MA421 Advanced Algorithms Lent 2019

2 Approximation Algorithms

Reading:

- David Williamson. Lecture Notes on Approximation Algorithms, `http://legacy.orie.cornell.edu/~dpw/cornell.ps`, 1998.
- Vijay Vazirani. *Approximation Algorithms.* Springer-Verlag, 2001.

We consider an algorithm whose running time is bounded by a polynomial in the size of its input to be *efficient*. On the other hand, there are so many interesting computational problems (for example, for scheduling, planning, and searching) that are proven to be $\mathcal{NP}$-complete. Hence, we have little hope that any of these problems can be solved efficiently. In addition, there are problems of which we need to solve very large instances in practice (for example, problems defined on large data sets). So, even if these latter problems can be solved in polynomial time, we cannot tolerate any polynomial time but need very efficient ones.

In both of these cases above, we can either (i) give up on efficient algorithms, use exact algorithms (hoping they will run fast in practice), or (ii) give up on optimality and try heuristics, local search, simulated annealing, genetic algortihms, or approximation algorithms. Here, we will study approximation algorithms as a principled of way of dealing with $\mathcal{NP}$-complete optimisation problems.

Definition 2.1 (Approximation Algorithm) *An algorithm is an* α-approximation algorithm *for an optimization problem* Π *if*

- *the algorithm runs in polynomial time and*
- *the algorithm always produces a solution which is within a factor of* α *of the value of the optimal solution.*

Note that, for a minimization problem, $\alpha > 1$, and, for a maximization problem, $\alpha < 1$.

On occasion, we can devise algorithms that can get arbitrarily close to the optimal solutions.

Definition 2.2 (Polynomial-Time Approximation Scheme) *A* polynomial-time approximation scheme (PTAS) *for a minimization problem is a family of algorithms* $\{A_\epsilon : \epsilon > 0\}$ *such that for each* ϵ*,* A_ϵ *is a* $(1+\epsilon)$*-approximation algorithm.*

Note that, even though A_ϵ is an efficient algorithm, there might a superpolynomial dependency of its running time on $1/\epsilon$. If the running time depends also polynomially on $1/\epsilon$, then the scheme is called a *fully polynomial-time approximation scheme (FPTAS).*

The following theorem states that a class of problems called MAX-SNP-hard Problems, which include common problems such as MAX-SAT, Independent Set, Vertex Cover, and Max Cut, is not likely to admit a PTAS.

Theorem 2.3 (Arora, Lund, Motwani, Sudan, Szegedy 92) *There does not exist a PTAS for any MAX-SNP-hard problem unless* $\mathcal{P} = \mathcal{NP}$*.*

On a even grimmer note, the situation with MAX-CLIQUE is even worse.

Theorem 2.4 (Håstad 96) *There does not exist an $O(n^{1-\epsilon})$-approximation algorithm for any given $\epsilon > 0$ for MAX-CLIQUE, where n is the number of vertices in the input graph, unless $\mathcal{NP} \subseteq \mathcal{RP}$.*

This last theorem should be contrasted with the simple observation that an algorithm that always outputs 1 is a n-approximation algorithm for MAX-CLIQUE.

2.1 Set Cover

Next, we focus on the Set Cover problem and study several approximation algorithms for it.

Set Cover Problem:
Input:

- Ground elements $T = \{t_1, t_2, \ldots, t_n\}$
- Subsets $S_1, S_2, \ldots, S_m \subseteq T$
- Weights $w_1, w_2, \ldots, w_m$

Goal: Find a set $I \subseteq \{1, 2, \ldots, m\}$ that minimizes $\sum_{i \in I} w_i$ such that $\cup_{i \in I} S_i = T$. Unweighted version of the problemis when $w_j = 1$ for all j.

Below in Algorithm 1, we present a simple greedy algorithm for the unwighted set cover problem.

Algorithm 1 A simple greedy algorithm for unweighted set cover

```
1: Algorithm GREEDY 1$(T, (S_1, \ldots, S_m), (w_1, \ldots, w_m))$
2:     $I \leftarrow \emptyset$
3:     while $T \neq \emptyset$ do
4:         Pick $t_i \in T$
5:         $I \leftarrow I \cup \{j : t_i \in S_j\}$
6:         $T \leftarrow T \setminus \cup_{j \in I} S_j$
```

Let's define f to be the frequency of the most frequent element:

$$f = \max_i |\{j : t_i \in S_j\}|.$$

Theorem 2.5 *Greedy 1 is an f-approximation algorithm for the unweighted Set Cover problem.*

A general approach for building approximation algorithms goes through the machinery of mathematical programming, in particular, linear programming.

1. Formulate the problem as an Integer Program (IP).
2. Relax the IP to a Linear Program (LP).
3. Use the LP (and its solution) to get a solution to the IP.

