CSE 548: Algorithms

Coping with NP-Completeness

R. Sekar

Coping with NP-Completeness

- Sometimes you are faced with hard problems — problems for which no efficient solutions exist.
- **Step 1:** Try to show that the problem is *NP*-complete
  - This way, you can avoid wasting a lot of time on a fruitless search for an efficient algorithm

Coping with NP-Completeness

- Sometimes you are faced with hard problems — problems for which no efficient solutions exist.
- **Step 1:** Try to show that the problem is *NP*-complete
  - This way, you can avoid wasting a lot of time on a fruitless search for an efficient algorithm
- **Step 2a:** Sometimes, you may be able to say "let us solve a different problem"
  - you may be able leverage some special structure of your problem domain that enables a more efficient solution

Coping with NP-Completeness

- Sometimes you are faced with hard problems — problems for which no efficient solutions exist.
- **Step 1:** Try to show that the problem is *NP*-complete
  - This way, you can avoid wasting a lot of time on a fruitless search for an efficient algorithm
- **Step 2a:** Sometimes, you may be able to say “let us solve a different problem”
  - you may be able leverage some special structure of your problem domain that enables a more efficient solution
- **Step 2b:** Other times, you are stuck with a difficult problem and you need to make the best of it.
  - We discuss different coping strategies in such cases.

Intelligent Exhaustive Search

- Exhaustive search will work for almost any problem

Intelligent Exhaustive Search

- Exhaustive search will work for almost any problem

  Hamiltonian Tour: Consider an edge e.

  - Either $e = (u, v)$ is part of the tour, in which case you can complete the tour by finding a path from u to v in $G - e$.

Intelligent Exhaustive Search

- Exhaustive search will work for almost any problem

  Hamiltonian Tour: Consider an edge e.

  - Either $e = (u, v)$ is part of the tour, in which case you can complete the tour by finding a path from u to v in $G - e$.
  - Or, e is not part of the tour, in which case you can find the tour by searching $G - e$.

Intelligent Exhaustive Search

- Exhaustive search will work for almost any problem

  Hamiltonian Tour: Consider an edge e.

  - Either $e = (u, v)$ is part of the tour, in which case you can complete the tour by finding a path from u to v in $G - e$.
  - Or, e is not part of the tour, in which case you can find the tour by searching $G - e$.

  Either case leads to a recurrence $T(m) = 2T(m-1)$, i.e., $T(m) = O(2^m)$. (Here m is the number of edge in G.)

Intelligent Exhaustive Search

- Exhaustive search will work for almost any problem

  Hamiltonian Tour: Consider an edge e.
  - Either $e = (u, v)$ is part of the tour, in which case you can complete the tour by finding a path from u to v in $G - e$.
  - Or, e is not part of the tour, in which case you can find the tour by searching $G - e$.

  Either case leads to a recurrence $T(m) = 2T(m-1)$, i.e., $T(m) = O(2^m)$. (Here m is the number of edge in G.)

  SAT: Try all 2^n possible truth assignments to the n variables in your formula.

Intelligent Exhaustive Search

- Exhaustive search will work for almost any problem

  Hamiltonian Tour: Consider an edge e.

  - Either $e = (u, v)$ is part of the tour, in which case you can complete the tour by finding a path from u to v in $G - e$.
  - Or, e is not part of the tour, in which case you can find the tour by searching $G - e$.

  Either case leads to a recurrence $T(m) = 2T(m-1)$, i.e., $T(m) = O(2^m)$. (Here m is the number of edge in G.)

  SAT: Try all 2^n possible truth assignments to the n variables in your formula.

- The key point is to be intelligent in the way this search is conducted, so that the algorithm is faster than 2^n in practice.

