Lecture 18

Polynomial-Time Approximations

Supplemental reading in CLRS: Chapter 35 except §35.4

If you try to design algorithms in the real world, it is inevitable that you will come across NP-hard problems. So what should you do?

1. Maybe the problem you want to solve is actually less general than the NP-hard problem.
   - Perhaps the input satisfies certain properties (bounded degree, planarity or other geometric structure, density/sparsity...)
   - Remember, if you are able to reduce your problem to an NP-hard problem, that doesn't mean your problem is NP-hard—it's the other way around!
2. Maybe you can settle for an approximation.
3. Maybe an exponential algorithm is not so bad.
   - There might be an $O(c^n)$ algorithm with $c \approx 1$.

In this lecture we will focus on item 2: approximation algorithms. As the name suggests, an **approximation algorithm** is supposed to return an answer that is close to the correct answer. There are several types of approximations one might consider, each based on a different notion of closeness.

Definition. Suppose A is an algorithm for the optimization problem Π. (So the answer to Π is a number which we are trying to maximize or minimize, e.g., the weight of a spanning tree.) Given input x, we denote the output of A by $A(x)$, and the optimal answer by $\Pi(x)$. The following are senses in which A can be an "approximation algorithm" to the problem Π:

- The most common type of approximation is multiplicative. For a positive number $\alpha \in \mathbb{R}$ (which may depend on the input), we say that A is a **multiplicative α-approximation** (or simply an **α-approximation**) to Π if, for every input x,

$$\left\{ \begin{array}{rcccll} \alpha\Pi(x) & \leq & A(x) & \leq & \Pi(x) & \text{if } \Pi \text{ is a maximization problem} \\ \Pi(x) & \leq & A(x) & \leq & \alpha\Pi(x) & \text{if } \Pi \text{ is a minimization problem} \end{array} \right\}.$$

  Of course, we must have $0 < \alpha \leq 1$ for a maximization problem and $\alpha \geq 1$ for a minimization problem. If someone talks about a "2-approximation" to a maximization problem, they are probably referring to what here would be called a $\frac{1}{2}$-approximation.

- For a positive number $\beta \in \mathbb{R}$, we say that A is a **additive β-approximation** to Π if, for every input x,
$$\Pi(x) - \beta \;\le\; A(x) \;\le\; \Pi(x) + \beta.$$
- There are lots of other possibilities.[1] There is no need to worry about memorizing different definitions of approximation. When someone talks about an "approximation," it is their job to define precisely in what way it approximates the answer.

In this lecture we will see approximation algorithms for three NP-hard problems:

Problem	**Approximation factor**
Vertex Cover	2
Set Cover	$\ln n + 1$ (where n is the total number of elements)
Partition	$1+\epsilon$ for any given parameter $\epsilon > 0$

Note that the $(1+\epsilon)$-approximation algorithm to Partition is a so-called **polynomial-time approximation scheme (PTAS)**. Given a parameter $\epsilon > 0$, the PTAS generates a polynomial-time algorithm which approximates the Partition problem to a factor of $1+\epsilon$. Naturally, the polynomials will get larger as we decrease ϵ, since we are asking for a better approximation.

The approximations to Vertex Cover and Set Cover in this lecture are conjectured to be optimal, in the sense that giving a better approximation would be NP-hard.[2,3] There exist better approximations to Partition, though.[4]

18.1 Vertex Cover

NP-hard problem (Vertex Cover)**.** Given an undirected graph $G = (V, E)$, a *vertex cover* of G is a subset $V' \subseteq V$ such that, for every edge $(u, v) \in E$, we have either $u \in V'$ or $v \in V'$. The Vertex Cover problem asks, given a graph G, what is the smallest possible vertex cover?

Approximation Algorithm:

```
1  V' ← ∅
2  while E is nonempty do
3      Pick any (u, v) ∈ E
4      V' ← V' ∪ {u, v}
5      Remove from E all edges touching u or v
6      Remove u and v from V
```

[1] As a final example, we might combine multiplicative and additive approximations to define an "α, β–affine-linear approximation," in which
$$\alpha^{-1}\Pi(x) - \beta \;\le\; A(x) \;\le\; \alpha\Pi(x) + \beta$$
for all inputs x, where $\alpha \ge 1$.

[2] If the so-called "Unique Games Conjecture" holds, then it is NP-hard to α-approximate Vertex Cover for any constant $\alpha < 2$. It was recently proved that it is NP-hard to $(c \ln n)$-approximate Set Cover for any constant $c < 1$.

[3] Proofs of such optimality typically use the *probabilistically checkable proofs (PCP) theorem*. The PCP theorem states that a problem is in NP if and only if solutions to the problem can be verified (with high probability) by a certain kind of randomized algorithm.

