Approximation Algorithms

Kumar Abhishek

Department of Industrial and Systems Engineering
Lehigh University

COR@L Seminar Series, Spring 2005

Outline

1. Motivation
   - Why Approximation Algorithms?
2. Introduction
   - Constant factor Approximations
   - Set Cover Example
   - TSP Example
3. Approximation schemes
   - PTAS, FPTAS...
   - LP based approximation schemes
   - Semidefinite Programming
4. Hardness of Approximations
   - Some Results
   - MAX-SNP

Why study Approximation Algorithms?

- Why not ??

Although this may seem a paradox, all exact science is dominated by the idea of approximation. Bertrand Russel.(1872-1970)

Why study Approximation Algorithms?

- Why not ??

Although this may seem a paradox, all exact science is dominated by the idea of approximation. Bertrand Russel.(1872-1970)

Why study Approximation Algorithms?

- Why not ??

Although this may seem a paradox, all exact science is dominated by the idea of approximation. Bertrand Russel.(1872-1970)

Why study Approximation Algorithms?

- Why not ??

Although this may seem a paradox, all exact science is dominated by the idea of approximation. Bertrand Russel.(1872-1970)

Introduction and some definitions

- A lot of optimization problems are NP-Hard.
- The widely believed assumption is that $P \neq NP$.
- Approaches include polynomial-time algorithms,heuristics etc.
- Need to get 'footholds' by understanding the combinatorial structure of the problem.

Introduction and some definitions

- A lot of optimization problems are NP-Hard.
- The widely believed assumption is that $P \neq NP$.
- Approaches include polynomial-time algorithms,heuristics etc.
- Need to get 'footholds' by understanding the combinatorial structure of the problem.

Introduction and some definitions

- A lot of optimization problems are NP-Hard.
- The widely believed assumption is that $P \neq NP$.
- Approaches include polynomial-time algorithms,heuristics etc.
- Need to get 'footholds' by understanding the combinatorial structure of the problem.

Introduction and some definitions

- A lot of optimization problems are NP-Hard.
- The widely believed assumption is that $P \neq NP$.
- Approaches include polynomial-time algorithms,heuristics etc.
- Need to get 'footholds' by understanding the combinatorial structure of the problem.

Introduction and some definitions

- An α-approximation algorithm is an algorithm that runs in polynomial time and always produces a solution within a factor of α of the value of the optimal solution.
- Do we know the optimal solution ??
- Lower Bounding OPT...
- Cardinality Vertex Cover (Find a maximal matching in G and output the set of matched vertices.)
- $|M| \leq OPT$.

Introduction and some definitions

- An α-approximation algorithm is an algorithm that runs in polynomial time and always produces a solution within a factor of α of the value of the optimal solution.
- Do we know the optimal solution ??
- Lower Bounding OPT...
- Cardinality Vertex Cover (Find a maximal matching in G and output the set of matched vertices.)
- $|M| \leq OPT$.

Introduction and some definitions

- An α-approximation algorithm is an algorithm that runs in polynomial time and always produces a solution within a factor of α of the value of the optimal solution.
- Do we know the optimal solution ??
- Lower Bounding OPT...
- Cardinality Vertex Cover (Find a maximal matching in G and output the set of matched vertices.)
- $|M| \leq OPT$.

Introduction and some definitions

- An α-approximation algorithm is an algorithm that runs in polynomial time and always produces a solution within a factor of α of the value of the optimal solution.
- Do we know the optimal solution ??
- Lower Bounding OPT...
- Cardinality Vertex Cover (Find a maximal matching in G and output the set of matched vertices.)
- $|M| \leq OPT$.

Introduction and some definitions

- An α-approximation algorithm is an algorithm that runs in polynomial time and always produces a solution within a factor of α of the value of the optimal solution.
- Do we know the optimal solution ??
- Lower Bounding OPT...
- Cardinality Vertex Cover (Find a maximal matching in G and output the set of matched vertices.)
- $|M| \leq OPT$.

Constant factor appproximations

- Algorithm mentioned above is a 2-factor algorithm for cardinality vertex matching.
- Cover picked has cardinality $2|M| \leq 2.OPT$
- Can the approximation guarentee be improved by better analysis ?

Constant factor appproximations

- Algorithm mentioned above is a 2-factor algorithm for cardinality vertex matching.
- Cover picked has cardinality $2|M| \leq 2.OPT$
- Can the approximation guarentee be improved by better analysis ?

Constant factor appproximations

- Algorithm mentioned above is a 2-factor algorithm for cardinality vertex matching.
- Cover picked has cardinality $2|M| \leq 2.OPT$
- Can the approximation guarentee be improved by better analysis ?

Constant factor appproximations

- Algorithm mentioned above is a 2-factor algorithm for cardinality vertex matching.
- Cover picked has cardinality $2|M| \leq 2.OPT$
- Can the approximation guarentee be improved by better analysis ?

Set Covering

- Given a universe U of n elements, a collection of subsets of U, $S = \{S_1, ..., S_k\}$, and a cost function c, find a minimum cost subcollection of S that covers all elements of S.
- Greedy Algorithm:
- $C = 0$
- While $C \neq U$ do
- Find the most cost effective set in the current iteration. say S.
- let $\alpha = \frac{cost(S)}{|S-C|}$.
- Pick S, and for each $e \in S - C$, set $price(e) = \alpha$. $C = C \cup S$.
- Output the picked sets.

Set Covering

- Given a universe U of n elements, a collection of subsets of U, $S = \{S_1, ..., S_k\}$, and a cost function c, find a minimum cost subcollection of S that covers all elements of S.
- Greedy Algorithm:
- $C = 0$
- While $C \neq U$ do
- Find the most cost effective set in the current iteration. say S.
- let $\alpha = \frac{cost(S)}{|S-C|}$.
- Pick S, and for each $e \in S - C$, set $price(e) = \alpha$. $C = C \cup S$.
- Output the picked sets.

