Approximation Algorithm

Rangaballav Pradhan

ITER, SOADU

Approximation Algorithms

- How to deal with NP-completeness?
  - If the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory.
  - We may be able to isolate important special cases that we can solve in polynomial time.
  - We might come up with approaches to find near-optimal solutions in polynomial time (either in the worst case or the expected case).
- In practice, near-optimality is often good enough.
- We call an algorithm that is not obsessed with the exact optimal solution and settles for a near-optimal solutions an approximation algorithm.

Approximation Ratio

- Consider an optimization problem P in which each potential solution has a positive cost.
- An algorithm A for the problem P has an approximation ratio of $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm A is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

- If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a $\rho(n)$–approximation algorithm.
- For a maximization problem, $0 < C \leq C^*$, and the ratio $\frac{C^*}{C}$ gives the factor by which the cost of an optimal solution is larger than the cost of the approximate solution.

Approximation Ratio

- For a minimization problem, $0 < C^* \leq C$, and the ratio $\frac{C}{C^*}$ gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution.
- The approximation ratio of an approximation algorithm is never less than 1, since $C/C^* \leq 1$ implies $C^*/C \geq 1$.
- A 1-approximation algorithm produces an optimal solution.
- Approximation ratio decrease and tend towards one as the optimality increases.
- Many problems have polynomial-time approximation algorithms with small constant approximation ratios, but for some other problems, the best known polynomial-time approximation algorithms have approximation ratios that grow as functions of the input size n.

Load balancing

Input. Given m identical machines and n ($\geq m$) jobs; job j has processing time t_j.

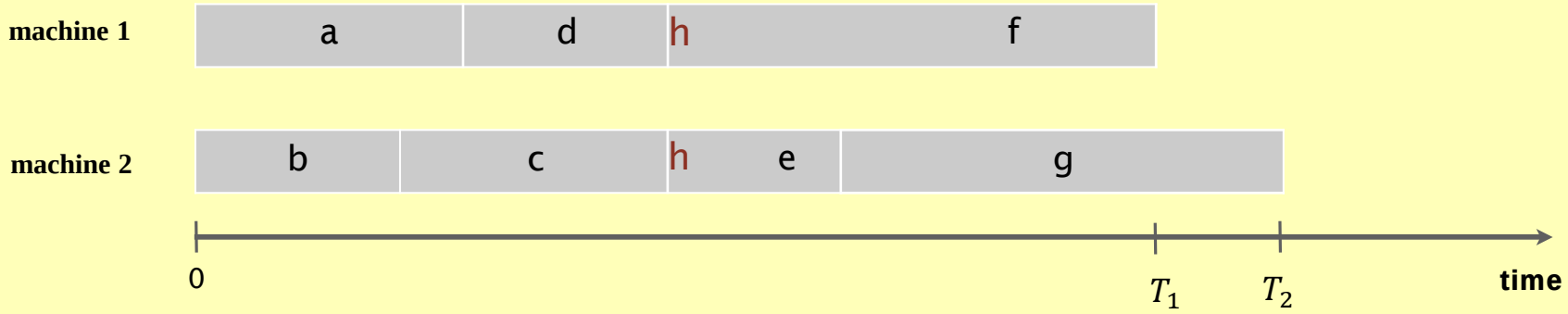
- Job j must run contiguously on one machine.
- A machine can process at most one job at a time.

Def. Let $A[i]$ be the subset of jobs assigned to machine i.

The load of machine i is $T_i = \sum_{j \in A[i]} t_j$

Def. The makespan is the maximum load on any machine $T = \max_i T_i$.

Load balancing. Assign each job to a machine to minimize makespan.



Load balancing: Greedy approach

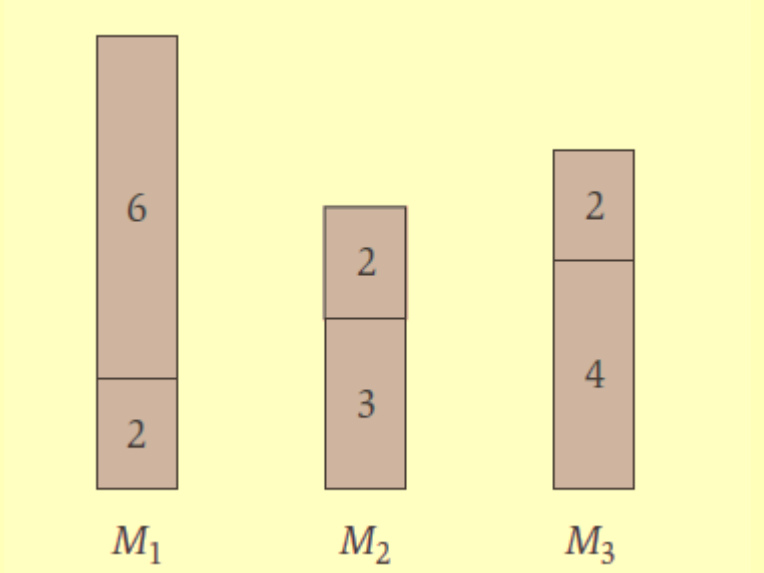
- Consider n jobs in some fixed order.
- Assign job j to machine i whose load is smallest so far.

```
Greedy-Balance:
Start with no jobs assigned
Set T_i = 0 and A(i) = ∅ for all machines M_i
For j = 1, . . . , n
  Let M_i be a machine that achieves the minimum min_k T_k
  Assign job j to machine M_i
  Set A(i) ← A(i) ∪ {j}
  Set T_i ← T_i + t_j
EndFor
```

