APPROXIMATION ALGORITHMS
FOR
GEOMETRIC PROBLEMS

Subhas C. Nandy
(nandysc@isical.ac.in)

Advanced Computing and Microelectronics Unit
Indian Statistical Institute
Kolkata 700108, India.
Organization

- Introduction
- Approximation Algorithms for Intersection Graphs
- Approximation Algorithm for Art Gallery Problem
- Approximation Algorithm for Some Important Geometric Optimization Problems
Introduction

Criteria for a good algorithm for a computational problem

- Fast execution time,
- Requirement of small extra space in addition to the space required for storing the input, and
- Easy to code.
Introduction

Criteria for a good algorithm for a computational problem

- Fast execution time,
- Requirement of small extra space in addition to the space required for storing the input, and
- Easy to code.

Measures of execution time

- Actual CPU time for running the program,
- Asymptotic running time.
Introduction

Assymptotic Running time

- Worst case time complexity.
- Average case time complexity.
Introduction

**Assymptotic Running time**
- Worst case time complexity.
- Average case time complexity.

**Worst case running time**
An upper bound on the running time of the algorithm for any input.
Introduction

**Assymptotic Running time**

- Worst case time complexity.
- Average case time complexity.

**Worst case running time**

An upper bound on the running time of the algorithm for any input.

**Average case running time**

The average over the time required to run the algorithm for all possible inputs, where the input is assumed to follow some distribution (known in advance).
Introduction

Assymptotic Running time

- Worst case time complexity.
- Average case time complexity.

Worst case running time

An upper bound on the running time of the algorithm for any input.

Average case running time

The average over the time required to run the algorithm for all possible inputs, where the input is assumed to follow some distribution (known in advance).

The time complexity is measured as a function of the input size, or the output size, or both.
Tractibility versus Intractibility

- A problem is said to be tractable if there exists an algorithm for that problem whose worst case time complexity is a polynomial function in $n$ (the input size).
- Otherwise, the problem is said to be intractable.
Tractibility versus Intractibility

- A problem is said to be **tractable** if there exists an algorithm for that problem whose worst case time complexity is a polynomial function in $n$ (the input size).
- Otherwise, the problem is said to be **intractable**.

**Complexity Classes**

**P:** A decision problem that can be solved in polynomial time.

**NP:** A decision problem that, for an input, can be verified in polynomial time.
NP-Completeness

An important question

Whether $P = NP$ or $P \neq NP$?

NP-Complete Class: A class of decision problems in NP such that if one of them can be solved in polynomial time, all other problems can also be solved in polynomial time.
Proof for NP-Completeness

For the given problem,

Step 1: Design a non-deterministic polynomial time algorithm for the problem, and

Step 2: Reduce a known NP-complete problem to the given problem in polynomial time.
NP-Complete Problems

- Circuit satisfiability,
- 3-SAT (satisfiability where the expression is in CNF, and each clause is of length at most 3),
- Clique decision problem,
- Vertex Cover decision,
- Travelling salesman problem,
- Knapsack problem,
- Bin packing
- Art gallery problem
to name only a few.
Handling the NP-Hard Problems

**Brute-force algorithm**

Design a clever enumeration strategy, so that
- it guarantees the optimality of the solution, but
- no guarantee on the running time.
Handling the NP-Hard Problems

Brute-force algorithm
Design a clever enumeration strategy, so that
- it guarantees the optimality of the solution, but
- no guarantee on the running time.

Heuristic
Develop an intuitive algorithm, such that
- it guarantees to run in polynomial time, but
- no guarantee on the quality of the solution.
Approximation Algorithm

- Guaranteed to run in polynomial time, and
- guaranteed to produce a ”high quality” solution.
Approximation Algorithm

- Guaranteed to run in polynomial time, and
- guaranteed to produce a "high quality" solution.

But, what is a high quality solution?

- Is the solution $\text{OPT} + \alpha$, or $\alpha \times \text{OPT}$, or $(1 + \epsilon) \times \text{OPT}$ where,
  - $\text{OPT} \rightarrow$ the optimum solution,
  - $\alpha \rightarrow$ a constant,
  - $\epsilon \rightarrow$ a very small constant.
Approximation Algorithm

- Guaranteed to run in polynomial time, and
- guaranteed to produce a "high quality" solution.

But, what is a high quality solution?