Set Covering

- Given a universe U of n elements, a collection of subsets of U, $S = \{S_1, ..., S_k\}$, and a cost function c, find a minimum cost subcollection of S that covers all elements of S.
- Greedy Algorithm:
- $C = 0$
- While $C \neq U$ do
- Find the most cost effective set in the current iteration. say S.
- let $\alpha = \frac{cost(S)}{|S-C|}$.
- Pick S, and for each $e \in S - C$, set $price(e) = \alpha$. $C = C \cup S$.
- Output the picked sets.

Set Covering

- Given a universe U of n elements, a collection of subsets of U, $S = \{S_1, ..., S_k\}$, and a cost function c, find a minimum cost subcollection of S that covers all elements of S.
- Greedy Algorithm:
- $C = 0$
- While $C \neq U$ do
- Find the most cost effective set in the current iteration. say S.
- let $\alpha = \frac{cost(S)}{|S-C|}$.
- Pick S, and for each $e \in S - C$, set $price(e) = \alpha$. $C = C \cup S$.
- Output the picked sets.

Set Covering

- Given a universe U of n elements, a collection of subsets of U, $S = \{S_1, ..., S_k\}$, and a cost function c, find a minimum cost subcollection of S that covers all elements of S.
- Greedy Algorithm:
- $C = 0$
- While $C \neq U$ do
- Find the most cost effective set in the current iteration. say S.
- let $\alpha = \frac{cost(S)}{|S-C|}$.
- Pick S, and for each $e \in S - C$, set $price(e) = \alpha$. $C = C \cup S$.
- Output the picked sets.

Set Covering

- Given a universe U of n elements, a collection of subsets of U, $S = \{S_1, ..., S_k\}$, and a cost function c, find a minimum cost subcollection of S that covers all elements of S.
- Greedy Algorithm:
- $C = 0$
- While $C \neq U$ do
- Find the most cost effective set in the current iteration. say S.
- let $\alpha = \frac{cost(S)}{|S - C|}$.
- Pick S, and for each $e \in S - C$, set $price(e) = \alpha$. $C = C \cup S$.
- Output the picked sets.

Set Covering

- Given a universe U of n elements, a collection of subsets of U, $S = \{S_1, ..., S_k\}$, and a cost function c, find a minimum cost subcollection of S that covers all elements of S.
- Greedy Algorithm:
- $C = 0$
- While $C \neq U$ do
- Find the most cost effective set in the current iteration. say S.
- let $\alpha = \frac{cost(S)}{|S-C|}$.
- Pick S, and for each $e \in S - C$, set $price(e) = \alpha$. $C = C \cup S$.
- Output the picked sets.

Set Covering

- Given a universe U of n elements, a collection of subsets of U, $S = \{S_1, ..., S_k\}$, and a cost function c, find a minimum cost subcollection of S that covers all elements of S.
- Greedy Algorithm:
- $C = 0$
- While $C \neq U$ do
- Find the most cost effective set in the current iteration. say S.
- let $\alpha = \frac{cost(S)}{|S - C|}$.
- Pick S, and for each $e \in S - C$, set $price(e) = \alpha$. $C = C \cup S$.
- Output the picked sets.

Set Covering continued...

- $price(e_k) \leq \frac{OPT}{|\bar{C}|} \leq \frac{OPT}{n-k+1}$.
- The greedy algorithm is an H_n factor algorithm for the minimum set cover problem, where $H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$.
- Tight example for showing that this is the tightest approximation one can hope for the problem.

Set Covering continued...

- $price(e_k) \leq \frac{OPT}{|\bar{C}|} \leq \frac{OPT}{n-k+1}$.
- The greedy algorithm is an H_n factor algorithm for the minimum set cover problem, where $H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$.
- Tight example for showing that this is the tightest approximation one can hope for the problem.

Set Covering continued...

- $price(e_k) \leq \frac{OPT}{|\bar{C}|} \leq \frac{OPT}{n-k+1}$.
- The greedy algorithm is an H_n factor algorithm for the minimum set cover problem, where $H_n = 1 + \frac{1}{2} + \ldots + \frac{1}{n}$.
- Tight example for showing that this is the tightest approximation one can hope for the problem.

Metric TSP

- Problem Definition: Given a complete graph with nonnegative edge costs, find the minimum cost cycle visiting every vertex exactly once.
- Theorem: For any polynomial time computable function $\alpha(n)$, TSP cannot be approximated within a factor of $\alpha(n)$, unless $P = NP$.
- Key: Reduction from Hamiltonian Cycle Problem...
- Had to assign edge costs that violate traingle inequality.

Metric TSP

- Problem Definition: Given a complete graph with nonnegative edge costs, find the minimum cost cycle visiting every vertex exactly once.
- Theorem: For any polynomial time computable function $\alpha(n)$, TSP cannot be approximated within a factor of $\alpha(n)$, unless $P = NP$.
- Key: Reduction from Hamiltonian Cycle Problem...
- Had to assign edge costs that violate traingle inequality.

Metric TSP

- Problem Definition: Given a complete graph with nonnegative edge costs, find the minimum cost cycle visiting every vertex exactly once.
- Theorem: For any polynomial time computable function $\alpha(n)$, TSP cannot be approximated within a factor of $\alpha(n)$, unless $P = NP$.
- Key: Reduction from Hamiltonian Cycle Problem...
- Had to assign edge costs that violate traingle inequality.

Metric TSP

- Problem Definition: Given a complete graph with nonnegative edge costs, find the minimum cost cycle visiting every vertex exactly once.
- Theorem: For any polynomial time computable function $\alpha(n)$, TSP cannot be approximated within a factor of $\alpha(n)$, unless $P = NP$.
- Key: Reduction from Hamiltonian Cycle Problem...
- Had to assign edge costs that violate traingle inequality.