The following integer program models the set cover problem.

$$\min \quad \sum_{j=1}^{m} w_j \cdot x_j$$

subject to:

$$\sum_{j:t_i \in S_j} x_j \geq 1 \qquad \forall t_i \in T$$
$$x_j \in \{0,1\} \qquad \forall i \in \{1,2,\dots m\}$$

Now, we can relax the integrality conditions to obtain a linear program, which can be solved optimally in polynomial time, whereas integer programs are $\mathcal{NP}$-hard to solve in general.

$$\min \quad \sum_{j=1}^{m} w_j \cdot x_j$$

subject to:

$$\sum_{j:t_i \in S_j} x_j \geq 1 \qquad \forall t_i \in T$$
$$x_j \geq 0 \qquad \forall i \in \{1,2,\dots m\}$$

Let Z_{LP} be the optimal value of the LP and OPT be the optimal value of the IP. Then, clearly,

$$Z_{LP} \leq \text{OPT}.$$

If we can find an *integral* solution of cost at most $\alpha \cdot Z_{LP} \leq \alpha \cdot \text{OPT}$, we get an α-approximation.

Algorithm 2 A set cover algorithm through LP rounding

```
1: Algorithm ROUNDING-SET-COVER($T, \mathcal{S}, w$)
2:     Solve the LP to get an optimal solution $x^*$.
3:     $I \leftarrow \emptyset$
4:     for all $S_j$ do
5:         if $x_j^* \geq 1/f$ then
6:             $I \leftarrow I \cup \{j\}$
```

Theorem 2.6 (Hochbaum 82) *Rounding is an f-approximation algorithm for Set Cover.*

$$\sum_{j \in I} w_j \leq \sum_j w_j x_j^* f = f \sum_j w_j x_j^* \leq f \cdot \text{OPT}$$

2.1.1 A Better Greedy Algorithm

Greedy Idea: "Most bang for the money". Select the set that minimizes the cost per additional element covered.

Algorithm 3 A greedy algorithm for weighted set cover

```
1: Algorithm GREEDY 2($T, (S_1, \dots, S_m), (w_1, \dots, w_m)$)
2:     $I \leftarrow \emptyset$
3:     $\tilde{S}_j \leftarrow S_j, \quad \forall j$
4:     while $\cup_{j \in I} S_j \neq T$ do
5:         $\ell \leftarrow \arg\min_{j:\tilde{S}_j \neq \emptyset} \frac{w_j}{|\tilde{S}_j|}$
6:         $I \leftarrow I \cup \{\ell\}$                ▷ for all $t \in \tilde{S}_\ell$, price($t$) = $\frac{w_\ell}{|\tilde{S}_\ell|}$
7:         $\tilde{S}_j \leftarrow \tilde{S}_j \setminus S_\ell, \quad \forall j$
```

We get

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln n + 0.577$$

$$t_1, t_2, \ldots, t_n : \text{ the order in which the items are covered}$$

Lemma 2.7 *For each $t_k \in T$, price$(t_k) \leq OPT/(n-k+1)$.*

Theorem 2.8 *Greedy 2 is a H_n-approximation algorithm for Set Cover.*

Theorem 2.9 (Chvátal 79) *Greedy 2 is a H_g-approximation algorithm for Set Cover for $g = \max_j |S_j|$.*

Proof. Fix any set S_j. Rearrange the indices of the items so that $t_1, \ldots, t_r$ be the elements of S_j in the order they were covered by Greedy 2 (breaking ties arbitrarily). Now, fix t_k for some $k \leq r$. Right before t_k was first covered in Greedy 2 by a set S_ℓ, the number $|\tilde{S}_j|$ of the remaining uncovered elements in S_j was at least $r-k+1$. Also, since S_ℓ was at least as good as S_j for the next set to include in the solution, we must have that

$$\frac{w_\ell}{|\tilde{S}_\ell|} \leq \frac{w_j}{|\tilde{S}_j|}.$$

Hence, we can bound price(t_k) as

$$\text{price}(t_k) = \frac{w_\ell}{|\tilde{S}_\ell|} \leq \frac{w_j}{|\tilde{S}_j|} \leq \frac{w_j}{r-k+1}.$$

Therefore, summing over all elements of S_j, we get

$$\sum_{i=1}^{r} \text{price}(t_i) \leq \sum_{i=1}^{r} \frac{w_j}{r-i+1} = w_j \cdot \left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{r}\right) = H_r \cdot w_j,$$

for an arbitrary set S_j of cardinality r.

