21120491 Advanced Data Structure and Algorithm Analysis **Spring-Summer 2023**

Lecture 11: Approximation Algorithm

Lecturer: Deshi Ye *Scribe: D. Ye*

11.1 Overview

Many real-world problems or optimization problems are NP-hard in general. We cannot find an algorithm that satisfies the following three properties (You can't have them all for NP-hard unless $P = NP$).

- Correctness: For every input, the algorithm correctly solves the problem.
- General-purpose: the algorithm works for all instances.
- Efficiency: For every input, the algorithm runs in polynomial time. (we say an algorithm is efficient if it runs in polynomial time)

Thus, one of the three properties or requirements has to be relaxed when we handle an NP-hard problem. Approximation algorithm will relax the requirement of correctness, and its goal is to find efficient algorithms that find solutions that are "good enough" for all instances. That means an approximation algorithm shall work for all instances and runs in polynomial time, and it outputs a feasible solution with an objective function value close to the best possible.

How well an approximation algorithm performs is measured by its approximation ratio. The approximation ratio represents the approximation algorithm how closely approximates the optimal solution in terms of its objective value.

Definition 11.1 *(Approximation ratio) An α-approximation algorithm for an optimization problem is a polynomial-time algorithm that for all instances of the problem produces a solution whose value is within a factor of α of the value of an optimal solution.*

In convience[1], let $\alpha \geq 1$. Denote the cost C of the solution produced by the approximation algorithm, and C^* the cost of an optimal solution. For minimization problems, $C \leq \alpha C^*$, while $C \geq \frac{1}{\alpha} C^*$ for maximization problems.

Definition 11.2 *(PTAS) A polynomial-time approximation scheme (PTAS) is a family of algorithm $\{A_\epsilon\}$, where there is an algorithm for each $\epsilon > 0$, such that $\{A_\epsilon\}$ is a $(1+\epsilon)$-approximation algorithm, and its running time is polynomial in the size n of its input instance.*

Definition 11.3 *(FPTAS) A polynomial-time approximation scheme (PTAS) is a family of algorithm $\{A_\epsilon\}$, where there is an algorithm for each $\epsilon > 0$, such that $\{A_\epsilon\}$ is a $(1+\epsilon)$-approximation algorithm, and its running time is polynomial in the size n and $1/\epsilon$.*

[1] It is also common that $\alpha \geq 1$ for minimization problems, while $\alpha \leq 1$ for maximization problems.

For example, the running time $O(n^{2/\epsilon})$ is PTAS, and the running time $O((1/\epsilon)^2 n^3)$ is FPTAS.

To achieve an approximation ratio α (for the minimization problem), we need to consider the worst case of the input, that is

$$\alpha = \sup_I \frac{C(I)}{OPT(I)}$$

for every instance I, where $C(I)$ is the cost of the approximation algorithm on the input I and $OPT(I)$ is the optimal cost on the input I.

In practice, we need to find an optimal lower bound $C^*(I)$ instead of the exact optimal value $OPT^*(I)$, *i.e.*, $OPT^*(I) \geq C^*(I)$. Usually, it is not hard to calculate $C^*(I)$.

Given any instance I, we are able to calculate $C(I)$ and $C^*(I)$, and we obtain that

$$\alpha = \sup_I \frac{C(I)}{OPT(I)} \leq \sup_I \frac{C(I)}{C^*(I)}.$$

The rule of finding $C^*(I)$ is to find a lower bound that is close to the optimal value $OPT(I)$, and also it is easy to calculate.

For a maximization problem, the approximation ratio α is defined to be

$$\alpha = \sup_I \frac{OPT(I)}{C(I)}.$$

In general, we say an approximation algorithm is better if its approximation ratio is smaller.

11.2 Bin packing

The bin packing problem is a well-known optimization problem. In the bin packing problem, we are given n items of sizes $S_1, S_2, \ldots, S_n$, such that $0 < s_i \leq 1$ for all $1 \leq i \leq n$. Pack these items in the fewest number of bins, each of which has unit capacity.

Decision version of bin packing: The BINPACKING problem: we are given n items of sizes $S_1, S_2, \ldots, S_n$, such that $0 < S_i \leq 1$ for all $1 \leq i \leq n$, the bin capacity is 1, and a positive integer k. Is there a partition of these n items into k bins, such that the total size of items in each bin is at most 1?

