APPROXIMATION ALGORITHMS

AMIR YEHUDAYOFF

Approximations are essential to science. They allow us to understand and to communicate. The number π does not have a finite description, but 3.14159265359 is good enough for most applications. Classical mechanics is an approximation but it is often good enough.

Exercise. *Find more examples of approximations.*

We use algorithms to analyze data and make better decisions. Algorithms have costs; time, memory size, energy, etc. To reduce costs, it is sometimes beneficial to settle for an approximate solution. Moreover, sometimes we do not have the full information about the system we are trying to analyze so we can not hope to obtain an accurate solution (e.g., streaming). In such cases, we have to approximate.

1. Greedy algorithms

Greedy algorithms make "locally optimal" decisions. They are a natural first choice in many scenarios. Sometimes they lead to accurate solutions and sometimes only to approximate solutions. We shall not try to provide a formal definition for greedy algorithms, but we shall cover a few examples (more on this in the exercise session).

1.1. **Set cover.** In the set cover problem, the input is a collection of sets $S = \{s_1, \ldots, s_m\}$ over the domain $[n] := \{1, 2, \ldots, n\}$, where each set s has a weight $w_s \geq 0$. A cover is a collection $I \subseteq [m]$ so that $\bigcup_{j \in I} s_j = [n]$. The weight of a cover is $\sum_{j \in I} w_{s_j}$. The output is the minimum weight of a cover.

The set cover problem is NP-hard so we do not know how to solve it efficiently (and it is unlikely that we will). What happens if we greedily try to solve it?

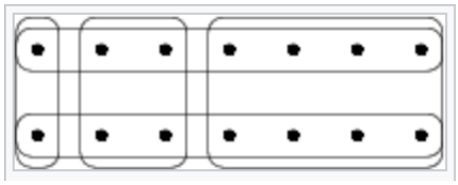
greedy set cover
  $I = \emptyset$
  $T = S$
  while I is not a cover
    let j be so that t_j is a minimizer of $\frac{w_t}{|t|}$ / division by zero is infinite
    set $I = I \cup \{j\}$
    set $t = t \setminus t_j$ for all $t \in T$
  output I

Exercise. *Describe the choice of j in words.*

Remark. *We also need to add a stopping condition when all sets in T are empty.*

The algorithm outputs a cover (if one exists). Denote by $g(S, w)$ the weight of the cover the algorithm outputs (g for greedy). We need to compare it to the minimum weight of a cover $opt = opt(S, w)$.

Example.



The universe size is 14 and all weights are 1. Denote the set of size two by s_1, the set of size four by s_2, the set of size eight by s_3, and the two sets of size seven by s_0, s_0'. The greedy algorithms chooses s_3, s_2, s_1 with weight 3 but the optimal cover is s_0, s_0' with weight 2.

Remark. *We measure the quality of the approximation via the approximation ratio*

$$\frac{g(S,w)}{opt(S,w)} = \frac{3}{2}.$$

The approximation ratio is always at least one and the smaller it is, the better the solution is.

Exercise. *Find an example with $O(\log n)$ sets over the domain $[n]$ so that the greedy algorithm algorithm has approximation ratio $\Omega(\log n)$.*

Theorem 1. *For every S, w over $[n]$,*

$$\frac{g(S,w)}{opt(S,w)} \leq H_n$$

where $H_n = \sum_{i=1}^{n} \frac{1}{i}$ is the n'th harmonic number.

Remark. *For all n,*

$$\ln(n) \leq H_n \leq \ln(n) + 1.$$

(Can you prove this?) The limit

$$\gamma := \lim_{n \to \infty} H_n - \ln(n)$$

exists and $\gamma \approx 0.5772156649$ is the Euler–Mascheroni constant.

Proof of theorem. Denote by I_* the set of indices of an optimal cover. The algorithm operates in iterations $k = 1, 2, \ldots$. Denote by n_k the number of elements not covered by the greedy algorithm at the start of iteration k; so $n_1 = n$. Denote by $s_j(k)$ the part of s_j that "remains" at the start of iteration k; so $s_j(1) = s_j$.

We need to relate *opt* to the choices of the algorithm. For every k,

$$\begin{aligned} opt &= \sum_{j \in I_*} w_j \\ &= \sum_{j \in I_*} |s_j(k)| \frac{w_j}{|s_j(k)|} \\ &\geq \left(\min_j \frac{w_j}{|s_j(k)|} \right) \cdot \sum_{j \in I_*} |s_j(k)| \\ (I_* \text{ is a cover}) \qquad &\geq \left(\min_j \frac{w_j}{|s_j(k)|} \right) \cdot n_k. \end{aligned}$$

It follows that the "j_k that is chosen by the algorithm" satisfies

$$\frac{w_{j_k}}{|s_{j_k}(k)|} \leq \frac{opt}{n_k}.$$

By definition,

$$n_{k+1} = n_k - |s_{j_k}(k)|.$$

So,

$$\begin{aligned} w_{j_k} &\leq (n_k - n_{k+1})\frac{opt}{n_k} \\ &\leq opt\Big(\frac{1}{n_k} + \frac{1}{n_k - 1} + \frac{1}{n_k - 2} + \ldots + \frac{1}{n_{k+1} + 1}\Big); \end{aligned}$$

in the last line there are $n_k - n_{k+1}$ terms which are not smaller than $\frac{1}{n_k}$.

