Approximation Algorithms

Md. Saidur Rahman,
Department of Computer Science and Engineering,
Bangladesh University of Engineering and Technology,
Dhaka, Bangladesh.

Decision Problems

Decision Problems

A *decision problem* is an algorithmic problem whose answer is “yes” or “no”.

Decision Problems

Decision Problems

A *decision problem* is an algorithmic problem whose answer is “yes” or “no”.

Decision Problems

Decision Problems

A *decision problem* is an algorithmic problem whose answer is “yes” or “no”.

Does a given Turing machine halt on a given input?

Decision Problems

Decision Problems

A *decision problem* is an algorithmic problem whose answer is “yes” or “no”.
Does a given Turing machine halt on a given input?
Not Computable!

Decision Problems

Decision Problems

A *decision problem* is an algorithmic problem whose answer is “yes” or “no”.
Does a given Turing machine halt on a given input?
Not Computable!
Is a given natural number a prime number?

Decision Problems

Decision Problems

A *decision problem* is an algorithmic problem whose answer is "yes" or "no".
Does a given Turing machine halt on a given input?
Not Computable!
Is a given natural number a prime number?
A computable problem.

Optimization Problems

Optimaization Problem

An *optimization problem* Π consists of a set of instances $\mathcal{I}$, a set of solutions $\mathcal{O}$, and three functions sol: $\mathcal{I} \rightarrow \mathcal{P}(\mathcal{O})$, quality: $\mathcal{I} \times \mathcal{O} \rightarrow \mathbb{R}$, and goal $\in$ *min*, *max*.

Optimization Problems

Optimaization Problem

An *optimization problem* Π consists of a set of instances $\mathcal{I}$, a set of solutions $\mathcal{O}$, and three functions sol: $\mathcal{I} \rightarrow \mathcal{P}(\mathcal{O})$, quality: $\mathcal{I} \times \mathcal{O} \rightarrow \mathbb{R}$, and goal $\in$ *min*, *max*.

Optimization Problems

Optimaization Problem

An *optimization problem* Π consists of a set of instances $\mathcal{I}$, a set of solutions $\mathcal{O}$, and three functions sol: $\mathcal{I} \rightarrow \mathcal{P}(\mathcal{O})$, quality: $\mathcal{I} \times \mathcal{O} \rightarrow \mathbb{R}$, and goal $\in \mathit{min}, \mathit{max}$.
For every instance $I \in \mathcal{I}$, sol(I) $\subseteq \mathcal{O}$ denotes the set of *feasible solutions* for I.

Optimization Problems

Optimaization Problem

An *optimization problem* Π consists of a set of instances $\mathcal{I}$, a set of solutions $\mathcal{O}$, and three functions sol: $\mathcal{I} \to \mathcal{P}(\mathcal{O})$, quality: $\mathcal{I} \times \mathcal{O} \to \mathbb{R}$, and goal $\in min, max$.
For every instance $I \in \mathcal{I}$, sol(I) $\subseteq \mathcal{O}$ denotes the set of *feasible solutions* for I.
For every instance $I \in \mathcal{I}$ and every feasible solution $O \in sol(I)$, quality (I, O) denotes the measure of I and O. An *optimal solution* for an instance $I \in \mathcal{I}$ of Π is a solution $OPT(I) \in sol(I)$ such that
quality $(I, OPT(I)) =$ goal $\{$ quality $(I, O) \mid O \in sol(I)\}$.