The bin packing is related to the PARTITION problem. In the partition problem, we are given n positive integers, $b_1, b_2, \ldots, b_n$ whose sum $B = \sum_{i=1}^n b_i$ is even. Our goal is to determine whether we can partition the indices $\{1, \ldots, n\}$ into two parts S and T such that $\sum_{i \in S} b_i = \sum_{j \in T} b_j$.

The partition problem is a decision problem, and it is a well-known NP-Complete problem [2].

Complexity analysis

Theorem 11.4 *The BINPACKING problem is in NP-complete.*

Proof: The BINPACKING problem is in NP. We can verify a "Yes" instance in polynomial time. Given the subset of integers packed in each of the k bins A_j, create the sum of these elements and compare with the capacity of 1.

We show that PARTITION $\leq_P$ BINPACKING. This is straightforward since Partition is a subproblem of BINPACKING (specific instances with 2 bins).

In detail, we show it below. For any instance of PARTITION, we construct the instance of BINPACKING by letting $S_i = 2b_i/B$, let $k = 2$. For sure, this construction runs in $O(n)$. If there is a "Yes" instance for PARTITION, i.e., $\sum_{i\in S} b_i = \sum_{i\in T} b_i = B/2$. Thus, $\sum_{i\in S} S_i = 1$, and $\sum_{i\in T} S_i = 1$. That means we can pack all items in two bins.

If we can pack all items in two bins S and T. Then, we have $\sum_{i\in S} S_i \leq 1$ and $\sum_{i\in T} S_i \leq 1$. Thus, $\sum_{i\in S} b_i \leq B/2$ and $\sum_{i\in T} b_i \leq B/2$. Known that $\sum_{i\in S} b_i + \sum_{i\in T} b_i = B$. Clearly, $\sum_{i\in S} b_i = \sum_{i\in T} b_i = B/2$, and we find a "Yes" partition. ■

11.2.1 Algorithms to solve the bin packing problem

Algorithms for bin packing require that a new bin is opened only if the item cannot be packed in any of the already opened bins. However, different ways of choosing an already opened bin lead to different algorithms.

- Next Fit (NF): Place each item in a single bin until an item does not fit in. When an item doesn't fit, **close it** and opens a new one.
- Best Fit (BF): Try to place an item in the **fullest** bin that can accommodate it. If there is no such bin, open a new one.
- First Fit (FF): Try to place an item in the **first** bin that accommodates it. If no such bin is found, open a new one.
- First Fit Decreasing (FFD): Same as FF after sorting the items by decreasing order.

Theorem 11.5 *Let M be the optimal number of bins required to pack a list I of items. Then next fit never uses more than $2M - 1$ bins. Namely, the approximation ratio of NF is at most 2. There exist sequences such that the next fit uses $2M - 1$ bins.*

Proof: For any given instance I, denote $OPT(I)$ to be the optimal number of bins required to pack a list I of items, and $NF(I)$ to be the number of bins used by the Next Fit algorithm, respectively.

Fact: $OPT(I) \geq \lceil \sum_{i=1}^{n} S_i \rceil$. The term $\lceil \sum_{i=1}^{n} S_i \rceil$ is a lower bound of any optimal solution.

Let $S(B_i)$ be the size of the i-th bin. Let S_j be the first item assigned in the $(i+1)$-th bin, then we have $S(B_i) + S_j > 1$, which implies that $S(B_i) + S(B_{i+1}) > 1$.

Then $\sum_{i=1}^{n} S_i = \sum_{i=1}^{NF(I)} S(B_i)$. We can show that if Next Fit generates $2M$ (or $2M+1$) bins, then the optimal solution must generate at least $M+1$ bins.

If $NF(I)$ is an odd number $2M+1$, then $\sum_{i=1}^{n} S_i > (NF(I)-1)/2 = M$. We obtain that $OPT(I) \geq M+1$ since $OPT(I)$ is an integer. If $NF(I)$ is an even number $2M$, then $\sum_{i=1}^{n} S_i = \sum_{i=1}^{NF(I)} S(B_i) > NF(I)/2 = M$, and again we have $OPT(I) \geq M+1$ since $OPT(I)$ is an integer.