Backtracking

- Depth-first approach to perform exhaustive search

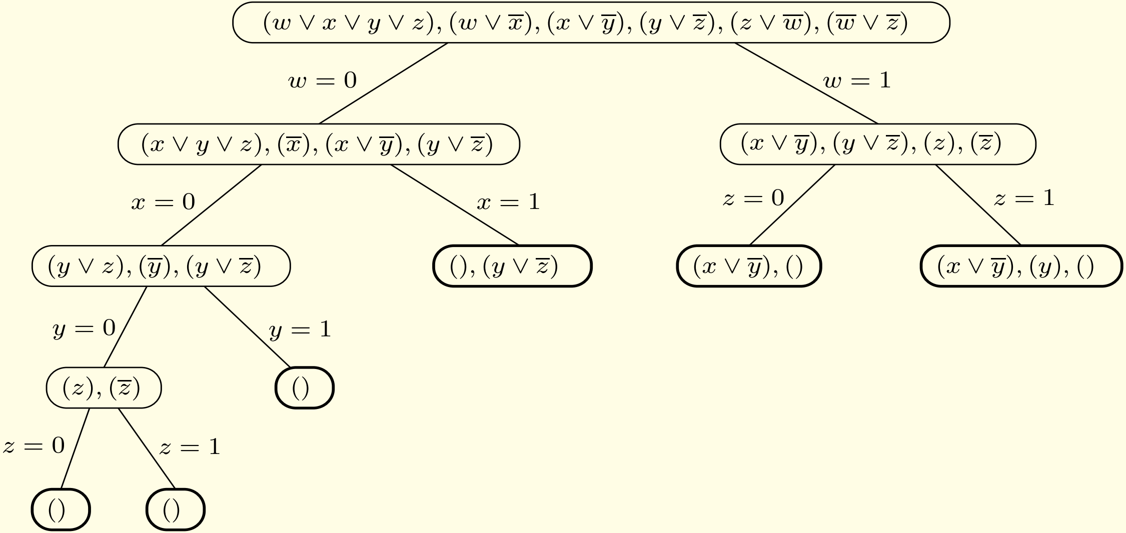
Backtracking

- Depth-first approach to perform exhaustive search
  - In the above example, first try to find a solution that includes e
    - Looking down further, the algorithm will make additional choices of edges to include: $e_1, e_2, \ldots, e_k$
  - Only when all paths that include e fail to be Hamiltonian, we consider the alternative (i.e., Hamiltonian path that doesn't include e)

Backtracking

- Depth-first approach to perform exhaustive search
  - In the above example, first try to find a solution that includes e
    - Looking down further, the algorithm will make additional choices of edges to include: $e_1, e_2, \ldots, e_k$
  - Only when all paths that include e fail to be Hamiltonian, we consider the alternative (i.e., Hamiltonian path that doesn't include e)
- Key goal is to recognize and prune failing paths as quickly as possible.

Backtracking Approach for *SAT*



Backtracking Approach for *SAT*: Complexity

- There are two cases, based on the variable w chosen for branching:

  Case 1: Both w and $\overline{w}$ occur in the formula In this case, both branches are present. Moreover, both w and $\overline{w}$ are eliminated from the formula at this point, so we have the recurrence:

$$T(n) = 2T(n-2) + O(n)$$

Backtracking Approach for *SAT*: Complexity

- There are two cases, based on the variable w chosen for branching:

  Case 1: Both w and $\overline{w}$ occur in the formula In this case, both branches are present. Moreover, both w and $\overline{w}$ are eliminated from the formula at this point, so we have the recurrence:

$$T(n) = 2T(n-2) + O(n)$$

  Case 2: Only one of them is present. In this case, only one of the branches needs exploring, so we have the recurrence

$$T(n) = T(n-1) + O(n)$$

Backtracking Approach for *SAT*: Complexity

- There are two cases, based on the variable w chosen for branching:

  Case 1: Both w and $\overline{w}$ occur in the formula In this case, both branches are present. Moreover, both w and $\overline{w}$ are eliminated from the formula at this point, so we have the recurrence:

$$T(n) = 2T(n-2) + O(n)$$

  Case 2: Only one of them is present. In this case, only one of the branches needs exploring, so we have the recurrence

$$T(n) = T(n-1) + O(n)$$

- Clearly, case 1 will dominate, so let us ignore case 2. Case 1 yields a solution of $O(2^{n/2})$ or $O(1.414^n)$, which is much better than 2^n.

Backtracking Approach for *SAT*: Improvements

- We can improve the worst-case bound by choosing a variable that occurs most times

Backtracking Approach for *SAT*: Improvements

- We can improve the worst-case bound by choosing a variable that occurs most times
  - If it occurs k times, then you have the recurrence

$$T(n) = 2T(n-k)$$

whose solution is $O(2^{n/k})$.