Summing over all k,

$$\begin{aligned} g &= \sum_k w_{j_k} \\ &\leq opt \sum_k \Big(\frac{1}{n_k} + \frac{1}{n_k - 1} + \frac{1}{n_k - 2} + \ldots + \frac{1}{n_{k+1} + 1}\Big) \\ &= opt \sum_{i=1}^{n} \frac{1}{i} = opt \cdot H_n. \end{aligned}$$

□

Remark. *We now understand the approximation ratio of the greedy algorithm; it is* $\Theta(\log n)$.

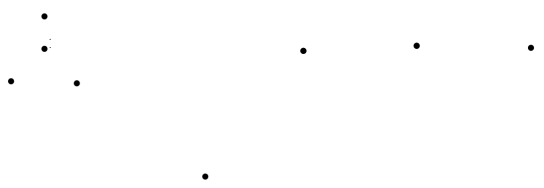
The greedy algorithm is just one example of an efficient algorithm. It is computationally hard to find a better approximation, as the following two hardness theorems show.

Theorem. *If there is* $c < 1$ *so that there is a* $c \ln n$*-approximation algorithm for set cover (with all weights being one) over the domain* $[n]$*, then NP is contained in time* $n^{O(\log \log n)}$.

Theorem. *There is* $c < 1$ *so that if there is a* $c \ln n$*-approximation algorithm for set cover (with all weights being one) over the domain* $[n]$*, then* $P = NP$.

1.2. **k-centers.** Clustering is an example of an "unsupervised machine learning problem". The input is a collection of data points and the goal is to find the clusters in the data. Draw some examples. There are many clustering algorithms, and there are also many ways to define the goal of clustering.

Example. *Consider the example (with Euclidean distances). How many clusters are there? If there are two, what are they? If there are three, what are they?*



In the k-centers problem, the input is an integer k and a metric[1] d on the point-set V (of size n). The objective is to find a set $C \subset V$ of k centers ($|C| = k$) that minimizes

$$\max_{x \in V} d(x, C)$$

where

$$d(x, U) = \min_{y \in U} d(x, y).$$

Remark. *There are other objectives that we can choose, like the sum of distances, the sum of the squares of the distances, etc. Each choice is important in some particular setting.*

greedy k-centers
set $C = \{x_0\}$ where $x_0 \in V$ is arbitrary
while $|C| < k$
  let x be a maximizer of $y \mapsto d(y, C)$
  set $C = C \cup \{x\}$
output C

Exercise. *Describe the choice of x in words.*

Exercise. *Run the greedy algorithm on the example above with $k = 3$.*

Theorem 2. *The greedy algorithm has approximation ratio two; that is, for all d,*

$$\frac{g(d)}{opt(d)} \leq 2.$$

Proof. Let C_* be an optimal solution and let

$$r_* = \max_x d(x, C_*).$$

Define k clusters; for $c_* \in C_*$, let $V(c_*)$ be the set of point that are closer to c_* than to any other center in C_* (break ties arbitrarily). Let C be the output of the greedy algorithm and let

$$r := \max_x d(x, C).$$

The analysis has two cases. If every cluster $V(c_*)$ contains a point in C, then for every $x \in V$, there is $c_* \in C_*$ and $c \in V(c_*)$ so that

$$d(x, c) \leq d(x, c_*) + d(c_*, c) \leq 2r_*.$$

It follows that

$$r \leq 2r_*.$$

Otherwise, the algorithm chooses two points c_1, c_2 in the same cluster. At this point in time (when both are in C), by the choice of the algorithm,

$$\max_y d(y, C) \leq d(c_1, c_2) \leq 2r_*.$$

□

Again, a better approximation ratio is hard to find.

Theorem. *If there a poly-time algorithm for the k-center problem with approximation ratio $\rho < 2$ then $P = NP$.*

[1] $d(x, x) = 0$, $d(x, y) = d(y, x)$ and $d(x, y) \leq d(x, z) + d(z, y)$.

1.3. **Metric TSP.** In the traveling sales person (TSP) problem, the input is a cost function $c : V \times V \to \mathbb{R}_+ \cup \{\infty\}$ so that $c(x, y) = c(y, x)$. We think of $c(x, y)$ as the cost of going from x to y. A tour τ in the graph is a cycle that passes through every vertex exactly once. The cost of a tour is the sum of the costs of its edges. The goal is to find a tour with minimum cost.

