COMP 6651 / Winter 2022 - B. Jaumard

Lecture 7 - Part II: Approximation Algorithms

February 18, 2022

Outline

Outline

Outline

Outline

Outline

Performance ratios for Approximation Algorithms

Suppose that we are working on an optimization problem in which each potential solution has a cost, and we wish to find a near-optimal solution.

- An optimal solution may be defined as one solution with max cost, or with min cost.
- An algorithm has an **approximation ratio** of $\rho(n)$, if for any input size n:

$$\max\left\{\frac{C}{C^\star}, \frac{C^\star}{C}\right\} \leq \rho(n)$$

- We call such algorithm a $\rho(n)$**-approximation algorithm**.

Performance ratios for Approximation Algorithms

- Max problem: $0 \leq C \leq C^*$. C^*/C gives the factor by which the cost of an optimal solution is larger than the cost of the **approximate** solution
- Min problem: $0 \leq C^* \leq C$. C/C^* gives the factor by which the cost of the **approximate** solution is larger than the optimal solution.
- The **approximate** ratio is never less than 1 (note that $C/C^* < 1$ implies $C^*/C > 1$).
- A **1-approximation** algorithm produces an **optimal** solution.

Performance ratios for Approximation Algorithms

- Some NP-complete problems allow polynomial-time approximation algorithms that can achieve increasingly smaller approximation ratios by using more and more computation time.
- That is, there is a trade-off between computation time and the quality of the approximation.

An **approximation scheme** for an optimization problem :

- takes as input:
  - not only an instance of the problem,
  - but also $\varepsilon > 0$
- $(1 + \varepsilon)$-approximation algorithm scheme.
- The running time of a polynomial-time approximation scheme can increase very rapidly as ε decreases.

Not a uniform convention across textbooks

Depending on the reference that is used, you may find a different convention, such as, e.g.,

For each instance I of a problem, let OPT(I) denote the value of an optimal solution to instance I. We say that an algorithm A is an α-approximation algorithm for a problem if, for every instance I, the value of the feasible solution returned by A is within a (multiplicative) factor of α of OPT(I). Equivalently, we say that A is an approximation algorithm with approximation ratio α. For a minimization problem we have $\alpha \geq 1$ and for a maximization problem we have $\alpha \leq 1$.

However, following the book of Cormen *et al.* (reference used for the course) convention is as described in the previous slide, i.e., $1 + \varepsilon$ for both maximization and minimization problem. In this course we will use the convention of the book of Cormen *et al.*

Performance ratios for Approximation Algorithms

- An approximation scheme is a **polynomial-time approximation scheme** if for any fixed $\varepsilon > 0$, the scheme runs in time polynomial in the size n of its input instance.
- Example: $O(n^{2/\varepsilon})$. Ideally, if ε decreases by a **Constant** factor, the running time should not increase by more than **Constant** factor. In other words, we would like the running time to be polynomial in $1/\varepsilon$ as well as in n.
- A **fully polynomial-time approximation**
  - It is an approximation algorithm
  - Its running time is polynomial in both $1/\varepsilon$ and the size n of the input instance
  - Example: $O((1/\varepsilon)^2 n^3)$, any constant-factor decrease in ε comes with a corresponding constant factor increase in the running time.

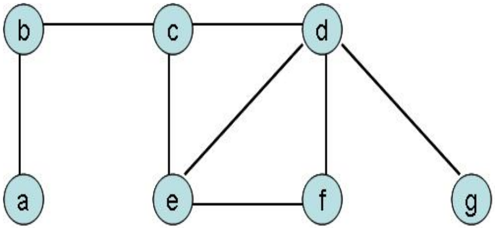
The vertex-cover problem

- The vertex-cover problem is a NP-complete problem.
- A **vertex cover** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $\{u, v\}$ is an edge of G, then either $u \in V'$ or $v \in V'$ (or both).
- The size of a vertex cover is the number of vertices in it.
- **Vertex-cover problem**: Find a vertex cover of minimum size in a given undirected graph.
- Difficult to find an optimal vertex cover in graph G, less difficult to find a vertex cover that is near-optimal.