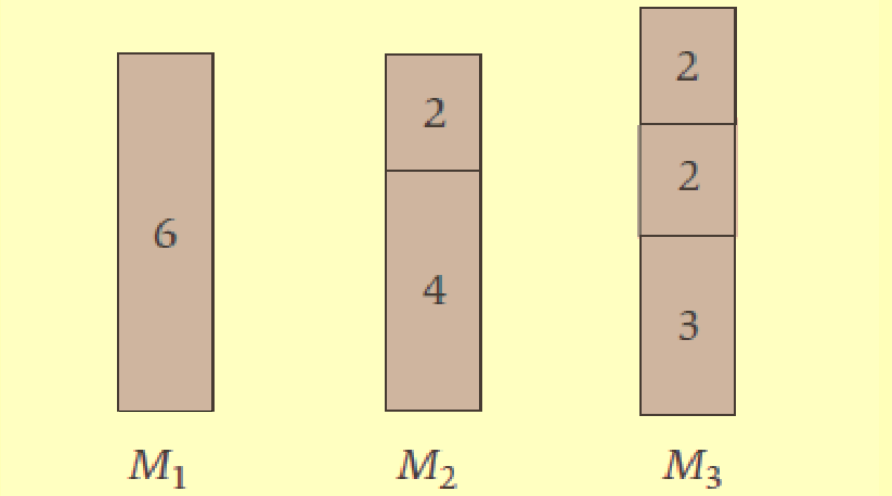
Implementation. $O(n \log m)$ using a priority queue for loads T_k.

Greedy-Balance: Example

- Given a sequence of six jobs with running times 2, 3, 4, 6, 2, 2;



The greedy solution



The optimal solution

- The greedy solution is not optimal.
- If the jobs arrived in a different order, say 6, 4, 3, 2, 2, 2, then it would have produced an allocation with a makespan of 7.

Greedy-Balance: Analysis

- We have to show that, the approximation produces a near optimal solution.
- We can do this by comparing the solution obtained by the Greedy method (say T) to the optimal solution (say T^*).
- But this is a logically impossible task as we do not know about the optimal solution.
- So, we can put some bounds on the optimal solution and compare this bound with the greedy solution.
- **1st Bound:** The total processing time $= \sum_j t_j \ \forall j$, which is divided among all the machines.
- So, one of the m machines must do at least $1/m$ parts of it.
- Total makespan $T^* \geq \frac{1}{m} \sum_j t_j$ (1)

Greedy-Balance: Analysis

- **2nd Bound:** Total makespan $T^* \geq \max_j t_j$ (2)
- Let's consider T = the makespan of Greedy-Balance
- Let M_i be a machine with the highest load i.e., T_i. So, $T = T_i$.
- Let j be the last job to be assigned to M_i.
- Before assigning the job j, M_i has the load $= T_i - t_j$ which is minimum among all the machines. So, at this point, every machine has a load of at least $T_i - t_j$.
- Now, the total load on all machines at this point is $\sum_k t_k \geq m \cdot (T_i - t_j)$
- As per the 1st bound, $\frac{1}{m}\sum_k t_k \leq T^*$

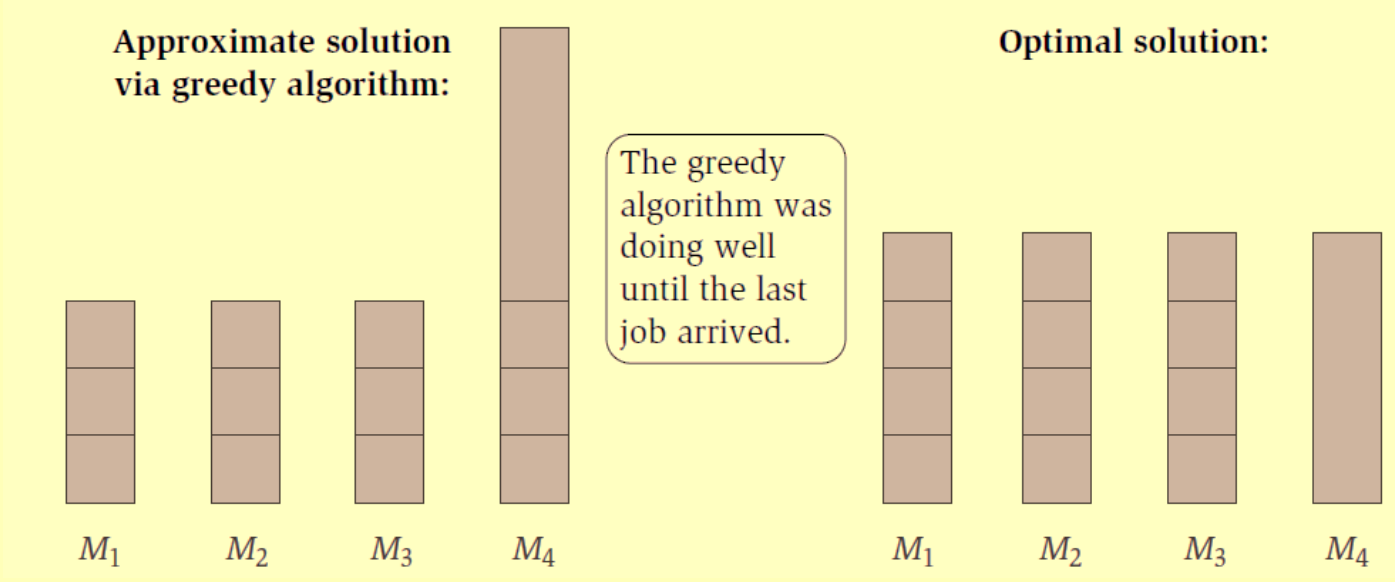
  $\Rightarrow (T_i - t_j) \leq T^*$ (3)
- From bound 2, $t_j \leq T^*$ (4)
- Adding equation (3) and (4), we get, $T_i \leq 2T^*$

