Approximation Algorithms

1 Basic Definitions

Intuitively, an optimization problem is a computational problem with the following structure. An input x may have multiple "solutions" y, where each y has some "value". In a *maximization optimization problem*, given x, our goal is to find a solution y with maximum value (among all possible solutions). Such a y is called an *optimal solution*. The value of an optimal solution is denoted by $\text{OPT}(x)$. As an example, think of the maximum matching problem. Input x corresponds to a graph, and y corresponds to a matching in the graph. The value of y is the number of edges in the matching (i.e. the size of the matching). The goal is to output a maximum matching. And $\text{OPT}(x)$ is the size of the maximum matching in the graph x.

The concept of *minimization optimization problem* is defined analogously, with the only difference being that our goal is to find a solution y with *minimum* value.

The formal definition of an optimization problem that we give below is actually not important as long as we have a firm conceptual/intuitive understanding of the concept, as described above.

Definition (Optimization problem)**.** A *minimization optimization problem* is a function $f : \Sigma^* \times \Sigma^* \to \mathbb{R}^{\geq 0} \cup \{\text{no}\}$. If $f(x, y) = r \in \mathbb{R}^{\geq 0}$, we say that y is a *solution* to x with value r. If $f(x, y) = \text{no}$, then y is not a solution to x. We let $\text{OPT}_f(x)$ denote the minimum $f(x, y)$ among all solutions y to x.[1] We drop the subscript f, and just write $\text{OPT}(x)$, when f is clear from the context.

In a *maximization optimization problem*, $\text{OPT}_f(x)$ is defined using a maximum rather than a minimum.

We say that an optimization problem f is *computable* if there is an algorithm such that given as input $x \in \Sigma^*$, it produces as output a solution y to x such that $f(x, y) = \text{OPT}(x)$.

[1] There are a few technicalities. We will assume that f is such that every x has at least one solution y, and that the minimum always exists.

We often describe an optimization problem by describing the input and a corresponding output (i.e. a solution y such that $f(x, y) = \text{OPT}(x)$).

Definition (Optimization version of the Vertex-cover problem)**.** Given an undirected graph $G = (V, E)$, a *vertex cover* in G is a set $S \subseteq V$ such that for all edges in E, at least one of its endpoints is in S.

The VERTEX-COVER problem is the following. Given as input an undirected graph G together with an integer k, output True if and only if there is a vertex cover in G of size at most k. The corresponding language is

$$\{\langle G, k\rangle : G \text{ is a graph that has a vertex cover of size at most } k\}.$$

In the optimization version of VERTEX-COVER, we are given as input an undirected graph G and the output is a vertex cover of minimum size. We refer to this problem as MIN-VC.

Using the notation in Definition (Optimization problem), the corresponding function f is defined as follows. Let $x = \langle G\rangle$ for some graph G. If y represents a vertex cover in G, then $f(x, y)$ is defined to be the size of the set that y represents. Otherwise, $f(x, y) = \text{no}$.

Note (Examples of optimization problems)**.** Each decision problem that we have defined at the beginning of our discussion on Polynomial-Time Reductions has a natural optimization version.

Note (NP-hardness for optimization problems)**.** The complexity class NP is a set of decision problems (or languages). Similarly, the set of NP-hard problems is a set of decision problems. Given an optimization problem f, suppose it is the case that if f can be computed in polynomial time, then every decision problem in NP can be decided in polynomial time. In this case, we will abuse the definition of NP-hard and say that f is NP-hard.

Definition (Approximation algorithm)**.**

- Let f be a minimization optimization problem and let $\alpha \geq 1$ be some parameter. We say that an algorithm A is an α-approximation algorithm for f if for all instances x, $f(x, A(x)) \leq \alpha \cdot \text{OPT}(x)$.
- Let f be a maximization optimization problem and let $0 < \alpha \leq 1$ be some parameter. We say that an algorithm A is a α-approximation algorithm for f if for all instances x, $f(x, A(x)) \geq \alpha \cdot \text{OPT}(x)$.

Note that α can either be a constant or a function of $n = |x|$.