APPROX-VERTEX-COVER Algorithm (1/6)

APPROX-VERTEX-COVER(G)

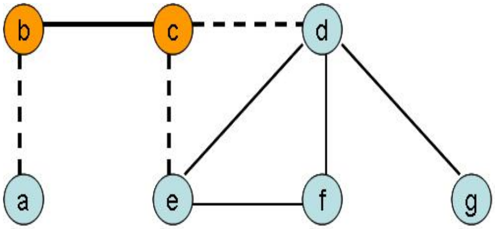
```
1   $C \leftarrow \phi$
2   $E' \leftarrow E[G]$
3   while $E' \neq \phi$
4      do let $(u, v)$ be an arbitrary edge of $E'$
5         $C \leftarrow C \cup \{u, v\}$
6         remove from $E'$ every edge incident on either $u$ or $v$.
7   return C
```



APPROX-VERTEX-COVER Algorithm (2/6)

APPROX-VERTEX-COVER(G)

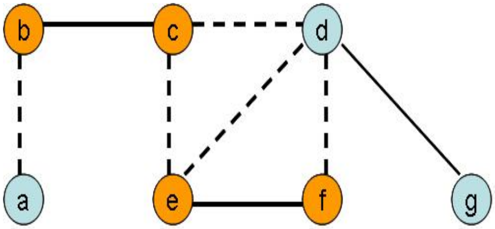
```
1   $C \leftarrow \phi$
2   $E' \leftarrow E[G]$
3   while $E' \neq \phi$
4      do let $(u, v)$ be an arbitrary edge of $E'$
5         $C \leftarrow C \cup \{u, v\}$
6         remove from $E'$ every edge incident on either $u$ or $v$.
7   return C
```



APPROX-VERTEX-COVER Algorithm (3/6)

APPROX-VERTEX-COVER(G)

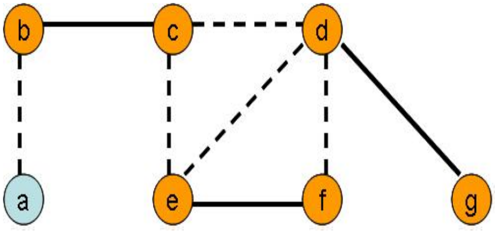
```
1  $C \leftarrow \phi$
2  $E' \leftarrow E[G]$
3  while $E' \neq \phi$
4     do let $(u, v)$ be an arbitrary edge of $E'$
5        $C \leftarrow C \cup \{u, v\}$
6        remove from $E'$ every edge incident on either $u$ or $v$.
7  return C
```



APPROX-VERTEX-COVER Algorithm (4/6)

APPROX-VERTEX-COVER(G)

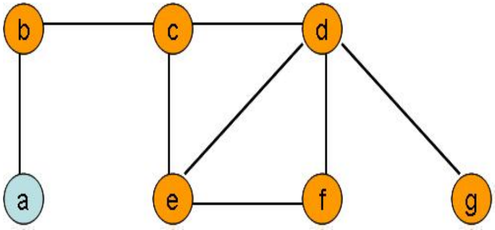
```
1  $C \leftarrow \phi$
2  $E' \leftarrow E[G]$
3  while $E' \neq \phi$
4     do let $(u, v)$ be an arbitrary edge of $E'$
5        $C \leftarrow C \cup \{u, v\}$
6        remove from $E'$ every edge incident on either $u$ or $v$.
7  return C
```



APPROX-VERTEX-COVER Algorithm (5/6)

APPROX-VERTEX-COVER(G)

```
1  $C \leftarrow \phi$
2  $E' \leftarrow E[G]$
3  while $E' \neq \phi$
4     do let $(u, v)$ be an arbitrary edge of $E'$
5        $C \leftarrow C \cup \{u, v\}$
6        remove from $E'$ every edge incident on either $u$ or $v$.
7  return C
```