Optimization Problems

Optimaization Problem

An *optimization problem* Π consists of a set of instances $\mathcal{I}$, a set of solutions $\mathcal{O}$, and three functions sol: $\mathcal{I} \to \mathcal{P}(\mathcal{O})$, quality: $\mathcal{I} \times \mathcal{O} \to \mathbb{R}$, and goal $\in min, max$.
For every instance $I \in \mathcal{I}$, sol(I) $\subseteq \mathcal{O}$ denotes the set of *feasible solutions* for I.
For every instance $I \in \mathcal{I}$ and every feasible solution $O \in sol(I)$, quality (I, O) denotes the measure of I and O. An *optimal solution* for an instance $I \in \mathcal{I}$ of Π is a solution $OPT(I) \in sol(I)$ such that
quality $(I, OPT(I)) =$ goal $\{$ quality $(I, O) \mid O \in sol(I)\}$.
If goal = min, we call Π a *minimization problem* and write “cost” instead of “quality.” Conversely, if goal=max, we say that Π is a *maximization problem* and write “gain” instead of “quality.”

Optimization Problems

Traveling Salesperson Problem (TSP)

Given a traffic network, what is the fastest tour that visits all cities on the map exactly once and returns to the starting point?

Optimization Problems

Traveling Salesperson Problem (TSP)

Given a traffic network, what is the fastest tour that visits all cities on the map exactly once and returns to the starting point?

Optimization Problems

Traveling Salesperson Problem (TSP)

Given a traffic network, what is the fastest tour that visits all cities on the map exactly once and returns to the starting point? Minimization problem.

Optimization Problems

Traveling Salesperson Problem (TSP)

Given a traffic network, what is the fastest tour that visits all cities on the map exactly once and returns to the starting point? Minimization problem.

Independent Set Problem

Find the independent set in a graph as large as possible.

Optimization Problems

Traveling Salesperson Problem (TSP)

Given a traffic network, what is the fastest tour that visits all cities on the map exactly once and returns to the starting point? Minimization problem.

Independent Set Problem

Find the independent set in a graph as large as possible.

Optimization Problems

Traveling Salesperson Problem (TSP)

Given a traffic network, what is the fastest tour that visits all cities on the map exactly once and returns to the starting point? Minimization problem.

Independent Set Problem

Find the independent set in a graph as large as possible. Maximization problem.

Optimization Problems

Traveling Salesperson Problem (TSP)

Given a traffic network, what is the fastest tour that visits all cities on the map exactly once and returns to the starting point? Minimization problem.

Independent Set Problem

Find the independent set in a graph as large as possible. Maximization problem.

Vertex Cover Problem

Find the vertex cover in a graph as small as possible.

Optimization Problems

Traveling Salesperson Problem (TSP)

Given a traffic network, what is the fastest tour that visits all cities on the map exactly once and returns to the starting point? Minimization problem.

Independent Set Problem

Find the independent set in a graph as large as possible. Maximization problem.

Vertex Cover Problem

Find the vertex cover in a graph as small as possible.

Optimization Problems

Traveling Salesperson Problem (TSP)

Given a traffic network, what is the fastest tour that visits all cities on the map exactly once and returns to the starting point? Minimization problem.

Independent Set Problem

Find the independent set in a graph as large as possible. Maximization problem.

Vertex Cover Problem

Find the vertex cover in a graph as small as possible. Minimixation problem.

All those problems are NP-hard problems.
It is unlikely to have polynomial time algorithms.

Coping with Hardness

How to deal with hard problems?

Coping with Hardness

How to deal with hard problems?

Desired Objective or requirements

Coping with Hardness

How to deal with hard problems?

Desired Objective or requirements

- find optimal solution

Coping with Hardness

How to deal with hard problems?

Desired Objective or requirements

- find optimal solution
- in polynomial time

Coping with Hardness

How to deal with hard problems?

Desired Objective or requirements

- find optimal solution
- in polynomial time
- for any instance

Coping with Hardness

How to deal with hard problems?

Desired Objective or requirements

- find optimal solution
- in polynomial time
- for any instance

Coping with Hardness

How to deal with hard problems?

Desired Objective or requirements

- find optimal solution
- in polynomial time
- for any instance

At least one of these requirements must be relaxed in any approch to dealing with NP-hard optimization problems.

Coping with Hardness

How to deal with hard problems?