The approximation ratio α of NF is

$$\alpha = \sup_I \frac{NF(I)}{OPT(I)} \leq \frac{2M+1}{M+1} \leq 2.$$

We claim that the approximation ratio 2 is **tight** by showing that there exists a sequence such that the next fit uses $2M-1$ bins while the optimal solution uses M bins. The instance is given below. Let $\epsilon > 0$ be a sufficiently small number. The $I = \{a_1, b_1, \ldots, a_{2(M-1)}, b_{2(M-1)}, a_{2M-1}, b_{2M-1}\}$ that consists of $2M-1$ large items with size $1/2$, and $2M-1$ small items with size ϵ and $\sum_{i=1}^{2M-1} \epsilon < 1/2$. The sequence of I is one large item followed by a small item.

One can easily check that $NF(I) = 2M-1$, in which every bin accommodates one large item and one small item. In the optimal solution, $2(M-1)$ large items occupy $M-1$ bins, while the last large item is assigned together with all small items, and $OPT(I) = M$ follows.

This instance shows that the analysis of the approximation ratio of Next Fit is tight, i.e., $\alpha \geq \frac{2M-1}{M}$. ■

11.2.2 Online algorithms for bin packing

Theorem 11.6 *There are inputs that force any on-line bin-packing algorithm to use at least* $5/3$ *the optimal number of bins.*

Proof: We show the theorem by the adversary method that will construct the sequence of arriving items by observing the actions of the on-line algorithm and force the on-line algorithm to perform the worst.

Especially, let bin size be 50. The adversary first releases tiny items with a total size of 34. Any online algorithm with a competitive ratio [2] smaller than 2 must pack all these tiny items in one bin, otherwise, the adversary stops the algorithm, which will lead to the approximation ratio being at least 2. Next, the adversary releases two items with sizes $\{17, 17\}$; if these two items are put in one bin, then the adversary releases three items with sizes $\{26, 26, 26\}$; otherwise, the adversary releases two items with sizes $\{34, 34\}$. One can check that in each case the on-line algorithm uses 5 bins, while the optimal algorithm uses 3 bins, and the competitive ratio of any online algorithm is at least $5/3$. ■

11.2.3 FF algorithm

First Fit algorithm is an on-line algorithm.

Theorem 11.7 *Let M be the optimal number of bins required to pack a list I of items. Then first fit never uses more than* $17M/10$ *bins. There exist sequences such that the first fit uses* $17(M-1)/10$ *bins.*

The proof of Theorem 11.7 was shown by Dosa and Sgall [1].

Implement of FF The FF algorithm can be implemented to take $O(n \log n)$ time. The idea is to use any balanced binary search tree, such as RB-tree and AVL tree.

Each node has three values: index of the bin, remaining capacity of the bin, and best (largest) remaining capacity in all the bins represented by the subtree rooted at the node. The ordering of the tree is by the bin index. Fig. 11.1 provides an example.

The Best Fit algorithm can also be implemented in $O(n \log n)$ time, with the ordering of the tree given by the remaining capacity of the bin.

[2] Approximation ratio in the on-line algorithm is usually called the competitive ratio

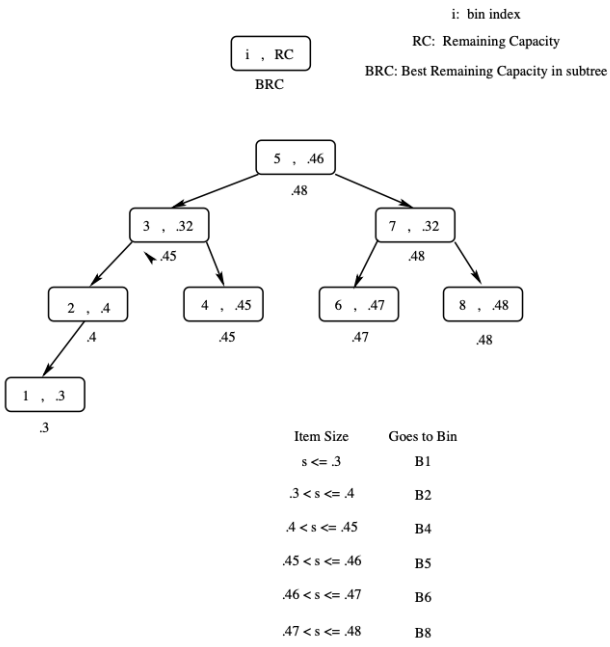

Figure 11.1: Example: FF in a balanced binary search tree

11.2.4 Inapproximability

What is the smallest approximation ratio for the bin packing problem? In the following, we show that one cannot find an approximation algorithm with an approximation ratio smaller than 1.5 if $P \neq P$.