  $\Rightarrow T \leq 2T^*$

Greedy-Balance: Worst-case Example

- Suppose in a general situation, we have m machines and $n = m(m-1) + 1$ jobs. Each of the first $m(m-1)$ jobs has 1 unit of processing time and the last job is much larger with processing time $t_n = m$.
- If we apply Greedy-Balance to this instance, the first $n-1$ jobs will be distributed evenly over the m machines with current processing time/load on each machine as $m-1$.
- Now, the last job will be added to one of the m machines incurring a total makespan $= (m-1) + m = 2m - 1$

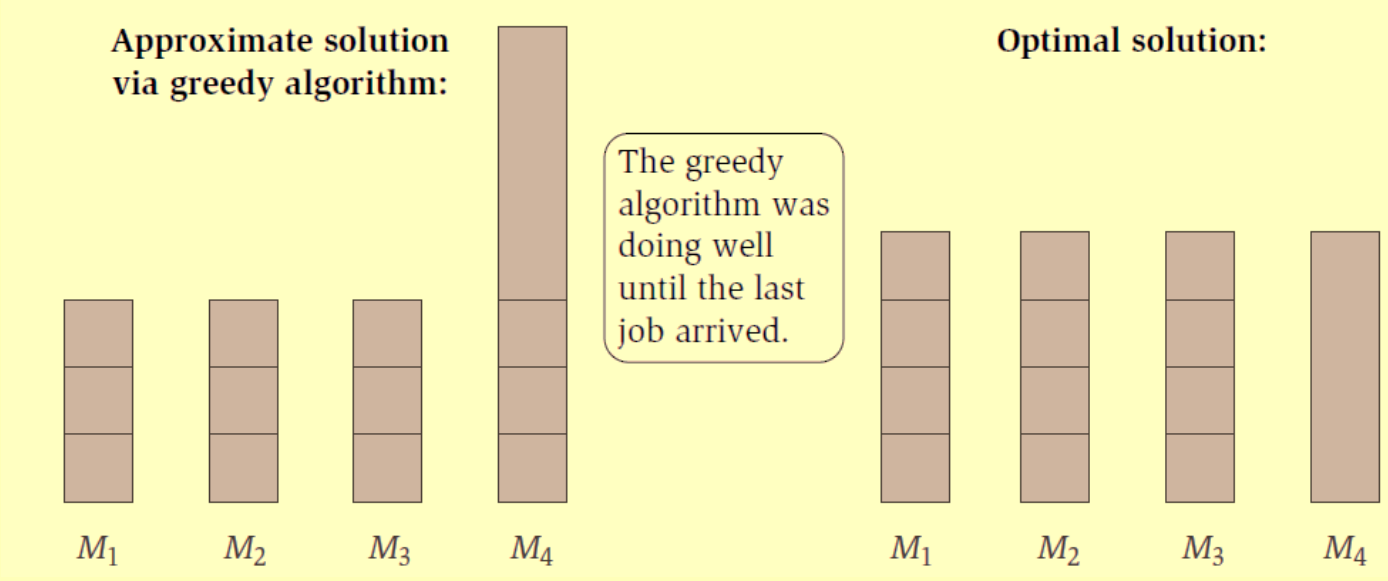
An example of the general instance with m = 4



Greedy-Balance: Worst-case Example

- However, the optimal schedule will assign the largest job to a single machine and all the remaining $m(m-1)$ jobs to the remaining $(m-1)$ machines incurring a makespan of m.
- Here, the Greedy-balance solution is an approximation solution which deviates from the optimal solution by a factor of $\frac{2m-1}{m} = (2 - {}^{1}/_{m})$ which is close to 2 when m is large.

An example of the general instance with m = 4



Sorted-Balance: Improved solution

- In the previous example, the Greedy-Balance algorithm distributed the first n-1 jobs evenly and was doing good until the last job (largest) arrived, which increased the makespan value.

- This gave us some idea that if we can take care of the larger jobs fist then the smaller ones, then we can improve the performance of our greedy solution.

- We can modify our greedy solution to consider the jobs in decreasing order of the processing time.

Sorted-Balance: Improved solution

Sorted-Balance:

Start with no jobs assigned

Set $T_i = 0$ and $A(i) = \emptyset$ for all machines M_i

Sort jobs in decreasing order of processing times t_j

Assume that $t_1 \geq t_2 \geq \ldots \geq t_n$

For $j = 1, \ldots, n$

  Let M_i be the machine that achieves the minimum $\min_k T_k$

  Assign job j to machine M_i

  Set $A(i) \leftarrow A(i) \cup \{j\}$

  Set $T_i \leftarrow T_i + t_j$

EndFor

Sorted-Balance: Analysis

- If there are at least n machines available, then the greedy solution is equal to the optimal solution as this lead to the trivial instance of the problem ($m \geq n$).

Claim. If there are more than m jobs, $T^* \geq 2\,t_{m+1}$.

- Pf. The first m jobs will be allocated one-by-one to the m machines and the $(m+1)^{th}$ job will be allocated any of these m machines depending upon the smallest current load value.
- At this point, each of the machines have load at most t_{m+1} as we have selected the jobs in decreasing order of the processing times.
- Adding $(m+1)^{th}$ job to a machine will increase its load to a value at least $2\,t_{m+1.}$
- This gives a lower bound on the optimal maespan as, $T^* \geq 2\,t_{m+1}$

Sorted-Balance: Analysis

Claim. If there are more than m jobs, $T^* \geq 2\,t_{m+1}$.

