Heuristics, Approximation Algorithms

Lecture 24

April 24, 2018

Most slides are courtesy Prof. Chekuri

Part I

Heuristics

Coping with Intractability

Question: Many useful/important problems are **NP-Hard** or worse. How does one cope with them?

Coping with Intractability

Question: Many useful/important problems are **NP-Hard** or worse. How does one cope with them?

Some general things that people do.

1. Consider special cases of the problem which may be tractable.

Coping with Intractability

Question: Many useful/important problems are **NP-Hard** or worse. How does one cope with them?

Some general things that people do.

1. Consider special cases of the problem which may be tractable.
2. Run inefficient algorithms (for example exponential time algorithms for **NP-Hard** problems) augmented with (very) clever heuristics
    1. stop algorithm when time/resources run out
    2. use massive computational power

Coping with Intractability

Question: Many useful/important problems are **NP-Hard** or worse. How does one cope with them?

Some general things that people do.

1. Consider special cases of the problem which may be tractable.
2. Run inefficient algorithms (for example exponential time algorithms for **NP-Hard** problems) augmented with (very) clever heuristics
    1. stop algorithm when time/resources run out
    2. use massive computational power
3. Exploit properties of instances that arise in practice which may be much easier. Give up on hard instances, which is OK.

Coping with Intractability

Question: Many useful/important problems are **NP-Hard** or worse. How does one cope with them?

Some general things that people do.

1. Consider special cases of the problem which may be tractable.
2. Run inefficient algorithms (for example exponential time algorithms for **NP-Hard** problems) augmented with (very) clever heuristics
    1. stop algorithm when time/resources run out
    2. use massive computational power
3. Exploit properties of instances that arise in practice which may be much easier. Give up on hard instances, which is OK.
4. Settle for sub-optimal (aka approximate) solutions, especially for optimization problems

NP and EXP

EXP: all problems that have an exponential time algorithm.

Proposition

NP $\subseteq$ **EXP**.

Proof.

Let $X \in$ **NP** with certifier C. To prove $X \in EXP$, here is an algorithm for X. Given input s,

1. For every t, with $|t| \le p(|s|)$ run $C(s,t)$; answer "yes" if any one of these calls returns "yes", otherwise say "no". □

NP and EXP

EXP: all problems that have an exponential time algorithm.

Proposition

NP $\subseteq$ **EXP**.

Proof.

Let $X \in$ **NP** with certifier C. To prove $X \in EXP$, here is an algorithm for X. Given input s,

1. For every t, with $|t| \leq p(|s|)$ run $C(s, t)$; answer "yes" if any one of these calls returns "yes", otherwise say "no". □

Every problem in **NP** has a brute-force "try all possibilities" algorithm that runs in exponential time.

Examples

1. **SAT**: try all possible truth assignment to variables.
2. **Independent set**: try all possible subsets of vertices.
3. **Vertex cover**: try all possible subsets of vertices.

Improving brute-force via intelligent backtracking

1. Backtrack search: enumeration with bells and whistles to "heuristically" cut down search space.

Improving brute-force via intelligent backtracking

1. Backtrack search: enumeration with bells and whistles to "heuristically" cut down search space.
2. Works quite well in practice for several problems.

Improving brute-force via intelligent backtracking

1. Backtrack search: enumeration with bells and whistles to "heuristically" cut down search space.
2. Works quite well in practice for several problems. especially for small enough problem sizes.

Backtrack Search Algorithm for SAT

Input: CNF Formula φ on n variables $x_1, \ldots, x_n$ and m clauses
Output: Is φ satisfiable or not.

Backtrack Search Algorithm for SAT

Input: CNF Formula φ on n variables $x_1, \ldots, x_n$ and m clauses
Output: Is φ satisfiable or not.

1. Pick a variable x_i
2. Set $x_i = 0$ and let φ' be the simplified CNF formula

Backtrack Search Algorithm for SAT

Input: CNF Formula φ on n variables $x_1, \ldots, x_n$ and m clauses
Output: Is φ satisfiable or not.

1. Pick a variable x_i
2. Set $x_i = 0$ and let φ' be the simplified CNF formula
3. Run a simple (heuristic) check on φ': returns "yes", "no" or "not sure"

Backtrack Search Algorithm for SAT

Input: CNF Formula φ on n variables $x_1, \ldots, x_n$ and m clauses
Output: Is φ satisfiable or not.

1. Pick a variable x_i
2. Set $x_i = 0$ and let φ' be the simplified CNF formula
3. Run a simple (heuristic) check on φ': returns "yes", "no" or "not sure"
    1. If "not sure" recursively solve φ'
    2. If φ' is satisfiable, return "yes"

Backtrack Search Algorithm for SAT

Input: CNF Formula φ on n variables $x_1, \ldots, x_n$ and m clauses
Output: Is φ satisfiable or not.

1. Pick a variable x_i
2. Set $x_i = 0$ and let φ' be the simplified CNF formula
3. Run a simple (heuristic) check on φ': returns "yes", "no" or "not sure"
    1. If "not sure" recursively solve φ'
    2. If φ' is satisfiable, return "yes"
4. Set $x_i = 1$ and let φ'' be the simplified CNF formula.
5. Run simple check on φ'': returns "yes", "no" or "not sure"

Backtrack Search Algorithm for SAT

Input: CNF Formula φ on n variables $x_1, \ldots, x_n$ and m clauses
Output: Is φ satisfiable or not.

