An Introduction to Approximation Algorithms

Capstone Student: Grady Hollar (vhe5ak),
Capstone Professor: Mark Floryan (mrf8t)

1 Introduction

Many problems of interest in computer science lie within the realm of discrete optimization. These are problems which ask us to find solutions that either minimize or maximize a given quantity. Some of them can be easily solved in polynomial time. For example:

Given an unweighted graph $G = (V, E)$ and two vertices $s, t \in V$,
what is the shortest path between s and t?

As we all know by now, a simple breadth-first-search on G gives the optimal solution in $O(|V| + |E|)$ time. Others, however, are not so simple:

Given an undirected graph, what is the minimum number of colors needed to
color its vertices so that no two neighboring vertices share the same color?

Like the minimum vertex coloring problem, there are many optimization problems for which finding these optimal solutions is NP-hard.[1] If we have essentially no hope at solving these problems exactly, how do we handle them in practice? One idea is to develop algorithms which produce solutions that get *close* to the optimal in polynomial time. This approach has given rise to the very interesting field of approximation algorithms.

The following discussion is meant to give a brief overview of the world of approximation algorithms. We will introduce various NP-hard optimization problems and give polynomial time approximations for each of them, proving their performance bounds and introducing various techniques and results along the way. But first, we should be a bit more precise in what we mean by an approximation algorithm.

For any given optimization problem, each solution to an instance of the problem has an associated positive integer *cost*. We often denote the cost of the optimal solution by Opt. An *approximation algorithm* for the problem produces solutions whose costs are guaranteed to be within a certain factor of Opt. As with many analysis techniques in computer science, we take a worst-case approach when studying approximation algorithms.

Definition (Approximation Algorithm). For a given optimization problem and a function $\rho : \mathbb{N} \to \mathbb{Q}^+$ with $\rho \geq 1$, we say an algorithm $\mathcal{A}$ is a *ρ-factor approximation algorithm* for the problem if, given any instance of size n, $\mathcal{A}$ returns a solution with cost C such that

$$\max\left\{\frac{C}{\textsc{Opt}}, \frac{\textsc{Opt}}{C}\right\} \leq \rho(n).$$

ρ is also sometimes referred to as the *approximation ratio* or *approximation guarantee* of $\mathcal{A}$.

This definition requires that $C/\textsc{Opt} \leq \rho(n)$ for minimization problems, and $\textsc{Opt}/C \leq \rho(n)$ for maximization problems. Notice that, since we assume the costs of our solutions to be positive, these ratios are well-defined. Moreover, the respective ratios for minimization and maximization problems will always be at least 1, with equality occurring only when the solution returned by $\mathcal{A}$ is optimal. With the fundamental definitions out of the way, we'll start our discussion by examining one of the most canonical examples of an approximation algorithm.

[1] The full formal definition of an optimization problem as well as what it means for one to be NP-hard is, although interesting, rather technical and not fully necessary to know for the purposes of our discussion, so we have deferred them to Appendix A.

2 A Basic Example

As our first treatment of an approximation algorithm, we'll study the optimization version of the classic NP-hard vertex cover problem. The problem is one of the most fundamental optimization problems, and the analysis of the approximation we outline illustrates many of the core ideas that will recur throughout our discussion.

Optimization Problem (Minimum Vertex Cover). Given an undirected graph $G = (V, E)$, find a minimum cardinality *vertex cover*, that is, a subset $C \subseteq V$ such that every edge in E has at least one endpoint incident to a vertex in C and for which $|C|$ is minimized.

In this case, the cost of a solution to the problem is simply the cardinality of the cover returned. While deciding whether a graph has a cover of size at most k is NP-complete, there is a remarkably simple polynomial time algorithm that is guaranteed to return a solution at most twice as large as the optimal.

Algorithm Approx-Min-VC(G)

1. $C = \emptyset$
2. **while** $G.E \neq \emptyset$ **do**
3. Choose any $(u, v) \in G.E$.
4. $C = C \cup \{u, v\}$
5. Remove (u, v) and any edge incident to u or v from $G.E$.
6. **return** C

The steps of the algorithm are quite natural: as long as there remains an uncovered edge, we add its vertices to the cover and remove all newly covered edges from consideration. Figure 1 below depicts Approx-Min-VC run on a sample graph.

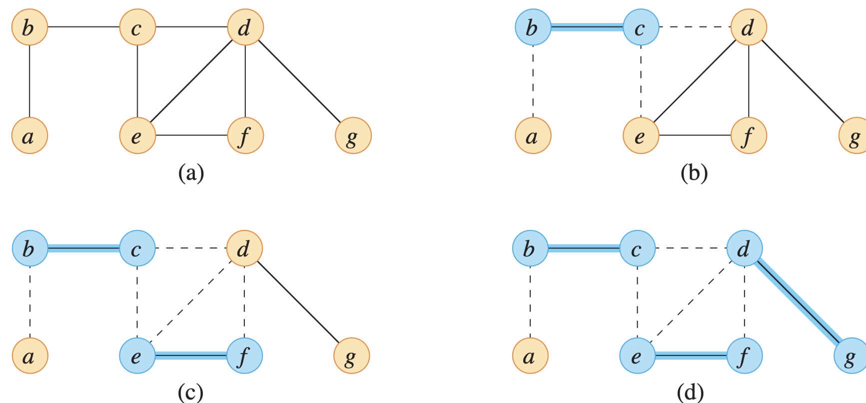

Figure 1: An instance of Approx-Min-VC

When analyzing an approximation algorithm, it's always important to first ask ourselves why the algorithm actually produces a valid solution. Can we be sure that the subset C returned by Approx-Min-VC will always be a cover for G? By design, the while loop terminates only when no edge is left uncovered by the vertices chosen so far, so C will indeed form a valid cover.

Now, the real challenge of analyzing approximation algorithms lies in proving their performance bounds. To establish an approximation guarantee, we need to compare our algorithm's solution cost to the optimal cost Opt. And yet, for the problems we consider, finding an optimal solution and computing Opt is NP-hard! This is the primary dilemma we face when designing and analyzing these kinds of algorithms.

The way we usually circumvent this seemingly impossible hurdle is by finding some way to "bridge the gap", so to speak, between the cost of our approximate solution and Opt. Instead of comparing the two directly, we try to find some other quantity that bounds both our solution and the optimal simultaneously, giving us a way to indirectly relate the two costs.