Backtracking Approach for *SAT*: Improvements

- We can improve the worst-case bound by choosing a variable that occurs most times
  - If it occurs k times, then you have the recurrence
    $$T(n) = 2T(n-k)$$
    whose solution is $O(2^{n/k})$.
    - Of course, you won't be able to repeatedly find a variable that occurs k times, so this solution is meaningless in practice — it just goes to show the exponential pruning effect of a frequently occurring variable

Backtracking Approach for *SAT*: Improvements

- We can improve the worst-case bound by choosing a variable that occurs most times
  - If it occurs k times, then you have the recurrence
    $$T(n) = 2T(n-k)$$
    whose solution is $O(2^{n/k})$.
    - Of course, you won't be able to repeatedly find a variable that occurs k times, so this solution is meaningless in practice — it just goes to show the exponential pruning effect of a frequently occurring variable
- Another strategy: pick a clause with fewest number of variables, and pick those variables in sequence.

Backtracking Approach for *SAT*: Improvements

- We can improve the worst-case bound by choosing a variable that occurs most times
  - If it occurs k times, then you have the recurrence

    $$T(n) = 2T(n-k)$$

    whose solution is $O(2^{n/k})$.
    - Of course, you won't be able to repeatedly find a variable that occurs k times, so this solution is meaningless in practice — it just goes to show the exponential pruning effect of a frequently occurring variable
- Another strategy: pick a clause with fewest number of variables, and pick those variables in sequence.
- *Exercise:* Show that the backtracking algorithm solves 2SAT in polynomial time

Branch and Bound

- Generalization of backtracking to support optimization problems

Branch and Bound

- Generalization of backtracking to support optimization problems
- Requires a lower bound on the cost of solutions that may result from a partial solution
  - If the cost is higher than that of a previously encountered solution, then this subproblem need not be explored further.

Branch and Bound

- Generalization of backtracking to support optimization problems
- Requires a lower bound on the cost of solutions that may result from a partial solution
  - If the cost is higher than that of a previously encountered solution, then this subproblem need not be explored further.
- Sometimes, we may rely on estimates of cost rather than strict lower bounds.

Branch and Bound for TSP

- Begin with a vertex a — the goal is to compute a TSP that begins and ends at a.

Branch and Bound for TSP

- Begin with a vertex a — the goal is to compute a TSP that begins and ends at a.
- We begin the search by considering an edge from a to its neighbor x, another edge from x to a neighbor of x, and so on.

Branch and Bound for TSP

- Begin with a vertex a — the goal is to compute a TSP that begins and ends at a.
- We begin the search by considering an edge from a to its neighbor x, another edge from x to a neighbor of x, and so on.
- *Partial solutions* represent a path from a to some vertex b, passing through a set $S \subset V$ of vertices.

Branch and Bound for TSP

- Begin with a vertex a — the goal is to compute a TSP that begins and ends at a.
- We begin the search by considering an edge from a to its neighbor x, another edge from x to a neighbor of x, and so on.
- *Partial solutions* represent a path from a to some vertex b, passing through a set $S \subset V$ of vertices.
- *Completing a partial solution* requires the computation of a low cost path from b to a using only vertices in $V - S$

Lower bound on costs of partial TSP solutions

- To complete the path from b to a, we must incur at least the following costs
  - Cost of going from b to a vertex in $V - S$, i.e, the minimum weight edge from b to a vertex in $V - S$

Lower bound on costs of partial TSP solutions

- To complete the path from b to a, we must incur at least the following costs
  - Cost of going from b to a vertex in $V - S$, i.e, the minimum weight edge from b to a vertex in $V - S$
  - Cost of going from a $V - S$ vertex to a, i.e, the minimum weight edge from a to a vertex in $V - S$