1. Pick a variable x_i
2. Set $x_i = 0$ and let φ' be the simplified CNF formula
3. Run a simple (heuristic) check on φ': returns "yes", "no" or "not sure"
    1. If "not sure" recursively solve φ'
    2. If φ' is satisfiable, return "yes"
4. Set $x_i = 1$ and let φ'' be the simplified CNF formula.
5. Run simple check on φ'': returns "yes", "no" or "not sure"
    1. If "not sure" recursively solve φ''
    2. If φ'' is satisfiable, return "yes"

Backtrack Search Algorithm for SAT

Input: CNF Formula φ on n variables $x_1, \ldots, x_n$ and m clauses
Output: Is φ satisfiable or not.

1. Pick a variable x_i
2. Set $x_i = 0$ and let φ' be the simplified CNF formula
3. Run a simple (heuristic) check on φ': returns "yes", "no" or "not sure"
    1. If "not sure" recursively solve φ'
    2. If φ' is satisfiable, return "yes"
4. Set $x_i = 1$ and let φ'' be the simplified CNF formula.
5. Run simple check on φ'': returns "yes", "no" or "not sure"
    1. If "not sure" recursively solve φ''
    2. If φ'' is satisfiable, return "yes"
6. Return "no"

Certain part of the search space is pruned.

Example

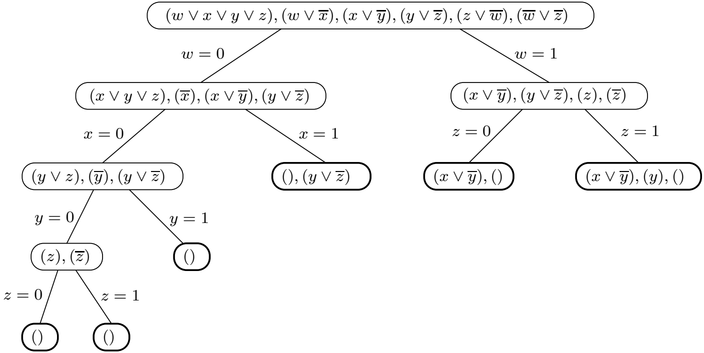


Figure: Backtrack search. Formula is not satisfiable.

Figure taken from Dasgupta etal book.

Backtrack Search Algorithm for SAT

How do we pick the order of variables?

Backtrack Search Algorithm for SAT

How do we pick the order of variables? Heuristically! Examples:

1. pick variable that occurs in most clauses first
2. pick variable that appears in most size 2 clauses first
3. ...

Backtrack Search Algorithm for SAT

How do we pick the order of variables? Heuristically! Examples:

1. pick variable that occurs in most clauses first
2. pick variable that appears in most size 2 clauses first
3. ...

What are quick tests for Satisfiability?

Backtrack Search Algorithm for SAT

How do we pick the order of variables? Heuristically! Examples:

1. pick variable that occurs in most clauses first
2. pick variable that appears in most size 2 clauses first
3. ...

What are quick tests for Satisfiability?
Depends on known special cases and heuristics. Examples.

1. Obvious test: return "no" if empty clause, "yes" if no clauses left and otherwise "not sure"

Backtrack Search Algorithm for SAT

How do we pick the order of variables? Heuristically! Examples:

1. pick variable that occurs in most clauses first
2. pick variable that appears in most size 2 clauses first
3. ...

What are quick tests for Satisfiability?
Depends on known special cases and heuristics. Examples.

1. Obvious test: return "no" if empty clause, "yes" if no clauses left and otherwise "not sure"
2. Run obvious test and in addition if all clauses are of size **2** then run 2-SAT polynomial time algorithm
3. ...

Branch-and-Bound

Backtracking for optimization problems

Intelligent backtracking can be also used for optimization problems.
Consider a minimization problem.
Notation: for instance I, $opt(I)$ is optimum value on I.

Branch-and-Bound

Backtracking for optimization problems

Intelligent backtracking can be also used for optimization problems.
Consider a minimization problem.
Notation: for instance I, $opt(I)$ is optimum value on I.

P_0 initial instance of given problem.

1. We will keep track of the best solution value B found so far.

Branch-and-Bound

Backtracking for optimization problems

Intelligent backtracking can be also used for optimization problems.
Consider a minimization problem.
Notation: for instance I, $opt(I)$ is optimum value on I.

P_0 initial instance of given problem.

1. We will keep track of the best solution value B found so far. Initialize B to be crude upper bound on $opt(I)$.

Branch-and-Bound

Backtracking for optimization problems

Intelligent backtracking can be also used for optimization problems.
Consider a minimization problem.
Notation: for instance I, $opt(I)$ is optimum value on I.

P_0 initial instance of given problem.

1. We will keep track of the best solution value B found so far. Initialize B to be crude upper bound on $opt(I)$.
2. Let P be a subproblem at some stage of exploration.
3. If P is a complete solution, update B.

Branch-and-Bound

Backtracking for optimization problems

Intelligent backtracking can be also used for optimization problems. Consider a minimization problem.

Notation: for instance I, $opt(I)$ is optimum value on I.

P_0 initial instance of given problem.

1. We will keep track of the best solution value B found so far. Initialize B to be crude upper bound on $opt(I)$.
2. Let P be a subproblem at some stage of exploration.
3. If P is a complete solution, update B.
4. Else quickly/efficiently find a lower bound b on $opt(P)$.
    1. If $b \geq B$ then prune (discard) P
    2. Else explore P further by breaking it into subproblems and recurse on them.