- Pf. The first m jobs will be allocated one-by-one to the m machines and the $(m+1)^{th}$ job will be allocated any of these m machines depending upon the smallest current load value.
- At this point, each of the machines have load at most t_{m+1} as we have selected the jobs in decreasing order of the processing times.
- Adding $(m+1)^{th}$ job to a machine will increase its load to a value at least $2\,t_{m+1}$.
- This gives a lower bound on the optimal maespan as, $T^* \geq 2\,t_{m+1}$ (5)
- Recall from the analysis of Greedy-Balance, we put two bounds on the optimal makespanas:
- $T^* \geq \frac{1}{m}\sum_j t_j$ (1)
- $T^* \geq \max_j t_j$ (2)

Sorted-Balance: Analysis

- We will continue our analysis in a similar way as the Greedy-Balance.
- Let's consider T = the makespan of Greedy-Balance
- Let M_i be a machine with the highest load i.e., T_i. So, $T = T_i$.
- Let j be the last job to be assigned to M_i.
- Before assigning the job j, M_i has the load $= T_i - t_j$ which is minimum among all the machines. So, at this point, every machine has a load of at least $T_i - t_j$.
- Now, the total load on all machines at this point is $\sum_k t_k \geq m \cdot (T_i - t_j)$
- As per the 1st bound, $\frac{1}{m}\sum_k t_k \leq T^*$

$$\Rightarrow \left(T_i - t_j\right) \leq T^* \qquad (3)$$

- From equation (5), $t_{m+1} \leq \frac{1}{2}T^*$ (6)

Sorted-Balance: Analysis

- As j $\geq m + 1$, $\Rightarrow t_j \leq t_{m+1}$
- From equation (4), we have $t_j \leq t_{m+1} \leq \frac{1}{2}T^*$

$$\Rightarrow t_j \leq \frac{1}{2}T^* \qquad (7)$$

- Adding equation (3) and (7), we get, $T_i \leq \frac{3}{2}T^*$

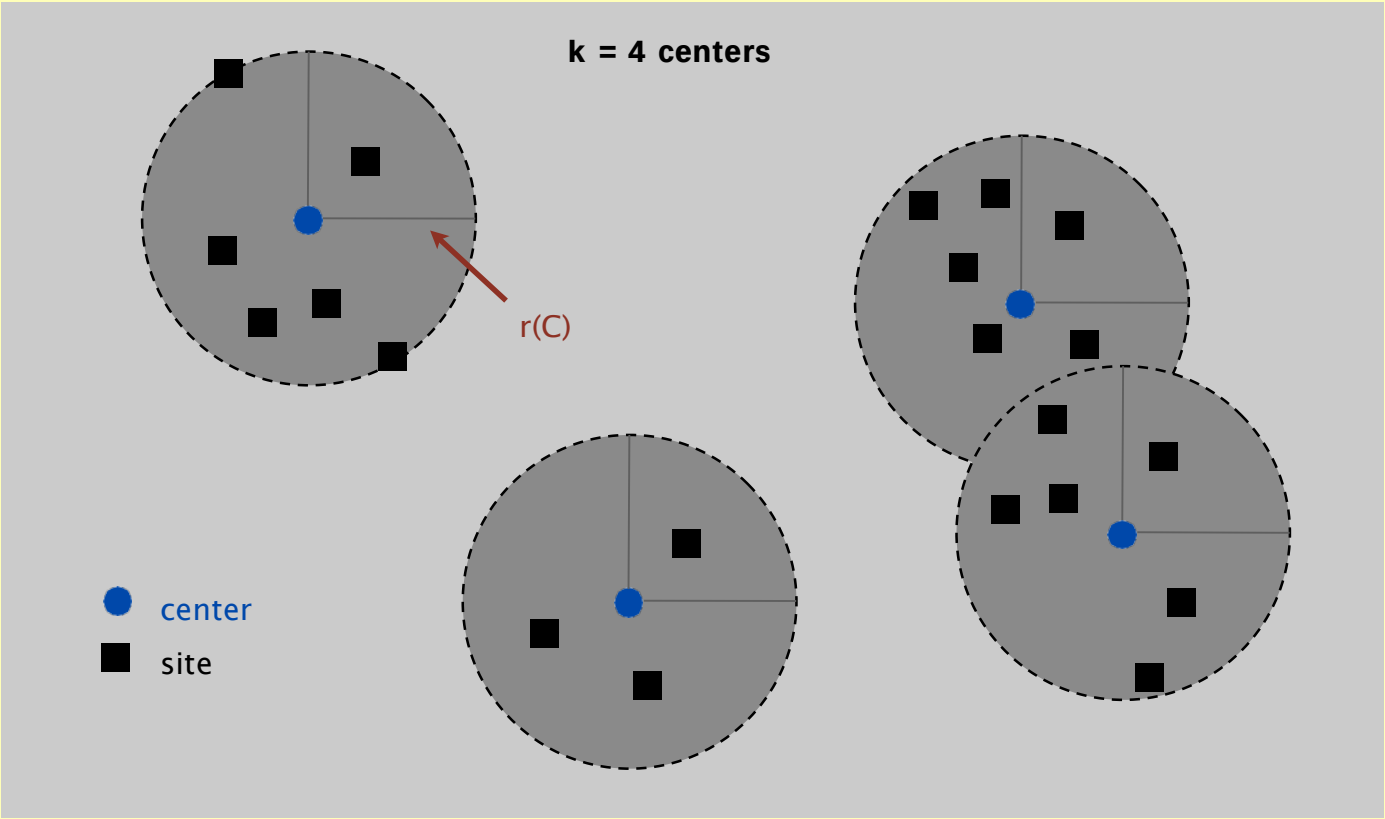
$$\Rightarrow T \leq \frac{3}{2}T^*$$

- Sorted-Balance for load balancing is a $\frac{3}{2}$-approximation algorithm.

Center Selection Problem

Input. Set S of n sites $S = \{s_1, \ldots, s_n\}$ and an integer $k > 0$.