Is the solution $= OPT + \alpha$, or $\alpha \times OPT$, or $(1 + \epsilon) \times OPT$

where,

$OPT \rightarrow$ the optimum solution,

$\alpha \rightarrow$ a constant,

$\epsilon \rightarrow$ a very small constant.
Approximation Algorithm

Note: Designing approximation algorithm for a problem having polynomial time algorithm is also relevant if the time complexity of the problem is very high.
Approximation Algorithm

**Note:** Designing approximation algorithm for a problem having polynomial time algorithm is also relevant if the time complexity of the problem is very high.

**Difficulty:** One needs to prove that the solution is close to optimum, without knowing the optimum solution.
### Absolute Approximation Algorithm

An algorithm $A$ for a problem $P$ that produces a solution $Q$ such that $|Q - OPT| \leq \alpha$, for a constant $\alpha$. 

**Example:** For a planar graph, testing whether the graph is 3 colorable is NP-complete. But, coloring the nodes of the graph with 5 colors needs $O(n^2)$ time. Thus, we have a 2-absolute approximation algorithm for the planar graph coloring problem.
Approximation Algorithm

**Absolute Approximation Algorithm**

An algorithm $A$ for a problem $P$ that produces a solution $Q$ such that $|Q - OPT| \leq \alpha$, for a constant $\alpha$.

**Example:**

- For a planar graph, testing whether the graph is 3 colorable is NP-complete.
- But, coloring the nodes of the graph with 5 colors needs $O(n^2)$ time.

Thus, we have a 2-absolute approximation algorithm for the planar graph coloring problem.
Approximation Algorithm

**f(n)-approximation Algorithm**

Let $P$ be a problem of size $n$. An algorithm $A$ for a problem $P$ is said to be an $f(n)$-approximation algorithm if and only if for every instance $I$ of $P$, $|A(I) - OPT| / OPT \leq f(n)$. It is assumed that $OPT \geq 0$.

**Note:** $f(n)$ can be a constant also.
Approximation Algorithm

$f(n)$-approximation Algorithm

Let $P$ be a problem of size $n$. An algorithm $A$ for a problem $P$ is said to be an $f(n)$-approximation algorithm if and only if for every instance $I$ of $P$, $|A(I) - OPT|/OPT \leq f(n)$. It is assumed that $OPT \geq 0$.

Note: $f(n)$ can be a constant also.

Performance ratio:

$$f(n) = \frac{A(I)}{OPT} \quad \text{in case } P \text{ is a minimization problem}$$

$$OPT/A(I) \quad \text{in case } P \text{ is a maximization problem}$$
Vertex Cover Problem

Definition

Given a graph $G = (V, E)$, a vertex cover $S$ is a set of nodes $S \subseteq V$ such that at least one end-point of every edge $e \in E$ is incident on a member of $S$. 
Vertex Cover Problem

**Definition**

Given a graph $G = (V, E)$, a vertex cover $S$ is a set of nodes $S \subseteq V$ such that at least one end-point of every edge $e \in E$ is incident on a member of $S$.

**Vertex Cover Problem**

Given a graph $G = (V, E)$, compute a vertex cover of minimum size.
Vertex Cover Problem

Definition

Given a graph $G = (V, E)$, a vertex cover $S$ is a set of nodes $S \subseteq V$ such that at least one end-point of every edge $e \in E$ is incident on a member of $S$.

Vertex Cover Problem

Given a graph $G = (V, E)$, compute a vertex cover of minimum size.

Status: NP-hard.
A Greedy Algorithm

- Find a vertex having maximum degree.
- Choose it in the vertex cover
- Delete that vertex and all its adjacent edges from the graph
- Repeat the above three steps until all the edges of the graph are removed
A Greedy Algorithm

- Find a vertex having maximum degree.
- Choose it in the vertex cover
- Delete that vertex and all its adjacent edges from the graph
- Repeat the above three steps until all the edges of the graph are removed

An example:
A Greedy Algorithm

A generalized Example

\[ SOL = k! \left( \frac{1}{k} + \frac{1}{k-1} + \frac{1}{k-2} + \ldots + 1 \right) = k! \times \log k \]
A Greedy Algorithm

A generalized Example

\[ \text{OPT} = k! \]

\[ \text{SOL} = k!(\frac{1}{k} + \frac{1}{k-1} + \frac{1}{k-2} + \ldots + 1) = k! \times \log k \]
A Greedy Algorithm

A generalized Example

\[
SOL = k! \left( \frac{1}{k} + \frac{1}{k-1} + \frac{1}{k-2} + \ldots + 1 \right) = k! \times \log k
\]

Optimum Solution

\[
OPT = k!
\]

Inference: The greedy algorithm is not a constant factor approximation algorithm.
Constant Factor Approximation Algorithm

Step 1: Choose an edge \( e = (u, v) \in E \)

Step 2: Update the vertex cover \( S \) by \( S \cup \{u, v\} \)

Step 3: Remove the vertices \( u \) and \( v \) from \( V \). Remove all the edges incident at \( u \) and \( v \) from \( E \).