Metric TSP continued: A 2-factor algorithm.

- Find an MST T of G.
- Double every edge of MST to get an Eulerian graph.
- Find an Eulerian tour, T_1, on this graph.
- Output the tour that visits vertices of G in the order if their first appearance in T_1. Let C be that tour.
- This is a 2-factor approximation algorithm for metric TSP.
- $Cost(T) \leq OPT, cost(T_1) = 2cost(T), cost(C) \leq cost(T_1)$...

Metric TSP continued: A 2-factor algorithm.

- Find an MST T of G.
- Double every edge of MST to get an Eulerian graph.
- Find an Eulerian tour, T_1, on this graph.
- Output the tour that visits vertices of G in the order if their first appearance in T_1. Let C be that tour.
- This is a 2-factor approximation algorithm for metric TSP.
- $Cost(T) \leq OPT, cost(T_1) = 2cost(T), cost(C) \leq cost(T_1)$...

Metric TSP continued: A 2-factor algorithm.

- Find an MST T of G.
- Double every edge of MST to get an Eulerian graph.
- Find an Eulerian tour, T_1, on this graph.
- Output the tour that visits vertices of G in the order if their first appearance in T_1. Let C be that tour.
- This is a 2-factor approximation algorithm for metric TSP.
- $Cost(T) \leq OPT, cost(T_1) = 2cost(T), cost(C) \leq cost(T_1)$...

Metric TSP continued: A 2-factor algorithm.

- Find an MST T of G.
- Double every edge of MST to get an Eulerian graph.
- Find an Eulerian tour, T_1, on this graph.
- Output the tour that visits vertices of G in the order if their first appearance in T_1. Let C be that tour.
- This is a 2-factor approximation algorithm for metric TSP.
- $Cost(T) \leq OPT, cost(T_1) = 2cost(T), cost(C) \leq cost(T_1)$...

Metric TSP continued: A 2-factor algorithm.

- Find an MST T of G.
- Double every edge of MST to get an Eulerian graph.
- Find an Eulerian tour, T_1, on this graph.
- Output the tour that visits vertices of G in the order if their first appearance in T_1. Let C be that tour.
- This is a 2-factor approximation algorithm for metric TSP.
- $Cost(T) \leq OPT, cost(T_1) = 2cost(T), cost(C) \leq cost(T_1)$...

Metric TSP continued: A 2-factor algorithm.

- Find an MST T of G.
- Double every edge of MST to get an Eulerian graph.
- Find an Eulerian tour, T_1, on this graph.
- Output the tour that visits vertices of G in the order if their first appearance in T_1. Let C be that tour.
- This is a 2-factor approximation algorithm for metric TSP.
- $Cost(T) \leq OPT, cost(T_1) = 2cost(T), cost(C) \leq cost(T_1)$...

Improving the approximation to factor $3/2$...

- Find an MST T of G.
- Compute a minimum cost perfect matching, M, on the set of odd-degree vertices of T. Add M to T to obtain an Eulerian graph.
- Find an Eulerian tour, T_1, on this graph.
- Output the tour that visits vertices of G in the order if their first appearance in T_1. Let C be that tour.
- Note that $Cost(M) \leq OPT/2$.
- This is a 3/2 factor approximation guarentee for metric TSP.
- Conjecture: An approximation factor of 4/3 may be achievable.

Improving the approximation to factor $3/2$...

- Find an MST T of G.
- Compute a minimum cost perfect matching, M, on the set of odd-degree vertices of T. Add M to T to obtain an Eulerian graph.
- Find an Eulerian tour, T_1, on this graph.
- Output the tour that visits vertices of G in the order if their first appearance in T_1. Let C be that tour.
- Note that $Cost(M) \leq OPT/2$.
- This is a 3/2 factor approximation guarentee for metric TSP.
- Conjecture: An approximation factor of 4/3 may be achievable.

Improving the approximation to factor $3/2$...

- Find an MST T of G.
- Compute a minimum cost perfect matching, M, on the set of odd-degree vertices of T. Add M to T to obtain an Eulerian graph.
- Find an Eulerian tour, T_1, on this graph.
- Output the tour that visits vertices of G in the order if their first appearance in T_1. Let C be that tour.
- Note that $Cost(M) \leq OPT/2$.
- This is a 3/2 factor approximation guarentee for metric TSP.
- Conjecture: An approximation factor of 4/3 may be achievable.

Improving the approximation to factor $3/2$...

- Find an MST T of G.
- Compute a minimum cost perfect matching, M, on the set of odd-degree vertices of T. Add M to T to obtain an Eulerian graph.
- Find an Eulerian tour, T_1, on this graph.
- Output the tour that visits vertices of G in the order if their first appearance in T_1. Let C be that tour.
- Note that $Cost(M) \leq OPT/2$.
- This is a 3/2 factor approximation guarentee for metric TSP.
- Conjecture: An approximation factor of 4/3 may be achievable.

Improving the approximation to factor $3/2$...

- Find an MST T of G.
- Compute a minimum cost perfect matching, M, on the set of odd-degree vertices of T. Add M to T to obtain an Eulerian graph.
- Find an Eulerian tour, T_1, on this graph.
- Output the tour that visits vertices of G in the order if their first appearance in T_1. Let C be that tour.
- Note that $Cost(M) \leq OPT/2$.
- This is a 3/2 factor approximation guarentee for metric TSP.
- Conjecture: An approximation factor of 4/3 may be achievable.

Improving the approximation to factor $3/2$...