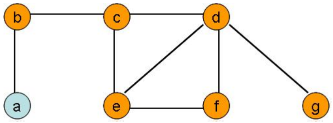


Near Optimal Solution: Running Time $O(V + E)$

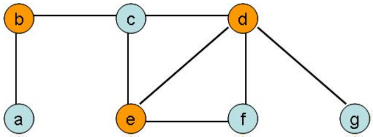
APPROX-VERTEX-COVER Algorithm (6/6)

APPROX-VERTEX-COVER(G)

```
1  $C \leftarrow \phi$
2  $E' \leftarrow E[G]$
3  while $E' \neq \phi$
4      do let $(u, v)$ be an arbitrary edge of $E'$
5          $C \leftarrow C \cup \{u, v\}$
6          remove from $E'$ every edge incident on either $u$ or $v$.
7  return C
```



Near optimal solution



Optimal solution

APPROX-VERTEX-COVER Algorithm: Its approximation ratio (1/2)

Theorem

APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Proof

- APPROX-VERTEX-COVER runs is polynomial time.
- The set C of vertices that is returned by APPROX-VERTEX-COVER is a vertex cover, since the algorithm loops until every edge in $E[G]$ has been covered by some vertex in C.
- Let A denote the set of edges that were picked in line 4 of APPROX-VERTEX-COVER.
- An optimal cover C^* must include at least one endpoint of each edge in A.

APPROX-VERTEX-COVER Algorithm: Its approximation ratio (2/2)

Theorem

APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.

Cont'd

- No two edges in A share an endpoint, since once an edge is picked in line 4, all other edges that are incident on its endpoints are deleted from E' in line 6.
- Thus, no two edges in A are covered by the same vertex from C^*, and we have the lower bound $|C^\star| \geq |A|$
- Each execution of line 4 picks an edge for which neither of its endpoints is already in C, yielding an upper bound (an exact upper bound, in fact) on the size of the vertex cover returned:
  - $|C| = 2|A|$
  - Thus $|C| \leq 2|C^*|$ thereby proving the theorem.

The Traveling-Salesman Problem

- Complete undirected graph $G = (V, E)$
- Nonnegative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$
- Question: **Find a hamiltonian cycle of G with minimum cost.**
- As an extension of our notation, let $c(A)$ denote the total cost of the edges in the subset $A \subseteq E$:

$$c(A) = \sum_{(u,v)\in A} c(u, v).$$

The Traveling-Salesman Problem

- In practice, it is always cheapest to go directly from a place u to a place w; going by way of any intermediate stop v cannot be less expensive.
- Putting it another way, cutting out an intermediate stop never increases the cost.
- We formalize this notion by saying that the cost function c satisfies the **triangle inequality** if for all vertices $u, v, w \in V$
$$c(u, w) \leq c(u, v) + c(v, w).$$

The Traveling-Salesman Problem

- The triangle inequality is a natural one, and in many applications it is automatically satisfied.
- For example, if the vertices of the graph are points in the plane and the cost of traveling between two vertices is the ordinary euclidean distance between them, then the triangle inequality is satisfied.
- The traveling-salesman problem is **NP-complete** even if we require that the cost function satisfies the triangle inequality.
- We therefore look instead for good approximation algorithms.

The Traveling-Salesman Problem

- We compute a structure - a minimum spanning tree - whose weight is a **lower bound** on the length of an optimal traveling salesman tour.
- We use the minimum spanning tree to create a tour whose cost is no more than twice that of the minimum spanning tree's weight, as long as the cost function satisfies the **triangle inequality**.
- $\rightsquigarrow$ The minimum-spanning-tree algorithm MST-PRIM.

TSP Approximation Algorithm

APPROX-TSP-TOUR(G,c)

```
1.    Select a vertex $r \in V[G]$ to be a "root" vertex
2.    Compute a minimum spanning tree $T$ for $G$ from root $r$
using MST-PRIM($G$, $c$, $r$)
3.    Let $H$ be the list of vertices, ordered accordingly to when
they are first visited in a preorder tree walk of $T$.
4. Return the Hamiltonian cycle $H$
```

Preorder tree walk

A preorder tree walk recursively visits every vertex in the tree, listing a vertex when it is first encountered, before visiting any of its children

Prim's Algorithm

MST-PRIM(G,w,r)