Branch-and-Bound

Backtracking for optimization problems

Intelligent backtracking can be also used for optimization problems.
Consider a minimization problem.
Notation: for instance I, $opt(I)$ is optimum value on I.

P_0 initial instance of given problem.

1. We will keep track of the best solution value B found so far. Initialize B to be crude upper bound on $opt(I)$.
2. Let P be a subproblem at some stage of exploration.
3. If P is a complete solution, update B.
4. Else quickly/efficiently find a lower bound b on $opt(P)$.
    1. If $b \geq B$ then prune (discard) P
    2. Else explore P further by breaking it into subproblems and recurse on them.
5. Output best solution found.

Example: Vertex Cover

Given $G = (V, E)$, find a minimum sized vertex cover in G.

1. Initialize $B = n - 1$.

Example: Vertex Cover

Given $G = (V, E)$, find a minimum sized vertex cover in G.

1. Initialize $B = n - 1$.
2. Pick a vertex u. Branch on u: either choose u or discard it.

Example: Vertex Cover

Given $G = (V, E)$, find a minimum sized vertex cover in G.

1. Initialize $B = n - 1$.
2. Pick a vertex u. Branch on u: either choose u or discard it.
3. **Choose** u: let b_1 be a lower bound on $G_1 = G - u$.
4. If $1 + b_1 < B$, recursively explore G_1 (and update B)

Example: Vertex Cover

Given $G = (V, E)$, find a minimum sized vertex cover in G.

1. Initialize $B = n - 1$.
2. Pick a vertex u. Branch on u: either choose u or discard it.
3. **Choose** u: let b_1 be a lower bound on $G_1 = G - u$.
4. If $1 + b_1 < B$, recursively explore G_1 (and update B)
5. **Dicard** u: let b_2 be a lower bound on $G_2 = G - u - N(u)$ where $N(u)$ is the set of neighbors of u.
6. If $|N(u)| + b_2 < B$, recursively explore G_2 (and update B)

Example: Vertex Cover

Given $G = (V, E)$, find a minimum sized vertex cover in G.

1. Initialize $B = n - 1$.
2. Pick a vertex u. Branch on u: either choose u or discard it.
3. **Choose** u: let b_1 be a lower bound on $G_1 = G - u$.
4. If $1 + b_1 < B$, recursively explore G_1 (and update B)
5. **Dicard** u: let b_2 be a lower bound on $G_2 = G - u - N(u)$ where $N(u)$ is the set of neighbors of u.
6. If $|N(u)| + b_2 < B$, recursively explore G_2 (and update B)
7. Output B.

Example: Vertex Cover

Given $G = (V, E)$, find a minimum sized vertex cover in G.

1. Initialize $B = n - 1$.
2. Pick a vertex u. Branch on u: either choose u or discard it.
3. **Choose** u: let b_1 be a lower bound on $G_1 = G - u$.
4. If $1 + b_1 < B$, recursively explore G_1 (and update B)
5. **Dicard** u: let b_2 be a lower bound on $G_2 = G - u - N(u)$ where $N(u)$ is the set of neighbors of u.
6. If $|N(u)| + b_2 < B$, recursively explore G_2 (and update B)
7. Output B.

How do we compute a lower bound?
One possibility: solve an LP relaxation.

Local Search

Local Search: a simple and broadly applicable heuristic method

1. Start with some arbitrary solution s

Local Search

Local Search: a simple and broadly applicable heuristic method

1. Start with some arbitrary solution s
2. Let $N(s)$ be solutions in the "neighborhood" of s obtained from s via "local" moves/changes

Local Search

Local Search: a simple and broadly applicable heuristic method

1. Start with some arbitrary solution s
2. Let $N(s)$ be solutions in the "neighborhood" of s obtained from s via "local" moves/changes
3. If there is a solution $s' \in N(s)$ that is better than s, move to s' and continue search with s'
4. Else, stop search and output s.

Local Search

Main ingredients in local search:

1. Initial solution.
2. Definition of neighborhood of a solution.
3. Efficient algorithm to find a good solution in the neighborhood.

Example: TSP

TSP: Given a complete graph $G = (V, E)$ with c_{ij} denoting cost of edge (i, j), compute a Hamiltonian cycle/tour of minimum edge cost.

Example: TSP

TSP: Given a complete graph $G = (V, E)$ with c_{ij} denoting cost of edge (i, j), compute a Hamiltonian cycle/tour of minimum edge cost.

2-change local search:

1. Start with an arbitrary tour s_0
2. For a solution s define s' to be a neighbor if s' can be obtained from s by replacing two edges in s with two other edges.

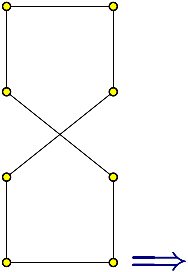
Example: TSP

TSP: Given a complete graph $G = (V, E)$ with c_{ij} denoting cost of edge (i, j), compute a Hamiltonian cycle/tour of minimum edge cost.

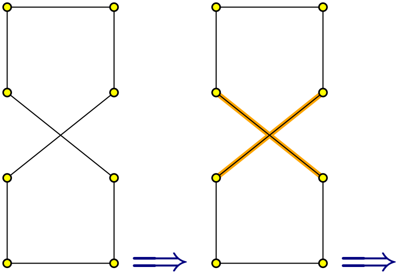
2-change local search:

1. Start with an arbitrary tour s_0
2. For a solution s define s' to be a neighbor if s' can be obtained from s by replacing two edges in s with two other edges.
3. For a solution s at most $O(n^2)$ neighbors and one can try all of them to find an improvement.