Step 3: Repeat Steps 1 and 2 until all the edges in \( E \) are removed.
Constant Factor Approximation Algorithm

Step 1: Choose an edge $e = (u, v) \in E$

Step 2: Update the vertex cover $S$ by $S \cup \{u, v\}$

Step 3: Remove the vertices $u$ and $v$ from $V$. Remove all the edges incident at $u$ and $v$ from $E$.

Step 3: Repeat Steps 1 and 2 until all the edges in $E$ are removed.

**Theorem**

Approximation factor $\Rightarrow 2$. Time Complexity $\Rightarrow O(|E|)$. 
Minimum Weight Steiner Tree Problem

Problem:

Given:
An edge weighted undirected graph $G = (V, E)$, with vertices classified into two groups, namely *terminal* $A$ and *non-terminal* $B$, where $A \cup B = V$. 
Minimum Weight Steiner Tree Problem

Problem:

Given:
An edge weighted undirected graph \( G = (V, E) \), with vertices classified into two groups, namely *terminal* \( A \) and *non-terminal* \( B \), where \( A \cup B = V \).

Objective:
To compute a tree \( T \) that contains all the vertices of \( A \), and the total weight of the edges of the tree \( T \) is minimum.
Minimum Weight Steiner Tree Problem

Problem:

Given:
An edge weighted undirected graph $G = (V, E)$, with vertices classified into two groups, namely *terminal* $A$ and *non-terminal* $B$, where $A \cup B = V$.

Objective:
To compute a tree $T$ that contains all the vertices of $A$, and the total weight of the edges of the tree $T$ is minimum.

Status
The decision version of the problem is NP-complete.
Approximation Algorithm for Steiner Tree Problem

**Input:** A graph $G = (V, E, w)$ and a terminal set $L \subset V$.

**Output:** A Steiner Tree
Approximation Algorithm for Steiner Tree Problem

**Input:** A graph $G = (V, E, w)$ and a terminal set $L \subset V$.

**Output:** A Steiner Tree

**Step 1** Consider the metric closure $G_L = (L, E_L, \omega)$ on the terminal set $L$.

- $G_L$ - A complete graph with node set $L$. For each $u, v \in L$ cost of the edge $(u, v) = \text{shortest path cost from } u \text{ to } v \text{ in } G$. 

**Step 2** find a minimum spanning tree (MST) $T$ on $G_L$.

**Step 3** set $T = \emptyset$.

**Step 4** for each edge $(u, v) \in E_L$ do

- **Step 4a** find a shortest path $\Pi$ from $u$ to $v$ in $G_L$.

- **Step 4b** if $\Pi$ contains less than two vertices in $T$, then add $\Pi$ to $T$.

- else let $x$ and $y$ be first and last vertices on $\Pi$ that are also in $T$.

  add sub-paths $u$ to $x$ and $y$ to $v$ in $T$.

**Step 5** output $T$. 

Approximation Algorithm for Steiner Tree Problem

**Input:** A graph \( G = (V, E, w) \) and a terminal set \( L \subset V \).

**Output:** A Steiner Tree

**Step 1** Consider the metric closure \( G_L = (L, E_L, \omega) \) on the terminal set \( L \).

- \( G_L \) - A complete graph with node set \( L \). For each \( u, v \in L \),
  - cost of the edge \((u, v)\) = shortest path cost from \( u \) to \( v \) in \( G \).

**Step 2** find a minimum spanning tree (MST) \( T_L \) on \( G_L \).
Approximation Algorithm for Steiner Tree Problem

**Input:** A graph \( G = (V, E, w) \) and a terminal set \( L \subset V \).

**Output:** A Steiner Tree

**Step 1** Consider the metric closure \( G_L = (L, E_L, \omega) \) on the terminal set \( L \).

- \( G_L \) - A complete graph with node set \( L \). For each \( u, v \in L \) cost of the edge \((u, v)\) = shortest path cost from \( u \) to \( v \) in \( G \).