Theorem 11.8 *For any constant $\epsilon > 0$, there is no $(1.5 - \varepsilon)$-approximation algorithm for the bin packing problem unless $P = NP$.*

Proof: Let $A = \{a_1, a_2, \ldots, a_n\}$ be an instance of a PARTITION problem. We denote a set of items $B = \{2a_i / \sum_{i=1}^{n} a_i | a_i \in A\}$. The bin size is 1.

Then items in B can be packed in 2 bins iff the set A can be partitioned into two sets with equal sums. Now suppose there is a $\rho = (1.5 - \epsilon)$ algorithm ALG for the bin packing for given $\epsilon > 0$.

We use this algorithm ALG to decide if B can be packed into 2 bins. If the output of ALG is 2, clearly, we found an equal partition for A.

Note that $ALG \leq \rho OPT$, where OPT is the optimal value to pack the set B. If $ALG > 2$, then it must hold that $OPT > 2$. Otherwise, $ALG \leq (1.5 - \epsilon)OPT < 3$, which implies that $ALG = 2$. In this case, we cannot find an equal partition.

In all, the approximation algorithm ALG runs in polynomial time, but it can decide whether a partition exists or not. That is, we solve the PARTITION problem, which is NP-complete, in polynomial time.

Therefore, we cannot have a $(1.5 - \varepsilon)$-approximation algorithm for bin packing unless $P = NP$. ■

11.3 Knapsack problem

A knapsack with a capacity of M is to be packed. Given N a set of n items. Each item i has a weight w_i and a profit p_i. If x_i is the percentage of the item i being packed, then the packed profit will be $p_i x_i$. In the vanilla 0/1 knapsack problem, x_i is either 0 or 1. If x_i can be any real number in $(0, 1]$, we say it is a fractional knapsack problem.

The knapsack problem is an NP-hard problem [2]. The PARTITION problem is a special case of the knapsack problem.

11.3.1 A greedy algorithm

We can design a greedy algorithm GreedyDensity that is based on profit density to solve the knapsack problem. We sort all the items by the ratio of their profits to their sizes so that $p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n$. The greedy algorithm GreedyDensity will select items in this order until exceeding the capacity of the knapsack.

It turns out that the algorithm GreedyDensity performs badly. Consider two items, and let $p_1 = 2$, $w_1 = 1$, and $p_2 = M$, $w_2 = M$. The greedy algorithm selects only item 1 with a total value of 2. But, the optimal algorithm will select item 2 with a total value of M.

What if we are greedy on taking the maximum profit or profit density? Namely, we simply take the best of GreedyDensity's solution or the most profitable item. We call this algorithm the ModifiedGreedy algorithm.

Theorem 11.9 *The approximation ratio of the ModifiedGreedy algorithm is* 2.

Proof: Let k be the index of the first item that is not accepted by GreedyDensity. Let $0 \leq \alpha < 1$ be the percentage of item k accepted in the knapsack. That is $w_1 + w_2 + \ldots + w_{k-1} + \alpha w_k = M$.

Denote P_{Gd} to the value generated by the GreedyDensity algorithm. We have $P_{Gd} = \sum_{i=1}^{k-1} p_i$.

Denote OPT to be the optimal value. We **claim that** $p_1 + p_2 + \ldots + p_{k-1} + p_k \geq OPT$.

On the the other hand, $OPT \geq p_{max} = \max_i p_i$. Let P_{Greedy} be the value returned by the ModifiedGreedy algorithm. We obtain that $P_{Greedy} = \max\{p_{max}, P_{Gd}\}$. From the claim, $P_{Gd} + p_k \geq OPT$, it follows that $P_{Gd} + p_{max} \geq OPT$. Thus, either P_{Gd} or p_{max} must be $OPT/2$.

The approximation ratio $\rho = \sup \frac{OPT}{P_{Greedy}} \leq 2$.