Center selection problem. Select set C of k centers so that maximum distance $r(C)$ from a site to its nearest center is minimized.



Center Selection Problem

Input. Set S of n sites $S = \{s_1, \ldots, s_n\}$ and an integer $k > 0$.

A distance function is defined between every pair of sites and has the following properties:

i. Identity: $d(s, s) = 0 \quad \forall s \in S$

ii. Symmetry: $d(s, t) = d(t, s) \quad \forall s, t \in S$

iii. Triangle inequality: $d(s, t) \leq d(s, u) + d(u, t) \quad \forall s, t, u \in S$

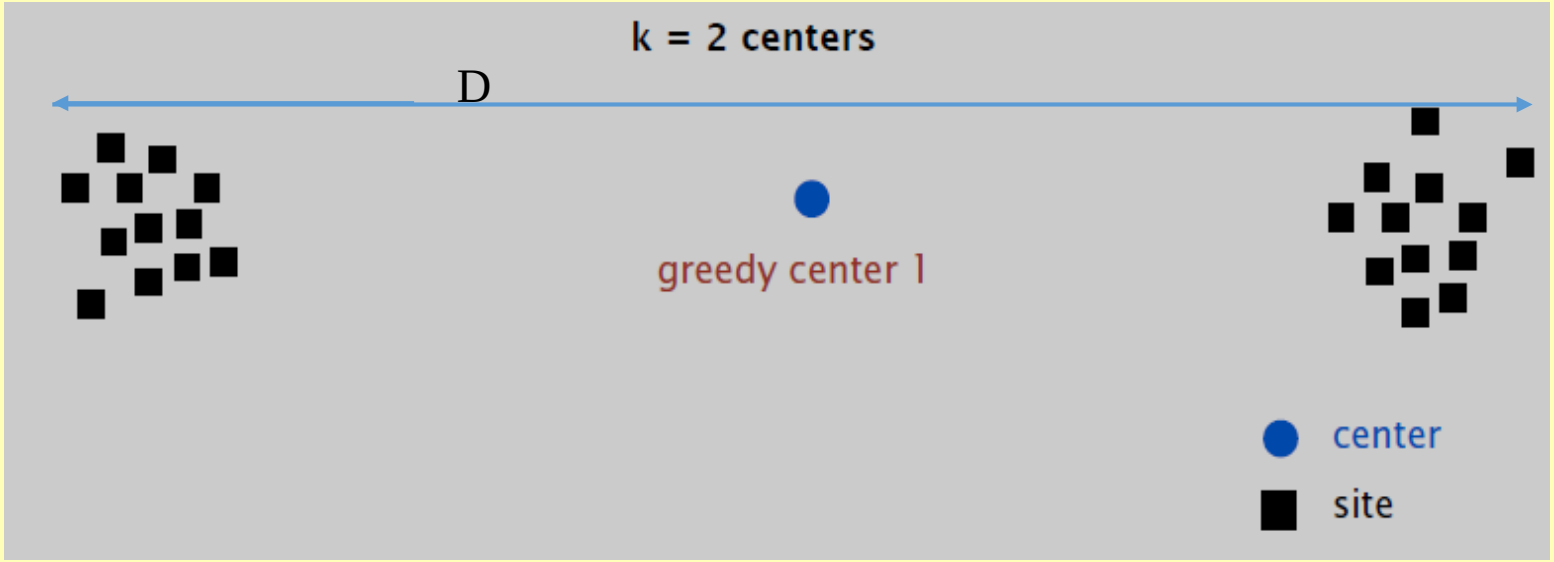
- Let C be the set of k centers, the distance between any site $s \in S$ and C is defined as, $d(s, C) = \min_{c' \in C} d(s, c')$
- C is said to form a r-cover if each site is within a distance r from one of the centers. i.e., $d(s, C) \leq r \quad \forall s \in S$

Center Selection Problem

- The minimum value of r for which C is an r-cover is called the Covering Radius of C and is denoted as $\mathrm{r}(C)$.
- The Covering Radius of a set of centers C is the farthest anyone needs to travel to get to their nearest center.
- Goal. To select a set C of k centers for which $\mathrm{r}(C)$ is as minimum as possible.

Center Selection: Towards a solution

- A simple Greedy Strategy. Put the first center at the best possible location for a single center, and then keep adding centers so as to reduce the covering radius each time by as much as possible.
- Remark: arbitrarily bad!
- Irrespective of the position of the greedy center 2, the covering radius of set of 2 centers will always D/2.



- However, the optimal solution would select two centers from each cluster to minimize the covering radius.

Greedy Solution with knowledge of Optimal Covering Radius

- If the optimal covering radius $r(C^*) = r$ is known, then we can use it to find a set of k centers with covering radius at most $2r$.
- Idea. Consider any site $s \in S$. There must be a center $c^* \in C^*$ that covers s, and this center c^* is at distance at most r from s.
- Now our idea would be to take this site s as a center in our solution instead of c^* and make s cover all the sites that c^* covers in the (unknown) solution C^*. This is accomplished by expanding the radius from r to $2r$.
- All the sites that were at distance at most r from center c^* are at distance at most $2r$ from s (by the triangle inequality).

Greedy Solution with knowledge of Optimal Covering Radius

GreedyCS_r (S, d, k, r)

$S' \leftarrow$ Sites not covered yet

Initialize $S' \leftarrow S$

Set $C \leftarrow \emptyset$

While $S' \neq \emptyset$

Select any site s $\in S'$

Add s to C: $C \leftarrow C \cup \{s\}$

Delete all sites from S'within a distance $2r$ from s

End while

If $|C| \leq k$

return C as the set of selected centers

Else

claim that no set is possible of k centers with covering radius at most r

Greedy Solution with knowledge of Optimal Covering Radius

Theorem 1. Any set of centers C returned by the greedy algorithm has covering radius $r(C) \leq 2r$ where r is the optimal covering radius.