Important (Analyzing approximation algorithms)**.** When showing that a certain minimization problem has an α-approximation algorithm, you need to first present an algorithm A, and then argue that for any input x, the value of the output produced by the algorithm is within a factor α of the optimum:

$$f(x, A(x)) \leq \alpha \cdot \text{OPT}(x).$$

When doing this, it is usually hard to know exactly what the optimum value would be. So a good strategy is to find a convenient *lower bound* on the optimum, and then argue that the output of the algorithm is within a factor α of this lower bound. In other words, if $\text{LB}(x)$ denotes the lower bound (so $\text{LB}(x) \leq \text{OPT}(x)$), we want to argue that

$$f(x, A(x)) \leq \alpha \cdot \text{LB}(x).$$

For example, for the MIN-VC problem, we will use Lemma (Vertex cover vs matching) below to say that the optimum (the size of the minimum size vertex cover) is lower bounded by the size of a matching in the graph.

The same principle applies to maximization problems as well. For maximization problems, we want to find a convenient *upper bound* on the optimum.

2 Examples of Approximation Algorithms

Lemma (Vertex cover vs matching)**.** *Given a graph $G = (V, E)$, let $M \subseteq E$ be a matching in G, and let $S \subset V$ be a vertex cover in G. Then, $|S| \geq |M|$.*

Proof. Observe that in a vertex cover, one vertex cannot be incident to more than one edge of a matching. Therefore, a vertex cover must have at least $|M|$ vertices in order to touch every edge of M. (Recall that the size of a matching, $|M|$, is the number of edges in the matching.) □

Theorem (2-Approximation for MIN-VC)**.** *There is a polynomial-time 2-approximation algorithm for the optimization problem* MIN-VC*.*

Proof. We start by presenting the algorithm (known as Gavril's algorithm), which greedily chooses a *maximal* matching M in the graph, and then outputs all the vertices that are incident to an edge in M.

def $A(\langle \text{graph } G = (V, E) \rangle)$:

1. $M = \varnothing$.
2. `For each edge` $e \in E$`:`
3. `  If` $M \cup \{e\}$ `is a matching:` $M = M \cup \{e\}$.
4. $S =$ `set of all vertices incident to an edge in` M.
5. `Return` S.

We need to argue that the algorithm runs in polynomial time and that it is a 2-approximation algorithm. It is easy to see that the running-time is polynomial. We have a loop that repeats $|E|$ times, and in each iteration, we do at most $O(|E|)$ steps. So the total cost of the loop is $O(|E|^2)$. The construction of S takes $O(|V|)$ steps, so in total, the algorithm runs in polynomial time.

Now we argue that the algorithm is a 2-approximation algorithm. To do this, we need to argue that

- (i) S is indeed a valid vertex-cover,
- (ii) if S^* is a vertex cover of minimum size, then $|S| \leq 2|S^*|$.

For (i), notice that the M constructed by the algorithm is a *maximal* matching, i.e., there is no edge $e \in E$ such that $M \cup \{e\}$ is a matching. This implies that the set S is indeed a valid vertex-cover because if it was not a vertex cover, and e was an edge not covered by S, then $M \cup \{e\}$ would be a matching, contradicting the maximality of M.

For (ii), a convenient lower bound on $|S^*|$ is given by Lemma (Vertex cover vs matching): for any matching M, $|S^*| \geq |M|$. Observe that $|S| = 2|M|$. Putting the two together, we get $|S| \leq 2|S^*|$ as desired. □

Exercise (Optimality of the analysis of Gavril's Algorithm)**.** Describe an infinite family of graphs for which the above algorithm returns a vertex cover which has twice the size of a minimum vertex cover.

Solution. For any $n \geq 1$, consider a perfect matching with $2n$ vertices (i.e. a set of n disjoint edges). Then the algorithm would output all the $2n$ vertices as the vertex cover. However, there is clearly a vertex cover of size n (for each edge, pick one of its endpoints). This argument shows that our analysis in the proof of Theorem (2-Approximation for MIN-VC) is tight. The algorithm is not better than a 2-approximation

algorithm. In fact, note that just taking G to be a single edge allows us to conclude that the algorithm cannot be better than a 2-approximation algorithm (why?). We did not need to specify an infinite family of graphs.