Desired Objective or requirements

- find optimal solution
- in polynomial time
- for any instance

At least one of these requirements must be relaxed in any approch to dealing with NP-hard optimization problems. Approximation Algorithms relax the first requirement.

Vertex Cover Problem: An Approximation

Algorithm VertexCoverApprox(G, C)

Vertex Cover Problem: An Approximation

```
Algorithm VertexCoverApprox($G, C$)
$C := \emptyset$;
```

Vertex Cover Problem: An Approximation

```
Algorithm VertexCoverApprox($G, C$)
$C := \emptyset$;
**while** $G$ still has edges **do**
```

Vertex Cover Problem: An Approximation

```
Algorithm VertexCoverApprox($G, C$)
$C := \emptyset$;
while $G$ still has edges do
    select an edge $e = (v, w)$ of $G$;
```

Vertex Cover Problem: An Approximation

```
Algorithm VertexCoverApprox($G, C$)
$C := \emptyset$;
while $G$ still has edges do
    select an edge $e = (v, w)$ of $G$;
    add vertices $v$ and $w$ to $C$;
```

Vertex Cover Problem: An Approximation

```
Algorithm VertexCoverApprox($G, C$)
$C := \emptyset$;
**while** $G$ still has edges **do**
    select an edge $e = (v, w)$ of $G$;
    add vertices $v$ and $w$ to $C$;
    **for** each edge $f$ incident to $v$ or $w$
    **do**
```

Vertex Cover Problem: An Approximation

```
Algorithm VertexCoverApprox($G, C$)
$C := \emptyset$;
while $G$ still has edges do
    select an edge $e = (v, w)$ of $G$;
    add vertices $v$ and $w$ to $C$;
    for each edge $f$ incident to $v$ or $w$ do
        remove $f$ from $G$;
```

Vertex Cover Problem: An Approximation

```
Algorithm VertexCoverApprox($G, C$)
$C := \emptyset$;
**while** $G$ still has edges **do**
    select an edge $e = (v, w)$ of $G$;
    add vertices $v$ and $w$ to $C$;
    **for** each edge $f$ incident to $v$ or $w$
    **do**
        remove $f$ from $G$;
    **end for**
```

Vertex Cover Problem: An Approximation

```
Algorithm VertexCoverApprox($G, C$)
$C := \emptyset$;
while $G$ still has edges do
    select an edge $e = (v, w)$ of $G$;
    add vertices $v$ and $w$ to $C$;
    for each edge $f$ incident to $v$ or $w$ do
        remove $f$ from $G$;
    end for
end while
```

Vertex Cover Problem: An Approximation

```
Algorithm VertexCoverApprox($G, C$)
$C := \emptyset$;
while $G$ still has edges do
    select an edge $e = (v, w)$ of $G$;
    add vertices $v$ and $w$ to $C$;
    for each edge $f$ incident to $v$ or $w$ do
        remove $f$ from $G$;
    end for
end while
return $C$
```

Approximation Algorithms

Approximation Algorithm

Let Π be an optimization problem, and let ALG be a consistent algorithm for Π.

Approximation Algorithms

Approximation Algorithm

Let Π be an optimization problem, and let ALG be a consistent algorithm for Π.

Approximation Algorithms

Approximation Algorithm

Let Π be an optimization problem, and let ALG be a consistent algorithm for Π.
For $r \geq 1$, ALG is an *r-approximation algorithm* for Π if, for every $I \in \mathcal{I}$,
gain (OPT (I))$\leq r$. gain (ALG (I)) if Π is maximization problem, or
cost (ALG (I))$\leq r$. cost (OPT (I))
if Π is minimization problem.