In the case of our vertex cover approximation, we'll use a matching on G to provide this indirect relation. A matching is quite similar in notion to a bipartite matching, which the reader should be familiar with, but for the sake of clarity, let's give the definition in full generality.

Definition (Matching). For an undirected graph $G = (V, E)$, a *matching* on G is a subset $M \subseteq E$ such that every vertex in V is incident to at most one edge from M.

Let's consider the set of edges chosen by Approx-Min-VC over the course of its execution (the highlighted blue edges in Figure 1 (d), e.g.). We'll refer to this set as M. Since during any given iteration of the loop, every edge incident to the one chosen is removed from future consideration, it's clear that no vertex of G can be incident to more than one edge from M. In other words, M is a matching on G. With this in mind, we're ready to prove the claimed approximation ratio for our algorithm.

Theorem 2.1. Approx-Min-VC is a 2-factor polynomial time approximation for Minimum Vertex Cover.

Proof. Since each edge is removed from E exactly once during execution, the algorithm runs in $O(|E|)$ time. Now, notice that, since the edges of M are disjoint, any vertex cover for G must contain at least one vertex for every edge in M. Since this also applies to the optimal cover, it follows that $|M| \leq \textsc{Opt}$. On the other hand, the algorithm adds both vertices of every edge in M to the cover, giving $|C| = 2 \cdot |M|$. Putting these two observations together, we see that

$$|C| = 2 \cdot |M| \leq 2 \cdot \textsc{Opt} \quad \Longrightarrow \quad \frac{|C|}{\textsc{Opt}} \leq 2,$$

proving the claimed approximation guarantee. □

Despite its simplicity, this algorithm may the best possible. It is currently conjectured that no polynomial-time algorithm for Min Vertex Cover can achieve an approximation ratio better than $2 - \varepsilon$ for any $\varepsilon > 0$. We will encounter some more interesting approximation bounds like this one throughout the course of our discussion.

3 The Traveling Salesperson Problem

3.1 Metric TSP

3.1.1 A basic 2-factor algorithm

The Traveling Salesperson Problem (TSP) is perhaps one of the best-known NP-hard optimization problems in all of computer science. Informally, it asks, given a list of cities and distances between each pair of cities, what is the shortest route that visits every city exactly once and starts and ends at the same city? In the version of the problem we consider, we will further require that the *triangle inequality* holds for all distances between cities. This is an important distinction we will touch on more later. Formally, this problem is known as *Metric* TSP, and is proposed as follows:

Optimization Problem (Metric TSP). Given a complete undirected graph $G = (V, E)$ and cost function $c : E \to \mathbb{N}$ such that the triangle inequality holds for all $u, v, w \in V$:

$$c(u, v) \leq c(u, w) + c(w, v),$$

find a minimum cost cycle in G that visits every vertex exactly once.

In this section, we'll devise a 2-factor approximation algorithm for Metric TSP. Our approach revolves around finding a minimum spanning tree of G and constructing a cycle using the MST. The algorithm works as follows:

Algorithm Approx-TSP(G, c)

1. Compute a minimum spanning tree T of G.
2. Perform a preorder traversal $\mathcal{P}$ of T.
3. Construct a Hamiltonian cycle $\mathcal{H}$ by listing the vertices of G in the order of their first appearance in $\mathcal{P}$.
4. **return** $\mathcal{H}$

Below is a depiction of the algorithm in action. Figure (a) is the original complete graph G, with Figure (b) showing a minimum spanning tree T of G. Figure (c) outlines the preorder traversal $\mathcal{P}$ of T. Lastly, Figure (d) is the resulting Hamiltonian cycle $\mathcal{H}$ constructed from $\mathcal{P}$, where the backtracking steps have been "short-cut" for more direct paths.

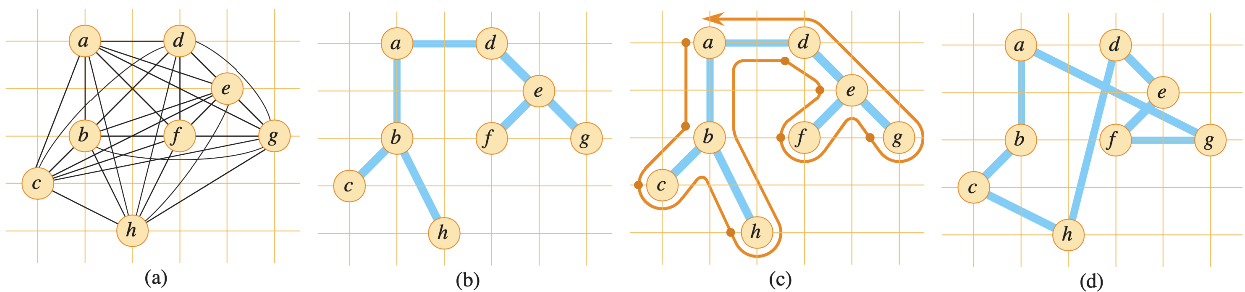

Figure 2: The stages of Approx-TSP

To place an approximation ratio on the cost of the resulting cycle $\mathcal{H}$, we begin by first establishing a bound on the cost of the traversal $\mathcal{P}$. Then, since the triangle inequality holds in G, this "short-cutting" step we use to construct $\mathcal{H}$ will not increase the cost of the traversal, and the same bound will hold for $\mathcal{H}$. Let's flesh out this approach and prove our claimed approximation ratio.

Theorem 3.1. Approx-TSP is a 2-factor polynomial time approximation for the Metric TSP problem.

Proof. There are numerous well-established and efficient algorithms to compute minimum spanning trees, so the algorithm clearly runs in polynomial time. Now, let's first consider the optimal tour O. Since O visits each vertex of G exactly once, deleting any edge from O will produce a spanning tree of G. Since T is a *minimum* spanning tree, it follows that $c(T) \leq c(O)$.