**Step 2** find a minimum spanning tree (MST) \( T_L \) on \( G_L \).

**Step 3** set \( T = \emptyset \).
Approximation Algorithm for Steiner Tree Problem

**Input:** A graph $G = (V, E, w)$ and a terminal set $L \subseteq V$.

**Output:** A Steiner Tree

**Step 1** Consider the metric closure $G_L = (L, E_L, \omega)$ on the terminal set $L$.

- $G_L$ - A complete graph with node set $L$. For each $u, v \in L$, cost of the edge $(u, v) = \text{shortest path cost from } u \text{ to } v \text{ in } G$.

**Step 2** find a minimum spanning tree (MST) $T_L$ on $G_L$.

**Step 3** set $T = \emptyset$.

**Step 4** for each edge $(u, v) \in E_L$ do

- **Step 4a** find a shortest path $\Pi$ from $u$ to $v$ in $G$.

- **Step 4b** if $\Pi$ contains less than two vertices in $T$, then add $\Pi$ to $T$.
  else let $x$ and $y$ be first and last vertices on $\Pi$ that are also in $T$.
  add sub-paths $u$ to $x$ and $y$ to $v$ in $T$. 

**Step 5** output $T$.
Approximation Algorithm for Steiner Tree Problem

**Input:** A graph $G = (V, E, w)$ and a terminal set $L \subseteq V$.

**Output:** A Steiner Tree

**Step 1** Consider the metric closure $G_L = (L, E_L, \omega)$ on the terminal set $L$.

- $G_L$ - A complete graph with node set $L$. For each $u, v \in L$ cost of the edge $(u, v) = \text{shortest path cost from } u \text{ to } v \text{ in } G$.

**Step 2** find a minimum spanning tree (MST) $T_L$ on $G_L$.

**Step 3** set $T = \emptyset$.

**Step 4** for each edge $(u, v) \in E_L$ do

**Step 4a** find a shortest path $\Pi$ from $u$ to $v$ in $G$.

**Step 4b** if $\Pi$ contains less than two vertices in $T$, then add $\Pi$ to $T$.

else let $x$ and $y$ be first and last vertices on $\Pi$ that are also in $T$.

- add sub-paths $u$ to $x$ and $y$ to $v$ in $T$.

**Step 5** output $T$
Demonstration of the Algorithm

Given Graph
Demonstration of the Algorithm

Given Graph

Metric Closure

\[ G_L = (L, E_L, \omega) \]
Demonstration of the Algorithm

Given Graph

Metric Closure
\( G_L = (L, E_L, \omega) \)

MST on
\( G_L = (L, E_L, \omega) \)
Demonstration of the Algorithm

Given Graph

Metric Closure
\[ G_L = (L, E_L, \omega) \]

MST on
\[ G_L = (L, E_L, \omega) \]

Expansion of edge
\((v_1, v_4)\)
Demonstration of the Algorithm

Given Graph

Metric Closure
$G_L = (L, E_L, \omega)$

MST on
$G_L = (L, E_L, \omega)$

Expansion of edge
$(v_1, v_4)$

Expansion of edge
$(v_2, v_3)$
Demonstration of the Algorithm

Given Graph

Metric Closure
\[ G_L = (L, E_L, \omega) \]

MST on
\[ G_L = (L, E_L, \omega) \]

Expansion of edge
\((v_1, v_4)\)

Expansion of edge
\((v_2, v_3)\)

Expansion of edge
\((v_2, v_5)\)
Analysis

Theorem

MST based Steiner Tree algorithm produces a 2-factor approximation solution for the Steiner Minimal Tree problem on an undirected weighted graph in $O(|V||L|^2)$ time.
Analysis

Theorem

MST based Steiner Tree algorithm produces a 2-factor approximation solution for the Steiner Minimal Tree problem on an undirected weighted graph in $O(|V||L|^2)$ time.

Proof:

Let $T^*$ be a Steiner Tree of minimum cost, and $T$ be the Steiner Tree obtained as output of our algorithm.