Approximation Algorithms

Approximation Algorithm

Let Π be an optimization problem, and let ALG be a consistent algorithm for Π.
For $r \geq 1$, ALG is an *r-approximation algorithm* for Π if, for every $I \in \mathcal{I}$,
gain (OPT (I))$\leq r$. gain (ALG (I)) if Π is maximization problem, or
cost (ALG (I))$\leq r$. cost (OPT (I))
if Π is minimization problem.
The *approximation ratio* of ALG is defined as
$r_{ALG} := inf\{r \geq 1 \mid \text{ALG is an } r\text{- approximation algorithm for } \Pi\}$.

Approximation Algorithms

Simple Knapsack Problem

The simple knapsack problem is a maximization problem.

Approximation Algorithms

Simple Knapsack Problem

The simple knapsack problem is a maximization problem.

Approximation Algorithms

Simple Knapsack Problem

The simple knapsack problem is a maximization problem.
An instance I is given by a sequence of $n+1$ positive integers $B, w_1, w_2, \cdots, w_n$, where we consider w_i with $1 \leq i \leq n$ to be the weight of the ith object;
B is the capacity of the knapsack.

Approximation Algorithms

Simple Knapsack Problem

The simple knapsack problem is a maximization problem.
An instance I is given by a sequence of $n+1$ positive integers $B, w_1, w_2, \cdots, w_n$, where we consider w_i with $1 \leq i \leq n$ to be the weight of the ith object;
B is the capacity of the knapsack.
A feasible solution for I is any set $\mathcal{O} \subseteq \{1, 2, \cdots, n\}$ such that

$$\sum_{i \in O} w_i \leq B$$

Approximation Algorithms

Simple Knapsack Problem

The simple knapsack problem is a maximization problem.
An instance I is given by a sequence of $n+1$ positive integers $B, w_1, w_2, \cdots, w_n$, where we consider w_i with $1 \leq i \leq n$ to be the weight of the ith object;
B is the capacity of the knapsack.
A feasible solution for I is any set $\mathcal{O} \subseteq \{1, 2, \cdots, n\}$ such that

$$\sum_{i \in O} w_i \leq B$$

The gain of a solution $\mathcal{O}$ and a corresponding instance I is given by
gain $(I, \mathcal{O}) = \sum_{i \in O} w_i$.
The goal is to maximize this number.

Algorithm: KNGREEDY for the simple knapsack problem

```
$O := \theta$;
```

Algorithm: KNGREEDY for the simple knapsack problem

```
$O := \theta$;
$s := 0$;
```

Algorithm: KNGREEDY for the simple knapsack problem

```
$O := \theta$;
$s := 0$;
$i := 0$;
```

Algorithm: KNGREEDY for the simple knapsack problem

```
$O := \theta$;
$s := 0$;
$i := 0$;
**sort** $w_1, w_2, \cdots, w_n$
```

Algorithm: KNGREEDY for the simple knapsack problem

```
$O := \theta$;
$s := 0$;
$i := 0$;
sort $w_1, w_2, \cdots, w_n$
while $i < n$ and $s + w_{i+1} \leq B$ do
```

Algorithm: KNGREEDY for the simple knapsack problem

```
$O := \theta$;
$s := 0$;
$i := 0$;
sort $w_1, w_2, \cdots, w_n$
while $i < n$ and $s + w_{i+1} \leq B$ do
    $O := O \cup i + 1$
```

Algorithm: KNGREEDY for the simple knapsack problem

```
$O := \theta$;
$s := 0$;
$i := 0$;
sort $w_1, w_2, \cdots, w_n$
while $i < n$ and $s + w_{i+1} \leq B$ do
    $O := O \cup i + 1$
    $s := s + w_{i+1}$
```

Algorithm: KNGREEDY for the simple knapsack problem

```
$O := \theta$;
$s := 0$;
$i := 0$;
sort $w_1, w_2, \cdots, w_n$
while $i < n$ and $s + w_{i+1} \leq B$ do
    $O := O \cup i + 1$
    $s := s + w_{i+1}$
    $i := i + 1$
```