Next, note that since the traversal $\mathcal{P}$ contains each edge of T exactly twice, the cost of $\mathcal{P}$ is precisely double that of the cost of T, or rather, $c(\mathcal{P}) = 2 \cdot c(T)$. Finally, as noted above, the triangle inequality guarantees that $c(\mathcal{H}) \leq c(\mathcal{P})$. Combining these inequalities, we have

$$c(\mathcal{H}) \leq c(\mathcal{P}) = 2 \cdot c(T) \leq 2 \cdot c(O),$$

which establishes the desired approximation ratio. □

3.1.2 Improving the ratio to 3/2

Another way to describe the above algorithm is through *Eulerian tours*. Recall that an Eulerian tour of a graph is a cycle that visits every edge exactly once, allowing for revisited vertices. An equivalent formulation of Approx-TSP finds an MST, doubles each edge in the tree, performs an Eulerian tour on the doubled tree, then short-cuts the tour to find a Hamiltonian cycle. Is there a cheaper graph to perform an Eulerian tour on than the one obtained by doubling every edge in the MST?

A key result in graph theory is that a connected graph has an Eulerian tour if and only if every vertex has even degree. So, we only need to concern ourselves with somehow "remedying" the odd-degree vertices in the MST to obtain an Eulerian graph. The way we remedy these vertices is by using a *perfect matching.*

A perfect matching is identical to the usual matching we encountered earlier, with the added criterion that every vertex from the set must be covered by the edges:

Definition (Perfect Matching). Given an undirected graph $G = (V, E)$ and subset $U \subseteq V$, a *perfect matching on* U is a subset $M \subseteq E$ such that every vertex in U is incident to *exactly* one edge from M.

Now, what happens if we find a perfect matching on the odd-degree vertices of the MST and construct a new graph by adding in these edges to the tree? Since the matching is perfect, every odd vertex's degree will increase by 1, and we obtain a graph with all even-degree vertices.

Note! This is only true if we allow for doubled edges within our graph: if an edge from the matching was already in the MST to begin with, we must double that edge when adding the matching to the original tree in order to increase the relevant vertices' degrees.

Regardless, we now have a graph that we can perform an Eulerian tour on! The approach we have just described is due to Christofides. Using his idea, it's possible to ensure a 3/2 approximation guarantee for Metric TSP.

Algorithm Christofides-TSP(G, c)

1. Compute a minimum spanning tree T of G.
2. Compute a minimum cost perfect matching, M, on the odd-degree vertices of T, and set $E = T \cup M$, doubling edges as needed.
3. Perform an Eulerian tour $\mathcal{T}$ of E.
4. Construct a Hamiltonian cycle $\mathcal{H}$ by listing the vertices of G in the order of their first appearance in $\mathcal{T}$.
5. **return** $\mathcal{H}$

To prove the claimed approximation ratio, we'll need a couple preliminary results. First, how can we be sure that the odd vertices of T have a perfect matching in the first place? Our first lemma will help show that a matching always exists.

Lemma 3.2. The number of odd-degree vertices in any undirected graph is even.

Proof. Let's quickly recall that the handshake lemma tells us the sum of degrees of all vertices in an undirected graph must be even. From this, our claim follows immediately. If there were an odd number of odd-degree vertices, then the total sum of vertex degrees would be odd, which is impossible. □

It turns out that this is all we need to guarantee that M can be constructed! This is because any complete undirected graph with an even number of vertices will always have at least one perfect matching: we can simply divide its vertices into pairs any way we'd like, and the corresponding edges will form a perfect matching.

Now, the key step of proving our claimed approximation ratio revolves around placing a bound on the matching that we add to the tree. Note that our algorithm chooses a *minimum cost* perfect matching to add to T. The reason for this distinction will become clear in our proof.

Lemma 3.3. Let $G = (V, E)$ be a graph given as an instance of the Metric TSP problem. Let $U \subseteq V$ such that $|U|$ is even, and let M be a minimum cost perfect matching on U. Then, $c(M) \leq \textsc{Opt}/2$.

Proof. Let τ be an optimal TSP tour of G. Construct a tour τ' of U by listing the vertices of U in order of their appearance in τ. By the triangle inequality, we know that $c(\tau') \leq c(\tau)$. Now, note that since $|U|$ is even, τ' is the union of two perfect matchings of U, say M_1 and M_2, each consisting of alternating edges of τ. Note also that the smaller-cost of the two must be at most $c(\tau')/2$. Lastly, since M is a *minimum* perfect matching, it follows that

$$c(M) \leq \min\{c(M_1), c(M_2)\} \leq c(\tau')/2 \leq \textsc{Opt}/2. \quad \square$$

Figure 3 visualizes the steps of our proof, depicting the full tour τ of G and the shorter tour τ' of U, showing how τ' can be split into the two perfect matchings M_1 and M_2.

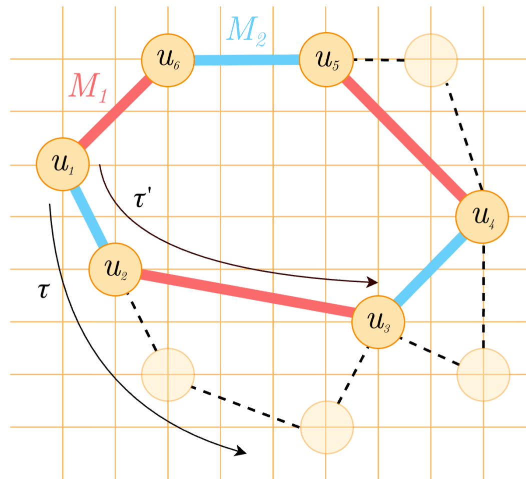

Figure 3: Construction of τ' from τ.

Since we already showed the number of odd-degree vertices of T is even, Lemma 3.3 gives us a bound on the cost of the matching M that the algorithm finds. Using this bound, we can show the final approximation guarantee.

Theorem 3.4. CHRISTOFIDES-TSP is a 3/2-factor polynomial time approximation for Metric TSP.

Proof. Though they are quite complex in nature, there are known efficient algorithms to calculate minimum-cost perfect matchings,[2] so the algorithm runs in polynomial time. As stated above, our lemmas show that $c(M) \leq \text{OPT}/2$. We also already showed in Theorem 3.1 that $c(T) \leq \text{OPT}$. Now, since the Eulerian tour $\mathcal{T}$ visits every edge of T and M exactly once, we know that $c(\mathcal{T}) = c(T) + c(M)$. Further, by the triangle inequality, the cost of the Hamiltonian cycle $\mathcal{H}$ is bounded above by $c(\mathcal{T})$. Putting everything together, we have

$$c(\mathcal{H}) \leq c(\mathcal{T}) = c(T) + c(M) \leq \text{OPT} + \frac{1}{2}\text{OPT} = \frac{3}{2}\text{OPT},$$

as desired. □

Whether the 3/2 factor can be improved further is currently one of the biggest open problems within the field of approximation algorithms.