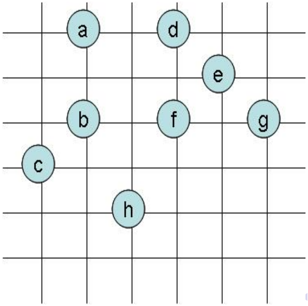
```
1    for each $u \in V[G]$
2       do $key[u] \leftarrow \infty$
3          $\pi[u] \leftarrow NIL$
4    $key[r] \leftarrow 0$
5    $Q \leftarrow V[G]$
6    while $Q \neq \phi$
7       do $u \leftarrow EXTRACT - MIN(Q)$
8          for each $v \in Adj[u]$
9             do if $v \in Q$ and $w(u,v) < key[v]$
10                  then $\pi[v] \leftarrow u$
11                       $key[v] \leftarrow w(u,v)$
```

Complexity: $O(|E| \log |V|)$

APPROX-TSP-TOUR Algorithm

APPROX-TSP-TOUR(G, c)

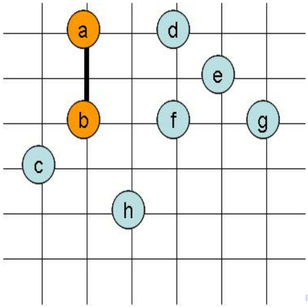
1. Select a vertex $r \in V[G]$ to be a "root" vertex
2. Compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. Let H be the list of vertices, ordered accordingly to when they are first visited in a preorder tree walk of T.
4. Return the Hamiltonian cycle H



APPROX-TSP-TOUR Algorithm

APPROX-TSP-TOUR(G, c)

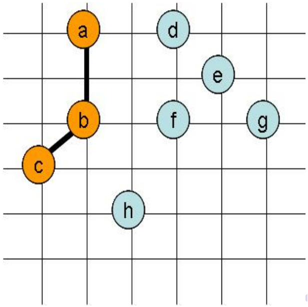
1. Select a vertex $r \in V[G]$ to be a "root" vertex
2. Compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. Let H be the list of vertices, ordered accordingly to when they are first visited in a preorder tree walk of T.
4. Return the Hamiltonian cycle H



APPROX-TSP-TOUR Algorithm

APPROX-TSP-TOUR(G, c)

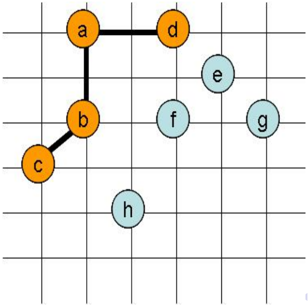
1. Select a vertex $r \in V[G]$ to be a "root" vertex
2. Compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. Let H be the list of vertices, ordered accordingly to when they are first visited in a preorder tree walk of T.
4. Return the Hamiltonian cycle H



APPROX-TSP-TOUR Algorithm

APPROX-TSP-TOUR(G, c)

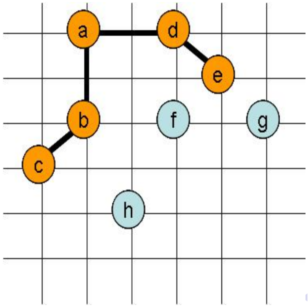
```
1. Select a vertex $r \in V[G]$ to be a "root" vertex
2. Compute a minimum spanning tree $T$ for $G$ from root $r$ using MST-PRIM($G, c, r$)
3. Let $H$ be the list of vertices, ordered accordingly to when they are first visited in a preorder tree walk of $T$.
4. Return the Hamiltonian cycle $H$
```



APPROX-TSP-TOUR Algorithm

APPROX-TSP-TOUR(G, c)

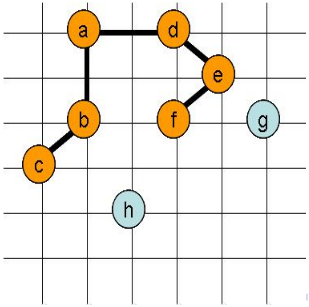
1. Select a vertex $r \in V[G]$ to be a "root" vertex
2. Compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. Let H be the list of vertices, ordered accordingly to when they are first visited in a preorder tree walk of T.
4. Return the Hamiltonian cycle H