Answer to 'why?': if the algorithm was a $(2-\epsilon)$-approximation algorithm for some $\epsilon > 0$, then **for all inputs**, the output of the algorithm would have to be within $(2-\epsilon)$ of the optimum. Therefore a single example where the gap is exactly factor 2 is enough to establish that the algorithm is not a $(2-\epsilon)$-approximation algorithm. ■

Exercise (Maximal vs maximum matching in Gavril's algorithm)**.** In Gavril's algorithm we output the vertices of a maximal matching. Suppose that instead, we output the vertices of a maximum matching. Is this algorithm still a 2-approximation algorithm?

Solution. Yes, the exact same analysis still goes through. ■

Definition (Max-cut problem)**.** Let $G = (V, E)$ be a graph. Given a coloring of the vertices with 2 colors, we say that an edge $e = \{u, v\}$ is a *cut edge* if u and v are colored differently. In the *max-cut problem*, the input is a graph G, and the output is a coloring of the vertices with 2 colors that maximizes the number of cut edges. We denote this problem by MAX-CUT.

Theorem ($(1/2)$-Approximation for MAX-CUT)**.** *There is a polynomial-time $\frac{1}{2}$-approximation algorithm for the optimization problem* MAX-CUT*.*

Proof. Here is the algorithm:

def $A(\langle \text{graph } G = (V, E)\rangle)$:

1. `Color every vertex with the same color.`
2. $c = 0$ `(number of cut edges).`
3. `While` $\exists v \in V$ `such that changing` v`'s color increases number of cut edges:`
4. `  Change` v`'s color.  Update` c`.`
5. `Return the coloring.`

We first argue that the algorithm runs in polynomial time. Note that the maximum number of cut edges possible is $|E|$. Therefore the loop repeats at most $|E|$ times. In each iteration, the number of steps we need to take is at most $O(|V|^2)$ since we can just go through every vertex once, and for each one of them, we can check all the edges incident to it. So in total, the number of steps is polynomial in the input length.

We now show that the algorithm is a $\frac{1}{2}$-approximation algorithm. It is clear that the algorithm returns a valid coloring of the vertices. Therefore, if c is the number of cut edges returned by the algorithm, all we need to show is that $c \geq \frac{1}{2}\text{OPT}(\langle G\rangle)$. We will use the trivial upper bound of m (the total number of edges) on $\text{OPT}(\langle G\rangle)$, i.e. $\text{OPT}(\langle G\rangle) \leq m$. So our goal will be to show $c \geq \frac{1}{2}m$.

Observe that in the coloring that the algorithm returns, for each $v \in V$, at least $\deg(v)/2$ edges incident to v are cut edges. To see this, notice that if there was a vertex such that this was not true, then we could change the color of the vertex to obtain a solution that has strictly more cut edges, so our algorithm would have changed the color of this vertex. From Theorem (**??**), we know that when we count the number of edges of a graph by adding up the degrees of all the vertices, we count every edge exactly twice, i.e. $2m = \sum_v \deg(v)$. In a similar way we can count the number of cut edges, which implies $2c \geq \sum_v \deg(v)/2$. The RHS of this inequality is equal to m, so we have $c \geq \frac{1}{2}m$, as desired. □

Definition (Traveling salesperson problem (TSP))**.** In the *Traveling salesperson problem*, the input is a connected graph $G = (V, E)$ together with edge costs $c : E \to \mathbb{N}$. The output is a Hamiltonian cycle that minimizes the total cost of the edges in the cycle, if one exists.

A popular variation of this problem is called METRIC-TSP. In this version of the problem, instead of outputting a Hamiltonian cycle of minimum cost, we output a "tour" that starts and ends at the same vertex and visits every vertex of the graph at least once (so the tour is allowed to visit a vertex more than once). In other words, the output is a list of vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}, v_{i_1}$ such that the vertices are not necessarily unique, all the vertices of the graph appear in the list, any two consecutive vertices in the list form an edge, and the total cost of the edges is minimized.