Remark. *The TSP problem was studied in many works, and it is well-known that it is NP-complete when all weights are one. It is even hard to approximate (more on this later on).*

We therefore consider the metric version of the problem, where c is assumed to be a metric.

greedy algorithm for metric TSP
choose $x_1 \neq x_2$ as minimizers of $(x, y) \mapsto c(x, y)$
set the first cycle to be $x_1 \to x_2 \to x_1$
repeat the greedy step until we have a tour:
given $x_1 \to x_2 \to \cdots \to x_k \to x_1$,
let x be in the cycle and y not in the cycle so that $c(x, y)$ is minimum
let x' be the point after x in the cycle
extend the cycle by adding y between x and x'
output the tour

Theorem 3. *The greedy algorithm (a.k.a. nearest addition algorithm) for metric TSP is a 2-approximation.*

Proof. The algorithm operates in iterations $k = 1, 2, \ldots$. In iteration k, the algorithm chooses an edge $e_k = \{x_k, y_k\}$. For all k, the edges $e_1, \ldots, e_k$ form a tree. The choice of the algorithm is the same as Prim's algorithm for finding the minimum spanning trees. The minimum cost of a TSP is at least the cost of a minimum spanning tree (a tour minus an edge is a tree). It follows that

$$opt \geq \sum_k c(x_k, y_k).$$

The cost of the first tour is $2c(x_1, x_2)$. In an iteration, if x, y are chosen then the cost of the tour increases by

$$c(x, y) + (c(y, x') - c(x, x')) \leq c(x, y) + c(x, y),$$

where we used the triangle inequality. It follows that the cost of the tour the algorithm found is at most $2 \cdot opt$. □

Summary

— Approximations are useful to save on resources, and sometimes we can not hope to get a non-approximate solution.

— We measured the quality of an approximation algorithm for a minimization problem via the approximation ratio

$$\rho(n) = \max \frac{A(x)}{opt(x)}$$

where x is of size n.

— Greedy algorithms are generally efficient and sometimes give approximate solutions. The analysis of a greedy algorithm is not always trivial.

2. Better approximations by rounding

2.1. **The basics.** Let us start with some basic definitions behind approximation algorithms. Let us consider a maximization problem.

Definition. *An algorithm A for a maximization problem has approximation ratio $\alpha(n)$ if for every input X of size n,*

$$\frac{A(x)}{opt(X)} \geq \alpha(n).$$

For minimization problems, the condition is

$$\frac{A(x)}{opt(X)} \leq \alpha(n).$$

Definition (PTAS)**.** *A polynomial-time approximation scheme is a family of algorithms $\{A_\varepsilon\}$, so that for each $\varepsilon > 0$, the algorithm A_ε runs in poly-time in n and has approximation ratio at least $1 - \varepsilon$ for a maximization problem. For minimization problems, it is $1 + \varepsilon$.*

Definition (FPTAS)**.** *A PTAS $\{A_\varepsilon\}$ is called a fully PTAS if for every $\varepsilon > 0$, the running time of A_ε is polynomial in $\frac{1}{\varepsilon}$.*

Example. *How can we compute the digits of π? We can use polygons and then the running time is $\frac{1}{\varepsilon}$. We can use calculus and then the running time $\log \frac{1}{\varepsilon}$. The latter allows to compute the first n digits of π in poly-time.*

Remark. *There are two different conventions for approximation ratios. In the option we use, we always measure $\frac{A(x)}{opt(x)}$ which is ≤ 1 for maximization problems and ≥ 1 for minimization problems. In the option we do not use, we make the approximation ratio ≥ 1 always by measuring $\frac{opt(x)}{A(x)}$ for maximization problems.*

2.2. **Dynamic programming.** Dynamic programming is a family of algorithms that produce a solution by carefully combining sub-solutions. It leads to efficient and optimal solutions for shortest path problems.

As an example, we consider the knapsack problem. The knapsack problem is a basic combinatorial optimization problem. There is a list of items $[n]$, and each item i has value v_i and size $s_i \geq 0$ (both are integers). There is a knapsack of capacity B. The goal is to compute

$$\max\Big\{\sum_{i\in T} v_i : \sum_{i\in T} s_i \leq B\Big\}.$$

Remark. *The knapsack problem is NP-complete when the numbers are given in binary.*

The following dynamic program finds an optimal solution for each interval of the form $[j]$. For each j, the programs keeps a list of pairs (s, v) which indicate that there is a solution of size s and value v in $[j]$.

dynamic programming for knapsack
  set $A(1) = (s_1, w_1)$
  for $j = 2, \ldots, n$:
    set $A(j) = A(j-1)$
    for all (s, v) in $A(j-1)$:

if $s + s_j \leq B$:
add $(s + s_j, v + v_j)$ to $A(j)$
keep only the dominating pairs[2] in $A(j)$
output $\max\{w : \exists (w, s) \in A(n)\}$

Exercise. *The dominating relation is transitive.*

Corollary. *For each j, the size of $A(j)$ is at most $\min\{B + 1, V + 1\}$ where $V = \sum_i v_i$.*

Exercise. *The dynamic program for knapsack is correct.*

Remark. *The running time could be exponential in the input length (when the numbers are encoded in binary).*

2.3. **Approximating knapsack.** The main idea is to round the values and then run the dynamic program from above.