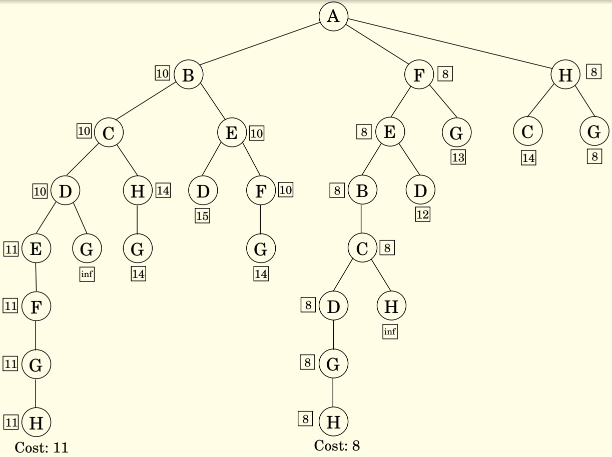
Lower bound on costs of partial TSP solutions

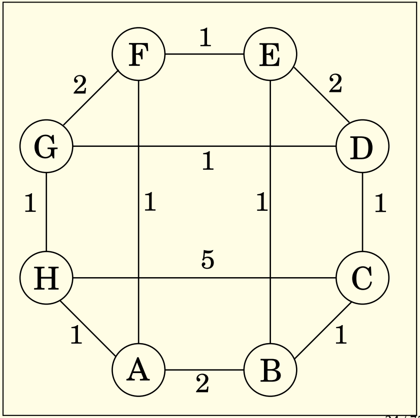
- To complete the path from b to a, we must incur at least the following costs
  - Cost of going from b to a vertex in $V - S$, i.e, the minimum weight edge from b to a vertex in $V - S$
  - Cost of going from a $V - S$ vertex to a, i.e, the minimum weight edge from a to a vertex in $V - S$
  - Minimal cost path in $V - S$ that visits all $v \in V - S$
    - *Note:* Lower bound is the cost of MST for $V - S$

Lower bound on costs of partial TSP solutions

- To complete the path from b to a, we must incur at least the following costs
  - Cost of going from b to a vertex in $V - S$, i.e, the minimum weight edge from b to a vertex in $V - S$
  - Cost of going from a $V - S$ vertex to a, i.e, the minimum weight edge from a to a vertex in $V - S$
  - Minimal cost path in $V - S$ that visits all $v \in V - S$
    - *Note:* Lower bound is the cost of MST for $V - S$
- By adding the above three cost components, we arrive at a lower bound on solutions derivable from a partial solution.

Illustration of Branch-and Bound for TSP





Approximation Algorithms

- Relax optimality requirement: permit *approximate* solutions
  - Solutions that are within a certain distance from optimum

Approximation Algorithms

- Relax optimality requirement: permit *approximate* solutions
  - Solutions that are within a certain distance from optimum
- *Not heuristics:* Approximate algorithms *guarantee* that solutions are within a certain distance from optimal
  - Differs from heuristics that can sometimes return very bad solutions.

Approximation Algorithms

- Relax optimality requirement: permit *approximate* solutions
  - Solutions that are within a certain distance from optimum
- *Not heuristics:* Approximate algorithms *guarantee* that solutions are within a certain distance from optimal
  - Differs from heuristics that can sometimes return very bad solutions.
- How to define "distance from optimal?"

Approximation Algorithms

- Relax optimality requirement: permit *approximate* solutions
  - Solutions that are within a certain distance from optimum
- *Not heuristics:* Approximate algorithms *guarantee* that solutions are within a certain distance from optimal
  - Differs from heuristics that can sometimes return very bad solutions.
- How to define "distance from optimal?"

  Additive: Optimal solution S_O and the Solution S_A returned by approximation algorithm differ only by a constant.
  - Quality of approximation is extremely good, but unfortunately, most problems don't admit such approximations

Approximation Algorithms

- Relax optimality requirement: permit *approximate* solutions
  - Solutions that are within a certain distance from optimum
- *Not heuristics:* Approximate algorithms *guarantee* that solutions are within a certain distance from optimal
  - Differs from heuristics that can sometimes return very bad solutions.
- How to define "distance from optimal?"

  Additive: Optimal solution S_O and the Solution S_A returned by approximation algorithm differ only by a constant.
  - Quality of approximation is extremely good, but unfortunately, most problems don't admit such approximations

  Factor: S_O and S_A are related by a factor.
  - Most known approximation algorithms fall into this category.

Approximation Factors

Constant: $S_A \leq kS_O$ for some fixed constant k.

- *Examples:* Vertex cover, Facility location, ...