Pf. The proof is trivial as our Greedy algorithm after selecting a center deletes all the sites within a distance $2r$ from it.

- For the remaining sites, which are at a distance $> 2r$, the algorithm selects another center and so on.
- So, the distance from any site to its nearest center can never be more than $2r$.

Greedy Solution with knowledge of Optimal Covering Radius

Theorem 2. If the algorithm selects more than k centers, then, for any set C^* of size at most k, the covering radius is $r(C^*) > r$.

Pf. (by contradiction)

- Assume that, the algorithm selects more than k centers and there is a set C^* of at most k centers with covering radius $r(C^*) \leq r$.
- Each center $c \in C$ selected by the greedy algorithm is one of the original sites in S, and the set C^* covers all sites.
- So each center $c \in C$ comes under the covering radius of an optimal center $c^* \in C^*$.
- There are at most k optimal centers and more than k greedy centers. So, at least once optimal center exists that covers two greedy centers. Let $c^* \in C^*$ be the optimal center that covers two greedy centers c_1 and c_2. i.e., $d(c_1, c^*) \leq r$ and $d(c^*, c_2) \leq r$.

Greedy Solution with knowledge of Optimal Covering Radius

Theorem 2. If the algorithm selects more than k centers, then, for any set C^* of size at most k, the covering radius is $r(C^*) > r$.

Pf. (by contradiction)

- Now, If we consider the distance $d(c_1, c_2)$ between c_1 and c_2, as per the triangle inequality property of the distance function,

$$\begin{aligned} d(c_1, c_2) &\leq d(c_1, c^*) + d(c^*, c_2) \\ &\leq r + r \\ &\leq 2r \qquad \text{(A contradiction)} \end{aligned}$$

- The above relation is a contradiction because, as per our greedy algorithm the distance between any two centers have a distance of at least $2r$ as before selecting any new center, the greedy algorithm deletes all the sites that are within a distance of $2r$ from the already selected center.
- Hence, proved that no C^* can be possible with $r(C^*) > r$.

Greedy Solution: Center Selection

- The previous greedy algorithm used the knowledge of r to select the next center by removing all the sites (potential centers) within $2r$ distance from the currently selected center (s).
- Idea. We can achieve this without knowing r, by selecting the next center to be a site which is the farthest from the nearest center.
- If we can find any site which is at a distance more than $2r$ from all the previously selected centers, then the farthest site must be one of such centers.

Greedy Solution: Center Selection

Greedy-Center-Selection(S, d, k)

Assume that $k \leq |S|$ (If not then return)
Select any site s $\in S$ and let $C \leftarrow \{s\}$
While $|C| < k$
    Select a site s $\in S$ that maximizes $d(s, C)$
    Add s to C: $C \leftarrow C \cup \{s\}$
End while
return C as the set of selected centers

Greedy Solution: Center Selection

- Theorem 3. The greedy algorithm Greedy-Center-Selection () returns a set C of k centers such that $r(C) \leq 2r(C^*)$, where C^* is an optimal set of k centers.

Pf. Let $r = r(C^*)$ denote the minimum possible radius of a set $C^*$of k centers.

- We can prove the claim using contradiction.
- Assume that, Using our Greedy-Center-Selection algorithm, we obtained a set C of k centers with $r(C) > 2r$.
- As $r(C) > 2r$, there exist a site s which is at a distance greater than $2r$ from C.
- Let c_1 be the first center selected by greedy, c_2 be the second center selected by greedy,… and so on.
- Let C^1 be the center set after the first iteration of greedy, C^2 be the center set after the second iteration of greedy,… and so on.

Greedy Solution: Center Selection

- Theorem 3. The greedy algorithm Greedy-Center-Selection () returns a set C of k centers such that $r(C) \leq 2r(C^*)$, where C^* is an optimal set of k centers.

Pf. As the greedy algorithm selects the next center as the farthest available site from the current center set, the distance between the center set and the subsequently selected centers decreases gradually and we have a site s which is at a distance greater than $2r$ from $C = C^k$, we have:

$$d(c_2, C^1) \geq d(c_3, C^2) \ldots\ldots\ldots \geq d(c_k, C^{k-1}) \geq d(s, C^k) > 2r$$

- The above relation validates that the centers selected in each of the k iterations are more than $2r$ distance apart$-$This is exactly what we obtained in the previous greedy algorithm (GreedyCS_r ()). So, the first k iterations of both the grrdy algorithms are consistent with each other.

Greedy Solution: Center Selection

- Theorem 3. The greedy algorithm Greedy-Center-Selection () returns a set C of k centers such that $r(C) \leq 2r(C^*)$, where C^* is an optimal set of k centers.

Pf. Now, at the end of our second greedy algorithm (Greedy-Center-Selection ()), we have a site s present at more than $2r$ distance from $C = C^k$.

- As we have already established that both the greedy algorithms created the k centers at least $2r$ distance apart from each other, the first greedy algorithm must also have such a site s after the k^{th} iteration.
- Then the first greedy algorithm continue its execution in the $(k+1)^{th}$ iteration as well and will go on to select more than k centers.
- If the first greedy algorithm selects more than k centers, then there cannot exist an optimal solution with $r(C^*) = r$ (we already proved it in Theorem 2) – a contradiction to our assumption that $r(C^*) = r$.

Set Cover Problem

Def. A set U of n elements and a list $S_1, \ldots, S_m$ of subsets of U, a *set cover* is a collection of these sets whose union is equal to all of U.