FPTAS for knapsack
set $M = \max_i v_i$
run the dynamic program with $v'_i = \lfloor \frac{n v_i}{\varepsilon M} \rfloor$

Remark. $\frac{n v_i}{\varepsilon M} - 1 \leq v'_i \leq \frac{n v_i}{\varepsilon M}$.

Exercise. $\sum_i v'_i \leq \varepsilon n^2$.

Corollary. *The running time of the algorithm is polynomial in n and $\frac{1}{\varepsilon}$.*

Theorem. *The approximation ratio of the algorithm is at least $1 - \varepsilon$.*

Proof. Let S_* be an optimal set of item and let S be the output of the algorithm.

$$\begin{aligned}
\sum_{i \in S} v_i &\geq \frac{\varepsilon M}{n} \sum_{i \in S} v'_i \\
\text{(DP is optimal)} \qquad &\geq \frac{\varepsilon M}{n} \sum_{i \in S_*} v'_i \\
&\geq \sum_{i \in S_*} \left(v_i - \frac{\varepsilon M}{n} \right) \\
&\geq opt - \varepsilon M \\
&\geq (1 - \varepsilon) opt.
\end{aligned}$$

□

Remark. *Rounding leads to an FPTAS for knapsack.*

2.4. **Scheduling on identical machines (a.k.a. makespan).** There are n jobs. Job j requires processing time $p_j \in \mathbb{N}$ without interruptions. There are m machines. Each machine can process a single job at a time. In every schedule for the jobs, every job j has a completion time C_j. Our goal is to minimize $\max_j C_j$.

We can solve makespan with a local search until we find a local minimum. A local move is as follows. Let j be a job that is finished last. If we can reassign j to

[2]The pair (s, w) dominates (s', w') if $s \leq s'$ and $w \geq w'$

a different machine so that it finished earlier, do so. A local search starts with any schedule and performs local moves until no move is left.

Remark. *We can assume that there are no "holes" in the schedule.*

Remark. *Finding local minimums is often a task that we can perform efficiently. In some cases, like convex functions on convex domains, a local minimum is also a global minimum. In most scenarios, however, there are many local minima and we do not know that we found an optimal solution.*

Denote by C_* an optimal makespan.

Exercise. $C_* \geq \max_j p_j$.

Exercise. $C_* \geq \frac{1}{m} \sum_j p_j$.

Denote by C a locally optimal schedule. Denote by ℓ a last job in this schedule. Denote by C_ℓ its completion time, and by $S_\ell = C_\ell - p_\ell$ its start time.

Exercise. $p_\ell \leq C_*$.

Exercise (all machines are busy up to time S_ℓ)**.** $S_\ell \leq \frac{1}{m} \sum_{j \neq \ell} p_j \leq C_*$.

Corollary. *A local search algorithm has approximation ratio at most two.*

Exercise (running time)**.** *If we move the last job to the machine that is available first, then no job is moved twice and the number of moves is at most n.*

We now improve the approximation ratio to $1 + \varepsilon$. The idea is to partition the jobs to "long" and "short" according to a threshold. Job s is short if

$$p_s \leq \frac{\varepsilon}{m} \sum_j p_j.$$

Exercise. *There are at most $\frac{m}{\varepsilon}$ long jobs.*

First, the algorithm finds the optimal scheduling for the long jobs. Second, the algorithm schedules the short jobs by local search.

Remark. *The total time is at most poly in $m^{m/\varepsilon} + n$ which is polynomial-time when m, ε are constant.*

Claim. *The approximation ratio is at most $1 + \varepsilon$.*

Proof. Above we saw that a local search leads to makespan of

$$p_\ell + \frac{1}{m} \sum_{j \neq \ell} p_j.$$

If job ℓ is short then this is at most $(1 + \varepsilon)opt$ as needed. If the last job is long then the solution is optimal (even just for the long jobs). □

Remark. *There is an algorithm with running time $n^{O(1/\varepsilon^2)}$ when the number of machines is also a parameter. Let us describe a few related ideas.*
— The first observation is that we just need to find $(1 + \varepsilon)$-approximation for the long jobs.
— The second idea is that if it suffices to design an algorithm that is given also a target time T as input and outputs either "T is not possible" or a scheduling with makespan at most $(1 + \varepsilon)T$. Given such an algorithm, we get the final algorithm via binary search.
— The third idea is to round the jobs costs to multiples of $\varepsilon^2 T$.