3.2 Inapproximability of general TSP

We devised our approximations for TSP under the assumption that the triangle inequality holds on the graph's edge weights. While this holds true practically for many graphs, e.g. a network modeling roads and intersections where edge weights correspond to lengths of roads, in complete generality, this is quite a big assumption. Can we still approximate TSP without this assumption?

As it turns out, the answer is almost certainly not. More formally, we'll prove that general TSP cannot be approximated at all, assuming $P \neq NP$. Results such as these are known as *inapproximability* or *hardness of approximation* results. These results show that, for some NP-hard optimization problems, approximating them past a certain factor would prove that $P = NP$. This is quite interesting since, as we will see later, other NP-hard optimization problems can be approximated arbitrarily well. In some sense, some problems are much harder to approximate than others.

[2] An in-depth treatment of one such algorithm can be found here.

The general strategy for proving hardness of approximation results is to assume the existence of an approximation algorithm of a certain degree and to use the algorithm to create a decider for some known NP-hard problem. In the case of TSP, we will show that an approximation algorithm of *any* factor can be used to decide the Hamiltonian cycle problem in polynomial time.

Theorem 3.5. For any function $\rho(n) \geq 1$, TSP cannot be approximated within a factor of $\rho(n)$ in polynomial time, unless $P = NP$.

Proof. Assume for contradiction there is a factor $\rho(n)$ polynomial time approximation algorithm, $\mathcal{A}$, for the general TSP problem. The key idea is a reduction from the Hamiltonian cycle problem to the TSP problem that takes a graph $G = (V, E)$ on n vertices and transforms it into a complete graph G' on n vertices such that:

(1) If G *does* have a Hamiltonian cycle, the cost of an optimal TSP tour in G' is n, and

(2) If G *doesn't* have a Hamiltonian cycle, the cost of an optimal TSP tour in G' is strictly greater than $\rho(n) \cdot n$.

What happens when we run $\mathcal{A}$ on our new graph G'? Since $\mathcal{A}$ must produce a solution within $\rho(n)$ of the optimal, it must return a tour of cost $\leq \rho(n) \cdot n$ in the first case and $> \rho(n) \cdot n$ in the second. To decide whether the original graph G had a Hamiltonian cycle, we simply need to compute the cost of the given TSP tour and compare it with $\rho(n) \cdot n$. So, if we can find such a reduction, $\mathcal{A}$ can be used to solve the Hamiltonian cycle problem in polynomial time.

The reduction is quite simple. We construct G' by completing the graph G on the same vertex set V, with edge weights defined as follows: (notice that these weights will almost certainly not satisfy the triangle inequality!)

$$c(u, v) = \begin{cases} 1, & (u, v) \in E \\ \rho(n) \cdot n, & (u, v) \notin E. \end{cases}$$

If G contains a Hamiltonian cycle, then the optimal tour on G' is simply this cycle, which has cost n. If G does not contain a Hamiltonian cycle, the optimal tour on G' must use some edge *not* in G, which has cost $\rho(n) \cdot n$, making the total tour cost strictly greater than $\rho(n) \cdot n$. □

4 Randomization and LP Methods

4.1 A simple MAX-3SAT algorithm

Let's take a moment to remember the most fundamental NP-complete problem of them all: the *satisfiability* (or SAT) *problem*. Recall that a Boolean formula in k-conjunctive normal form (k-CNF) consists of clauses connected by logical and's, where each clause consists of exactly k distinct literals connected by logical or's. For example,

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_4 \vee x_5) \wedge (\neg x_3 \vee x_4 \vee x_6) \wedge (x_1 \vee x_3 \vee x_5)$$

is a 3-CNF formula with 4 clauses and 6 variables. Given a k-CNF formula, the SAT problem asks whether we can find an assignment for its variables such that the formula evaluates to true. In the optimization version of this decision problem, we'll instead try to satisfy as many clauses in the formula as possible.

Optimization Problem (MAX-3SAT). Given a 3-CNF formula[3] ϕ, find an assignment of ϕ that satisfies the largest number of clauses.

[3] In classical literature, the definition of a k-CNF formula only requires each clause to contain *at most* k literals, not exactly k. Adding in this extra condition to the formulation of the problem as we have is formally known as MAX-EkSAT, with the E standing for "exactly". It turns out, however, that the usual version of MAX-3SAT *still* admits an approximate solution with a factor identical to the one we are about to show for MAX-E3SAT, but the proof of this is far beyond the scope of this writeup. The interested reader may wish to look into the Karloff-Zwick algorithm.

How good of a solution could we get to the above problem by setting each variable in the given formula completely randomly? We'll soon see that, on average, this simple approach will surprisingly get us close to an optimal solution—quite close, in fact! But what do we mean by "on average?" We need to formalize what it means to have an approximation factor when our algorithm involves an element of randomization.

Definition (Randomized Approximation Algorithm). We say a randomized algorithm $\mathcal{A}$ is a *ρ-factor randomized approximation algorithm* for a problem if, given any instance of size n, $\mathcal{A}$ returns a random solution such that

$$\max\left\{\frac{\mathbb{E}[C]}{\textsc{Opt}}, \frac{\textsc{Opt}}{\mathbb{E}[C]}\right\} \leq \rho(n),$$

where C is the random variable which gives the cost of the solution returned by $\mathcal{A}$ on the fixed instance.

To see how we calculate this expected cost in practice, let's prove that our proposed seemingly naive approach is, interestingly, an 8/7-approximation for the MAX-3SAT problem.

Algorithm Approx-MAX-3SAT(ϕ)

1. **for** $x_i \in \phi$ **do**
2. Assign x_i randomly to be 0 or 1 with equal probability 1/2.
3. **return** the assignment.

Theorem 4.1. Approx-MAX-3SAT is an 8/7-factor randomized approximation for MAX-3SAT.