APPROX-TSP-TOUR Algorithm

APPROX-TSP-TOUR(G, c)

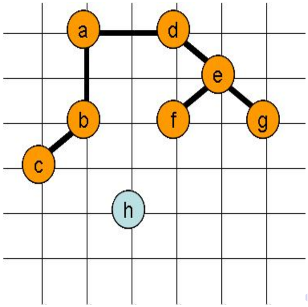
1. Select a vertex $r \in V[G]$ to be a "root" vertex
2. Compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. Let H be the list of vertices, ordered accordingly to when they are first visited in a preorder tree walk of T.
4. Return the Hamiltonian cycle H



APPROX-TSP-TOUR Algorithm

APPROX-TSP-TOUR(G, c)

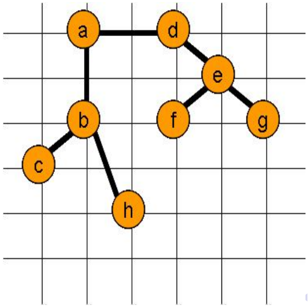
1. Select a vertex $r \in V[G]$ to be a "root" vertex
2. Compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. Let H be the list of vertices, ordered accordingly to when they are first visited in a preorder tree walk of T.
4. Return the Hamiltonian cycle H



APPROX-TSP-TOUR Algorithm

APPROX-TSP-TOUR(G, c)

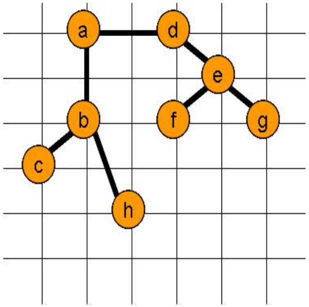
1. Select a vertex $r \in V[G]$ to be a "root" vertex
2. Compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. Let H be the list of vertices, ordered accordingly to when they are first visited in a preorder tree walk of T.
4. Return the Hamiltonian cycle H

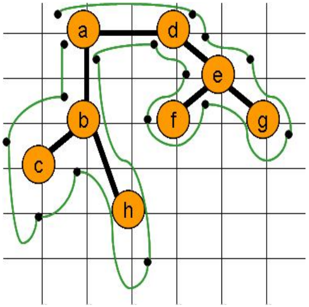


APPROX-TSP-TOUR Algorithm

APPROX-TSP-TOUR(G, c)

1. Select a vertex $r \in V[G]$ to be a "root" vertex
2. Compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. Let H be the list of vertices, ordered accordingly to when they are first visited in a preorder tree walk of T.
4. Return the Hamiltonian cycle H

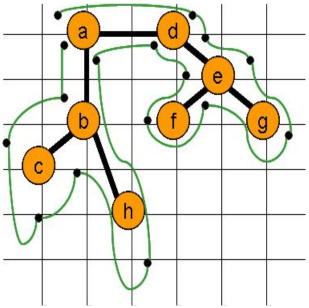


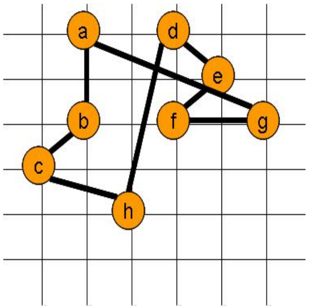


APPROX-TSP-TOUR Algorithm

APPROX-TSP-TOUR(G, c)

1. Select a vertex $r \in V[G]$ to be a "root" vertex
2. Compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. Let H be the list of vertices, ordered accordingly to when they are first visited in a preorder tree walk of T.
4. Return the Hamiltonian cycle H

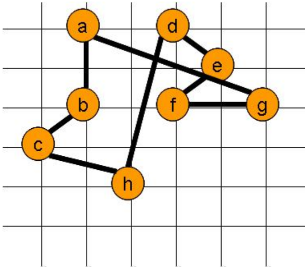




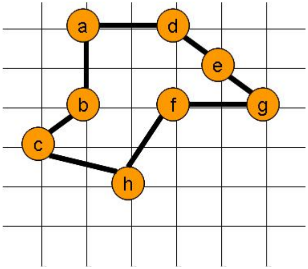
APPROX-TSP-TOUR Algorithm

APPROX-TSP-TOUR(G, c)