TSP: 2-change example



TSP: 2-change example



TSP: 2-change example

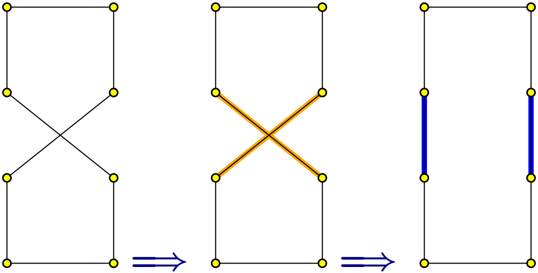


Figure below shows a bad local optimum for **2**-change heuristic...

TSP: 2-change example

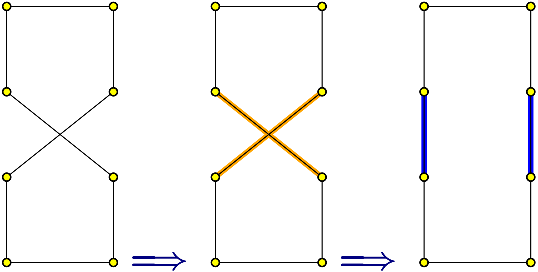
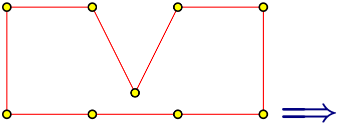


Figure below shows a bad local optimum for **2**-change heuristic...



TSP: 2-change example

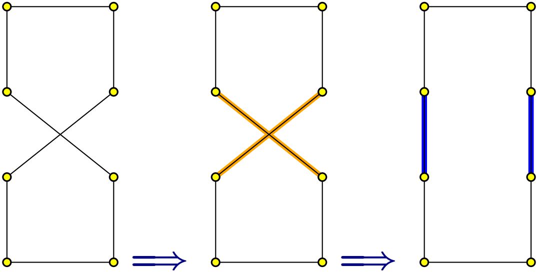
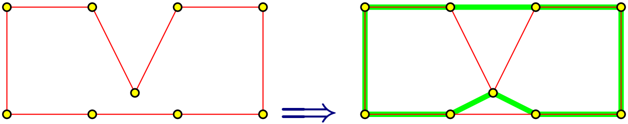
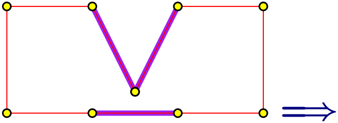


Figure below shows a bad local optimum for **2**-change heuristic...



TSP: 3-change example

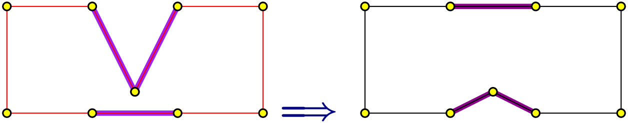
3-change local search: swap $\mathbf{3}$ edges out.



Neighborhood of s has now increased to a size of $\Omega(n^3)$

TSP: 3-change example

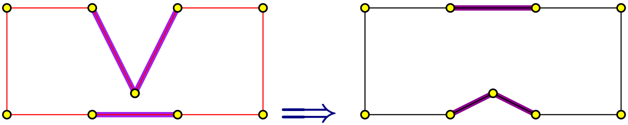
3-change local search: swap $\mathbf{3}$ edges out.



Neighborhood of s has now increased to a size of $\Omega(n^3)$

TSP: 3-change example

3-change local search: swap $\mathbf{3}$ edges out.



Neighborhood of s has now increased to a size of $\Omega(n^3)$

Can define k-change heuristic where k edges are swapped out. Increases neighborhood size and makes each local improvement step less efficient.

Local Search Variants

Local search terminates with a local optimum which may be far from a global optimum. Many variants to improve plain local search.

1. Randomization and restarts: Initial solution may strongly influence the quality of the final solution. Try many random initial solutions.

Local Search Variants

Local search terminates with a local optimum which may be far from a global optimum. Many variants to improve plain local search.

1. Randomization and restarts: Initial solution may strongly influence the quality of the final solution. Try many random initial solutions.
2. Simulated annealing: Allows the algorithm to *move to worse solutions with some probability*. At the beginning this is done more aggressively and then slowly the algorithm converges to plain local search. Controlled by a parameter called "temperature".

Local Search Variants

Local search terminates with a local optimum which may be far from a global optimum. Many variants to improve plain local search.

1. Randomization and restarts: Initial solution may strongly influence the quality of the final solution. Try many random initial solutions.
2. Simulated annealing: Allows the algorithm to *move to worse solutions with some probability*. At the beginning this is done more aggressively and then slowly the algorithm converges to plain local search. Controlled by a parameter called "temperature".
3. Tabu search. Store already visited solutions and do not visit them again (they are "taboo").

Heuristics

Several other heuristics used in practice.

1. Heuristics for solving integer linear programs such as cutting planes, branch-and-cut etc are quite effective. They exploit the geometry of the problem.
2. Heuristics to solve SAT (SAT-solvers) have gained prominence in recent years
3. Genetic algorithms
4. ...

Heuristics

Several other heuristics used in practice.

1. Heuristics for solving integer linear programs such as cutting planes, branch-and-cut etc are quite effective. They exploit the geometry of the problem.
2. Heuristics to solve SAT (SAT-solvers) have gained prominence in recent years
3. Genetic algorithms
4. ...