Remark. *There is no FPTAS for makespan unless P=NP.*

Summary

By rounding or thresholding the data, we can get efficient algorithms with $1+\varepsilon$ approximation ratios. The best scenario is that we get an FPTAS, but sometimes we only get a PTAS.

3. Linear programming

Linear programming (LP) is an important algorithmic perspective. An LP is an optimization problem of the form

$$\max \langle c, x\rangle : Ax \leq b$$

where $x, c \in \mathbb{R}^n$ and $b \in \mathbb{R}^k$ and $A \in \mathbb{R}^{k\times n}$, and we use the partial order $x \leq y$ iff $x_i \leq y_i$ for all i.

Remark. *Geometrically $D = \{x : Ax \leq b\}$ is the intersection of k half-spaces in $\mathbb{R}^n$. So it is a convex domain. If it is also bounded, it is a polytope. It could also be empty.*

Remark. *Denote by $\mathbb{S}^{n-1}$ the unit sphere. For $u \in \mathbb{S}^{n-1}$, the support function of the polytope $D \subset \mathbb{R}^n$ is*

$$h_D(u) := \max\{\langle u, x\rangle : x \in D\}.$$

The supporting hyperplane $H(D, u)$ is

$$H(D, u) := u^{\perp} + h_D(u).$$

The intersection of $H(D, u)$ and D is a face of D. Every face is also a polytope. The dimension of a polytope is the minimum dimension of an affine space that contains it. Faces of co-dimension one are called facets. Faces of dimension zero are called vertices.

Exercise. *If D is a polytope then the maximum is attained at a vertex of D.*

Remark. *An important idea in LP is duality. The dual of*

$$\max \langle c, x\rangle : Ax \leq b, x \geq 0$$

is

$$\max \langle b, y\rangle : A^T y \geq c, y \geq 0.$$

Strong duality of LP says that the two values are the same; see appendix.

Remark. *The condition $x \geq 0$ is needed here.*

Remark. *There are several efficient algorithms for solving LPs; ellipsoid algorithm and interior-point methods.*

3.1. **Vertex cover.** In the vertex cover problem, the input is a graph $G = (V, E)$ and a weight function $w : V \to \mathbb{R}_+$. A cover $C \subset V$ is a set so that

$$\forall e \in E\ \exists v \in V\ v \in e.$$

The goal is the find the cover of minimum weight.

Remark. *This is a well-known combinatorial problem (NP-complete for $w \equiv 1$).*

Remark. *We can try to solve combinatorial optimization problems by "relaxing to an LP" and then "rounding the solution". This is a powerful paradigm.*

$$\textit{combinatorial optimization} \to_{relax} \textit{LP} \to_{round} \textit{solution}$$

How can we relax vertex cover to an LP? Think of $x \in \mathbb{R}^V$ as the indicator of a cover. Add the constraints $x_v \in [0, 1]$ for all v. The cover condition becomes

$$\forall e = \{v, u\} \in E\ x_u + x_v \geq 1.$$

Such an x is called fractional cover. If a fractional cover has integer values, then it is a cover.

Remark. *If opt is the minimum weight of a cover and opt^* of a fractional cover then $opt^* \leq opt$.*

Remark. *We can find a fractional cover x efficiently. The remaining algorithmic step is to round x to a cover C.*

Exercise. *Given $x \in [0,1]^V$, how can we round it to $\{0,1\}^V$?*

Claim. *If x is an optimal fractional cover, and C is the natural rounding of x then*

$$w(C) \leq 2 \cdot opt.$$

Proof. For all v, we have $1_{v \in C} \leq 2x_v$. So,

$$w(C) \leq 2 \sum_v x_v = 2 \cdot opt^* \leq 2 \cdot opt.$$

□

Example. *The cover of a triangle has size two, and the fractional cover has size 3/2.*

Remark. *The quantity*

$$\sup \frac{opt}{opt^*}$$

is called the integrality gap. It is the integrality gap of the LP and not of the "original problem". The claim say that the integrality gap is at most two.

Exercise. *Show that the integrality gap is indeed two.*

Remark. *Vertex cover is a special case of set cover where the domain is E with sets $S_v = \{e : v \in e\}$.*

Remark. *A sequence of works showed that a 1.36-approximation of vertex cover is NP-hard [Dinur-Safra]. There is another notion of hardness (Unique Games); It is known that a $(2-\varepsilon)$-approximation is UG-hard [Khot, Regev].*

3.2. **Set cover.** Relax a set cover instance F over $[n]$ (with $w \equiv 1$) to the LP:

$$\min \sum_{s \in F} x_s$$

subject to

$$x \geq 0$$

$$\sum_{s \in F : s \ni i} x_s \geq 1 \ \forall i \in [n]$$

A solution to the above is called a fractional cover. Again, we have *opt* and opt^*.