1. Select a vertex $r \in V[G]$ to be a "root" vertex
2. Compute a minimum spanning tree T for G from root r using MST-PRIM(G, c, r)
3. Let H be the list of vertices, ordered accordingly to when they are first visited in a preorder tree walk of T.
4. Return the Hamiltonian cycle H



Near optimal tour



Optimal Tour

APPROX-TSP-TOUR Algorithm: Its approximation ratio (1/4)

Theorem

APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling salesman problem with the triangle inequality

Proof

- APPROX-TSP-TOUR runs in polynomial time.
- Let $H^\star$ denote an optimal tour for the given set of vertices.
- Since we obtain a spanning tree T by deleting any edge from a tour, the weight of such a minimum spanning tree T is a **lower bound** on the cost of an optimal tour,
- $c(T) \leq c(H^\star)$

APPROX-TSP-TOUR Algorithm: Its approximation ratio (2/4)

Theorem

APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling salesman problem with the triangle inequality

Proof Cont'd

- Let T_{MST} be the minimum spanning tree resulting from the application of Kruskal's algorithm
- A **full walk** of T_{MST} lists the vertices when they are first visited $+$ when they are returned to after a visit to a subtree.
- Let us call this walk W. Full walk of the example gives the order: $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$.
- Since the full walk traverses every edge of T_{MST} exactly twice: $c(W) = 2c(T_{\text{MST}}) \leq 2c(T) \leq 2c(H^\star)$,
  $\rightsquigarrow$ cost of W is within a factor of the cost of an optimal tour.

APPROX-TSP-TOUR Algorithm: Its approximation ratio (3/4)

Theorem

APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling salesman problem with the triangle inequality

Proof Cont'd

- But W is generally not a tour, since it visits some vertices more than once.
- By the **triangle inequality**, we can delete a visit to any vertex from W and the cost does **not increase**.
- If a vertex v is deleted from W between visits to u and w $\hookrightarrow$ resulting ordering specifies going directly from u to w.
- By repeatedly applying this operation, we can remove from W all but the first visit to each vertex.

$$a, b, c, h, d, e, f, g.$$

APPROX-TSP-TOUR Algorithm: Its approximation ratio (4/4)

Theorem

APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling salesman problem with the triangle inequality

Proof Cont'd

- Such an ordering: the same as that obtained by a preorder walk of the tree T_{MST}.
- Let H be the cycle corresponding to this preorder walk.
- H is a hamiltonian cycle, and it corresponds to the cycle computed by APPROX-TSP-TOUR.
- H is obtained by deleting vertices from the full walk W:

$$\rightsquigarrow \quad c(H) \leq c(W) \leq 2c(H^\star),$$

which completes the proof.

TSP Problem - No Triangle Inequality

If we **drop** the assumption that the cost function c satisfies the **triangle inequality**, good approximate tours cannot be found in polynomial time unless $P = NP$.

Theorem

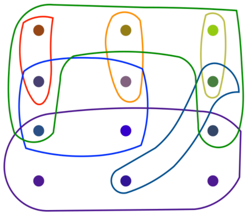
If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio ρ for the general traveling-salesman problem.

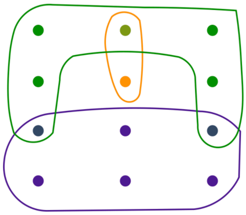
Set-Covering Problem

- Set-covering problem
    - an optimization problem that models many resource-selection problems
    - generalizes the NP-complete vertex-cover problem
- Objective: Study a simple greedy heuristic with a logarithmic approximation ratio.
- Observation: The logarithm function grows rather slowly. However, such an approximation algorithm may nonetheless give useful results.