Heuristics design is somewhat ad hoc and depends heavily on the problem and the instances that are of interest.

Part II

Approximation Algorithms

Approximation algorithms

Consider the following *optimization* problems:

1. **Max Knapsack**: Given knapsack of capacity W, n items each with a value and weight, pack the knapsack with the most profitable subset of items whose weight does not exceed the knapsack capacity.
2. **Min Vertex Cover:** given a graph $G = (V, E)$ find the minimum cardinality vertex cover.
3. **Min Set Cover:** given Set Cover instance, find the smallest number of sets that cover all elements in the universe.
4. **Max Independent Set:** given graph $G = (V, E)$ find maximum independent set.
5. **Min Traveling Salesman Tour:** given a directed graph G with edge costs, find minimum length/cost Hamiltonian cycle in G.

Approximation algorithms

Consider the following *optimization* problems:

1. **Max Knapsack**: Given knapsack of capacity W, n items each with a value and weight, pack the knapsack with the most profitable subset of items whose weight does not exceed the knapsack capacity.
2. **Min Vertex Cover:** given a graph $G = (V, E)$ find the minimum cardinality vertex cover.
3. **Min Set Cover:** given Set Cover instance, find the smallest number of sets that cover all elements in the universe.
4. **Max Independent Set:** given graph $G = (V, E)$ find maximum independent set.
5. **Min Traveling Salesman Tour:** given a directed graph G with edge costs, find minimum length/cost Hamiltonian cycle in G.

Solving one in polynomial time implies solving all the others.

Approximation algorithms

However, the problems behave very differently if one wants to solve them *approximately*.

Approximation algorithms

However, the problems behave very differently if one wants to solve them *approximately*.

Informal definition: An approximation algorithm for an optimization problem is an efficient (polynomial-time) algorithm that *guarantees* for every instance a solution of some given quality when compared to an optimal solution.

Some known approximation results

1. **Knapsack**: For every fixed $\epsilon > 0$ there is a polynomial time algorithm that guarantees a solution of quality $(1 - \epsilon)$ times the best solution for the given instance. Hence can get a 0.99-approximation efficiently.

Some known approximation results

1. **Knapsack**: For every fixed $\epsilon > 0$ there is a polynomial time algorithm that guarantees a solution of quality $(1 - \epsilon)$ times the best solution for the given instance. Hence can get a 0.99-approximation efficiently.
2. **Min Vertex Cover**: There is a polynomial time algorithm that guarantees a solution of cost at most 2 times the cost of an optimum solution.

Some known approximation results

1. **Knapsack**: For every fixed $\epsilon > 0$ there is a polynomial time algorithm that guarantees a solution of quality $(1 - \epsilon)$ times the best solution for the given instance. Hence can get a 0.99-approximation efficiently.
2. **Min Vertex Cover**: There is a polynomial time algorithm that guarantees a solution of cost at most 2 times the cost of an optimum solution.
3. **Min Set Cover**: There is a polynomial time algorithm that guarantees a solution of cost at most $(\ln n + 1)$ times the cost of an optimal solution.

Some known approximation results

1. **Knapsack**: For every fixed $\epsilon > 0$ there is a polynomial time algorithm that guarantees a solution of quality $(1 - \epsilon)$ times the best solution for the given instance. Hence can get a 0.99-approximation efficiently.
2. **Min Vertex Cover**: There is a polynomial time algorithm that guarantees a solution of cost at most 2 times the cost of an optimum solution.
3. **Min Set Cover**: There is a polynomial time algorithm that guarantees a solution of cost at most $(\ln n + 1)$ times the cost of an optimal solution.
4. **Max Independent Set**: Unless $P = NP$, for any fixed $\epsilon > 0$, no polynomial time algorithm can give a $n^{1-\epsilon}$ relative approximation. Here n is number of vertices in the graph.

Some known approximation results

1. **Knapsack**: For every fixed $\epsilon > 0$ there is a polynomial time algorithm that guarantees a solution of quality $(1 - \epsilon)$ times the best solution for the given instance. Hence can get a 0.99-approximation efficiently.
2. **Min Vertex Cover**: There is a polynomial time algorithm that guarantees a solution of cost at most 2 times the cost of an optimum solution.
3. **Min Set Cover**: There is a polynomial time algorithm that guarantees a solution of cost at most $(\ln n + 1)$ times the cost of an optimal solution.
4. **Max Independent Set**: Unless $P = NP$, for any fixed $\epsilon > 0$, no polynomial time algorithm can give a $n^{1-\epsilon}$ relative approximation. Here n is number of vertices in the graph.
5. **Min TSP**: No polynomial factor relative approximation possible.

Approximation algorithms

1. Although **NP-Complete** problems are all equivalent with respect to polynomial-time solvability they behave quite differently under approximation (in both theory and practice).

Approximation algorithms

1. Although **NP-Complete** problems are all equivalent with respect to polynomial-time solvability they behave quite differently under approximation (in both theory and practice).
2. Approximation is a useful lens to examine **NP-Complete** problems more closely.

Approximation algorithms

1. Although **NP-Complete** problems are all equivalent with respect to polynomial-time solvability they behave quite differently under approximation (in both theory and practice).
2. Approximation is a useful lens to examine **NP-Complete** problems more closely.
3. Approximation also useful for problems that we can solve efficiently:
    1. We may have other constraints such a space (streaming problems) or time (need linear time or less for very large problems)
    2. Data may be uncertain (online and stochastic problems).