- In the version of the problem we consider here, each set S_i has an associated weight $w_i \geq 0$. The goal is to find a set cover C so that the total weight $\sum_{S_i \in C} w_i$ is minimized.
- If we set all $w_i = 1$, then the minimum weight of a set cover is at most k if and only if there is a collection of at most k sets that covers U.

Set Cover Problem: Greedy Solution

Idea. The next set selected by greedy is the one with the following desired properties:

- Smaller weight
- Greater coverage (large number of elements)

- To design a solution, the above two criteria can be combined as a single measure, $\frac{w_i}{|S_i|}$ which denotes the weight/cost per element for set S_i.
- However, the measure if considered alone does not guarantee the coverage of new elements as a set S_i with the minimum $\frac{w_i}{|S_i|}$ value can have the elements that are already covered previously through some other set.
- So, to make our approach precise, we can consider a parameter $\frac{w_i}{|S_i \cap R|}$ to select the next set into the set cover, where R denotes the set of remaining/uncovered elements of U.

Greedy Algorithm for Set Cover

Greedy-Set-Cover $(U, S_1, \ldots, S_m, W)$

Initialize $R \leftarrow U$ (No element is covered yet)

Initialize $C \leftarrow \emptyset$ (No set is selected yet)

While $R \neq \emptyset$

Select a set S_i that minimizes $\frac{w_i}{|S_i \cap R|}$

Add s to C: $C \leftarrow C \cup \{S_i\}$

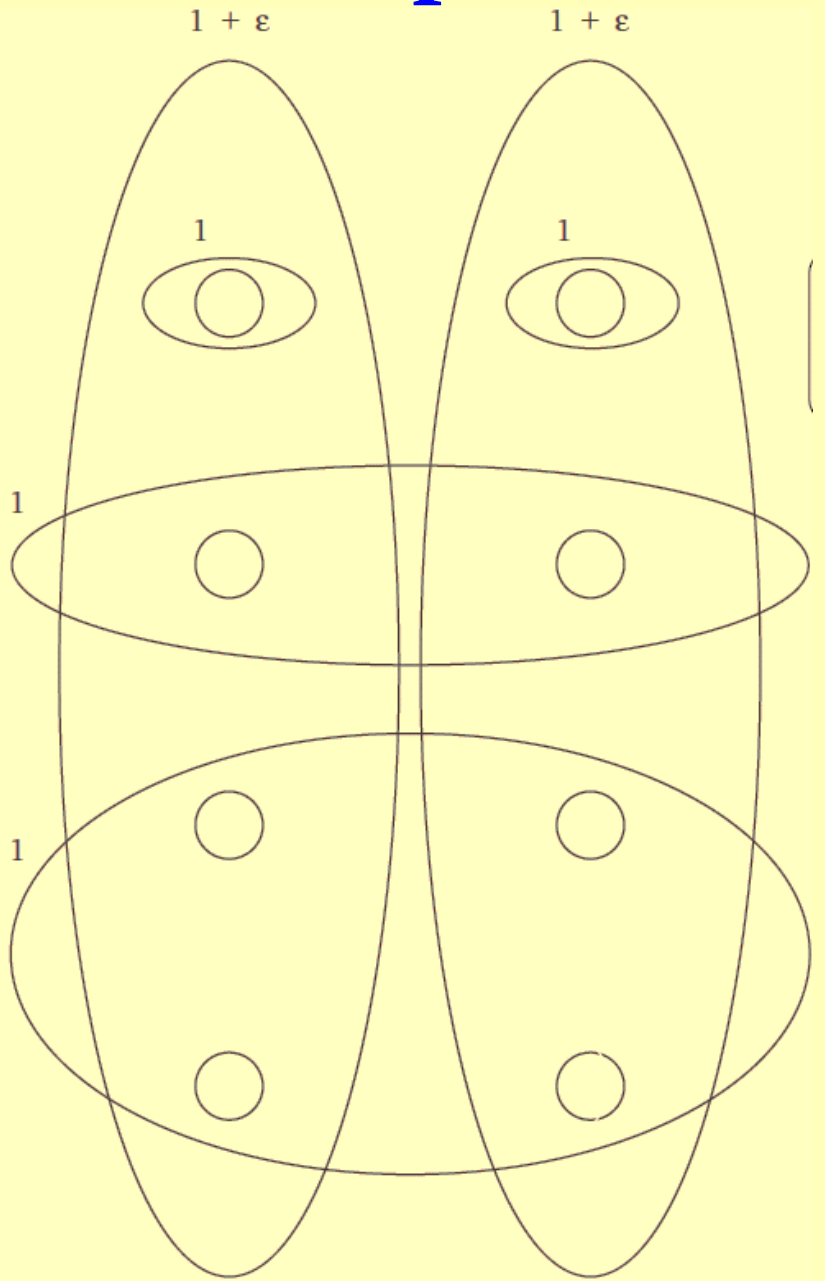
Delete the elements of S_i from R: $R \leftarrow R - S_i$

End while

return C as the set cover

Greedy Algorithm for Set Cover: Example

- In the given figure, the greedy-set-cover would first choose the set containing the four nodes at the bottom (since this has the best (minimum) weight-to-coverage ratio, 1/4).
- It then chooses the set containing the two nodes in the second row, and finally it chooses the sets containing the two individual nodes at the top.
- It thereby chooses a collection of sets of total weight 4.
- The greedy algorithm misses the fact that there's a way to cover everything using a weight of just $2+ 2\varepsilon$, by selecting the two sets that each cover a full column.