- Find an MST T of G.
- Compute a minimum cost perfect matching, M, on the set of odd-degree vertices of T. Add M to T to obtain an Eulerian graph.
- Find an Eulerian tour, T_1, on this graph.
- Output the tour that visits vertices of G in the order if their first appearance in T_1. Let C be that tour.
- Note that $Cost(M) \leq OPT/2$.
- This is a 3/2 factor approximation guarentee for metric TSP.
- Conjecture: An approximation factor of 4/3 may be achievable.

Improving the approximation to factor $3/2$...

- Find an MST T of G.
- Compute a minimum cost perfect matching, M, on the set of odd-degree vertices of T. Add M to T to obtain an Eulerian graph.
- Find an Eulerian tour, T_1, on this graph.
- Output the tour that visits vertices of G in the order if their first appearance in T_1. Let C be that tour.
- Note that $Cost(M) \leq OPT/2$.
- This is a 3/2 factor approximation guarentee for metric TSP.
- Conjecture: An approximation factor of 4/3 may be achievable.

Some definitions

- Some NP-Hard problems may allow approximability to any required degree.
- Approximation Scheme: Let Π be an NP-Hard problem with objective function f_Π. An algorithm A is an approximation scheme for Π if on input (I, ϵ), where I is an instance of Π, and $\epsilon > 0$ is an error parameter, it outputs a solution s such that:
  $f_\Pi(I, s) \leq (1 + \epsilon)OPT$ if Π is a minimization problem.
  $f_\Pi(I, s) \geq (1 - \epsilon)OPT$ if Π is a maximization problem.

Some definitions

- Some NP-Hard problems may allow approximability to any required degree.
- Approximation Scheme: Let Π be an NP-Hard problem with objective function f_Π. An algorithm A is an approximation scheme for Π if on input (I, ϵ), where I is an instance of Π, and $\epsilon > 0$ is an error parameter, it outputs a solution s such that:
  $f_\Pi(I, s) \leq (1 + \epsilon)OPT$ if Π is a minimization problem.
  $f_\Pi(I, s) \geq (1 - \epsilon)OPT$ if Π is a maximization problem.

PTAS and FPTAS

- A is said to be a polynomial time approximation scheme(PTAS), if for each fixed $\epsilon > 0$, its running time is bounded by a polynomial in the size of the instance I.
- A is said to be a fully polynomial time approximation scheme(FPTAS), if for each fixed $\epsilon > 0$, its running time is bounded by a polynomial in the size of the instance I and $1/\epsilon$.

PTAS and FPTAS

- A is said to be a polynomial time approximation scheme(PTAS), if for each fixed $\epsilon > 0$, its running time is bounded by a polynomial in the size of the instance I.
- A is said to be a fully polynomial time approximation scheme(FPTAS), if for each fixed $\epsilon > 0$, its running time is bounded by a polynomial in the size of the instance I and $1/\epsilon$.

AS continued...

- Knapsack being NP-Hard does not admit a polynomial time algorithm.
- But it does admit a pseudo-polynomial time algorithm.
- This fact is critically used to obtain a FPTAS for Knapsack.
- All known pseudo-polynomial time algorithms for NP-Hard problems are based on dynamic programming.

AS continued...

- Knapsack being NP-Hard does not admit a polynomial time algorithm.
- But it does admit a pseudo-polynomial time algorithm.
- This fact is critically used to obtain a FPTAS for Knapsack.
- All known pseudo-polynomial time algorithms for NP-Hard problems are based on dynamic programming.

AS continued...

- Knapsack being NP-Hard does not admit a polynomial time algorithm.
- But it does admit a pseudo-polynomial time algorithm.
- This fact is critically used to obtain a FPTAS for Knapsack.
- All known pseudo-polynomial time algorithms for NP-Hard problems are based on dynamic programming.

AS continued...

- Knapsack being NP-Hard does not admit a polynomial time algorithm.
- But it does admit a pseudo-polynomial time algorithm.
- This fact is critically used to obtain a FPTAS for Knapsack.
- All known pseudo-polynomial time algorithms for NP-Hard problems are based on dynamic programming.

Knapsack Problem

- Definition: Given a set $S = \{a_1, \ldots, a_n\}$ of objects, with sizes $size(a_i) \in \mathbb{Z}^+$, and profits $p(a_i) \in \mathbb{Z}^+$, and a knapsack capacity $B \in \mathbb{Z}^+$, find a maximum profit subset of objects having total size $\leq B$.
- Dynamic Programming:
- Let $S_{i,p}$ denote a subset of $\{a_1, \ldots, a_i$ with total profit exactly p.
- $A(i+1, p) = min\{A(i,p), size(a_{i+1}) + A(i, p - profit(a_{i+1}) \quad if \quad p(a_{i+1}) < p$.
- $A(i+1, p) = A(i,p) \quad otherwise$.
- $max\{p | A(n,p) \leq B\}$.

Knapsack Problem

- Definition: Given a set $S = \{a_1, \ldots, a_n\}$ of objects, with sizes $size(a_i) \in \mathbb{Z}^+$, and profits $p(a_i) \in \mathbb{Z}^+$, and a knapsack capacity $B \in \mathbb{Z}^+$, find a maximum profit subset of objects having total size $\leq B$.
- Dynamic Programming:
- Let $S_{i,p}$ denote a subset of $\{a_1, \ldots, a_i$ with total profit exactly p.
- $A(i+1, p) = min\{A(i,p), size(a_{i+1}) + A(i, p - profit(a_{i+1}) \quad if \quad p(a_{i+1}) < p$.
- $A(i+1, p) = A(i,p) \quad otherwise$.
- $max\{p | A(n,p) \leq B\}$.