Now, we only need to prove the claim. We give an LP relaxation of the knapsack problem as follows: $x_i \in [0, 1]$ denotes the fraction of item i packed in the knapsack.

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} p_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} w_i x_i \leq M \\ & 0 \leq x_i \leq 1 \end{aligned}$$

This relaxed knapsack problem is called fractional knapsack. Let OPT' be the optimal value of the objective

function of the fractional knapsack problem. Note that Knapsack is a feasible solution to the LP, so $OPT' \geq OPT$.

Now, set $x_1 = x_2 = \ldots = x_{k-1} = 1, x_k = \alpha$, and $x_i = 0$ for $i > k$, such that the constraint $\sum_{i=1}^{n} w_i x_i = M$ holds. This is a feasible solution to the LP, and the objective value cannot be improved by changing any one tight constraint, as we sorted the items by the ratio of the profit to the weight. So, $OPT' = p_1 + p_2 + \ldots + p_{k-1} + \alpha p_k \geq OPT$. The claim follows immediately. ■

11.3.2 PTAS *

We observe that the GreedyDensity algorithm gives $(1+\epsilon)$-approximation if $p_k \leq \frac{\epsilon}{1+\epsilon} OPT$. We also observe that there are most $\frac{1+\epsilon}{\epsilon}$ items with profit at least $\frac{\epsilon}{1+\epsilon} OPT$ in any optimal solution.

Now, we describe the following algorithm EnumerateGreedy. Let $1 > \epsilon > 0$ be a fixed constant and let $h = \lceil \frac{1+\epsilon}{\epsilon} \rceil$. We try to enumerate all h most profitable items in an optimal solution and pack the rest greedily.

For each subset $S \subseteq N$, we select at most h items (can be less). Then remove all items in $N - S$ if their profits are larger than the smallest profit in S. Apply the GreedyDensity algorithm to select items among the remaining items.

Theorem 11.10 *The approximation ratio of the EnumerateGreedy algorithm is* $1 + \epsilon$*, and its running time is* $O(n^{h+1})$*, where* $h = \lceil \frac{1+\epsilon}{\epsilon} \rceil$.

Proof: The running time of enumerating all subsets S is $O(n^h)$. For each subset, a linear time is enough for the GreedyDensity if we sort all items at the beginning.

We now consider the approximation ratio. Our enumeration of S makes sure that one of the subset S contains the same most profitable items (no more than h items) in an optimal solution. The algorithm's greedy stage packs the set of items $A' \subseteq N - S$. Let OPT' be the optimal way to pack the smaller items in $N - S$, then $OPT = OPT' + p(S)$, where $p(S)$ is the total profit of items in S.

Let item k be the first item rejected by the greedy packing of $N - S$. Note that the profit of any items in A' is no more than $\frac{\epsilon}{1+\epsilon} OPT$. By the claim, $p(A') \geq OPT' - p_k \geq OPT' - \frac{\epsilon}{1+\epsilon} OPT$.

The profit generated by the EnumerateGreedy is $p(A') + p(S) \geq p(S) + OPT' - \frac{\epsilon}{1+\epsilon} OPT = \frac{1}{1+\epsilon} OPT$. The approximation ratio of $1 + \epsilon$ follows. ■

11.3.3 Dynamic programming for knapsack

Let $OPT(i, p)$ denote the smallest weight of a subset of items $1, \ldots, i$ such that its profit is exactly p, If no such item exists it is infinity. The recursion of the dynamic programming is

$$OPT(i, p) = \min\{OPT(i-1, p), w_i + OPT(i-1, p-p_i)\}.$$

The bound conditions are $OPT(i, p) = +\infty$ if $i = 0$, and $OPT(i, p) = OPT(i-1, p)$ if $p_i > p$.

The optimal profit is bounded by np_{max}, and then the running time of the dynamic programming is $O(n^2 p_{max})$, where $p_{max} = \max p_i$.

11.3.4 FPTAS *

For each instance of knapsack, denote $p_{max} = \max_i p_i$ to be the maximum profit of all items. For any given $\epsilon > 0$, we use rounding technique to redefine the profit of each item $p'_i = \lfloor \frac{n}{\epsilon p_{max}} p_i \rfloor$.