Formal definition of approximation algorithm

An algorithm $\mathcal{A}$ for an optimization problem X is an α-approximation algorithm if the following conditions hold:

- for each instance I of X the algorithm $\mathcal{A}$ correctly outputs a valid solution to I

Formal definition of approximation algorithm

An algorithm $\mathcal{A}$ for an optimization problem X is an α-approximation algorithm if the following conditions hold:

- for each instance I of X the algorithm $\mathcal{A}$ correctly outputs a valid solution to I
- $\mathcal{A}$ is a polynomial-time algorithm

Formal definition of approximation algorithm

An algorithm $\mathcal{A}$ for an optimization problem X is an α-approximation algorithm if the following conditions hold:

- for each instance I of X the algorithm $\mathcal{A}$ correctly outputs a valid solution to I
- $\mathcal{A}$ is a polynomial-time algorithm
- Let $OPT(I)$ and $\mathcal{A}(I)$ denote the values of an optimum solution and the solution output by $\mathcal{A}$ on instances I.

Formal definition of approximation algorithm

An algorithm $\mathcal{A}$ for an optimization problem X is an α-approximation algorithm if the following conditions hold:

- for each instance I of X the algorithm $\mathcal{A}$ correctly outputs a valid solution to I
- $\mathcal{A}$ is a polynomial-time algorithm
- Let $OPT(I)$ and $\mathcal{A}(I)$ denote the values of an optimum solution and the solution output by $\mathcal{A}$ on instances I. Then
  - If X is a minimization problem: $\mathcal{A}(I)/OPT(I) \leq \alpha$
  - If X is a maximization problem: $OPT(I)/\mathcal{A}(I) \leq \alpha$

Formal definition of approximation algorithm

An algorithm $\mathcal{A}$ for an optimization problem X is an α-approximation algorithm if the following conditions hold:

- for each instance I of X the algorithm $\mathcal{A}$ correctly outputs a valid solution to I
- $\mathcal{A}$ is a polynomial-time algorithm
- Let $OPT(I)$ and $\mathcal{A}(I)$ denote the values of an optimum solution and the solution output by $\mathcal{A}$ on instances I. Then
  - If X is a minimization problem: $\mathcal{A}(I)/OPT(I) \leq \alpha$
  - If X is a maximization problem: $OPT(I)/\mathcal{A}(I) \leq \alpha$

Definition ensures that $\alpha \geq 1$

Formal definition of approximation algorithm

An algorithm $\mathcal{A}$ for an optimization problem X is an α-approximation algorithm if the following conditions hold:

- for each instance I of X the algorithm $\mathcal{A}$ correctly outputs a valid solution to I
- $\mathcal{A}$ is a polynomial-time algorithm
- Let $OPT(I)$ and $\mathcal{A}(I)$ denote the values of an optimum solution and the solution output by $\mathcal{A}$ on instances I. Then
  - If X is a minimization problem: $\mathcal{A}(I)/OPT(I) \leq \alpha$
  - If X is a maximization problem: $OPT(I)/\mathcal{A}(I) \leq \alpha$

Definition ensures that $\alpha \geq 1$

To be formal we need to say $\alpha(n)$ where $n = |I|$ since in some cases the *approximation ratio* depends on the size of the instance.

Formal definition of approximation algorithm

Unfortunately notation is not consistently used. Some times people use the following convention:

- If X is a minimization problem then $\mathcal{A}(I)/OPT(I) \leq \alpha$ and here $\alpha \geq 1$.
- If X is a maximization problem then $\mathcal{A}(I)/OPT(I) \geq \alpha$ and here $\alpha \leq 1$.

Usually clear from the context.

Relative vs Additive

We defined approximation ratio in a relative sense. Some times it makes sense to ask for an additive approximation. For instance in continuous optimization such as linear/convex optimization we talk about ϵ-error where we want a solution I such that $|\mathcal{A}(I) - OPT(I)| \leq \epsilon$.

Relative vs Additive

We defined approximation ratio in a relative sense. Some times it makes sense to ask for an additive approximation. For instance in continuous optimization such as linear/convex optimization we talk about ϵ-error where we want a solution I such that $|\mathcal{A}(I) - OPT(I)| \leq \epsilon$.

For most NP-Hard optimization problems it is not hard to show that one cannot obtain a good additive approximation in polynomial time unless $P = NP$

Relative vs Additive

We defined approximation ratio in a relative sense. Some times it makes sense to ask for an additive approximation. For instance in continuous optimization such as linear/convex optimization we talk about ϵ-error where we want a solution I such that $|\mathcal{A}(I) - OPT(I)| \leq \epsilon$.

For most NP-Hard optimization problems it is not hard to show that one cannot obtain a good additive approximation in polynomial time unless $P = NP$ and hence relative approximation is a more robust and useful notion.

Part III

Approximation for Vertex Cover

Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:

1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.