Knapsack Problem

- Definition: Given a set $S = \{a_1, \ldots, a_n\}$ of objects, with sizes $size(a_i) \in \mathbb{Z}^+$, and profits $p(a_i) \in \mathbb{Z}^+$, and a knapsack capacity $B \in \mathbb{Z}^+$, find a maximum profit subset of objects having total size $\leq B$.
- Dynamic Programming:
- Let $S_{i,p}$ denote a subset of $\{a_1, \ldots, a_i$ with total profit exactly p.
- $A(i+1, p) = min\{A(i, p), size(a_{i+1}) + A(i, p - profit(a_{i+1}) \quad if \quad p(a_{i+1}) < p$.
- $A(i+1, p) = A(i, p) \quad otherwise$.
- $max\{p | A(n, p) \leq B\}$.

Knapsack Problem

- Definition: Given a set $S = \{a_1, \ldots, a_n\}$ of objects, with sizes $size(a_i) \in \mathbb{Z}^+$, and profits $p(a_i) \in \mathbb{Z}^+$, and a knapsack capacity $B \in \mathbb{Z}^+$, find a maximum profit subset of objects having total size $\leq B$.
- Dynamic Programming:
- Let $S_{i,p}$ denote a subset of $\{a_1, \ldots, a_i$ with total profit exactly p.
- $A(i+1,p) = min\{A(i,p), size(a_{i+1}) + A(i,p - profit(a_{i+1}) \quad if \quad p(a_{i+1}) < p$.
- $A(i+1,p) = A(i,p) \quad otherwise$.
- $max\{p|A(n,p) \leq B\}$.

Knapsack Problem

- Definition: Given a set $S = \{a_1, \ldots, a_n\}$ of objects, with sizes $size(a_i) \in \mathbb{Z}^+$, and profits $p(a_i) \in \mathbb{Z}^+$, and a knapsack capacity $B \in \mathbb{Z}^+$, find a maximum profit subset of objects having total size $\leq B$.
- Dynamic Programming:
- Let $S_{i,p}$ denote a subset of $\{a_1, \ldots, a_i$ with total profit exactly p.
- $A(i+1,p) = min\{A(i,p), size(a_{i+1}) + A(i, p - profit(a_{i+1}) \quad if \quad p(a_{i+1}) < p$.
- $A(i+1,p) = A(i,p) \quad otherwise$.
- $max\{p|A(n,p) \leq B\}$.

Knapsack Problem

- Definition: Given a set $S = \{a_1, \ldots, a_n\}$ of objects, with sizes $size(a_i) \in \mathbb{Z}^+$, and profits $p(a_i) \in \mathbb{Z}^+$, and a knapsack capacity $B \in \mathbb{Z}^+$, find a maximum profit subset of objects having total size $\leq B$.
- Dynamic Programming:
- Let $S_{i,p}$ denote a subset of $\{a_1, \ldots, a_i$ with total profit exactly p.
- $A(i+1,p) = min\{A(i,p), size(a_{i+1}) + A(i,p - profit(a_{i+1}) \quad if \quad p(a_{i+1}) < p$.
- $A(i+1,p) = A(i,p) \quad otherwise$.
- $max\{p|A(n,p) \leq B\}$.

Knapsack Problem Continued...

- Given $\epsilon > 0$, let $K = \frac{\epsilon P}{n}$.
- For each object a_i, define $p'(a_i) = \lfloor \frac{p(a_i)}{K} \rfloor$.
- With these as profits for objects, use dynamic programming to get the most profitable set, S'.
- $p(S') \geq (1 - \epsilon)OPT$.
- Uses $P \leq OPT$. and $Kp'(a_i) \leq p(a_i) \leq K(p'(a_i + 1))$.
- Running time is $\mathbb{O}(n^2 \lfloor \frac{P}{K} \rfloor) = \mathbb{O}(n^2 \lfloor \frac{n}{\epsilon} \rfloor)$

Knapsack Problem Continued...

- Given $\epsilon > 0$, let $K = \frac{\epsilon P}{n}$.
- For each object a_i, define $p'(a_i) = \lfloor \frac{p(a_i)}{K} \rfloor$.
- With these as profits for objects, use dynamic programming to get the most profitable set, S'.
- $p(S') \geq (1 - \epsilon)OPT$.
- Uses $P \leq OPT$. and $Kp'(a_i) \leq p(a_i) \leq K(p'(a_i + 1))$.
- Running time is $\mathbb{O}(n^2 \lfloor \frac{P}{K} \rfloor) = \mathbb{O}(n^2 \lfloor \frac{n}{\epsilon} \rfloor)$

Knapsack Problem Continued...

- Given $\epsilon > 0$, let $K = \frac{\epsilon P}{n}$.
- For each object a_i, define $p'(a_i) = \lfloor \frac{p(a_i)}{K} \rfloor$.
- With these as profits for objects, use dynamic programming to get the most profitable set, S'.
- $p(S') \geq (1 - \epsilon)OPT$.
- Uses $P \leq OPT$. and $Kp'(a_i) \leq p(a_i) \leq K(p'(a_i + 1))$.
- Running time is $\mathbb{O}(n^2 \lfloor \frac{P}{K} \rfloor) = \mathbb{O}(n^2 \lfloor \frac{n}{\epsilon} \rfloor)$

Knapsack Problem Continued...

- Given $\epsilon > 0$, let $K = \frac{\epsilon P}{n}$.
- For each object a_i, define $p'(a_i) = \lfloor \frac{p(a_i)}{K} \rfloor$.
- With these as profits for objects, use dynamic programming to get the most profitable set, S'.
- $p(S') \geq (1 - \epsilon)OPT$.
- Uses $P \leq OPT$. and $Kp'(a_i) \leq p(a_i) \leq K(p'(a_i + 1))$.
- Running time is $\mathbb{O}(n^2 \lfloor \frac{P}{K} \rfloor) = \mathbb{O}(n^2 \lfloor \frac{n}{\epsilon} \rfloor)$

Knapsack Problem Continued...