Next, we show that the solution of Greedy 2 is an H_g-approximation to the optimal solution. For ease of notation, suppose $S_1, \ldots, S_p$ is an optimal solution. Note that the total cost of the solution of Greedy 2 is $\sum_{i=1}^{n} \text{price}(t_i)$. Thus, we can bound this cost as

$$\sum_{i=1}^{n} \text{price}(t_i) \leq \sum_{j=1}^{p} \sum_{t_i \in S_j} \text{price}(t_i) \leq \sum_{j=1}^{p} H_{|S_j|} \cdot w_j \leq H_g \cdot \sum_{j=1}^{p} w_j = H_g \cdot \text{OPT}.$$

□

Finally, the following theorem demonstrates how close Greedy 2 gets to the best possible approximation ratio.

Theorem 2.10 (Dinur, Steurer '13) *For every $\alpha > 0$, it is $\mathcal{NP}$-hard to approximate Set Cover to within $(1-\alpha)\ln n$, where n is the size of the instance.*

2.2 Bin Packing

Next we study another fundamental combinatorial optimisation problem.

Definition 2.11 (Bin Packing) *Given $w_1, w_2, \ldots, w_n$ such that $0 < w_i \leq 1$, find a packing of these items into minimum number of unit-size bins.*

It is easy to see that bin packing is $\mathcal{NP}$-hard. A simple argument can show that even approximating it within a factor of $3/2 - \epsilon$ for any $\epsilon > 0$ is $\mathcal{NP}$-hard: we can reduce a set partition instance to a bin packing instance, by scaling the numbers so that the total weight is 2. Hence, a YES-instance of the set partition problem gives a bin packing instance that has the optimal value 2, whereas other instances yield an optimal value 3. A $(3/2 - \epsilon)$-approximation algorithm for bin packing would distinguish between these cases.

Some know results for bin packing approximations are below.

- Constant approximation: First Fit ($\frac{17}{10} \cdot \mathrm{OPT} + 2$), Best Fit, First/Best Fit Decreasing ($\frac{11}{9} \cdot \mathrm{OPT} + 1$),...
- Asymptotic PTAS: $(1+\epsilon) \cdot \mathrm{OPT} + 1$ for any $\epsilon > 0$ [Fernandez de la Vega and Lueker 81]. Linear in n and exponential in $\frac{1}{\epsilon}$.
- Better than PTAS: $\mathrm{OPT} + \log^2(\mathrm{OPT})$ by Karmarkar and Karp in 1982. This is an $O(n^8)$-time algorithm.
- Is $\mathrm{OPT} + c$ possible? Open!

Let's start with a simple algorithm, called First Fit. This algorithm considers the input items in an arbitrary order. In the ith step it has a list of partially packed bins $B_1, \ldots, B_k$. It attempts to put the next item in one of these bins, trying each in that order. If it does not fit in any of these bins, then it opens a new bin B_{k+1} to put this item in. To show that First Fit algorithm achieves an approximation factor of 2, we need to observe that all but the one bin has to be at least half full (otherwise we would not have opened a new bin).

Let $W = \sum_i w_i$. It clear that $\lceil W \rceil \leq \mathrm{OPT}$, where OPT is the cost of an optimal solution. Hence, we get for the number m of bins used by the First Fit algorithm

$$m - 1 < \lceil \frac{W}{\frac{1}{2}} \rceil = \lceil 2 \cdot W \rceil \leq 2 \cdot \lceil W \rceil \leq 2 \cdot \mathrm{OPT},$$

which implies $m \leq 2 \cdot \mathrm{OPT}$.

Total weight W provides a simple 2-approximation to OPT. And, First Fit guarantees such an approximation (actually, even more, see the list above).

2.2.1 Special Case: Small Items

Suppose $w_i \leq \beta$ for all i. Then, we can observe that each bin can be filled up to $1 - \beta$ level. Hence,

$$\lceil \frac{W}{1-\beta} \rceil \leq \lceil (1+2\beta) \cdot W \rceil \leq (1+2\beta) \cdot \lceil W \rceil + 1 \leq (1+2\beta) \cdot \mathrm{OPT} + 1$$

Total weight W is sufficient for a $(1+\epsilon)$-approximation for "small" items.

2.2.2 Special Case: Large Items

If $w_i > \beta$ for all i, each bin contains at most $\lfloor \frac{1}{\beta} \rfloor$ items.