Since at most all shortest paths of $T_L$ are inserted in $T$, we have

$$\omega(T) \leq \omega(T_L)$$
Analysis of Approximation Factor

Important points:
Analysis of Approximation Factor

Important points:

1. If $X$ is an Eulerian Tour by doubling each edge of $T^*$, then
\[ \omega(X) = 2\omega(T^*) \]
Analysis of Approximation Factor

**Important points:**

1. If $X$ is an Eulerian Tour by doubling each edge of $T^*$, then
   \[ \omega(X) = 2\omega(T^*) \]

2. If $Y$ is the TSP Tour (Minimum Hamiltonian Circuit) on $G_L$, then
   \[ \omega(X) \geq \omega(Y) \geq \omega(T_L) \]
Analysis of Approximation Factor

**Important points:**

1. If $X$ is an Eulerian Tour by doubling each edge of $T^*$, then
   \[ \omega(X) = 2 \omega(T^*) \]

2. If $Y$ is the TSP Tour (Minimum Hamiltonian Circuit) on $G_L$, then
   \[ \omega(X) \geq \omega(Y) \geq \omega(T_L) \]

3. $\omega(T) \leq \omega(T_L)$
Analysis of Approximation Factor

Important points:

1. If $X$ is an Eulerian Tour by doubling each edge of $T^*$, then
   \[ \omega(X) = 2\omega(T^*) \]

2. If $Y$ is the TSP Tour (Minimum Hamiltonian Circuit) on $G_L$, then
   \[ \omega(X) \geq \omega(Y) \geq \omega(T_L) \]

3. $\omega(T) \leq \omega(T_L)$

Thus, we have the final result
\[ w(T) \leq 2\omega(T^*) \]
Analysis of Approximation Factor

Important points:

1. If $X$ is an Eulerian Tour by doubling each edge of $T^*$, then
   \[ \omega(X) = 2\omega(T^*) \]

2. If $Y$ is the TSP Tour (Minimum Hamiltonian Circuit) on $G_L$, then
   \[ \omega(X) \geq \omega(Y) \geq \omega(T_L) \]

3. \[ \omega(T) \leq \omega(T_L) \]

Thus, we have the final result
\[ w(T) \leq 2\omega(T^*) \]

Time complexity: \( O(|V||L|^2) \)

Follows from the time complexity of computing $G_L$. 
Demonstration of an worst case example

Given Graph

Our Solution

Optimum Solution
Improving Approximation Factor

Can we have an Eulerian Tour without doubling the edges: YES

How

Replace Step 2 of the earlier algorithm by the following three steps:

**Step 2.1**: Identify the odd degree vertices $V'$ in the minimum spanning tree $T_L$.

**Step 2.2**: Find a set of edges $M$ that corresponds to the minimum maximal matching of the graph $V'$.

**Step 2.3**: Compute Eulerian tour in the graph $G' = (V, T_L \cup M)$. 
Demonstration

Given Graph

Metric Closure
\( G_L = (L, E_L, \omega) \)

MST on
\( G_L = (L, E_L, \omega) \)
Demonstration

Given Graph

Metric Closure
$G_L = (L, E_L, \omega)$

MST on
$G_L = (L, E_L, \omega)$

Minimum maximal matching $M$ in the graph induced by odd degree vertices
Demonstration

Given Graph

Metric Closure
$G_L = (L, E_L, \omega)$

MST on
$G_L = (L, E_L, \omega)$

Minimum maximal matching $M$ in the graph induced by odd degree vertices

The graph
$G' = (V, T_L \cup M)$
Theorem

Thus we have an \( \frac{3}{2} \) approximation of the Steiner tree problem.
Analysis

**Theorem**

Thus we have an $\frac{3}{2}$ approximation of the Steiner tree problem.

**Proof:**

- $\tau$: Optimal TSP tour in $G_L$.
- $\tau'$: Tour in $G'$ obtained by short-cutting $\tau$.
  - $\text{cost}(\tau') \leq \text{cost}(\tau)$ (* by triangles inequality *).
  - $\tau'$ consists of two perfect matchings on $V'$ each consisting of alternate edges of $\tau$.
  - Total cost in the eulerian tour due to the edges in $\tau \leq \text{OPT}$, and
  - Total cost in the eulerian tour due to the edges of $M \leq \frac{\text{OPT}}{2}$. 
Available Results on Steiner Tree approximation algorithm

Best known approximation result

- for edge weighted graph: \(1.39\)
  (* An LP based algorithm *)

- for a complete graph of edge weight 1 or 2: \(1.25\)

- for node weighted graph: \(2 \log k\)
  where \(k\) is the number of terminal nodes.