[4] There exists a PTAS for Partition in which the running time depends only polynomially on $1/\epsilon$; this sort of PTAS is known as a **fully polynomial-time approximation scheme (FPTAS)**. As a consequence, for any given integer r there exists a polynomial-time $\left(1 + 1/n^r\right)$-approximation to Partition. Moreover, there exists a pseudo-polynomial–time algorithm which solves the Partition problem exactly. A **pseudo-polynomial–time** algorithm is one whose running time is polynomial in the size of the input and the numeric value of the output (in this case, the "cost" of the optimal partition).

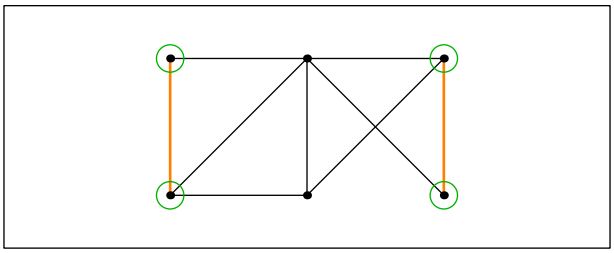

Figure 18.1. Example run of the approximation algorithm for Vertex Cover. The edges chosen by line 3 are highlighted in orange and the chosen vertices are circled. The algorithm returns a cover of size 4, whereas the optimal covering has size 3.

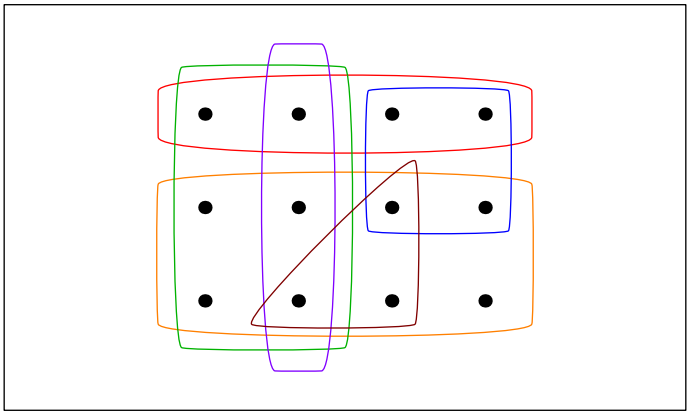

Figure 18.2. The Set Cover problem gives us a covering of a universe set U and asks us to find a small subcovering. How many of the sets in this picture are required to cover the array of dots?

It is clear that this algorithm always returns a vertex cover. Moreover, if G is given in the adjacency-list format, then the algorithm runs in linear time.

Claim. The cover returned by this algorithm is at most twice the size of an optimal cover.

Proof. Let E' denote the set of edges that get chosen by line 3. Notice that no two edges in E' share an endpoint. Thus, every vertex cover must contain at least one endpoint of each edge in E'. Meanwhile, the output of our algorithm consists precisely of the *two* endpoints of each edge in E'. □

18.2 Set Cover

NP-hard problem (Set Cover)**.** Given a set U and subsets $S_1, \ldots, S_m \subseteq U$ such that $\bigcup_{i=1}^{m} S_i = U$, find indices $I \subseteq \{1, \ldots, m\}$, with $|I|$ minimal, such that

$$\bigcup_{i \in I} S_i = U.$$

Approximation Algorithm:

```
1 while U is not empty do
2     Pick the largest subset S_i
3     Remove all elements of S_i from U and from the other subsets
4 return a list of the sets we chose
```

The running time for a good implementation of this algorithm is

$$O\left(\sum_{i=1}^{m} |S_i|\right)$$

(see Exercise 35.3-3 of CLRS).

Claim. The above algorithm gives a $(\ln|U|+1)$-approximation.

Proof. Assume the optimal cover has size k. Let U_i denote the value of U (our universe set) after i iterations of the loop. Clearly, for all i, the set U_i can be covered by k sets. So one of these k sets must contain at least $\frac{|U_i|}{k}$ elements, and therefore the set chosen on line 2 has size at least $\frac{|U_i|}{k}$. Thus, we have

$$\left|U_{i+1}\right| \le \left(1-\tfrac{1}{k}\right)\left|U_i\right|$$

for all i. This implies that

$$\left|U_i\right| \le \left(1-\tfrac{1}{k}\right)^i \left|U_0\right|$$

for all i. Since $1-\frac{1}{k} \le e^{-1/k}$, it follows that

$$\left|U_i\right| \le e^{-i/k}\left|U_0\right|.$$

In particular, letting $n = |U_0|$, we have

$$\left|U_{k(\ln n+1)}\right| < 1 \quad \Longrightarrow \quad \left|U_{k(\ln n+1)}\right| = 0.$$

Thus, the loop exits after at most $k(\ln n+1)$ iterations. □

18.3 Partition

NP-hard problem (Partition)**.** Given a sorted list of numbers $s_1 \ge s_2 \ge \cdots \ge s_n$, partition the indices $\{1,\ldots,n\}$ into two sets $\{1,\ldots,n\} = A \sqcup B$ such that the "cost"

$$\max\left\{\sum_{i\in A} s_i,\ \sum_{i\in B} s_i\right\}$$

is minimized.