Proof. Let ϕ be a 3-CNF formula with n variables $x_1, x_2, \ldots, x_n$ and m clauses. For $1 \leq i \leq m$, define the random variable

$$Y_i = \begin{cases} 1, & \text{clause } i \text{ is satisfied} \\ 0, & \text{otherwise.} \end{cases}$$

Then, the number of clauses satisfied overall is modeled by the random variable defined by

$$Y = \sum_{i=1}^{m} Y_i,$$

so that the expected cost of the algorithm will be exactly $\mathbb{E}[Y]$. Since each literal is set to 1 with probability 1/2 and 0 with probability 1/2, and a clause is not satisfied only if all three of its literals are set to 0, we have

$$\mathbb{P}[Y_i = 0] = (1/2)^3 = 1/8 \quad \Longrightarrow \quad \mathbb{P}[Y_i = 1] = 1 - 1/8 = 7/8.$$

Computing $\mathbb{E}[Y_i]$ then gives us

$$\mathbb{E}[Y_i] = 1 \cdot 7/8 + 0 \cdot 1/8 = 7/8.$$

Now we can use the familiar properties of expected values along with the above calculations to see that

$$\mathbb{E}[Y] = \mathbb{E}\left[\sum_{i=1}^{m} Y_i\right] = \sum_{i=1}^{m} \mathbb{E}[Y_i] = \sum_{i=1}^{m} 7/8 = 7m/8.$$

Since the maximum number of clauses that can be satisfied in ϕ is m, the approximation ratio is at most $m/(7m/8) = 8/7$. □

The keen reader may have realized that in our proof we have not addressed an important possibility: what if a clause contains both a variable and its negation? Surely this will change the probabilities we calculated, and in turn the approximation factor. Let's show that, in fact, there is no need for such worries.

Proposition 4.2. Without assuming no clause contains a variable and its negation, Algorithm 1 is *still* an 8/7-approximation for MAX-3SAT.

Proof. Suppose that k of ϕ's clauses contain both a variable and its negation, and m clauses do not. For each of these remaining m clauses, we'll define the same random variable Y_i as we did before. Now, notice that for each of the k clauses containing a variable and its negation, the clause will be satisfied *no matter what assignment* for ϕ is chosen. So, the total number of satisfied clauses in ϕ is exactly

$$Y = Y_1 + Y_2 + \cdots + Y_m + k.$$

The calculation of each $\mathbb{E}[Y_i]$ is the same as before, and so

$$\mathbb{E}[Y] = \mathbb{E}\left[\sum_{i=1}^{m} Y_i + k\right] = \sum_{i=1}^{m} \mathbb{E}\left[Y_i\right] + k = 7m/8 + k.$$

Since $m + k$ is the maximum number of clauses that can be satisfied, the approximation ratio is at most

$$\frac{m+k}{7m/8+k} \leq \frac{m+k}{7m/8+7k/8} = \frac{8}{7}.$$

□

Interestingly, it has been shown that, assuming $P \neq NP$, no polynomial-time algorithm for MAX-3SAT can achieve an approximation ratio strictly better than $8/7$.[4] So, in the end, our somewhat naive approach gave us essentially the best solution we could hope for!

4.2 Weighted vertex cover and LP-rounding

Many NP-hard optimization problems we consider can be modeled as *integer programs* (or *IP*s). An integer program is identical to a linear program, with the added constraint that its variables may only take integer values. As opposed to the problem of linear programming, which is in P,[5] finding optimal solutions to integer programs is NP-hard (the very fact we can formulate NP-hard problems as integer programs proves this). But it turns out we can leverage the efficiency of linear programming to find *near*-optimal solutions for integer programs in polynomial time using a technique known as *LP-rounding*.

LP-rounding involves formulating an optimization problem as an IP, relaxing the integer constraints to turn the IP into an LP, producing an optimal solution to the LP-relaxation, and constructing a feasible solution to the original IP by rounding the variables of the LP solution. To see LP-rounding in action, we'll look at a modified version of Minimum Vertex Cover, where each vertex now has with it an associated weight.

Optimization Problem (Min-Weight Vertex Cover). Given an undirected graph $G = (V, E)$ and a weight function on vertices $w : V \to \mathbb{N}$, find a vertex cover $C \subseteq V$ of minimum weight $w(C) = \sum_{v \in C} w(v)$.

Our earlier approximation to the usual unweighted vertex cover problem will not work here, since minimizing the number of vertices in the cover does not necessarily minimize its weight. Instead, we'll use LP-rounding to approximate the problem. Let's consider the following integer program:

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} w(v) \cdot x_v \\ \text{subject to} \quad & x_u + x_v \geq 1 \qquad \forall (u,v) \in E \\ & x_v \in \{0,1\} \qquad \forall v \in V \end{aligned}$$

Notice that a feasible solution to the above program corresponds exactly to a feasible solution to min-weight vertex cover, where the corresponding cover will include vertex v if $x_v = 1$ and exclude v if $x_v = 0$. The constraint $x_u + x_v \geq 1$ ensures that this will indeed be a cover, since it forces every edge $(u, v) \in E$ to have at least one of its endpoints included in the cover. Notice also that the solution's objective function value is precisely the corresponding cover's weight.

[4] The full proof of this incredible fact is far beyond the scope of this writeup, but the interested reader can begin to learn more here.

[5] We should note here that, interestingly, the simplex algorithm we covered earlier in the semester does *not* actually give a fully polynomial time solution for linear programming. Though efficient in practice, LPs can be constructed that take exponential time using simplex. For a true polynomial time algorithm, readers may be interested in learning about the ellipsoid algorithm.

Now, we construct the *LP-relaxation* of the above IP by allowing its variables to take any real values within $[0, 1]$:

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} w(v) \cdot x_v \\ \text{subject to} \quad & x_u + x_v \geq 1 \quad \forall (u,v) \in E \\ & x_v \leq 1 \quad \forall v \in V \\ & x_v \geq 0 \quad \forall v \in V \end{aligned}$$

This is the LP we'll use to construct approximate solutions to the weighted vertex cover problem.

Algorithm Approx-Min-Weight-VC(G, w)

1. $C = \emptyset$
2. Compute an optimal solution $\overline{x}$ to the above LP-relaxation.
3. **for** $v \in G.V$ **do**
4.   **if** $\overline{x}_v \geq 1/2$ **then**
5. $C = C \cup \{v\}$
6. **return** C

Theorem 4.3. Approx-Min-Weight-VC is a 2-approximation algorithm for Min-Weight Vertex Cover.