- for node weighted planar graph: \(O(1)\)
  E. D. Demaine, M. T. Hajiaghayi and P. N. Klein.
Set Cover Problem

**Input:**

A finite set $X$, and a set of subsets $\mathcal{F} = \{S_1, S_2, \ldots S_k\}$ of $X$ such that every element of $X$ belongs to at least one subset of $\mathcal{F}$. Thus, $X = \bigcup_{S_i \in \mathcal{F}} S_i$.
Set Cover Problem

**Input:**
A finite set $X$, and a set of subsets $\mathcal{F} = \{S_1, S_2, \ldots S_k\}$ of $X$ such that every element of $X$ belongs to at least one subset of $\mathcal{F}$. Thus, $X = \bigcup_{S_i \in \mathcal{F}} S_i$

**Objective:**
Choose minimum number of subsets $\mathcal{C}$ from $\mathcal{F}$ to cover all the elements of $X$, i.e., $X = \bigcup_{S \in \mathcal{C}} S$. 
Set Cover Problem

**Input:**
A finite set $X$, and a set of subsets $\mathcal{F} = \{S_1, S_2, \ldots, S_k\}$ of $X$ such that every element of $X$ belongs to at least one subset of $\mathcal{F}$. Thus, $X = \bigcup_{S_i \in \mathcal{F}} S_i$

**Objective:**
Choose minimum number of subsets $C$ from $\mathcal{F}$ to cover all the elements of $X$, i.e., $X = \bigcup_{S \in C} S$.

**Usefulness**
Many practical problems can be formulated as a set cover problem. Example: Art gallery Problem, Hitting Set Problem, etc.
Algorithm Greedy-Set-Cover($X, \mathcal{F}$)

\begin{align*}
U & \leftarrow X \\
\quad \text{(* } U \text{ contains uncovered elements *)} \\
C & \leftarrow \emptyset \text{ (* Set cover *)} \\
\textbf{while } U \neq \emptyset \textbf{ do} \\
\quad \text{select an } S \in \mathcal{F} \text{ that maximizes } |S \cap U| \\
\quad U & \leftarrow U - S \\
\quad C & \leftarrow C \cup S \\
\textbf{endwhile} \\
\textbf{end.}
\end{align*}
Algorithm Greedy-Set-Cover($X, \mathcal{F}$)

\[ U \leftarrow X \]
\[ (* U \text{ contains uncovered elements } *) \]
\[ C \leftarrow \emptyset (* \text{Set cover } *) \]
\[ \textbf{while} \ U \neq \emptyset \ \textbf{do} \]
\[ \quad \text{select an } S \in \mathcal{F} \text{ that maximizes } |S \cap U| \]
\[ \quad U \leftarrow U - S \]
\[ \quad C \leftarrow C \cup S \]
\[ \textbf{endwhile} \]
\[ \textbf{end.} \]

Here the smallest set cover is \( \{S_3, S_4, S_5\} \)
Algorithm Greedy-Set-Cover($X, \mathcal{F}$)

$U \leftarrow X$

(* $U$ contains uncovered elements *)

$C \leftarrow \emptyset$ (* Set cover *)

while $U \neq \emptyset$ do

select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$

$U \leftarrow U - S$

$C \leftarrow C \cup S$

endwhile

end.

Here the smallest set cover is $\{S_3, S_4, S_5\}$

But Greedy algorithm produces $\{S_4, S_1, S_3, S_5\}$. 
Approximation Factor

Theorem

*Greedy set cover has a ratio bound* $H(\max\{|S| : S \in \mathcal{F}\})$
Approximation Factor

**Theorem**

*Greedy set cover has a ratio bound $H(\max\{|S| : S \in \mathcal{F}\})$*

$c_x = \text{cost allocated to element } x \in X$.

Each element is assigned cost only once.

If $x$ is covered for the first time by $S_i$, then

$c_x = \frac{1}{|S_i - (S_1 + S_2 + \ldots + S_{i-1})|}$

Let the algorithm finds a solution $C$. The cost of the solution is $|C| = \sum_{x \in X} c_x$.