Approximation Factors

Constant: $S_A \leq kS_O$ for some fixed constant k.

- *Examples:* Vertex cover, Facility location, ...

Logarithmic: $S_A \leq O(\log^k n) \cdot S_O$.

- *Examples:* Set cover, dominating set, ...

Approximation Factors

Constant: $S_A \leq kS_O$ for some fixed constant k.

- *Examples:* Vertex cover, Facility location, ...

Logarithmic: $S_A \leq O(\log^k n) \cdot S_O$.

- *Examples:* Set cover, dominating set, ...

Polynomial: $S_A \leq O(n^k) \cdot S_O$.

- *Examples:* Max Clique, Independent set, graph coloring, ...

Approximation Factors

Constant: $S_A \leq kS_O$ for some fixed constant k.

- *Examples:* Vertex cover, Facility location, ...

Logarithmic: $S_A \leq O(\log^k n) \cdot S_O$.

- *Examples:* Set cover, dominating set, ...

Polynomial: $S_A \leq O(n^k) \cdot S_O$.

- *Examples:* Max Clique, Independent set, graph coloring, ...

PTAS: $S_A \leq (1 + \epsilon) \cdot S_O$ for any $\epsilon > 0$.

("Polynomial-time approximation scheme")

Approximation Factors

Constant: $S_A \leq kS_O$ for some fixed constant k.

- *Examples:* Vertex cover, Facility location, ...

Logarithmic: $S_A \leq O(\log^k n) \cdot S_O$.

- *Examples:* Set cover, dominating set, ...

Polynomial: $S_A \leq O(n^k) \cdot S_O$.

- *Examples:* Max Clique, Independent set, graph coloring, ...

PTAS: $S_A \leq (1 + \epsilon) \cdot S_O$ for any $\epsilon > 0$.
("Polynomial-time approximation scheme")

FPTAS: PTAS with runtime $O(\epsilon^{-k})$ for some k. ("Fully PTAS")

- *Examples:* Knapsack, Bin-packing, Euclidean TSP, ...

Bin Packing

Problem

Pack objects of different weight into bins that have a fixed capacity in such a way that minimizes bins used.

Bin Packing

Problem

Pack objects of different weight into bins that have a fixed capacity in such a way that minimizes bins used.

- Obvious similarity to Knapsack
- Bin-packing is *NP*-hard
- Very good (and often very simple) approximation algorithms exist

First-fit Algorithm

A simple, greedy algorithm

$FirstFit(x[1..n])$

for $i = 1$ **to** n **do**
  Put $x[i]$ into the first open bin large enough to hold it

First-fit Algorithm

A simple, greedy algorithm

FirstFit$(x[1..n])$

for $i = 1$ **to** n **do**
  Put $x[i]$ into the first open bin large enough to hold it

Theorem

All open bins, except possibly one, are more than half-full

First-fit Algorithm

A simple, greedy algorithm

FirstFit$(x[1..n])$

for $i = 1$ **to** n **do**
  Put $x[i]$ into the first open bin large enough to hold it

Theorem

All open bins, except possibly one, are more than half-full

Proof: Suppose that there are two bins b and b' that are less than half-full. Then, items in b' would have fitted into b, and so the FF algorithm would never have opened the bin b' — a contradiction. ■

First-fit Algorithm

A simple, greedy algorithm

FirstFit$(x[1..n])$

for $i = 1$ **to** n **do**
  Put $x[i]$ into the first open bin large enough to hold it

Theorem

All open bins, except possibly one, are more than half-full

Proof: Suppose that there are two bins b and b' that are less than half-full. Then, items in b' would have fitted into b, and so the FF algorithm would never have opened the bin b' — a contradiction. ■

Theorem

First-fit is optimal within a factor of 2: specifically, $S_A < 2S_O + 1$.

Best-Fit Algorithm

- Another simple, greedy algorithm
- Instead of using the first bin that will can hold $x[i]$, use the open bin whose remaining capacity is closest to $x[i]$
  - Prefers to keep bins close to full.
- Factor-2 optimality can established easily.

Other algorithms for Bin-packing

- *First-fit decreasing* strategy first sorts the items so that $x[i] \geq x[i+1]$ and then runs first-fit.