We denote this new instance by N'. We apply the dynamic programming to run this instance N' and output the solution A. Note that the solution A is an optimal solution for N' and a feasible solution for N.

Theorem 11.11 *The output A achieves a profit of at least $(1-\epsilon)OPT$, where OPT is the optimal value for the knapsack problem. The running time is $O(n^3/\epsilon)$. Hence, the knapsack problem admits an FPTAS.*

Proof: The running time of dynamic programming is $O(n^2 p_{max})$ for the instance N, and hence $O(n^3/\epsilon)$ for the instance N' because of $p'_{max} \le n/\epsilon$.

To show the approximation ratio. Let $\alpha = \frac{n}{\epsilon}\frac{1}{p_{max}}$. Note that for any subset of $S \subseteq N$, let $p(S)$ be the total profit of items in $S \subseteq N$, and $p'(S)$ be the total profit of items in $S \subseteq N'$ (i.e., $p'(S)$ is the sum of profit S after rounding).

By rounding, we have $\alpha p(S) - |S| \le p'(S) \le \alpha p(S)$.

Let A^* be the optimal solution. Thus, $p'(A) \ge p'(A^*)$ because A is the optimal solution of the dynamic programming for N'. So,

$$\begin{aligned} p(A) &\ge \frac{1}{\alpha} p'(A) \\ &\ge \frac{1}{\alpha} p'(A^*) \\ &\ge \frac{1}{\alpha}(\alpha p(A^*) - |A^*|) \\ &\ge p(A^*) - \frac{n}{\alpha} \\ &= p(A^*) - \epsilon p_{max} \\ &\ge p(A^*) - \epsilon p(A^*) = (1-\epsilon)OPT. \end{aligned}$$

■

11.4 K-center

We are given a set of n sites $S = \{s_1, \ldots, s_n\}$, and our goal is to select a subset $C \subseteq S$ of size at most k such that the maximum distance from a site to the nearest center is minimized. Denote $dist(s_i, C)$ to be the distance from s_i to the closest center, *i.e.*, $dist(s_i, C) = \min_{c \in C} dist(s_i, c)$. That is to minimize $\max_{s_i \in S} dist(s_i, C)$.

Let's put it in another way, we would like to select at most k centers C and connect every point $s_i \in S$ to its nearest center $c \in C$ and its connection cost the distance $dist(s_i, c)$, while the goal is to minimize the maximum connection cost overall points $s_i \in S$.

We use the following Greedy-2r algorithm to solve the problem, given a guess of optimal cost r.

```
1 Centers  Greedy-2r ( Sites S[ ], int n, int K, double r )
2 {   Sites  S'[ ] = S[ ]; /*S' is the set of the remaining sites */
3     Centers  C[ ] = ∅;
```

```
4       while (S'[ ] ≠ ∅) {
5           Select any s from S' and add it to C;
6           Delete all s' from S' that are at dist(s', s) ≤ 2r;
7       } /* end-while */
8       if ( |C| ≤ K ) return C;
9       else ERROR(No set of K centers with covering radius at most r);
10  }
```

Theorem 11.12 *Suppose the algorithm Greedy-2r selects more than K centers. Then for any set C^* of size at most K, the covering radius is $r(C^*) > r$.*

Proof: Let $C^* = \{c_1, c_2, \ldots, c_K\}$ be the K centers of an optimal solution with covering radius r, and $B_j = \{s_i | dist(s_i, c_j) \leq r\}$ be the set of sites whose distance to the center c_j is at most r.

Fact: the distance between any two sites in the same ball B_j is at most $2r$, *i.e.*, $dist(u, v) \leq 2r$ for all $u, v \in B_j$. (Here, the distance shall be the metric distance)

We claim that in each iteration of the Greedy-$2r$ algorithm, we delete at least one ball B_j from S'. That is, suppose S' is not empty, and we select any $s \in S'$. Note that s must be in one ball B_j for $j = 1, \ldots, K$. Recall that the distance between s and any site in B_j is at most $2r$. Since we delete all sites s' with $dist(s', s) \leq 2r$ from S', we know that at least the set of B_j will be removed from S'.

Then after K iterations, no ball B_j will be left, *i.e.*, S' will be empty.