Set-Covering Problem

- An instance (X, F) of the **set covering problem**: A finite set X and a family F of subsets of X, such that every element of X belongs to at least one subset in F: $X = \bigcup\limits_{S \in F} S$.
- Set-Covering Problem: Find a minimum-size subset $C \subseteq F$ whose members cover all of X: $X = \bigcup\limits_{S \in C} S$.
- We say that C **covers** X.


Input


Output

GREEDY-SET-COVER Algorithm (1/7)

GREEDY-SET-COVER(X, F)

```
1. $U \leftarrow X$
2. $C \leftarrow \phi$
3. while $U \neq \phi$
4.     do select an $S \in F$ that maximizes $|S \cap U|$
5.         $U \leftarrow U - S$
6.         $C \leftarrow C \cup \{S\}$
7. return $C$
```

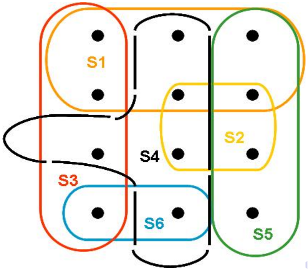
The greedy method works by picking, at each stage, the set S that covers the **greatest** number of remaining elements that are **uncovered**.

GREEDY-SET-COVER algorithm can be implemented to run in polynomial time in $|X|$ and $|F|$.

GREEDY-SET-COVER Algorithm (2/7)

GREEDY-SET-COVER(X, F)

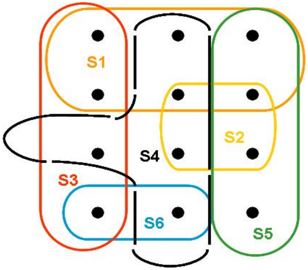
```
1. $U \leftarrow X$
2. $C \leftarrow \phi$
3. **while** $U \neq \phi$
4.    **do** select an $S \in F$ that maximizes $|S \cap U|$
5.       $U \leftarrow U - S$
6.       $C \leftarrow C \cup \{S\}$
7. **return** $C$
```

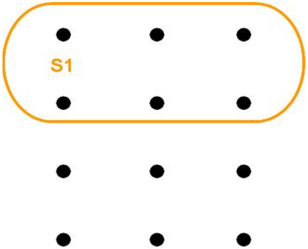


GREEDY-SET-COVER Algorithm (3/7)

GREEDY-SET-COVER(X, F)

```
1. $U \leftarrow X$
2. $C \leftarrow \phi$
3. while $U \neq \phi$
4.     do select an $S \in F$ that maximizes $|S \cap U|$
5.         $U \leftarrow U - S$
6.         $C \leftarrow C \cup \{S\}$
7. return $C$
```

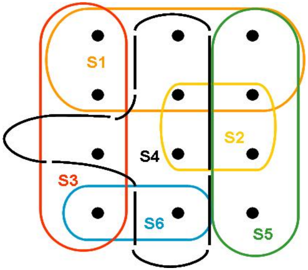


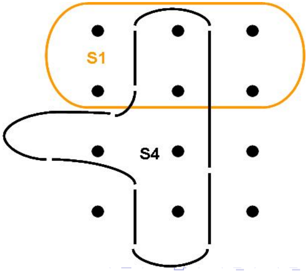


GREEDY-SET-COVER Algorithm (4/7)

GREEDY-SET-COVER(X, F)

```
1. $U \leftarrow X$
2. $C \leftarrow \phi$
3. while $U \neq \phi$
4.     do select an $S \in F$ that maximizes $|S \cap U|$
5.         $U \leftarrow U - S$
6.         $C \leftarrow C \cup \{S\}$
7. return $C$
```

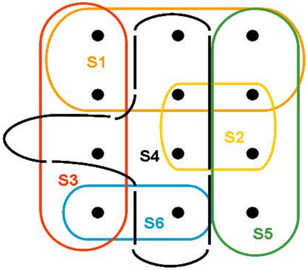


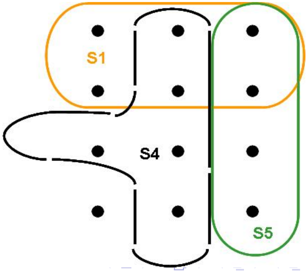


GREEDY-SET-COVER Algorithm (5/7)