Since the optimum solution $C^*$ also covers $X$, we have

$|C| = \sum_{x \in X} c_x \leq \sum_{S \in C^*} \sum_{x \in S} c_x$
Approximation Factor

An Useful Result

\[ \sum_{x \in S} c_x \leq H(|S|), \text{ where } H(n) = \sum_{i=1}^{n} \frac{1}{i} = \ln n \]

Thus \[ |C| \leq \sum_{S \in C^*} H(|S|) \leq |C^*| \times H(\max\{|S| : S \in \mathcal{F}\}) \]

□
Approximation Factor

Proof of the result: $\sum_{x \in S} c_x \leq H(|S|)$

Consider a set $S \in \mathcal{F}$.

Let $u_0 = |S|$, and
$u_i = |S - (S_1 \cup S_2 \cup \ldots \cup S_i)|$. (* the number of elements in $S$ that remains uncovered after $S_1, S_2, \ldots S_i$ are already selected by the algorithm.*)
Approximation Factor

Proof of the result: \( \sum_{x \in S} c_x \leq H(|S|) \)

Consider a set \( S \in F \).

Let \( u_0 = |S| \), and
\( u_i = |S - (S_1 \cup S_2 \cup \ldots \cup S_i)| \). (* the number of elements in \( S \) that remains uncovered after \( S_1, S_2, \ldots S_i \) are already selected by the algorithm.*)

Let \( k \) be the least index such that \( u_k = 0 \) (* all elements in \( S \) are covered by \( S_1, S_2, \ldots, S_k \) *)
Approximation Factor

Proof of the result: $\sum_{x \in S} c_x \leq H(|S|)$

Consider a set $S \in \mathcal{F}$.

Let $u_0 = |S|$, and 
$u_i = |S - (S_1 \cup S_2 \cup \ldots \cup S_i)|$. (* the number of elements in $S$ that remains uncovered after $S_1, S_2, \ldots S_i$ are already selected by the algorithm.*)

Let $k$ be the least index such that $u_k = 0$ (* all elements in $S$ are covered by $S_1, S_2, \ldots, S_k$ *)

Thus, $u_{i-1} \geq u_i$, and $(u_{i-1} - u_i)$ elements of $S$ are covered for the first time by $S_i$. Thus,

$\sum_{x \in S} c_x = \sum_{i=1}^{k}(u_{i-1} - u_i) \times \frac{1}{|S - (S_1 \cup S_2 \cup \ldots \cup S_{i-1})|}$
Approximation Factor

In the $i$-th step, $S_i$ is the better choice than $S$ by our greedy algorithm.

Thus, $|S_i - (S_1 \cup S_2 \cup S_{i-1})| \geq |S - (S_1 \cup S_2 \cup S_{i-1})| \geq u_{i-1}$

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \times \frac{1}{u_{i-1}}$$

Another Important Result

For any integer $a$ and $b$, where $a < b$, we have

$$H(b) - H(a) = \sum_{i=a+1}^{b} \frac{1}{i} \geq (b - a) \times \frac{1}{b}$$

Thus, $\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) = H(u_0) - H(u_k) = H(u_0) = H(|S|)$
Polynomial Time Approximation Scheme (PTAS)

**Definition**

An algorithm $A$ for a problem $P$ is said to be a polynomial time approximation scheme (PTAS) if

- given an instance $\Pi$ of $P$, the algorithm $A$ returns an $\epsilon$-approximation result in time polynomial on the length of the input $\Pi$, where (the polynomial function depends on $\epsilon$)

Note that, this function may be of the form $\frac{1}{\epsilon} \times n^2$ or $n^{\frac{1}{\epsilon}}$. 
Polynomial Time Approximation Scheme (PTAS)

**Definition**

An algorithm $\mathcal{A}$ for a problem $P$ is said to be a polynomial time approximation scheme (PTAS) if

- given an instance $\Pi$ of $P$, the algorithm $\mathcal{A}$ returns an $\epsilon$-approximation result in time polynomial on the length of the input $\Pi$, where *(the polynomial function depends on $\epsilon$)*

Note that, this function may be of the form $\frac{1}{\epsilon} \times n^2$ or $n^{\frac{1}{\epsilon}}$.

**Definition**

An PTAS algorithm $\mathcal{A}$ for a problem $P$ is said to be a fully polynomial time approximation scheme (FPTAS) if

- given an instance $\Pi$ of $P$, the algorithm $\mathcal{A}$ returns an $\epsilon$-approximation result in time polynomial in the length of the input $\Pi$, and $\frac{1}{\epsilon}$. 