Proof. First, why will the constructed set C actually form a cover? In the LP-relaxation, notice the first constraint ensures that for every edge $(u, v) \in E$, at least one of x_u or x_v must be at least $1/2$: if they were both less than $1/2$, the inequality would not be satisfied. But, by construction, this precisely means that every edge will have at least one of its vertices included in C.

Now, let C^* be an optimal cover and let z^* be the objective function value for the optimal solution of the LP-relaxation. We'll use z^* to place bounds on the weights of C^* and C. First, since C^* corresponds to an optimal solution of the IP, and this solution is also feasible for the LP-relaxation, it follows that $z^* \leq w(C^*)$.

Next, using a bit of clever summation tricks, we can show that performing the rounding step in our algorithm doesn't take us more than 2 factors away from the optimal LP-solution:

$$z^* = \sum_{v \in V} w(v) \cdot \overline{x}_v \;\geq \sum_{\substack{v \in V, \\ \overline{x}_v \geq 1/2}} w(v) \cdot \overline{x}_v \;\geq \sum_{\substack{v \in V, \\ \overline{x}_v \geq 1/2}} w(v) \cdot \frac{1}{2} \;\geq\; \frac{1}{2} w(C).$$

Intuitively this makes sense, as rounding a variable up to 1 at most doubles it, and rounding a variable down to 0 ensures it does not contribute to the objective function at all. Now, putting the two inequalities together, we have

$$\frac{1}{2} w(C) \leq z^* \leq w(C^*) = \textsc{Opt},$$

proving the desired ratio. □

5 A Fully Polynomial Time Approximation Scheme

5.1 PTAS's and FPTAS's

As hinted at in Section 3.2, there are some optimization problems that can be approximated arbitrarily well. To make this notion precise, we introduce the idea of *approximation schemes.*

Definition (PTAS). A *polynomial time approximation scheme* (or *PTAS*) is an algorithm $\mathcal{A}$ that takes as input not only an instance of the problem, but also some $\varepsilon > 0$ such that, for any fixed ε, $\mathcal{A}$ is a $(1 + \varepsilon)$-approximation algorithm that runs in polynomial time with respect to n, where n is the size of the instance.

Naturally, as ε grows smaller, a PTAS will almost certainly require a longer running time to satisfy the tighter-growing approximation ratios. So, we'd like to make sure this growing runtime doesn't blow up too quickly as ε shrinks.

Definition (FPTAS). We say an algorithm $\mathcal{A}$ is a *fully polynomial time approximation scheme* (or *FPTAS*) if $\mathcal{A}$ is a PTAS that runs in polynomial time with respect to both $1/\varepsilon$ and n.

Our final section will be dedicated to establishing an FPTAS for a classic NP-hard optimization problem.

5.2 Knapsack

The *knapsack problem* is another fundamental optimization problem that has been studied extensively in both exact and approximate settings. Some readers are likely to have encountered the problem before:

Optimization Problem (Knapsack). Given a collection $A = \{a_1, \ldots, a_n\}$ of n objects, a weight function $w : A \to \mathbb{N}$, a profit function $p : A \to \mathbb{N}$, and a "knapsack capacity" $W \in \mathbb{N}$, find a subset S of A whose total weight is bounded by W and whose total profit $p(S)$ is maximized.

To obtain a FPTAS for Knapsack, we'll start by proposing a dynamic programming solution to the problem. We'll denote by $DP(i, \ell)$ the weight of the lightest subset, denoted $S_{i,\ell}$, we can obtain from $\{a_1, \ldots, a_i\}$ whose profit is exactly ℓ. Then, the answer to the Knapsack problem is precisely the subset associated with the value $\max\{\ell : DP(n, \ell) \leq W\}$.

Let $P = \max\{p(a) : a \in A, p(a) \leq W\}$ be the value of the most profitable object in A that can fit in the knapsack. Clearly, nP is an upper bound for the profit of any solution we can find, so we only need to fill in values of the DP table for $1 \leq i \leq n$ and $0 \leq \ell \leq nP$. At each stage, we may either add or leave out the a_i'th element from the knapsack. If we leave it out, $DP(i, \ell)$ remains the same as $DP(i-1, \ell)$. If we add it, we add its weight to $DP(i-1, \ell - p(a_i))$ (the previous optimal solution with respect to the *remaining* profit). We can model this choice with the following recurrence:

$$DP(i, \ell) = \begin{cases} \min\{DP(i-1, \ell),\ w(a_i) + DP(i-1, \ell - p(a_i))\} & \text{if } p(a_i) \leq \ell \\ DP(i-1, \ell) & \text{otherwise,} \end{cases}$$

which gives rise to the following bottom-up DP algorithm.

Algorithm Knapsack-DP(A, w, p, W, P)

1. Initialize $DP(1, \ell)$ for $\ell \in \{0, 1, \ldots, nP\}$.
2. **for** $i = 2$ **to** n **do**
3.   **for** $\ell = 0$ **to** nP **do**
4. $add = \infty$
5.   **if** $p(a_i) \leq \ell$ **then**
6. $add = w(a_i) + DP(i-1, \ell - p(a_i))$
7. $leaveOut = DP(i-1, \ell)$
8. $DP(i, \ell) = \min\{add,\ leaveOut\}$
9.   Store the corresponding $S_{i,\ell}$.
10. $M = \max\{\ell : DP(n, \ell) \leq W\}$
11. **return** $S_{n,M}$

But wait... Doesn't this solve the Knapsack problem in polynomial time? Not quite. Notice that the runtime of $O(n^2 P)$ depends on the maximum profit P. But since this profit could be arbitrarily large with respect to n (potentially $n! \cdot 2^{n^2}$ large, for instance!), this is not a true polynomial-bounded runtime. When an algorithm is polynomially bounded in both its input size and its *numerical parameters* like the one above, we call it a *pseudo-polynomial time algorithm*.

If it just so happened that $P = q(n)$ for some polynomial q, then our algorithm would in fact run in

polynomial time. To obtain an FPTAS, we are going to exploit precisely this fact by ignoring a certain number of least significant bits of the profits of our objects to ensure P will be small enough.