- Given $\epsilon > 0$, let $K = \frac{\epsilon P}{n}$.
- For each object a_i, define $p'(a_i) = \lfloor \frac{p(a_i)}{K} \rfloor$.
- With these as profits for objects, use dynamic programming to get the most profitable set, S'.
- $p(S') \geq (1 - \epsilon)OPT$.
- Uses $P \leq OPT$. and $Kp'(a_i) \leq p(a_i) \leq K(p'(a_i + 1))$.
- Running time is $\mathbb{O}(n^2 \lfloor \frac{P}{K} \rfloor) = \mathbb{O}(n^2 \lfloor \frac{n}{\epsilon} \rfloor)$

Knapsack Problem Continued...

- Given $\epsilon > 0$, let $K = \frac{\epsilon P}{n}$.
- For each object a_i, define $p'(a_i) = \lfloor \frac{p(a_i)}{K} \rfloor$.
- With these as profits for objects, use dynamic programming to get the most profitable set, S'.
- $p(S') \geq (1 - \epsilon)OPT$.
- Uses $P \leq OPT$. and $Kp'(a_i) \leq p(a_i) \leq K(p'(a_i + 1))$.
- Running time is $\mathbb{O}(n^2 \lfloor \frac{P}{K} \rfloor) = \mathbb{O}(n^2 \lfloor \frac{n}{\epsilon} \rfloor)$

LP based schemes

- The linear relaxation of an LP provides a lower bound to the optimal solution.
- Integrality gap/ratio $sup_I \frac{OPT(I)}{OPT_f(I)}$. If the relaxation is not exact, then the best approximation ratio an algorithm may hope for is the integrality ratio.
- Rounding of fractional values(including randomized rounding)
- Dual LP. Dual of the linear programming relaxation. $(z_{DP} \leq z_{LP} \leq OPT)$

LP based schemes

- The linear relaxation of an LP provides a lower bound to the optimal solution.
- Integrality gap/ratio $sup_I \frac{OPT(I)}{OPT_f(I)}$. If the relaxation is not exact, then the best approximation ratio an algorithm may hope for is the integrality ratio.
- Rounding of fractional values(including randomized rounding)
- Dual LP. Dual of the linear programming relaxation. $(z_{DP} \leq z_{LP} \leq OPT)$

LP based schemes

- The linear relaxation of an LP provides a lower bound to the optimal solution.
- Integrality gap/ratio $sup_I \frac{OPT(I)}{OPT_f(I)}$. If the relaxation is not exact, then the best approximation ratio an algorithm may hope for is the integrality ratio.
- Rounding of fractional values(including randomized rounding)
- Dual LP. Dual of the linear programming relaxation. $(z_{DP} \leq z_{LP} \leq OPT)$

LP based schemes

- The linear relaxation of an LP provides a lower bound to the optimal solution.
- Integrality gap/ratio $sup_I \frac{OPT(I)}{OPT_f(I)}$. If the relaxation is not exact, then the best approximation ratio an algorithm may hope for is the integrality ratio.
- Rounding of fractional values(including randomized rounding)
- Dual LP. Dual of the linear programming relaxation. $(z_{DP} \leq z_{LP} \leq OPT)$

LP based schemes

- Primal-Dual schema. Suitable relaxations to the complementary slackness conditions.
- $\alpha \geq 1, \beta \geq 1$. Then

$$x_j = 0 \quad or \quad c_j/\alpha \leq \sum a_{ij} y_i \leq c_j \tag{1}$$

$$y_i = 0 \quad or \quad b_i \leq \sum a_{ij} x_j \leq \beta b_i \tag{2}$$

$$\sum c_j x_j \leq \alpha\beta \sum b_i y_i \tag{3}$$

LP based schemes

- Primal-Dual schema. Suitable relaxations to the complementary slackness conditions.
- $\alpha \geq 1, \beta \geq 1$. Then

$$x_j = 0 \quad or \quad c_j/\alpha \leq \sum a_{ij} y_i \leq c_j \tag{1}$$

$$y_i = 0 \quad or \quad b_i \leq \sum a_{ij} x_j \leq \beta b_i \tag{2}$$

$$\sum c_j x_j \leq \alpha\beta \sum b_i y_i \tag{3}$$

SemiDefinite Programming

- Another class of relaxations.
- Many NP-Hard problems can be expressed as strict quadratic programs(MAX-CUT).
- maximize C.Y

$$D_i.Y = d_i \tag{4}$$

$$Y \quad positive \quad semidefinite \tag{5}$$

- A matrix is semidefinite if $\forall x \in \mathbb{R}^K, x^T A x \geq 0$.
- $A.B = tr(A^T B)$
- There is a theorem on finding seperating hyperplane for Y in polynomial time.
- As a result, semidefinite programs can be solved in time polynomial in n and $log(1/\epsilon)$ using ellipsoid algorithm.

SemiDefinite Programming

- Another class of relaxations.
- Many NP-Hard problems can be expressed as strict quadratic programs(MAX-CUT).
- maximize C.Y

$$D_i.Y = d_i \tag{4}$$

$$Y \quad positive \quad semidefinite \tag{5}$$

- A matrix is semidefinite if $\forall x \in \mathbb{R}^K, x^T Ax \geq 0$.
- $A.B = tr(A^T B)$
- There is a theorem on finding seperating hyperplane for Y in polynomial time.
- As a result, semidefinite programs can be solved in time polynomial in n and $log(1/\epsilon)$ using ellipsoid algorithm.