In other words, if Greedy-$2r$ selects more than K centers, then an optimal solution must generate a covering radius larger than r. ■

Theorem 11.12 reveals that if we know the optimal cost r, the Greedy-$2r$ algorithm will produce K centers with covering radius at most of $2r$. Of course, the optimal cost r is not known to the algorithm. There are at most $O(n^2)$ possible values for r since r must be the distance between two sites in S. Instead of iterating all these $O(n^2)$ values, we use binary search to find such r.

A smarter solution that does not need to enumerate the possible radius.

```
1   Centers  Greedy-Kcenter ( Sites S[ ], int n, int K )
2   {   Centers  C[ ] = ∅;
3       Select any s from S and add it to C;
4       while ( |C| < K) {
5           Select s from S with maximum dist(s, C);
6           Add s it to C;
7       } /* end-while */
8       return C;
9   }
```

Theorem 11.13 *The approximation ratio of the algorithm Greedy-Kcenter is at most* 2.

Proof: Let $C^* = \{c_1, c_2, \ldots, c_K\}$ be the K centers of an optimal solution with covering radius r, and $B_j = \{s_i | dist(s_i, c_j) \leq r\}$ be the set of sites whose distance to the center c_j is at most r.

The algorithm Greedy-Kcenter selects exactly K centers $C = \{g_1, g_2, \ldots, g_K\}$ by the order of selecting. If each center g_i will be in different B_j, *i.e.*, no ball B_j contains two centers in C, then we have the covering radius of C is at most $2r$.

If some ball B_j contains more than one site from C, w.l.o.g, let g_i be the first center that belong the ball which already contains a center g_l with $1 \leq l < i$. Then all sites covered by $\{g_1, \ldots, g_{i-1}\}$ generate the covering radius at most $2r$. Note that the distance between the remaining sites to C is no than the distance g_i to $\{g_1, \ldots, g_{i-1}\}$, which is at most $2r$ (because g_i is one of the two center sites in B_j). In all, a covering radius of the greedy algorithm is at most $2r$. ■

11.4.1 Inapproximability: hardness of approximation

Theorem 11.14 *Unless $P = NP$, there is no ρ-approximation for center-selection problem with metric distance for any $\rho < 2$.*

Proof: If we can obtain a $(2-\epsilon)$-approximation ALG in polynomial time, then we can solve DOMINATING-SET (which is in NP-complete) in polynomial time.

Let $G = (V, E)$ be an instance of DOMINATING-SET. A dominating set in a graph $G = (V, E)$ is a subset of vertices V' such that every vertex in the graph is either in V' or is adjacent to some vertex in V'. The dominating set problem in the decision version is: given a graph $G = (V, E)$ and an integer k, does G has a dominating set of size k?

We construct instance G' of k-center problem with sites V and distances $dist(u, v) = 1$ if $(u, v) \in E$, and $dist(u, v) = 2$ if $(u, v) \notin E$.

Note that G' satisfies the triangle inequality, and the distance $dist(,)$ is metric.

By this construction, we claim that G has dominating set of size k iff there exists k centers $C \in V$ with the covering radius $r(C) = 1$. If G has a dominating set V^* of size k, we select the sites in V^* as the k centers, *i.e.*, $C = V^*$. The covering radius is 1 because every site other than V^* has an edge connected with a site in $C = V^*$ (by the dominating definition). In the other direction, if we have k centers with the covering radius $r(C) = 1$, we say C is a dominate set. Because all other sites in V C connect a site in C with a distance of 1, which implies that there is an edge between this site and a site in C.

Now, we use the approximation algorithm ALG to solve the DOMINATING-SET problem. We run the ALG for G'. If the output $r(C) = 1$, we have shown that there exists a dominating set of size k.

Let C^* be the optimal solution k-center problem. If the output $r(C) > 1$, we claim that $r(C^*) > 1$, which implies that we cannot have a dominating set of size k. We show it by contradiction. Note that $r(C) \leq (2 - \epsilon)r(C^*) = 2 - \epsilon$ if $r(C^*) = 1$, then $r(C) = 1$ because the distance is an integer. ■

References

[1] G. Dósa and J. Sgall. First Fit bin packing: A tight analysis. In N. Portier and T. Wilke, editors, *30th International Symposium on Theoretical Aspects of Computer Science (STACS 2013)*, volume 20 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 538–549, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[2] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, volume 174. W. H. Freeman, San Francisco, 1979.