Other algorithms for Bin-packing

- *First-fit decreasing* strategy first sorts the items so that $x[i] \geq x[i+1]$ and then runs first-fit.
- *Best-fit decreasing* strategy first sorts the items so that $x[i] \geq x[i+1]$ and then runs best-fit.

Other algorithms for Bin-packing

- *First-fit decreasing* strategy first sorts the items so that $x[i] \geq x[i+1]$ and then runs first-fit.
- *Best-fit decreasing* strategy first sorts the items so that $x[i] \geq x[i+1]$ and then runs best-fit.
- Both FFD and BFD achieve approximation factors of $11/9S_O + 6/9$.

Set Cover

Problem

Given a collection $S_1, \ldots, S_m$ of subsets of B, find a minimum collection $S_{i_1}, \ldots, S_{i_k}$ such that $\bigcup_{j=1}^{k} S_{i_j} = B$

Set Cover

Problem

Given a collection $S_1, \ldots, S_m$ of subsets of B, find a minimum collection $S_{i_1}, \ldots, S_{i_k}$ such that $\bigcup_{j=1}^{k} S_{i_j} = B$

Greedy Set Cover Algorithm

$GSC(S, B)$

$cover = \emptyset$; $covered = \emptyset$

while $covered \neq B$ **do**

Let *new* be the set in $S - cover$ containing

the maximum number of elements of $B - covered$

add *new* to *cover*; $covered = covered \cup new$

return *cover*

Analysis of Greedy Set Cover

Theorem

Greedy set cover is approximate with a factor of $\ln n$, *where* $n = |B|$

Analysis of Greedy Set Cover

Theorem

Greedy set cover is approximate with a factor of $\ln n$, *where* $n = |B|$

Proof:

- Let k be the size of optimal cover, and n_t be the number of elements left uncovered after t steps of *GSC*

Analysis of Greedy Set Cover

Theorem

Greedy set cover is approximate with a factor of $\ln n$*, where* $n = |B|$

Proof:

- Let k be the size of optimal cover, and n_t be the number of elements left uncovered after t steps of *GSC*
- These n_t elements are covered by k sets in optimal cover $\Rightarrow$ these k sets must cover n_t/k uncovered elements on average.

Analysis of Greedy Set Cover

Theorem

Greedy set cover is approximate with a factor of $\ln n$*, where* $n = |B|$

Proof:

- Let k be the size of optimal cover, and n_t be the number of elements left uncovered after t steps of *GSC*
- These n_t elements are covered by k sets in optimal cover $\Rightarrow$ these k sets must cover n_t/k uncovered elements on average.
- Thus, *GSC* will find at least one set that covers n_t/k elements.

Analysis of Greedy Set Cover

Theorem

Greedy set cover is approximate with a factor of $\ln n$, *where* $n = |B|$

Proof:

- Let k be the size of optimal cover, and n_t be the number of elements left uncovered after t steps of GSC
- These n_t elements are covered by k sets in optimal cover $\Rightarrow$ these k sets must cover n_t/k uncovered elements on average.
- Thus, GSC will find at least one set that covers n_t/k elements.
- This yields the recurrence for bounding uncovered elements: $U(t+1) = n_t - n_t/k = n_t(1 - 1/k) = U(t)(1 - 1/k)$

Analysis of Greedy Set Cover

Theorem

Greedy set cover is approximate with a factor of $\ln n$*, where* $n = |B|$

Proof:

- Let k be the size of optimal cover, and n_t be the number of elements left uncovered after t steps of GSC
- These n_t elements are covered by k sets in optimal cover $\Rightarrow$ these k sets must cover n_t/k uncovered elements on average.
- Thus, GSC will find at least one set that covers n_t/k elements.
- This yields the recurrence for bounding uncovered elements: $U(t+1) = n_t - n_t/k = n_t(1 - 1/k) = U(t)(1 - 1/k)$
- The solution to recurrence is $n(1 - 1/k)^t < ne^{-t/k}$

Analysis of Greedy Set Cover

Theorem

Greedy set cover is approximate with a factor of $\ln n$*, where* $n = |B|$

Proof:

- Let k be the size of optimal cover, and n_t be the number of elements left uncovered after t steps of GSC
- These n_t elements are covered by k sets in optimal cover $\Rightarrow$ these k sets must cover n_t/k uncovered elements on average.
- Thus, GSC will find at least one set that covers n_t/k elements.
- This yields the recurrence for bounding uncovered elements: $U(t+1) = n_t - n_t/k = n_t(1 - 1/k) = U(t)(1 - 1/k)$
- The solution to recurrence is $n(1 - 1/k)^t < ne^{-t/k}$
- Thus, after $t = k \ln n$ steps, less than 1 (i.e., no) elements uncovered

Analysis of Greedy Set Cover

Theorem

Greedy set cover is approximate with a factor of $\ln n$*, where* $n = |B|$

Proof:

- Let k be the size of optimal cover, and n_t be the number of elements left uncovered after t steps of *GSC*
- These n_t elements are covered by k sets in optimal cover $\Rightarrow$ these k sets must cover n_t/k uncovered elements on average.
- Thus, *GSC* will find at least one set that covers n_t/k elements.
- This yields the recurrence for bounding uncovered elements: $U(t+1) = n_t - n_t/k = n_t(1 - 1/k) = U(t)(1 - 1/k)$
- The solution to recurrence is $n(1 - 1/k)^t < ne^{-t/k}$
- Thus, after $t = k \ln n$ steps, less than 1 (i.e., no) elements uncovered
- Thus, GSC computes a cover at most $\ln n$ times the optimal cover.

Vertex Cover

- Note that a vertex cover is a set cover for $(\mathcal{S}, E)$, where

$$\mathcal{S} = \{\{(v,u)|(v,u) \in E\}|v \in V\}$$

  - i.e., $\mathcal{S}$ contains a set for each vertex; this set lists all edges incident on v

Vertex Cover

- Note that a vertex cover is a set cover for $(\mathcal{S}, E)$, where

$$\mathcal{S} = \{\{(v, u) | (v, u) \in E\} | v \in V\}$$

  - i.e., $\mathcal{S}$ contains a set for each vertex; this set lists all edges incident on v
- Thus *GSC* is an approximate algorithm for vertex cover.

Vertex Cover

- Note that a vertex cover is a set cover for $(\mathcal{S}, E)$, where
$$\mathcal{S} = \{\{(v,u) | (v,u) \in E\} | v \in V\}$$
  - i.e., $\mathcal{S}$ contains a set for each vertex; this set lists all edges incident on v
- Thus *GSC* is an approximate algorithm for vertex cover.
- But $\ln n$ is not a factor to be thrilled about — can we do better?
  - Actually, we can do much better! That too with a very simple algorithm.

Vertex Cover

Consider any edge (u, v).

- Either u or v must belong to any vertex cover.
- If we accept $S_A = 2S_O$, we can avoid the guesswork by simply picking both vertices!

Vertex Cover

Consider any edge (u, v).

- Either u or v must belong to any vertex cover.
- If we accept $S_A = 2S_O$, we can avoid the guesswork by simply picking both vertices!