Problem (**Vertex Cover**)

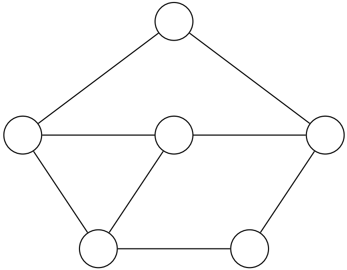
Input: *A graph* G

Goal: *Find a vertex cover of minimum size in* G

Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:

1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



Problem (Vertex Cover)

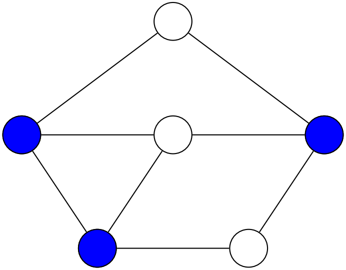
Input: *A graph* G

Goal: *Find a vertex cover of minimum size in* G

Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:

1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



Problem (**Vertex Cover**)

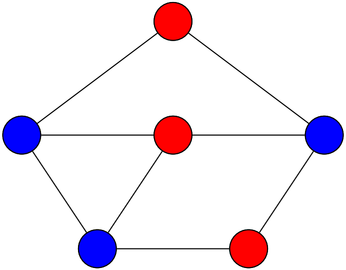
Input: *A graph G*

Goal: *Find a vertex cover of minimum size in G*

Vertex Cover

Given a graph $G = (V, E)$, a set of vertices S is:

1. A **vertex cover** if every $e \in E$ has at least one endpoint in S.



Problem (Vertex Cover)

Input: *A graph* G

Goal: *Find a vertex cover of minimum size in* G

Greedy Algorithm

Greedy(G):

```
Initialize S to be ∅
While there are edges in G do
    Select vertex v with
```

Greedy Algorithm

```
Greedy(G):
        Initialize S to be ∅
        While there are edges in G do
            Select vertex v with maximum degree
            S ← S ∪ {v}
            G ← G − v
        endWhile
        Output S
```

Greedy Algorithm

```
Greedy(G):
        Initialize S to be ∅
        While there are edges in G do
            Select vertex v with maximum degree
            S ← S ∪ {v}
            G ← G − v
        endWhile
        Output S
```

Theorem

$|S| \leq (\ln n + 1)OPT$ where OPT is the value of an optimum set. Here n is number of nodes in G.

Greedy Algorithm

```
Greedy(G):
        Initialize S to be ∅
        While there are edges in G do
            Select vertex v with maximum degree
            S ← S ∪ {v}
            G ← G − v
        endWhile
        Output S
```

Theorem

$|S| \leq (\ln n + 1)OPT$ where OPT is the value of an optimum set. Here n is number of nodes in G.

Theorem

There is an infinite family of graphs where the solution S output by Greedy is $\Omega(\ln n)OPT$.

Matching Heuristic

Relation between matching and vertex cover

Lemma

Let $M \subset E$ *be a matching in graph* $G = (V, E)$*, then* $OPT \geq |M|$ *where* OPT *is the size of minimum vertex cover.*

Matching Heuristic

Relation between matching and vertex cover

Lemma

Let $M \subset E$ be a matching in graph $G = (V, E)$, then $OPT \geq |M|$ where OPT is the size of minimum vertex cover.

```
MatchingHeuristic(G):
        Find a maximal matching M in G
        S is the set of both end points of edges in M
        Output S
```

Matching Heuristic

Relation between matching and vertex cover

Lemma

Let $M \subset E$ be a matching in graph $G = (V, E)$, then $OPT \geq |M|$ where OPT is the size of minimum vertex cover.

MatchingHeuristic(G):
    Find a maximal matching M in G
    S is the set of both end points of edges in M
    Output S

Lemma

S is a feasible vertex cover.

Matching Heuristic

Relation between matching and vertex cover

Lemma

Let $M \subset E$ be a matching in graph $G = (V, E)$, then $OPT \geq |M|$ where OPT is the size of minimum vertex cover.

MatchingHeuristic(G):
    Find a maximal matching M in G
    S is the set of both end points of edges in M
    Output S

Lemma

S is a feasible vertex cover.

Analysis: $|S| = 2|M| \leq 2OPT$. Algorithm is a 2-approximation.

Vertex Cover: LP Relaxation based approach

Write (weighted) vertex cover problem as an integer linear program

$$
\begin{array}{lll}
\text{Minimize} & \sum_{v \in V} w_v x_v & \\
\text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\
 & x_v \in \{0, 1\} & \text{for each } v \in V
\end{array}
$$

Vertex Cover: LP Relaxation based approach

Write (weighted) vertex cover problem as an integer linear program

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \in \{0,1\} & \text{for each } v \in V \end{array}$$

Relax integer program to a linear program

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \geq 0 & \text{for each } v \in V \end{array}$$

Vertex Cover: LP Relaxation based approach

Write (weighted) vertex cover problem as an integer linear program

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \in \{0, 1\} & \text{for each } v \in V \end{array}$$

Relax integer program to a linear program

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \geq 0 & \text{for each } v \in V \end{array}$$

Can solve linear program in polynomial time.
Let x^* be an optimum solution to the linear program.

Vertex Cover: LP Relaxation based approach

Write (weighted) vertex cover problem as an integer linear program

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \in \{0, 1\} & \text{for each } v \in V \end{array}$$

Relax integer program to a linear program

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \geq 0 & \text{for each } v \in V \end{array}$$

Can solve linear program in polynomial time.

Let x^* be an optimum solution to the linear program.

Lemma

$OPT \geq \sum_v w_v x_v^*$.

Vertex Cover: Rounding fractional solution

LP Relaxation

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \geq 0 & \text{for each } v \in V \end{array}$$

Let x^* be an optimum solution to the linear program.
Rounding: $S = \{v \mid x^*_v \geq 1/2\}$. Output S.

Vertex Cover: Rounding fractional solution

LP Relaxation

$$
\begin{array}{lll}
\text{Minimize} & \sum_{v \in V} w_v x_v & \\
\text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\
 & x_v \geq 0 & \text{for each } v \in V
\end{array}
$$

Let x^* be an optimum solution to the linear program.
Rounding: $S = \{v \mid x^*_v \geq 1/2\}$. Output S.