SemiDefinite Programming

- Another class of relaxations.
- Many NP-Hard problems can be expressed as strict quadratic programs(MAX-CUT).
- maximize C.Y

$$D_i.Y = d_i \tag{4}$$

$$Y \quad positive \quad semidefinite \tag{5}$$

- A matrix is semidefinite if $\forall x \in \mathbb{R}^K, x^T Ax \geq 0$.
- $A.B = tr(A^T B)$
- There is a theorem on finding seperating hyperplane for Y in polynomial time.
- As a result, semidefinite programs can be solved in time polynomial in n and $log(1/\epsilon)$ using ellipsoid algorithm.

SemiDefinite Programming

- Another class of relaxations.
- Many NP-Hard problems can be expressed as strict quadratic programs(MAX-CUT).
- maximize C.Y

$$D_i.Y = d_i \tag{4}$$

$$Y \quad positive \quad semidefinite \tag{5}$$

- A matrix is semidefinite if $\forall x \in \mathbb{R}^{\ltimes}, x^T A x \geq 0$.
- $A.B = tr(A^T B)$
- There is a theorem on finding seperating hyperplane for Y in polynomial time.
- As a result, semidefinite programs can be solved in time polynomial in n and $log(1/\epsilon)$ using ellipsoid algorithm.

SemiDefinite Programming

- Another class of relaxations.
- Many NP-Hard problems can be expressed as strict quadratic programs(MAX-CUT).
- maximize C.Y

$$D_i.Y = d_i \tag{4}$$

$$Y \quad positive \quad semidefinite \tag{5}$$

- A matrix is semidefinite if $\forall x \in \mathbb{R}^{\ltimes}, x^T A x \geq 0$.
- $A.B = tr(A^T B)$
- There is a theorem on finding seperating hyperplane for Y in polynomial time.
- As a result, semidefinite programs can be solved in time polynomial in n and $log(1/\epsilon)$ using ellipsoid algorithm.

SemiDefinite Programming

- Another class of relaxations.
- Many NP-Hard problems can be expressed as strict quadratic programs(MAX-CUT).
- maximize C.Y

$$D_i.Y = d_i \tag{4}$$

$$Y \quad positive \quad semidefinite \tag{5}$$

- A matrix is semidefinite if $\forall x \in \mathbb{R}^{\ltimes}, x^T A x \geq 0$.
- $A.B = tr(A^T B)$
- There is a theorem on finding seperating hyperplane for Y in polynomial time.
- As a result, semidefinite programs can be solved in time polynomial in n and $log(1/\epsilon)$ using ellipsoid algorithm.

SemiDefinite Programming

- Another class of relaxations.
- Many NP-Hard problems can be expressed as strict quadratic programs(MAX-CUT).
- maximize C.Y

$$D_i.Y = d_i \tag{4}$$
$$Y \quad positive \quad semidefinite \tag{5}$$

- A matrix is semidefinite if $\forall x \in \mathbb{R}^{\ltimes}, x^T A x \geq 0$.
- $A.B = tr(A^T B)$
- There is a theorem on finding seperating hyperplane for Y in polynomial time.
- As a result, semidefinite programs can be solved in time polynomial in n and $log(1/\epsilon)$ using ellipsoid algorithm.

Some results

- Strongly NP-Hard: A problem is strongly NP-Hard if the problem is NP-Hard even when all the numbers in the input are encoded in unary.
- A strongly NP-Hard problem cannot have a FPTAS assuming $P \neq NP$.
- KNAPSACK is not strongly NP-hard.

Some results

- Strongly NP-Hard: A problem is strongly NP-Hard if the problem is NP-Hard even when all the numbers in the input are encoded in unary.
- A strongly NP-Hard problem cannot have a FPTAS assuming $P \neq NP$.
- KNAPSACK is not strongly NP-hard.

Some results

- Strongly NP-Hard: A problem is strongly NP-Hard if the problem is NP-Hard even when all the numbers in the input are encoded in unary.
- A strongly NP-Hard problem cannot have a FPTAS assuming $P \neq NP$.
- KNAPSACK is not strongly NP-hard.

Inapproximability Results

- Sometimes, achieving certain reasonable approximation ratios is no easier than computing optimal solutions.
- Approximability preserving reductions. If two problems are interreducble as such, then they have the same approximability.
- This can be used to categorize NP-Hard problems into a small number of equivalence classes and get *complete* problems for each class.

Inapproximability Results

- Sometimes, achieving certain reasonable approximation ratios is no easier than computing optimal solutions.
- Approximability preserving reductions. If two problems are interreducble as such, then they have the same approximability.
- This can be used to categorize NP-Hard problems into a small number of equivalence classes and get *complete* problems for each class.

Inapproximability Results

- Sometimes, achieving certain reasonable approximation ratios is no easier than computing optimal solutions.
- Approximability preserving reductions. If two problems are interreducble as such, then they have the same approximability.
- This can be used to categorize NP-Hard problems into a small number of equivalence classes and get *complete* problems for each class.

PCP(Probabilistically checkable proofs) Theorem

- Probabilistic characterizations of class NP yield a general technique for obtaining gap-introducing reductions. The PCP Theorem captures this characterization.
- Class PCP(r(n), q(n)) : a complexity class consisting of every language with an (r(n), q(n))-restricted verifier. Verifier reads the input of size n and uses O(r(n)) random bits to compute a sequence of O(q(n)) addresses in the proof. if input $\in$ L , then probability of acceptance is 1, else it is less than half.
- $NP = PCP(logn, 1)$

PCP(Probabilistically checkable proofs) Theorem

- Probabilistic characterizations of class NP yield a general technique for obtaining gap-introducing reductions. The PCP Theorem captures this characterization.
- Class PCP(r(n), q(n)) : a complexity class consisting of every language with an (r(n), q(n))-restricted verifier. Verifier reads the input of size n and uses O(r(n)) random bits to compute a sequence of O(q(n)) addresses in the proof. if input $\in$ L , then probability of acceptance is 1, else it is less than half.
- $NP = PCP(logn, 1)$