Approximate Vertex Cover Algorithm

$AVC(G = (V, E))$

$C = \emptyset$

while G is not empty

pick any $(u, v) \in E$

$C = C \cup \{u, v\}$

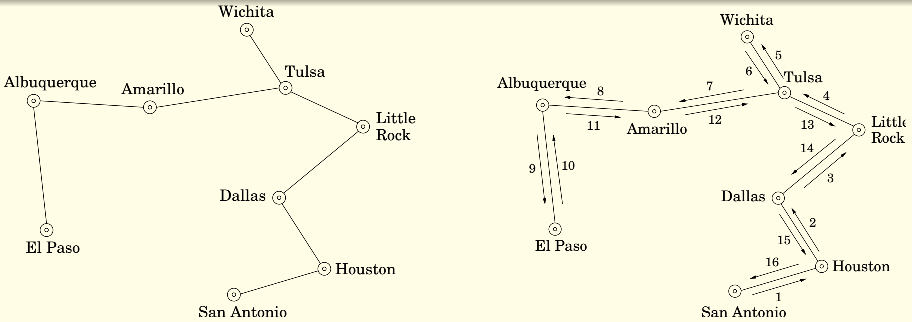
$G = G - \{u, v\}$

return C

Euclidean TSP

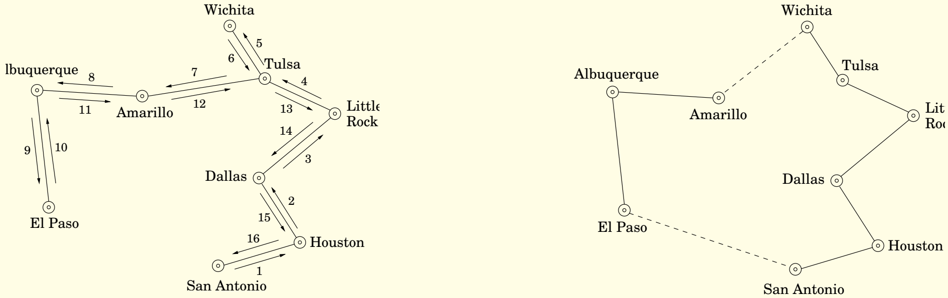
- Our starting point is once again the MST
- Note that no TSP solution can be smaller than MST
  - Deleting an edge from TSP solution yields a spanning tree
- **Simple algorithm:**
  - Start with the MST

Approximating Euclidean TSP: An Illustration



- Start with the MST
- Make a tour that uses each MST edge twice (forward and backward)
  - This tour is like TSP in ending at the starting node, and differs from TSP by visiting some vertices and edges twice

Approximating Euclidean TSP: An Illustration (2)



- Avoid revisits by short-circuiting to next unvisited vertex
- By triangle inequality, short-circuit distance can only be less than the distance following MST edges.
  - Thus, tour length less than 2xMST, i.e., approximate within a factor 2.

Knapsack

$Knap01(w, v, n, W)$

$V = \sum_{j=0}^{n} v[j]$

$K[j, 0] = 0, \forall 0 \leq j \leq V$

for $j = 1$ to n **do**

  for $v = 1$ to V **do**

    if $v[j] > v$ **then** $K[j, v] = K[j-1, v]$

    else $K[j, v] = min(K[j-1, v], K[j-1, v-v[j]] + w[j])$

return maximum v such that $K[n, v] \leq W$

- Computes minimum weight of knapsack for a given value.
- Iterates over all possible items and all possible values: $O(nV)$
  - we derive a polynomial time approximate algorithm from this

FPTAS for 0-1 Knapsack

$Knap01FPTAS(w, v, n, W, \epsilon)$

$$v'_i = \left\lfloor \frac{v_i}{max_{1 \le j \le n} v_j} \cdot \frac{n}{\epsilon} \right\rfloor, \text{ for } 1 \le i \le n$$

$$Knap01(w, v', n, W)$$

- Rescaling consists of two steps:
  - Express value of each item relative to the most valuable item
    - If we worked with real values, this step won't change the optimal solution
  - Multiply relative values by a factor n/ϵ to get an integer
- Floor operation introduces an error ≤ 1 in v'_i (e.g., $\lfloor 3.99 \rfloor = 3$)
- Error in $Knap01$ output $=$ error in $\sum v'_i$, which is at most $n \cdot 1$
- We scale each v'_i by n/ϵ, so relative error is $n/(n/\epsilon) = \epsilon$

FPTAS for 0-1 Knapsack: Runtime

$Knap01FPTAS(w, v, n, W, \epsilon)$

$$v_i' = \left\lfloor \frac{v_i}{max_{1 \le j \le n}\ v_j} \cdot \frac{n}{\epsilon} \right\rfloor, \text{ for } 1 \le i \le n$$

$$Knap01(w, v', n, W)$$

- Note that we are using *Knap*01 with rescaled values, so the complexity is $O(nV')$.
- Note: $V' = \sum_1^n v_i' \le n \cdot max_{1 \le j \le n}\ v_j'$
- It is easy to see from definition of v_i' that $max_{1 \le j \le n}\ v_j' = n/\epsilon$. Substituting this into the above equation yields a complexity of:

$$O(nV') \le O(n(n \cdot max_{1 \le i \le n}\ v_i')) = O(n(n \cdot (n/\epsilon))) = O(n^3/\epsilon)$$

- By varying ϵ, we can trade off accuracy against runtime.