Algorithm: KNGREEDY for the simple knapsack problem

```
$O := \theta$;
$s := 0$;
$i := 0$;
sort $w_1, w_2, \cdots, w_n$
while $i < n$ and $s + w_{i+1} \leq B$ do
    $O := O \cup i + 1$
    $s := s + w_{i+1}$
    $i := i + 1$
    output $O$
```

Algorithm: KNGREEDY for the simple knapsack problem

```
$O := \theta$;
$s := 0$;
$i := 0$;
sort $w_1, w_2, \cdots, w_n$
while $i < n$ and $s + w_{i+1} \leq B$ do
    $O := O \cup i + 1$
    $s := s + w_{i+1}$
    $i := i + 1$
    output $O$
end while
```

Approximation Algorith (Simple Knapsack Problem)

Theorem

KNGREEDY is a polynomial-time 2-approximation algorithm for the simple knapsack problem.

KNGREEDY is a polynomial-time 2-approximation algorithm for the simple knapsack problem

Proof.

- **Case 1.** If all objects fit into the knapsack, then KNGREEDY is even optimal

KNGREEDY is a polynomial-time 2-approximation algorithm for the simple knapsack problem

Proof.

- **Case 1.** If all objects fit into the knapsack, then KNGREEDY is even optimal
- **Case 2.** Assume total weight is larger than B

KNGREEDY is a polynomial-time 2-approximation algorithm for the simple knapsack problem

Proof.

- **Case 1.** If all objects fit into the knapsack, then KNGREEDY is even optimal
- **Case 2.** Assume total weight is larger than B
  - **Case 2.1.** Suppose w_i of weight at least $B/2$. $w_1 \geq B/2$ and w_1 is always packed into knapsack. Since B is an upper bound for any solution, the approximation ratio of KNGREEDY is at most 2 in this case.

KNGREEDY is a polynomial-time 2-approximation algorithm for the simple knapsack problem

Proof.

- **Case 1.** If all objects fit into the knapsack, then KNGREEDY is even optimal
- **Case 2.** Assume total weight is larger than B
  - **Case 2.1.** Suppose w_i of weight at least $B/2$.
    $w_1 \geq B/2$ and w_1 is always packed into knapsack. Since B is an upper bound for any solution, the approximation ratio of KNGREEDY is at most 2 in this case.
  - **Case 2.2.** Suppose weight of all objects are smaller than $B/2$, j be the index of the first object that is too heavy to be packed into the knapsack by KNGREEDY.
    $w_j < B/2$
    this implies that space that is already occupied by the objects $w_1, w_2, \cdots, w_{j-1}$ must be larger than $B/2$. the approximation ratio of KNGREEDY is at most 2 in this case.

Chapter 1 (Simple Knapsack Problem)

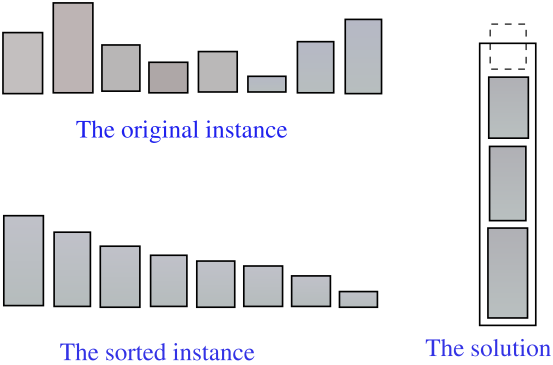
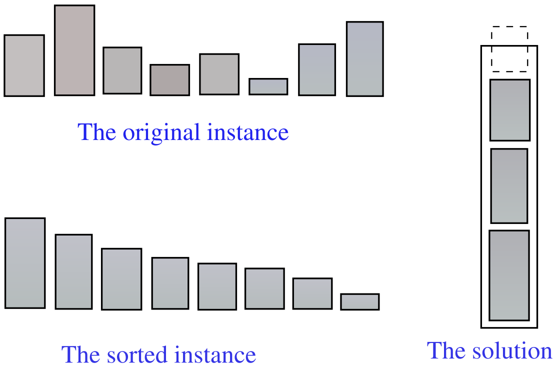


Figure: The greedy strategy; first sort, then pack greedily what fits.