Algorithm Knapsack-FPTAS$(A, w, p, W, P, \varepsilon)$

1. $K = \varepsilon \frac{P}{n}$
2. Define a new profit function p' by $p'(a_i) = \lfloor \frac{p(a_i)}{K} \rfloor$ for each a_i.
3. $S =$ Knapsack-DP(A, w, p', W, P)
4. **return** S

To show that this is indeed an FPTAS, we begin by placing a bound on its approximation ratio.

Lemma 5.1. Fix some $0 < \varepsilon < 1$. Let S be the solution returned by Knapsack-FPTAS run with parameter ε, and let O be the true optimal solution. Then,

$$\frac{p(O)}{p(S)} \leq \frac{1}{1-\varepsilon}.$$

Proof. First, due to rounding, we have that

$$p(S) = \sum_{a \in S} p(a) \geq \sum_{a \in S} K \cdot \left\lfloor \frac{p(a)}{K} \right\rfloor.$$

Next, note that since S is an optimal solution with respect to the new profits $p'(a) = \lfloor \frac{p(a)}{K} \rfloor$, it will dominate the new profit of the original optimal solution O:

$$\sum_{a \in S} K \cdot \left\lfloor \frac{p(a)}{K} \right\rfloor \geq \sum_{a \in O} K \cdot \left\lfloor \frac{p(a)}{K} \right\rfloor.$$

Lastly, note that for any single object a, due to rounding again, $K \cdot \lfloor \frac{p(a)}{K} \rfloor$ will never be smaller than $p(a)$ by more than K. That is,

$$\sum_{a \in O} K \cdot \left\lfloor \frac{p(a)}{K} \right\rfloor \geq \sum_{a \in O} \big(p(a) - K \big).$$

Putting the three previous inequalities together gives us

$$p(S) \geq \sum_{a \in O} \big(p(a) - K \big) \geq p(O) - nK.$$

Finally, by noting that $P \leq p(O)$, since naively picking the single most profitable object is a feasible solution, we see that

$$p(S) \geq p(O) - nK = p(O) - \varepsilon P \geq p(O) \cdot (1 - \varepsilon),$$

which is exactly the inequality we wanted to show. ☐

With the lemma done, we are ready for the final proof.

Theorem 5.2. Knapsack-FPTAS is a fully polynomial time approximation scheme for the Knapsack problem.

Proof. Given $\varepsilon > 0$, we'll run our algorithm with a new parameter $\delta = \frac{\varepsilon}{1+\varepsilon} < 1$. We can use the bound from Lemma 5.1 to see that

$$\frac{\text{Opt}}{p(S)} \leq \frac{1}{1-\delta} = \frac{1}{1 - \varepsilon/(1+\varepsilon)} = 1 + \varepsilon,$$

so that our solution is within $1 + \varepsilon$ of the optimal. Next, with the adjusted profits and new parameter δ, the running time of the call to Knapsack-DP will be

$$O\left(n^2 \left\lfloor \frac{P}{K} \right\rfloor\right) = O\left(n^2 \left\lfloor \frac{n}{\delta} \right\rfloor\right) = O\left(n^3 \cdot \frac{1+\varepsilon}{\varepsilon}\right) = O\left(n^3 \cdot \frac{1}{\varepsilon} + n^3\right),$$

which is polynomial in n and $1/\varepsilon$. So, our algorithm is indeed a fully polynomial time approximation scheme for the Knapsack problem! ☐

Appendix

A NP-hardness of optimization problems

All the optimization problems we consider are NP-hard. Indeed, their hardness is the very reason we look for approximate, non-optimal solutions in the first place. But what do we really mean by "NP-hard" in this case? The standard Turing machine model of computation rests on the idea of languages and decision problems—questions with simple "yes" or "no" answers, not "optimal" or "non-optimal" ones. The goal of this section is to answer this question by giving a more complete and formal definition for these optimization problems and showing how they fit in with the familiar concepts of computation theory.

Definition (Optimization Problem). An *optimization problem* Π is a 5-tuple $(\Omega, \Sigma, S, f, \textsc{goal})$, where Ω is the set of all *valid instances*, Σ is the set of all possible solutions, $S : \Omega \to \Sigma$ is a function that maps an instance $I \in \Omega$ to its set of *feasible solutions* $S(I)$, $f : \Sigma \to \mathbb{N}$ is an *objective function* that assigns a positive cost to each feasible solution, and GOAL is either MINIMIZATION or MAXIMIZATION.

An *optimal solution* for an instance $I \in \Omega$ of a minimization (maximization) problem is a feasible solution $\omega \in S(I)$ such that $f(\omega) \leq f(s)$ $(f(\omega) \geq f(s))$ for all $s \in S(I)$. We often denote the optimal objective function value $f(\omega)$ by $\textsc{Opt}(I)$, or simply $\textsc{Opt}$.

Lastly, the *size* of I, denoted by $|I|$, is defined as the number of bits needed to represent I on the tape of a Turing machine.

For a given optimization problem Π, an *optimization algorithm* for Π is an algorithm that, given an instance I of Π, produces an optimal solution for I. We would like to talk about the hardness of obtaining these optimal solutions. To do so, we'll begin by naturally associating with every optimization problem a closely-related decision problem.

Definition. For an optimization problem $(\Omega, \Sigma, S, f, \textsc{goal})$, we define its *underlying language* to be

$$\{(I, k) : I \in \Omega,\ k \in \mathbb{N},\ \exists s \in S(I) \text{ with } f(s) \leq k\}$$

if GOAL is MINIMIZATION, and analogously $(f(s) \geq k)$ if GOAL is MAXIMIZATION.

It's easy to see why the hardness of the original problem and of this language are so closely related. An optimization algorithm for a problem Π can be used to solve the underlying decision problem: given (I, k), find an optimal solution for I, use it to compute $\textsc{Opt}(I)$, and compare it with k—if Π is a minimization (maximization) problem, the answer is "yes" if $\textsc{Opt}(I) \leq k$ $(\textsc{Opt}(I) \geq k)$ and "no" otherwise. We'll use the notion of underlying languages to define new complexity classes that pertain to optimization problems specifically.

Definition (P-Optimization Problems). Define the class PO to be the set of optimization problems which have their underlying languages in P.

For NP-optimization problems we'll be a bit more particular, since we want to rule out cases where uninteresting issues such as checking input validity or computing objective function values are already hard.