Theorem (2-Approximation for METRIC-TSP)**.** *There is a polynomial-time* 2*-approximation algorithm for* METRIC-TSP*.*

Proof. The algorithm first computes a minimum spanning tree, and then does a depth-first search on the tree starting from an arbitrary vertex. More precisely:

def $A(\langle$graph $G = (V, E),$ function $c : E \to \mathbb{N}\rangle)$:

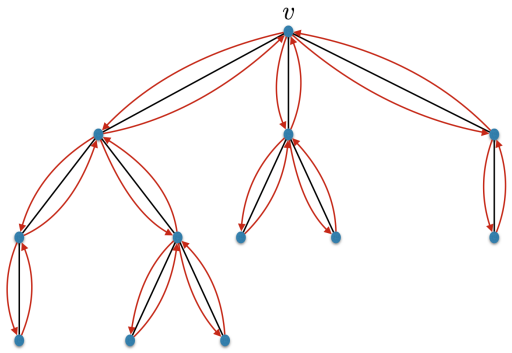
1. `Compute a MST` T `of` G`.`
2. `Let` v `be an arbitrary vertex in` V`.`
3. `Let` L `be an empty list.`
4. `Run` DFS$(\langle T, v\rangle)$.

    def DFS$(\langle$graph $G = (V, E),$ vertex $v \in V\rangle)$:

    1. `Mark` v `as visited.`
    2. `Add` v `to` L`.`
    3. `For each neighbor` u `of` v`:`
    4. `  If` u `is not marked visited, run` DFS$(\langle G, u\rangle)$.
    5. `  Add` v `to` L`.`
5. `Return` L`.`

This is clearly a polynomial-time algorithm since computing a minimum spanning tree (Theorem (**??**)) and doing a depth-first search both take polynomial time.

To see that the algorithm outputs a valid tour, note that it visits every vertex (since T is a spanning tree), and it starts and ends at the same vertex v.



Let $c(L)$ denote the total cost of the tour that the algorithm outputs. Let L^* be a optimal solution (so $c(L^*) = \text{OPT}(\langle G, c\rangle)$). Our goal is to show that $c(L) \leq 2c(L^*)$. The graph induced by L^* is a connected graph on all the vertices. Let T^* be a spanning tree

within this induced graph. It is clear that $c(L^*) \geq c(T^*)$ and this will be the convenient lower bound we use on the optimum. In other words, we'll show $c(L) \leq 2c(T^*)$. Note that $c(L) = 2c(T)$ since the tour uses every edge of T exactly twice. Furthermore, since T is a *minimum* spanning tree, $c(T) \leq c(T^*)$. Putting these together, we have $c(L) \leq 2c(T^*)$, as desired. □

Exercise (Another MIN-VC approximation)**.** Consider the following approximation algorithm to find minimum vertex cover in a connected graph G: run the DFS algorithm starting from some arbitrary vertex in the graph, and then output the set S consisting of the non-leaf vertices of the DFS tree (where the root does not count as a leaf even if it has degree 1). Show that this algorithm is a 2-approximation algorithm for MIN-VERTEX-COVER.

Hint. Show that G has a matching of size at least $|S|/2$.

Solution. To establish that the given algorithm is a 2-approximation algorithm for MIN-VC, we need to argue that

- (i) S is indeed a valid vertex-cover,
- (ii) if S^* is a vertex cover of minimum size, then $|S| \leq 2|S^*|$.

To prove (i) we claim that every edge in the graph is such that either both endpoints are in S, or one endpoint is a leaf (i.e. in $V \backslash S$) and the other is in S. Note that this claim implies S is a vertex cover. And the claim is true because there cannot be an edge in the DFS tree with both endpoints being a leaf: If two leaves were to be connected, one leaf would have to be a child of the other leaf, but a leaf cannot have a child.