Example. For the partition

$$\boxed{12}\quad \boxed{10}\quad 9\quad 7\quad 4\quad 3\quad \boxed{2}$$

it is optimal to take $A = \{1,2,7\}$ and $B = \{3,4,5,6\}$, so that

$$\sum_{i\in A} s_i = 24 \quad \text{and} \quad \sum_{i\in B} s_i = 23.$$

Approximation Algorithm:

1 ▷ ϵ is a parameter: for a given ϵ, the running time is polynomial in n when we view ϵ as a constant
2 $m \leftarrow \lfloor 1/\epsilon \rfloor$
3 By brute force, find an optimal partition $\{1,\ldots,m\} = A' \sqcup B'$ for $s_1,\ldots,s_m$
4 $A \leftarrow A'$
5 $B \leftarrow B'$
6 **for** $i \leftarrow m+1$ **to** n **do**
7 **if** $\sum_{j\in A} s_j \le \sum_{j\in B} s_j$ **then**
8 $A \leftarrow A \cup \{i\}$
9 **else**
10 $B \leftarrow B \cup \{i\}$
11 **return** $\langle A, B\rangle$

Note that this algorithm always returns a partition. The running time of a reasonable implementation is[5]

$$\Theta\left(2^m + n\right) = \Theta(n).$$

Claim. The above algorithm gives a $(1+\epsilon)$-approximation.

Proof. Without loss of generality,[6] assume

$$\sum_{i\in A'} s_i \ge \sum_{i\in B'} s_i.$$

Let

$$H = \frac{1}{2}\sum_{i=1}^{n} s_i.$$

Notice that solving the partition problem amounts to finding a set $A \subseteq \{1,\ldots,n\}$ such that $\sum_{i\in A} s_i$ is as close to H as possible. Moreover, since $\sum_{i\in A} s_i + \sum_{i\in B} s_i = 2H$, we have

$$\max\left\{\sum_{i\in A} s_i,\ \sum_{i\in B} s_i\right\} \ = \ H + \frac{D}{2},$$

where

$$D = \left|\sum_{i\in A} s_i - \sum_{i\in B} s_i\right|.$$

- *Case 1:*

  $$\sum_{i\in A'} s_i > H.$$

  In that case, the condition on line 7 is always false, seeing as $\sum_{i\in A'} s_i > \sum_{i\notin A'} s_i$. So $A = A'$ and $B = \{1,\ldots,n\} \setminus A'$. In fact, approximation aside, I claim that this must be the *optimal* partition. To see this, first note that $\sum_{i\in B} s_i < H < \sum_{i\in A'} s_i$. If there were a better partition

[5] Of course, the dependence on ϵ is horrible. In a practical context, we would have to choose ϵ small enough to get a useful approximation but big enough that our algorithm finishes in a reasonable amount of time. This would be helped if we replaced line 3 with a more efficient subroutine.

[6] We can have our algorithm check whether this equation holds, and switch the roles of A' and B' if it doesn't.

$\{1,\ldots,n\} = A^* \sqcup B^*$, then taking $A'' = A^* \cap \{1,\ldots,m\}$ and $B'' = B \cap \{1,\ldots,m\}$ would give a partition of $\{1,\ldots,m\}$ in which

$$\max\left\{\sum_{i\in A''} s_i,\ \sum_{i\in B''} s_i\right\} < \sum_{i\in A'} s_i,$$

contradicting the brute-force solution on line 3.

- *Case 2:*

$$\sum_{i\in A'} s_i \le H.$$

Note that, if A ever gets enlarged (i.e., if the condition on line 7 is ever true), then we have $D \le s_i$ (where i is as in line 6). And if A is never enlarged, then it must be the case that $\sum_{i\in B} s_i$ never exceeded $\sum_{i\in A} s_i$ until the very last iteration of lines 6–10, in which s_n is added to B. In that case we have $D \le s_n$ (why?). Either way, we must have

$$D \le s_{m+1}$$

and consequently

$$\max\left\{\sum_{i\in A} s_i,\ \sum_{i\in B} s_i\right\} \quad = \quad H + \frac{D}{2} \quad \le \quad H + \tfrac{1}{2}s_{m+1}.$$

Thus,

$$\begin{aligned}
\frac{\text{cost of the algorithm's output}}{\text{optimal cost}} &\le \frac{\text{cost of the algorithm's output}}{H} \\
&\le \frac{H + \frac{1}{2}s_{m+1}}{H} \\
&= 1 + \frac{\frac{1}{2}s_{m+1}}{H} \\
&= 1 + \frac{\frac{1}{2}s_{m+1}}{\frac{1}{2}\sum_{i=1}^{m+1} s_i} \\
&= 1 + \frac{s_{m+1}}{\sum_{i=1}^{m+1} s_i} \\
&\le 1 + \frac{s_{m+1}}{(m+1)s_{m+1}} \\
&= 1 + \frac{1}{m+1} \\
&< 1 + \epsilon.
\end{aligned}$$

□

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012