Definition (NP-Optimization Problems). Define the class NPO to be the set of optimization problems $(\Omega, \Sigma, S, f, \textsc{goal})$ such that Ω is in P, for every $I \in \Omega$ the feasible solution set $S(I)$ is in P, and f is computable in polynomial time.

An almost identical argument to the previous shows that for any optimization problem in NPO, its underlying language must be in NP: given (I, k) a feasible solution s for I can be nondeterministically guessed and verified in polynomial time by computing $f(s)$ and comparing it with k. Now, with the basic complexity theory of optimization problems established, we finally specify, albeit by a slight abuse of terminology, what we mean by a "hard" optimization problem.

Definition. We say an optimization problem is *NP-hard* if its underlying language is NP-hard. If the problem is both NP-hard and contained in NPO, we say it is *NP-complete*.

B Approximation algorithms, formally

The goal of this section is to build on our definition of optimization problems from Appendix A to more formally define an approximation algorithm and use this notion to develop more distinction within the complexity classes of optimization problems. We fix an optimization problem $\Pi = (\Omega, \Sigma, S, f, \textsc{goal})$ throughout.

Definition (Approximation Ratio). For a given instance $I \in \Omega$, the *ratio* $r(s)$ of a solution $s \in S(I)$ is defined as

$$r(s) := \max\left\{\frac{f(s)}{\textsc{Opt}(I)}, \frac{\textsc{Opt}(I)}{f(s)}\right\}.$$

Notice that since we defined f to be positive, these ratios are well-defined. Further, $r(s)$ is always at least 1, with equality occurring only when s is optimal. For minimization problems, we will have $r(s) = f(s)/\textsc{Opt}(I)$, and for maximization, $r(s) = \textsc{Opt}(I)/f(s)$.

Definition (Approximation Algorithm). Let $\rho : \mathbb{N} \to \mathbb{Q}^+$ be a polynomial time computable function with $\rho \geq 1$. An algorithm $\mathcal{A}$ is said to be a *ρ-factor approximation algorithm for* Π if, for every instance $I \in \Omega$, $\mathcal{A}$ produces a feasible solution $s \in S(I)$ such that $r(s) \leq \rho(|I|)$. ρ is also sometimes referred to as the *approximation ratio* or *approximation guarantee* of $\mathcal{A}$.

When studying discrete optimization and approximation algorithms, we are interested in further separating optimization problems into classes based on how well we can approximate them.

Definition (APX-Optimization Problems). We say Π is *approximable* if there exists some polynomial time computable function $\rho \geq 1$ together with a polynomial time ρ-approximation algorithm for Π. We call the class of all approximable optimization problems APX.

Although an optimization problem might be approximable in the above sense, it very well may be the case that its best possible approximation bound is too large to be useful in practice. In turn, we are interested in problems which yield lower approximation ratios. As it turns out, some optimization problems can be approximated arbitrarily well.

Definition ($PTAS$-Optimization Problems). The class $PTAS$ consists of all optimization problems Π such that, for any $\varepsilon > 0$, there exists a $(1+\varepsilon)$-factor approximation algorithm for Π with running time $O\big(p(|I|)\big)$ for some polynomial p.

The running times of these approximation algorithms will inevitably depend on the newly introduced parameter ε, and will surely increase as ε shrinks in order to satisfy the tighter approximation ratios. We are also interested in $PTAS$ problems for which these runtimes do not increase at an unmanageable pace.

Definition ($FPTAS$-Optimization Problems). The class $FPTAS$ consists of all optimization problems Π such that, for any $\varepsilon > 0$, there exists a $(1+\varepsilon)$-factor approximation algorithm for Π with running time $O\big(p(|I|, 1/\varepsilon)\big)$ for some polynomial p.

The relationship between the classes of optimization problems we have outlined thus far is quite natural:

$$PO \subseteq FPTAS \subseteq PTAS \subseteq APX \subseteq NPO.$$

If $P \neq NP$, these inclusions are proper. Indeed, throughout the course of our discussion we have either proven or cited proof of all but one of the distinctions of the classes under this assumption. In section 3.2, we show that general TSP, while certainly in NPO, is not even a member of APX. The approximation bounds we mention for Vertex Cover and MAX-E3SAT in sections 2 and 4.1 show that, while the two problems are in APX, they cannot reside in $PTAS$. In section 5.2, we show that Knapsack is in $FPTAS$ while not being a member of PO. For an example of a problem that lies in $PTAS$ but not $FPTAS$, readers are encouraged to look into *minimum makespan scheduling*. A thorough coverage of the problem and its exclusion from $FPTAS$ can be found in Williamson and Shmoys' seminal text.

References

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, 4th ed. MIT Press, Cambridge, MA, 2022.

V. V. Vazirani. *Approximation Algorithms.* Springer, Berlin / Heidelberg, 2001.

D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms.* Cambridge University Press, Cambridge, UK, 2011.

D. S. Hochbaum (ed.). *Approximation Algorithms for NP-Hard Problems.* PWS Publishing, Boston, MA, 1997.

E. W. Mayr, H.-J. Prömel, and A. Steger (eds.). *Lectures on Proof Verification and Approximation Algorithms.* Lecture Notes in Computer Science, vol. 1367. Springer-Verlag, Berlin–Heidelberg, 1998.

S. Khot and O. Regev. Vertex cover might be hard to approximate to within $2 - \varepsilon$. *Journal of Computer and System Sciences*, 74(3):335–349, May 2008.

H. J. Karloff and U. Zwick. A 7/8-approximation algorithm for MAX 3SAT? In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science* (FOCS), pages 406–415, 1997.

C. Chekuri (lecturer) and R. Samdani (scribe). Min Cost Perfect Matching (CS 598CSC: Topics in Combinatorial Optimization, Lecture 11). University of Illinois Urbana–Champaign, Feb. 2010.

V. Klee and G. J. Minty. How good is the simplex algorithm? In O. Shisha (ed.), *Inequalities III*, Academic Press, 1972 (technical report version dated Feb. 1970).

S. Arora (lecturer) and S. Brahma (scribe). The Ellipsoid Algorithm for Linear Programming (COS 521 lecture notes). Princeton University, 2005.

Wikipedia contributors. Handshaking lemma. *Wikipedia.*

Wikipedia contributors. MAX-3SAT. *Wikipedia.*