Lemma

S is a feasible vertex cover for the given graph.

Vertex Cover: Rounding fractional solution

LP Relaxation

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \geq 0 & \text{for each } v \in V \end{array}$$

Let x^* be an optimum solution to the linear program.
Rounding: $S = \{v \mid x^*_v \geq 1/2\}$. Output S.

Lemma

S is a feasible vertex cover for the given graph.

Lemma

$w(S) \leq$

Vertex Cover: Rounding fractional solution

LP Relaxation

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \geq 0 & \text{for each } v \in V \end{array}$$

Let x^* be an optimum solution to the linear program.
Rounding: $S = \{v \mid x_v^* \geq 1/2\}$. Output S.

Lemma

S is a feasible vertex cover for the given graph.

Lemma

$w(S) \leq 2 \sum_v w_v x_v^* \leq$

Vertex Cover: Rounding fractional solution

LP Relaxation

$$\begin{array}{lll} \text{Minimize} & \sum_{v \in V} w_v x_v & \\ \text{subject to} & x_u + x_v \geq 1 & \text{for each } uv \in E \\ & x_v \geq 0 & \text{for each } v \in V \end{array}$$

Let x^* be an optimum solution to the linear program.
Rounding: $S = \{v \mid x^*_v \geq 1/2\}$. Output S.

Lemma

S is a feasible vertex cover for the given graph.

Lemma

$w(S) \leq 2\sum_v w_v x^*_v \leq 2OPT$.

Set Cover and Vertex Cover

Theorem

Greedy gives $(\ln n + 1)$*-approximation for Set Cover where* n *is number of elements.*

Set Cover and Vertex Cover

Theorem

Greedy gives $(\ln n + 1)$*-approximation for Set Cover where* n *is number of elements.*

Theorem

Unless $P = NP$ *no* $(\ln n + \epsilon)$*-approximation for Set Cover for* $\epsilon < 1$.

Set Cover and Vertex Cover

Theorem

Greedy gives $(\ln n + 1)$-approximation for Set Cover where n is number of elements.

Theorem

Unless $P = NP$ no $(\ln n + \epsilon)$-approximation for Set Cover for $\epsilon < 1$.

2-approximation is best known for Vertex Cover.

Theorem

Unless $P = NP$ no 1.36-approximation for Vertex Cover.

Set Cover and Vertex Cover

Theorem

Greedy gives $(\ln n + 1)$-approximation for Set Cover where n is number of elements.

Theorem

Unless $P = NP$ no $(\ln n + \epsilon)$-approximation for Set Cover for $\epsilon < 1$.

2-approximation is best known for Vertex Cover.

Theorem

Unless $P = NP$ no 1.36-approximation for Vertex Cover.

Conjecture: Unless $P = NP$ no $(2 - \epsilon)$-approximation for Vertex Cover for any fixed $\epsilon > 0$.

Independent Set and Vertex Cover

Proposition

Let $G = (V, E)$ *be a graph.* S *is an independent set if and only if* $V \setminus S$ *is a vertex cover.*

Independent Set and Vertex Cover

Proposition

Let $G = (V, E)$ be a graph. S is an independent set if and only if $V \setminus S$ is a vertex cover.

```
IndependentSetHeuristic(G = (V, E)):
        Find (an approximate) vertex cover S in G
        Output V − S
```

Independent Set and Vertex Cover

Proposition

Let $G = (V, E)$ be a graph. S is an independent set if and only if $V \setminus S$ is a vertex cover.

```
IndependentSetHeuristic(G = (V, E)):
        Find (an approximate) vertex cover S in G
        Output V − S
```

Question: Is this a good (approximation) algorithm?

Independent Set and Vertex Cover

Proposition

Let $G = (V, E)$ be a graph. S is an independent set if and only if $V \setminus S$ is a vertex cover.

```
IndependentSetHeuristic(G = (V, E)):
        Find (an approximate) vertex cover S in G
        Output V − S
```

Question: Is this a good (approximation) algorithm?

If S^* is a minimum sized vertex cover then $V - S^*$ is a max independent set.

Independent Set and Vertex Cover

IndependentSetHeuristic$(G = (V, E))$:
```
        Find (an approximate) vertex cover S in G
        Output V − S
```

- Let k be minimum vertex cover size.
- Suppose $k = n/2$ where $n = |V|$
- Then V is a 2-approximation
- But then algorithm will output an **empty** independent set even though there is an independent set of size $n/2$.

Independent Set and Vertex Cover

```
IndependentSetHeuristic(G = (V, E)):
        Find (an approximate) vertex cover S in G
        Output V − S
```

- Let k be minimum vertex cover size.
- Suppose $k = n/2$ where $n = |V|$
- Then V is a 2-approximation
- But then algorithm will output an **empty** independent set even though there is an independent set of size $n/2$.

Example?

Independent Set and Vertex Cover

```
IndependentSetHeuristic(G = (V, E)):
        Find (an approximate) vertex cover S in G
        Output V − S
```

- Let k be minimum vertex cover size.
- Suppose $k = n/2$ where $n = |V|$
- Then V is a 2-approximation
- But then algorithm will output an **empty** independent set even though there is an independent set of size $n/2$.

Example?

Theorem

Unless $P = NP$ no $n^{1-\delta}$-approximation for Independent Set for any fixed $\delta > 0$.