To prove (ii), we want to argue that $|S| \leq 2|S^*|$. We will follow the same strategy as in the analysis of Gavril's algorithm for MIN-VC: we will use the fact that for any matching M in the graph, $|M| \leq |S^*|$. And to establish the approximation ratio, we'll show that there exists a matching M such that $|S|/2 \leq |M|$. Putting these two inequalities together we get what we want: $|S|/2 \leq |S^*|$, i.e. $|S| \leq 2|S^*|$.

So the only thing that remains to be shown is that there exists a matching M such that $|M| \geq |S|/2$. To establish this, we'll show that we can find a matching M in which every vertex in S is matched. If this is true, then it follows that $|M| \geq |S|/2$.

We can show that such an M exists algorithmically: take the root of the DFS tree and match it with any one of its children. Remove these two nodes from the tree since they are already matched. Now we are left with a forest (collection of trees). We continue in the same way for each of the trees left. We can always match the root to one of its children if the tree has at least two vertices. And if we get a tree with only one node, then notice that it must be a leaf in the original DFS tree, so we don't need to match it. ■

3 Check Your Understanding

Problem.

1. Describe the high-level steps for showing/proving that a certain minimization problem has an α-approximation algorithm.
2. What does it mean for an algorithm to be a $1/2$-approximation algorithm for the MAX-CLIQUE problem?
3. What is the relationship between a matching and a vertex cover in a graph? How does the size of a matching compare to the size of a vertex cover?
4. Describe Gavril's approximation algorithm for MIN-VERTEX-COVER.
5. Explain at a high level why Gavril's approximation algorithm has an approximation ratio of 2.

6. Describe a deterministic polynomial-time $1/2$-approximation algorithm for MAX-CUT.

7. Explain at a high level why the above algorithm has an approximation ratio of $1/2$.

8. Describe a polynomial-time 2-approximation algorithm for METRIC-TSP.

9. Explain at a high level why the approximation algorithm for METRIC-TSP that we have seen has an approximation ratio of 2.

10. Suppose you have an α-approximation algorithm for some minimization problem. At a high level, how can you show that your algorithm is not a $(\alpha - \epsilon)$-approximation algorithm for any $\epsilon > 0$?

11. True or false: The approximation algorithm for METRIC-TSP that we have seen in this chapter is not a $(2 - \epsilon)$-approximation algorithm for any constant $\epsilon > 0$.

12. True or false: Suppose A is an α-approximation algorithm for the MAX-CLIQUE problem for some $\alpha < 1$. Then it must be the case that for all input graphs $G = (V, E)$, the size of the clique returned by A is at least $\alpha \cdot |V|$.

13. True or false: Let A_1 be a 2-approximation algorithm for MIN-VC, and let A_2 be a 4-approximation algorithm for MIN-VC. Define a new approximation algorithm A_3 that runs A_1 and A_2 on the given MIN-VC instance, and outputs the smaller among the two vertex covers they find. Then A_3 is a 2-approximation algorithm.

14. True or we don't know: Suppose A is a polynomial-time algorithm for MIN-VC such that the output of A is within a factor of 1.9 of the optimum for all but 251 possible inputs. Then we cannot say that A is a 1.9-approximation algorithm for MIN-VC. But we can conclude that there exists a polynomial-time 1.9-approximation algorithm for MIN-VC.

15. True or false: It is possible for the approximation ratio of an approximation algorithm to depend on the input length n.

4 High-Order Bits

Important.

1. The difference between decision problems and optimization problems is important to understand. However, the formal definition of optimization problem will not be very useful. In the context of specific optimization problems, it will be clear what an optimum solution represents. And this will really be all we need.

2. Internalize the important note on "Analyzing approximation algorithms", and observe it in action within the proofs in this chapter. In particular, you want to be comfortable with how one goes about establishing the approximation guarantee of a given algorithm.

3. Make sure you understand how to establish that a certain algorithm is not an α-approximation algorithm. In particular, understanding the solution to Exercise (Optimality of the analysis of Gavril's Algorithm) is very important.

4. In general, coming up with approximation algorithms can be tricky. Any algorithm that is slightly non-trivial can be very hard to analyze. Therefore, often approximation algorithms are quite simple so that the approximation guarantee can be proved relatively easily. If you are asked to come up with an approximation algorithm, keep this note in mind.