This in turn implies that $a_i < b/2$ for all i. The claim now is that $a_g > b/2$. The greedy algorithm from (b) will never terminate in a state where $a_g \leq b/2$ because as long as there is an untaken element, that element is less than $b/2$ and thus can be added. (Recall that we assumed $\sum_{i=1}^{n} a_i > b$ so there will be at least 1 untaken item). Only when $a_g > b/2$ will the algorithm terminate. Therefore, we have shown that in both cases of $a_{max} \geq b/2$ and $a_{max} < b/2$, we have that this modified greedy algorithm returns a set whose sum is more than $b/2$ and hence within a factor of 2 from optimal.

(e) Solution Not Released

Exercise 4. We know that that all of NP-complete reduce to each other. It would be nice if this meant that an approximation for one NP-hard problem would lead to another. But this is not the case. Consider the case of Minimum Vertex Cover, for which we have a 2-approximation. You can check quite easily that C is a cover in a graph $G = (V, E)$ if and only if $V - C$ is an independent set in V .

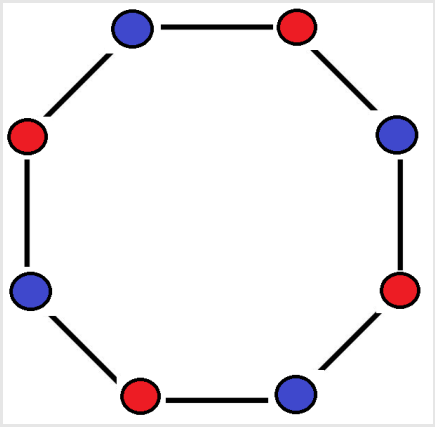
(a) Explain why this does not yield an approximation algorithm that is within a constant factor of optimal for Maximum Independent Set. That is, show that for any constant c, there exists a graph for which even if we obtain a 2-approximation of the Minimum Vertex Cover, the

corresponding independent set is not within a factor of c of the Maximum Independent Set.

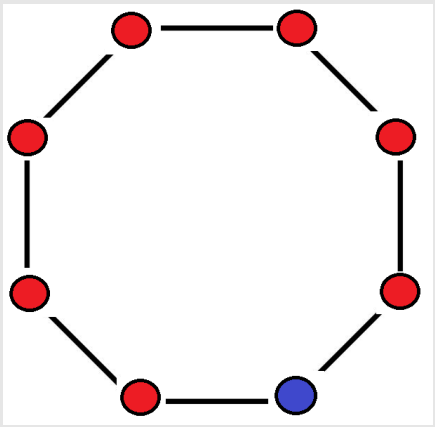
(b) The Maximum Independent Set problem and the Maximum Clique problem are related in the following way: an independent set of a graph G is a clique in the complement of G. (The complement of G is the graph that contains exactly the edges that are not in G.) Does an approximation algorithm (that is, an algorithm within a constant factor of the optimal) for the Maximum Clique problem yield an approximation algorithm (within a constant factor of optimal) for Maximum Independent Set?

Solution

(a) Consider the following cycle on 8 vertices:



The Red dots above would represent one possible minimum vertex cover. It has size 4. It's complement, the blue dots, represents the maximum independent set, which also has size 4.



However, since our minimum vertex cover algorithm is a 2-approximation, it could potentially tell me that the minimum vertex cover is the one shown above of size 7. In that case the complement would have size 1.

Therefore, this approximation algorithm for independent set using the complement of an approximation algorithm for vertex cover is off by a factor of 4.

More generally, if we take an cycle of n vertices, the minimum vertex cover and the maximum independent set are both of size $\frac{n}{2}$. However, the approximation algorithm used for vertex

cover could potentially give us a vertex cover of size $n-1$, which means our guess for the maximum independent set would be of size 1. The actual maximum independent set is of size $\frac{n}{2}$, so this algorithm for independent set based on the approximation algorithm for vertex cover is off by a factor of $\frac{n}{2}$. We can make this factor as big as we want by increasing n. We can make our approximation for maximum independent set as bad as we want! Therefore, the approximation algorithm for vertex cover cannot approximate the maximum independent set to within a constant factor.

The approximation algorithm for vertex cover does NOT give an approximation algorithm for maximum independent set because one set of vertices found is the **complement** of the other. The above family of counterexamples shows that an approximation algorithm for vertex cover may not be an approximation algorithm for maximum independent set, if we simply take the complementary set.

(b) Let k be both:

- the maximum independent set in the graph G
- the maximum clique in the complement of G

We know these two numbers are the same from the problem statement.

In addition, let n be the total number of vertices, and c be the approximation accuracy of our maximum clique algorithm. In other words, the result returned by our maximum clique algorithm is no more than ck, for $0 < c < 1$.

Suppose that I run the maximum clique algorithm on the complement of G and get a result that is of size x. My algorithm is also a c-approximation, so $x < ck$.

Our approximation for the maximum independent set in G is the **same** set of vertices, so clearly it has the same size, x. Since our maximum independent set in G also has size k, we have the relationship $x < ck$ as well because the x and k are the same as before in the maximum clique approximation.

Therefore, any c-approximation of the maximum clique in the complement of G is also a c-approximation of the maximum independent set in G.