Chapter 1 (Simple Knapsack Problem)



The original instance

The sorted instance

The solution

Figure: The greedy strategy; first sort, then pack greedily what fits.

Chapter 1 (Simple Knapsack Problem)

$$r_{KNGREEDY} \geq \frac{gain(OPT(I))}{gain(KNGREEDY(I))} = \frac{B}{\frac{B}{2}+1} = \frac{2}{1+\frac{2}{B}}$$

Approximability

APX

The class APX (an abbreviation of ”approximable”) is the set of NP optimization problems that allow polynomial-time approximation algorithms with approximation ratio bounded by a constant (or constant-factor approximation algorithms for short). In simple terms, problems in this class have efficient algorithms that can find an answer within some fixed multiplicative factor of the optimal answer.

Approximability

PTAS

If there is a polynomial-time δ-approximation algorithms with $\delta = 1 + \epsilon$, for any fixed value $\epsilon > 0$ to solve a problem, then the problem is said to have a polynomial-time approximation scheme (PTAS). The running time depends on input size and ϵ. Unless P=NP there exist problems that are in APX but without a PTAS, so the class of problems with a PTAS is strictly contained in APX

Approximability

PTAS

If there is a polynomial-time δ-approximation algorithms with $\delta = 1 + \epsilon$, for any fixed value $\epsilon > 0$ to solve a problem, then the problem is said to have a polynomial-time approximation scheme (PTAS). The running time depends on input size and ϵ. Unless P=NP there exist problems that are in APX but without a PTAS, so the class of problems with a PTAS is strictly contained in APX

Fully polynomial-time approximation scheme (FPTAS)

An algorithm that achieves an arbitrarily good approximation ratio of $1 + \epsilon$ in a time that is polynomial both in *n* and $1/\epsilon$ is called a *fully polynomial-time approximation scheme (FPTAS)*.

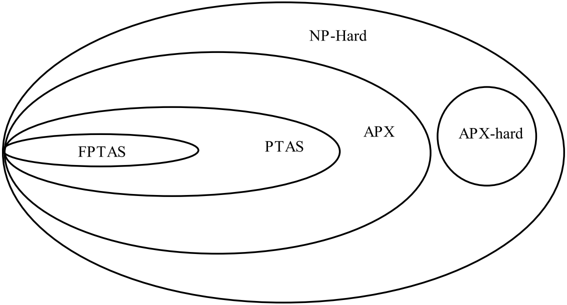
Hard for Approximation

APX-Hard

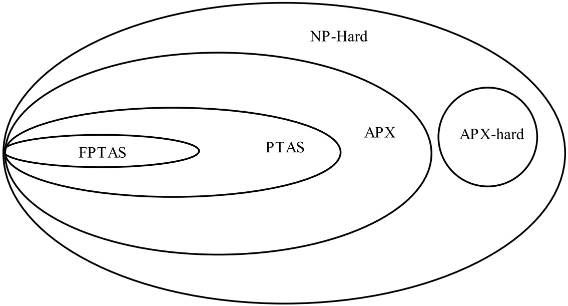
A problem is said to be APX-hard if there is a PTAS reduction from every problem in APX to that problem, and to be APX-complete if the problem is APX-hard and also in APX. As a consequence of $P \neq NP \Rightarrow PTAS \neq APX$, if $P \neq NP$ is assumed, no APX-hard problem has a PTAS.

TSP problem is APX-hard.

NP-hard, APX-hard, APX, PTAS and FPTAS



NP-hard, APX-hard, APX, PTAS and FPTAS



TSP problem is APX-hard.

Approximation for TSP Satisfying Triangle Inequality

On whiteboard.