Primal rounding. How can we round a fractional cover x to a cover? The frequency of $i \in [n]$ is

$$f_i = \#\{s \in F : i \in s\}.$$

The frequency is

$$f = \max_i f_i.$$

Now, round x to

$$C = \{s \in F : x_s \geq \tfrac{1}{f}\}.$$

Exercise. *C is a cover.*

Exercise. *$w(C) \leq f \cdot opt$.*

Dual rounding. We can also round the dual program! The dual is

$$\max \sum_i y_i$$

subject to

$$y \geq 0$$
$$\sum_{i \in s} y_i \leq 1 \ \forall s \in F$$

Remark. *This is equivalent to*

$$\min_y \max_s \sum_{i \in s} y_i$$

where y is a probability distribution on $[n]$. In other words, we want to find a distribution on $[n]$ that gives as little mass as possible to each of the sets.

The rounding of an optimal fractional y^* is as follows:

$$C = \Big\{ s \in F : \sum_{i \in s} y_i^* = 1 \Big\}.$$

Remark. *This is the set of "saturated" inequalities; it is a general rounding mechanism.*

Claim. *C is a cover (if there is a cover).*

Proof. If i is not covered by C then

$$\varepsilon := \max_{s \ni i} \Big(1 - \sum_{j \in s} y_j \Big) > 0.$$

Define a new solution y by

$$y_i = y_i^* + \tfrac{\varepsilon}{2}$$

and leaving the other entries the same. It can be verified that y is still a fractional cover with higher value than y^*, which is a contraction. □

Claim. *$|C| \leq f \cdot opt$.*

Proof.

$$\begin{aligned} |C| &= \sum_{s \in C} 1 \\ &= \sum_{s \in C} \sum_{i \in s} y_i^* \\ &= \sum_i y_i^* f_i \\ &\leq f \cdot opt^* \\ &\leq f \cdot opt. \end{aligned}$$

□

Remark. *The integrality gap of set cover is H_n [Lovász]. Here is an example with $\Omega(\log n)$ gap. The domain is $U = \mathbb{F}_2^k \setminus \{0\}$. The family F is defined as follows: for each $z \in \mathbb{F}_2^k$, consider the set*

$$s_z = \left\{ x : \sum_i x_i z_i = 1 \mod 2 \right\}$$

and let $w \equiv 1$.

Exercise: There is a fractional solution of value two: $x_s = 2^{1-k}$.

Exercise: A cover has size at least k.

Summary

We can relax an integer program to a linear program. The LP can be solved efficiently. We then need to round the LP to an integer value, which causes some loss (we just can an approximate solution).

Appendix: strong duality

Here we explain strong duality, and provide a geometric picture. We have our data A, b, c and we interested in

$$\max \langle c, x \rangle : Ax \leq b, x \geq 0.$$

Let us see how to relate this optimization problem to a decision problem. For each value $v \in \mathbb{R}$, we can look at

$$P_v := \{x : Ax \leq b, x \geq 0, \langle c, x \rangle \geq v\}.$$

Assume that the LP is feasible and bounded; that is, for small enough v we have $P_v \neq \emptyset$ and for large enough v we have $P_v = \emptyset$. Due to monotonicity (i.e., if $v < v'$ then $P_v \supseteq P_{v'}$), the set $\{v : P_v \neq \emptyset\}$ is a ray of the form

$$\{v : P_v \neq \emptyset\} := (-\infty, v_P].$$

The polytope P_v is non-empty iff the maximization problem has solution $\geq v$. So,

$$v_P = \max\{\langle c, x \rangle : Ax \leq b, x \geq 0\}.$$

For the dual problem, look at

$$Q_v := \{y : A^T x \geq c, y \geq 0, \langle b, y \rangle \leq v\}.$$

Now, if $v < v'$ then $Q_v \subseteq Q_{v'}$. First, we claim that the dual is feasible and bounded. We already say that (weak duality)

$$\{v : Q_v \neq \emptyset\} \subset [v_P, \infty)$$

so for small enough v we have that $Q_v = \emptyset$. Strong duality, which we shall prove next, states that for all $v > v_P$, we have that

$$Q_v \neq \emptyset.$$

This would imply that the dual is also bounded, and that in fact

$$Q_v = [v_Q, \infty)$$

with

$$v_Q = v_P = v_*.$$

Remark. *It is a priori not clear how to represent both the primal LP and the dual LP in the same picture. For example, the primal and the dual deal with spaces of different dimensions. In the geometric picture we shall draw next, we look only at the dual space. The columns of the matrix A are interpreted as points in the dual space. The variable vector x is thought of as defining a cone (i.e., a convex set which is also closed under products by positive numbers).*