Greedy Algorithm for Set Cover: Analysis

- The Greedy-Set-Cover selects the sets into the set cover C by checking the average cost per coverage $\frac{w_i}{|S_i \cap R|}$ which can also be viewed as the cost paid for covering a new element.
- Let c_s denotes the cost associated with an element s. We can compute c_s for each $s \in S_i$ when S_i is selected by the Greedy-Set-Cover into C as, $c_s = \frac{w_i}{|S_i \cap R|}$
- Whenever a set S_i is selected into C, its weight is distributed as costs over each of the newly covered elements of $(S_i \cap R)$. These costs of the elements account for the total weight of the set cover C.

$$\sum_{S_i \in C} w_i = \sum_{s \in U} c_s \qquad (1)$$

Greedy Algorithm for Set Cover: Analysis

- Now, we can look at "How much total cost a single set S_k can contribute with respect its weight w_k?". The following theorem gives a bound on this value.
- Theorem 1. For every set S_k, the sum of the cost of each element $s \in S_k$ is at most $H(|S_k|) \cdot w_k$. *i.e.*,

$$\sum_{s \in S_k} c_s \leq H(|S_k|) \cdot w_k \qquad (2)$$

- Here, H is the Harmonic function and is defined as,

$$H(n) = \sum_{i=1}^{n} \frac{1}{i}$$

- $H(n)$ is naturally bounded by $\ln(n+1) \leq H(n) \leq 1 + \ln n$. Hence, $H(n) = \theta(\ln n)$

Greedy Algorithm for Set Cover: Analysis

- Theorem 1. For every set S_k, the sum of the cost of each element $s \in S_k$ is at most $H(|S_k|) \cdot w_k$.
- Pf. To simplify the notations, assume that the elements of S_k are the first $\mathrm{d} = |S_k|$ elements of U. i.e., $\mathrm{d}\ S_k = \{s_1, s_2, \ldots, s_d\}$ and these elements are labelled in the order in which they are assigned a cost c_{s_j} by the greedy algorithm.
- Now, consider the iteration in which element s_j is covered by the greedy algorithm for some $j \leq d$.
- Just before this iteration, all the elements s_j, $s_{j+1}, \ldots$, $s_d \in R$ as the elements are labelled in the order $s_1, s_2, \ldots, s_d$ in which they are getting covered in C.

$$\Rightarrow |S_k \cap R| \geq (d - j + 1)$$

Greedy Algorithm for Set Cover: Analysis

- Theorem 1. For every set S_k, the sum of the cost of each element $s \in S_k$ is at most $H(|S_k|) \cdot w_k$.
- Pf. So, the average cost of the set S_k is bounded as,

$$\frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{(d - j + 1)}$$

- Let's say, in this iteration, the greedy algorithm selected a set S_i with minimum average cost per new element. So, S_i has an average cost which is at most that of the set S_k as we are selecting the sets in increasing order of the average costs.
- The element s_j is assigned with a cost c_{s_j} that is equal to the average cost of S_i. So,

$$c_{s_j} = \frac{w_i}{|S_i \cap R|} \leq \frac{w_k}{|S_k \cap R|} \leq \frac{w_k}{(d - j + 1)}$$

Greedy Algorithm for Set Cover: Analysis

- Theorem 1. For every set S_k, the sum of the cost of each element $s \in S_k$ is at most $H(|S_k|) \cdot w_k$.
- Pf. Now, simply adding these inequalities for all the elements of the set S_k ,we get,

$$\sum_{s \in S_k} c_s = \sum_{j=1}^{d} c_{s_j} \leq \sum_{j=1}^{d} \frac{w_k}{(d-j+1)} = w_k \left[\frac{1}{d} + \frac{1}{d-1} + \cdots + 1\right]$$

$$= w_k \cdot H(d)$$

$$= H(|S_k|) \cdot w_k$$

$$\Rightarrow \sum_{s \in S_k} c_s \leq H(|S_k|) \cdot w_k$$

Greedy Algorithm for Set Cover: Analysis

- Theorem 2. The set cover C selected by Greedy-Set-Cover has weight at most $H(d^*)$ times the weight of the optimal set cover, where $d^* = \max_j |S_j|$ is the maximum size of any given set.
- Pf. The theorem states that $w \leq H(d^*) \cdot w^*$, where w is the weight of the greedy set cover C and w^* is the weight of the optimal set cover C^*.

$$w^* = \sum_{S_i \in C^*} w_i \qquad (3)$$

From the previous theorem, for any set S_k,

$$\sum_{s \in S_k} c_s \leq H(|S_k|) \cdot w_k$$

$$\Rightarrow w_k \geq \frac{1}{H(d^*)} \sum_{s \in S_k} c_s$$

Greedy Algorithm for Set Cover: Analysis

- Theorem 2. The set cover C selected by Greedy-Set-Cover has weight at most $H(d^*)$ times the weight of the optimal set cover, where $d^* = \max_j |S_j|$ is the maximum size of any given set.
- Pf. Putting this bound for S_k in equation (3) we get,

$$w^* \geq \sum_{S_i \in C^*} \left(\frac{1}{H(d^*)} \sum_{s \in S_i} c_s \right)$$

$$\geq \frac{1}{H(d^*)} \sum_{S_i \in C^*} \left(\sum_{s \in S_i} c_s \right)$$

$$\geq \frac{1}{H(d^*)} \sum_{s \in U} c_s \qquad (4)$$

Greedy Algorithm for Set Cover: Analysis

- Theorem 2. The set cover C selected by Greedy-Set-Cover has weight at most $H(d^*)$ times the weight of the optimal set cover, where $d^* = \max_j |S_j|$ is the maximum size of any given set.
- Pf. Putting the value of $\sum_{s \in U} c_s$ from equation (1) in the inequality (4) we get,

$$w^* \geq \frac{1}{H(d^*)} \sum_{S_i \in C} w_i$$

$$\Rightarrow w^* \geq \frac{1}{H(d^*)} \cdot w$$

$$\Rightarrow w \leq H(d^*) \cdot w^*$$