PCP(Probabilistically checkable proofs) Theorem

- Probabilistic characterizations of class NP yield a general technique for obtaining gap-introducing reductions. The PCP Theorem captures this characterization.
- Class PCP(r(n), q(n)) : a complexity class consisting of every language with an (r(n), q(n))-restricted verifier. Verifier reads the input of size n and uses O(r(n)) random bits to compute a sequence of O(q(n)) addresses in the proof. if input $\in$ L , then probability of acceptance is 1, else it is less than half.
- $NP = PCP(logn, 1)$

MAX-SNP

- Class of Problems defined by Papadimitriou et al.for studying which problems have a PTAS.
- Max-SNP is defined as a class of problems having constant factor approximation algorithms, but no approximation schemes unless $P = NP$.
- Result: There does not exist a PTAS for MAX-SNP hard problems unless $P = NP$. (Proof uses PCP Theorem)
- Using approximability preserving reductions, completeness for MAX-SNP problems were defined.

MAX-SNP

- Class of Problems defined by Papadimitriou et al.for studying which problems have a PTAS.
- Max-SNP is defined as a class of problems having constant factor approximation algorithms, but no approximation schemes unless $P = NP$.
- Result: There does not exist a PTAS for MAX-SNP hard problems unless $P = NP$. (Proof uses PCP Theorem)
- Using approximability preserving reductions, completeness for MAX-SNP problems were defined.

MAX-SNP

- Class of Problems defined by Papadimitriou et al.for studying which problems have a PTAS.
- Max-SNP is defined as a class of problems having constant factor approximation algorithms, but no approximation schemes unless $P = NP$.
- Result: There does not exist a PTAS for MAX-SNP hard problems unless $P = NP$. (Proof uses PCP Theorem)
- Using approximability preserving reductions, completeness for MAX-SNP problems were defined.

MAX-SNP

- Class of Problems defined by Papadimitriou et al.for studying which problems have a PTAS.
- Max-SNP is defined as a class of problems having constant factor approximation algorithms, but no approximation schemes unless $P = NP$.
- Result: There does not exist a PTAS for MAX-SNP hard problems unless $P = NP$. (Proof uses PCP Theorem)
- Using approximability preserving reductions, completeness for MAX-SNP problems were defined.

MAX-SNP

- A reduction : A problem P is A-reducible if to problem T , implies if P is approximable to a factor a, then T is approximable to a factor O(a).
- AP reduction : A problem P is AP-reducible if to problem T , implies if P is approximable to a factor $1 + a$, then T is approximable to a factor $1 + O(a)$.
- L-Reductions: A L-reduction from A to B is a pair of functions R and S, computable in logarithmic space, such that if x is an instance of A with optimal cost OPT(x), then R(x) is an instance of B with optimal cost that satisfies: $OPT(R(x)) \leq \alpha OPT(x)$

MAX-SNP

- A reduction : A problem P is A-reducible if to problem T , implies if P is approximable to a factor a, then T is approximable to a factor O(a).
- AP reduction : A problem P is AP-reducible if to problem T , implies if P is approximable to a factor $1 + a$, then T is approximable to a factor $1 + O(a)$.
- L-Reductions: A L-reduction from A to B is a pair of functions R and S, computable in logarithmic space, such that if x is an instance of A with optimal cost OPT(x), then R(x) is an instance of B with optimal cost that satisfies: $OPT(R(x)) \leq \alpha OPT(x)$

MAX-SNP

- A reduction : A problem P is A-reducible if to problem T , implies if P is approximable to a factor a, then T is approximable to a factor O(a).
- AP reduction : A problem P is AP-reducible if to problem T , implies if P is approximable to a factor $1 + a$, then T is approximable to a factor $1 + O(a)$.
- L-Reductions: A L-reduction from A to B is a pair of functions R and S, computable in logarithmic space, such that if x is an instance of A with optimal cost OPT(x), then R(x) is an instance of B with optimal cost that satisfies: $OPT(R(x)) \leq \alpha OPT(x)$

MAX-SNP

- Using L-reductions(), it was shown that every MAX-SNP Hard problem is L-reducible to the MAX-3SAT, MAX-CUT, Metric TSP problems.
- MAX-3SAT, MAX-CUT, Metric TSP are MAX-SNP complete.
- MAX-CSP. (Constraint Satisfaction Problem)
- Only two types of Max-CSP problems: either solvable to optimality in polynomial time, or, MAX-SNP Hard.

MAX-SNP

- Using L-reductions(), it was shown that every MAX-SNP Hard problem is L-reducible to the MAX-3SAT, MAX-CUT, Metric TSP problems.
- MAX-3SAT, MAX-CUT, Metric TSP are MAX-SNP complete.
- MAX-CSP. (Constraint Satisfaction Problem)
- Only two types of Max-CSP problems: either solvable to optimality in polynomial time, or, MAX-SNP Hard.

MAX-SNP

- Using L-reductions(), it was shown that every MAX-SNP Hard problem is L-reducible to the MAX-3SAT, MAX-CUT, Metric TSP problems.
- MAX-3SAT, MAX-CUT, Metric TSP are MAX-SNP complete.
- MAX-CSP. (Constraint Satisfaction Problem)
- Only two types of Max-CSP problems: either solvable to optimality in polynomial time, or, MAX-SNP Hard.

MAX-SNP

- Using L-reductions(), it was shown that every MAX-SNP Hard problem is L-reducible to the MAX-3SAT, MAX-CUT, Metric TSP problems.
- MAX-3SAT, MAX-CUT, Metric TSP are MAX-SNP complete.
- MAX-CSP. (Constraint Satisfaction Problem)
- Only two types of Max-CSP problems: either solvable to optimality in polynomial time, or, MAX-SNP Hard.