To prove that, fix $v > v_P$. By definition,

$$\emptyset = P_v = \{x : Ax \leq b, x \geq 0, \langle -c, x\rangle \leq -v\}.$$

Let A' be the matrix A when we add to it the row $-c$, and let b' the the vector b when we add to it $-v$ at the end so that

$$P_v = \{x : A'x \leq b', x \geq 0\}.$$

Denote by $a'(1), \ldots, a'(n) \in \mathbb{R}^{k+1}$ the n columns of A'. Consider the two convex sets

$$K = \mathsf{cone}(a'(1), \ldots, a'(n)) = \{A'x : x \geq 0\} \subseteq \mathbb{R}^{k+1}$$

and

$$B = \{y' \in \mathbb{R}^{k+1} : y' \leq b'\}.$$

It follows that

$$K \cap B = \emptyset.$$

Here we use convexity; two convex sets are disjoint iff there is a separating hyperplane. There is $\alpha' \in \mathbb{R}^{k+1}$ and $\beta \in \mathbb{R}$ so that

$$K \subseteq \{y' : \langle \alpha', y'\rangle \geq \beta\}$$

and

$$B \subseteq \{y' : \langle \alpha', y'\rangle \leq \beta\}.$$

Remark. *To geometrically see the reason behind the convex separation theorem, if K, B are disjoint and convex, then look at the closest point p in K to B and the closest point q in B to K, and consider the hyperplane that is orthogonal to $p - q$ and bisects the interval $[p, q]$ to two parts.*

Remark. *The convex separation theorem can also be proved in other ways (e.g., by analyzing the perceptron algorithm).*

Because K is a cone, we can choose $\beta = 0$. In other words, for all $x \geq 0$,

$$\langle \alpha', A'x\rangle \geq 0$$

and for all $y' \leq b'$,

$$\langle \alpha', y'\rangle \leq 0.$$

We claim that

$$\alpha' \geq 0,$$

because otherwise some $\alpha'_i < 0$ and so for for some $y' \leq b'$ we have $\langle \alpha', y'\rangle > 0$ which is false. In addition, $\alpha'_{k+1} \neq 0$, because otherwise P_v is empty for all v, which is false. So, $\alpha'_{k+1} > 0$ and we can normalize α' so that

$$\alpha'_{k+1} = 1.$$

Writing $\alpha' = (\alpha, 1)$ we see that for all $x > 0$,

$$0 \leq \langle \alpha', A'x\rangle = \langle \alpha, Ax\rangle - \langle c, x\rangle = \langle A^T\alpha - c, x\rangle.$$

and

$$\langle \alpha, b \rangle \leq v.$$

Because the first inequality holds for all $x \geq 0$,

$$A^T \alpha \geq c.$$

It follows that

$$\alpha \in Q_v,$$

as needed.

4. Randomness

Randomness allows to make decisions because in many cases the "bad events are sparse" but we can not identify them.

4.1. **Rounding LPs.** We now describe how randomness can help to round LPs. For set cover, we saw two deterministic rounding algorithms that yield approximation ratio f, which could be quite high.

How can randomness help to round $p \in [0,1]$ to $\{0,1\}$? A natural option is to define a random variable that take 1 with probability p and 0 with probability $1-p$. The expected value is thus exactly p.

Let us do that for the set cover problem. Let F be an instance of set cover, and x^* be a fractional cover. Let $(Z_s)_{s\in F}$ be independent zero-one variables so that $\mathbb{E}Z_s = x^*_s$ for all s. Denote by R the random subset of F defines as

$$R = \{s \in F : Z_s = 1\}.$$

Exercise. $\mathbb{E}|R| = \sum_s x^*_s$.

Claim. *For all $i \in [n]$,*

$$\Pr\Big[i \notin U\Big] \le \frac{1}{e}.$$

where $U := \bigcup_{s\in R} s$.

Proof.

$$\begin{aligned}\Pr\Big[i \notin U\Big] &= \prod_{s\ni i}(1 - x^*_s)\\ &\le \prod_{s\ni i} e^{-x^*_s}\\ &= e^{-\sum_{s\ni i} x^*_s}\\ &\le \frac{1}{e}.\end{aligned}$$

The first inequality holds because $1 - \xi \le e^{-\xi}$ for all $\xi \in \mathbb{R}$; draw the two graphs (try to prove). The second inequality holds because x^* is a fractional cover. □

The set R is most likely not a cover, but it covers a constant fraction of the domain.