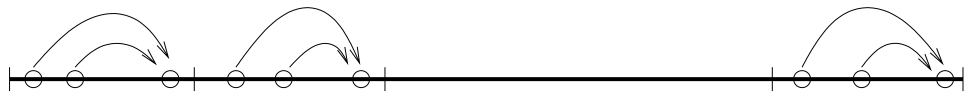
Lemma 2.12 *If the number of different weights in $w_1, \ldots, w_n$ is a constant c and $w_i \geq \beta$ for all i and constant $\beta > 0$, bin packing can be solved* exactly *in time $O(n^c \cdot c^{1/\beta})$.*

Proof idea. There are $O(n^c)$ different subsets of items and $O(c^{1/\beta})$ configurations for a bin. Dynamic Programming table of size $O(n^c)$ with time $O(c^{1/\beta})$ to fill in each table entry. □

An alternative proof for a similar lemma (from Vazirani book) that does not rely on Dynamic Programming can be given here. Due to its (more) bruteforce nature, this proof is not as efficient as the one above, but still yields a polynomial-time algorithm (as opposed to the specific running time given in the lemma above).

Proof. (of a variant of Lemma 2.12) The number of items in a bin is bounded by $m = \lfloor \frac{1}{\beta} \rfloor$. Therefore, the number of different bin types is bounded by $r = \binom{m+c}{m}$ (this follows from the number of ways of putting m indistinguishable balls into c distinguishable bins). Note that r is still a constant. Clearly, the number of bins used is bounded above by the number n of items. Therefore, the number of possible packings is bounded by $p = \binom{n+r}{r}$ (similar to above calculation of r, by choosing one of the types for each bin). Enumerating all p options and picking the best packing gives the optimal answer. Since $p = O(n^r)$, this approach yields a polynomial-time algorithm. □

Theorem 2.13 *For the special case of large items, there is a polynomial-time $(1+\beta)$-approximation algorithm.*



Proof. Below in Algorithm 4, we present an algorithm for such instances of the bin packing problem.

Algorithm 4 An algorithm for bin packing instances with only big elements

```
1: Algorithm BIN-PACKING-BIG-ITEMS($\vec{w}, \beta$)
2:     Sort and group the items into $\lceil \frac{1}{\beta^2} \rceil$ groups (in nondecreasing order).
3:     Round up the weight of each item to largest weight in the group.
                                        ▷ constant number of different weights
4:     Using previous lemma (Lemma 2.12, output the optimal packing of these items.
```

Let I be the original binpacking instance. And, let J be the rounded-up instance obtained at Step 1 of the algorithm. Then, clearly, $\mathrm{OPT}(I) \leq \mathrm{OPT}(J)$. We will show that $\mathrm{OPT}(J)$, which is the cost of the solution returned by the algorithm, is at most $(1+\beta) \cdot \mathrm{OPT}(I)$.

Now, consider another instance J' which is obtained by rounding down the weight of each item to the smallest weight in its group. Clearly, $\mathrm{OPT}(J') \leq \mathrm{OPT}(I)$. We will now observe that an optimal packing for J' can be used to pack all but the group of the largest weights in J. Hence,

$$\mathrm{OPT}(J) \leq \mathrm{OPT}(J') + \beta^2 n,$$

where the additional factor of $\beta^2 n$ comes from using one bin per item for those remaining items. Finally, using $\mathrm{OPT}(J') \leq \mathrm{OPT}(I)$ and observing that $\mathrm{OPT}(I) \geq \beta n$, we get

$$\mathrm{OPT}(J) \leq \mathrm{OPT}(I) + \beta \cdot \mathrm{OPT}(I) = (1+\beta) \cdot \mathrm{OPT}(I).$$

□

2.2.3 General case

We can finally put it all together to construct a general algorithm for bin packing and prove the following theorem.

Theorem 2.14 *For any ϵ, $0 < \epsilon \leq \frac{1}{2}$, there is a polynomial-time algorithm A_ϵ that finds a packing using at most $(1+\epsilon)OPT + 1$ bins.*

Proof. The algorithm that combines the two algorithms for the special cases of only-small and only-large elements is presented below in Algorithm 5.

Algorithm 5 A PTAS for Bin Packing

```
1: Algorithm Bin-Packing($\vec{w}, \epsilon$)
2:     Remove item of size less than $\epsilon/2$.
3:     Round up to obtain constant number of different item sizes.
4:     Find optimal packing using Algorithm BIN-PACKING-BIG-ITEMS.
5:     Use this packing for the original item sizes.
6:     Pack items of size less than $\epsilon/2$ using First Fit.
```

First, let's consider the case where no new bins were opened at Step 5. Then, since we have $(1+\epsilon/2)$-approximation to the subset consisting "large items" and no new bins were needed for the small items, we should have a $(1+\epsilon/2)$-approximation to the given instance.

In the case new bins were needed in Step 5, we should have that all but the last bin is used to the $1-\epsilon/2$ level. Let M be the number of bins used. Then, the total weight of all the items is at least $(M-1)(1-\epsilon/2)$. Hence, we get

$$M \leq \frac{\text{OPT}}{1-\epsilon/2} + 1 \leq (1+\epsilon)\cdot \text{OPT} + 1.$$

□