Claim. *There is a polynomial-time randomized algorithm that with probability at least $2/3$, outputs a cover C of expected size at most $2(3 + \ln n) \cdot opt$.*

Remark. *By repeating the algorithm a few times, the success probability can be boosted.*

Proof. By repeating the random choice k times, we get independent $U_1, \dots, U_k$ and

$$\Pr\Big[i \notin \bigcup_{j\in[k]} U_j\Big] \le e^{-k}.$$

By the union bound,

$$\Pr\Big[\bigcup_{j\in[k]} R_j \text{ is not a cover}\Big] \le ne^{-k}.$$

Let

$$k = \lceil 2 + \ln(n) \rceil$$

and denote by E the event the $C = \bigcup_j R_j$ is a cover. It follows that

$$\Pr[E] \geq \frac{2}{3}.$$

It remains to bound the size of the cover C, conditioned on the event E.

$$\begin{aligned}
\mathbb{E}\Big[|C|\Big|E\Big] &= \frac{1}{\Pr[E]}\mathbb{E}\Big[|C|1_E\Big] \\
&\leq 2\mathbb{E}\Big[|C|\Big] \\
&\leq 2\sum_{j\in[k]} \mathbb{E}\Big[|R_j|\Big] \\
&= 2k \cdot opt^*.
\end{aligned}$$

□

Remark. *Recall the flow*

$$\textit{combinatorial optimization} \rightarrow_{relax} \textit{LP} \rightarrow_{round} \textit{solution}$$

If we use randomness in the rounding, we can get better results.

4.2. **Randomness for approximation.** In the 3-max-sat problem, the input is a CNF

$$\bigwedge_{i\in[m]} c_i$$

over n variables where each clause is the OR of three literals and each literal is a variable or its negation. The output is an assignment that satisfied the maximum number of clauses. This problem is NP-hard because 3-sat is NP-complete so we do not expect to solve it in polynomial time.

But we can use randomness to approximately solve it. If $x \in \{0,1\}^n$ is chosen uniformly at random then

$$\Pr[c_i(x) = 1] = \tfrac{7}{8}.$$

Denote by σ the random variable that counts the number of satisfying assignments. This means that the algorithm that outputs x leads to

$$\mathbb{E}[\sigma] = \tfrac{7}{8}m \geq \tfrac{7}{8}opt.$$

Remark. *If for some $\rho > \frac{7}{8}$, there is a ρ-approximation algorithm for max-sat then P=NP.*

Derandomization. *Can we make a randomized algorithm deterministic without significantly increasing the running time?* The answer is not known (but is mostly believed to be "yes").

Because the algorithm above and its analysis are quite simple, we can make it deterministic. One way to achieve this is to use "conditional expectations". We shall follow a different approach.

Let X be a random variable taking values in $\{0,1\}^n$. We say that X is 3-wise independent if for every $i < j < k$, the three boolean variables x_i, x_j, x_k are independent.

Exercise. *If X is 3-wise independent and σ is the number of satisfied assignments then*

$$\mathbb{E}[\sigma] = \tfrac{7}{8}m.$$

Remark. *This also leads to a $\frac{7}{8}$-approximation ratio.*

The main point is that there is 3-wise independent distribution of small support. Here is one construction. Let e.g. n be of the form $n = 2^k k$. Consider the field $\mathbb{F}_2^k$. Let y_0, y_1, y_2 be three i.i.d. uniform variables in $\mathbb{F}_2^k$. Let

$$g(x) = y_0 + y_1 x + y_2 x^2,$$

where the squaring is in the field. For each y_0, y_1, y_2, for every $x \in \mathbb{F}_2^k$, the field element $g(x)$ has k bits. In total, we have n bits:

$$(y_0, y_1, y_2) \mapsto r \in \{0,1\}^n.$$

Claim. *r is 3-wise independent.*

Proof. In fact, for every distinct x_0, x_1, x_2, we shall prove that $g(x_0), g(x_1), g(x_2)$ are uniform and independent (which is more informative). Indeed, for all z_0, z_1, z_2,

$$(g(x_0), g(x_1), g(x_2)) = (z_0, z_1, z_2)$$

iff

$$\begin{bmatrix} 1 & x_0 & x_0^2 \\ 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \end{bmatrix} \cdot \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} z_0 \\ z_1 \\ z_2 \end{bmatrix}.$$

The rank of the matrix is full, so there is a unique solution:

$$\Pr[(g(x_0), g(x_1), g(x_2)) = (z_0, z_1, z_2)] = \frac{1}{|\mathbb{F}_2^k|^3}.$$

□

We now get an efficient deterministic algorithm. For every y_0, y_1, y_2, compute $\sigma(r)$ and output the r with maximum value. The number of (y_0, y_1, y_2) is at most $2^{3k} \le n^3$.

Remark. *The idea of k-wise independence is extremely useful in derandomization, and is related to coding theory as well. One particular property that it yields is a bound on moments. If $X_1, \ldots, X_n$ then we can prove that $X = \sum_i X_i$ is concentrated by looking at the exponential moment e^X. In the k-wise independent setting, we get weaker concentration bounds via the study of X^k. For example, we can use Chebyshev's inequality with pairwise independence.*

Summary

Randomness is a powerful computational resource. It can help to do rounding with better guarantees. It can also help to find approximation solutions. Sometimes, we can derandomize the algorithm we built.