GREEDY-SET-COVER(X, F)

```
1. $U \leftarrow X$
2. $C \leftarrow \phi$
3. while $U \neq \phi$
4.     do select an $S \in F$ that maximizes $|S \cap U|$
5.        $U \leftarrow U - S$
6.        $C \leftarrow C \cup \{S\}$
7. return $C$
```

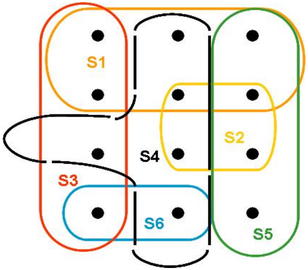


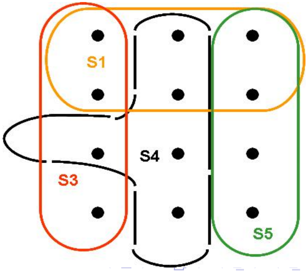


GREEDY-SET-COVER Algorithm (6/7)

GREEDY-SET-COVER(X, F)

```
1. $U \leftarrow X$
2. $C \leftarrow \phi$
3. while $U \neq \phi$
4.     do select an $S \in F$ that maximizes $|S \cap U|$
5.         $U \leftarrow U \setminus S$
6.         $C \leftarrow C \cup \{S\}$
7. return $C$
```

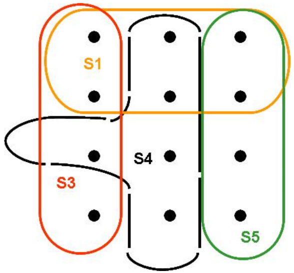




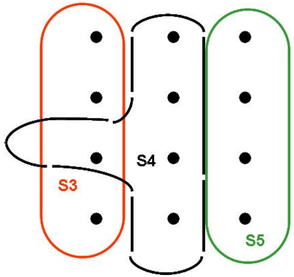
GREEDY-SET-COVER Algorithm (7/7)

GREEDY-SET-COVER(X, F)

```
1. $U \leftarrow X$
2. $C \leftarrow \phi$
3. **while** $U \neq \phi$
4.     **do** select an $S \in F$ that maximizes $|S \cap U|$
5.         $U \leftarrow U \setminus S$
6.         $C \leftarrow C \cup \{S\}$
7. **return** $C$
```



Near optimal solution



Optimal solution

Complexity of GREEDY-SET-COVER Algorithm

Since each iteration of the while loop in lines 3-7 adds a set $S \in F$ to the cover C, and S must cover at least one of the as yet uncovered elements of X, it is clear that the maximum number of iterations of the while loop is at most $\min\{|X|; |F|\}$.

The actual body of the loop in lines 4-6 can be implemented in time $O(|X| \times |F|)$ and so the overall complexity of GREEDY-SET-COVER is $O(|X| \times |F| \min\{|X|; |F|\})$ [1]

[1] Actually this bound can be improved further. The algorithm can be implemented in time that is linear in the size of the input. For our purposes, however, it is sufficient to note that the algorithm runs in polynomial time.

Analysis of GREEDY-SET-COVER Algorithm (1/2)

Analysis

- The GREEDY-SET-COVER algorithm returns a set cover that is not too much larger than an optimal set cover.
- Denote the *d*th harmonic number

$$H_d = \sum_{i=1}^{d} \frac{1}{i} \qquad H(0) = 0.$$

Theorem

GREEDY-SET-COVER is a polynomial-time $\rho(n)$-approximation algorithm, where

$$\rho(n) = H(\max\{|S| : S \in F\})$$

Analysis of GREEDY-SET-COVER Algorithm (2/2)

$$H(n) = \sum_{i=1}^{n} \frac{1}{i} \leq 1 + \ln n \qquad H(0) = 0.$$

Therefore, GREEDY-SET-COVER is a polynomial-time algorithm with approximation ratio $\rho(n)$ such that,

$$\begin{aligned} \rho(n) &= H(\max\{|S| : S \in F\}) \\ &\leq H(|X|) \quad (S \subset X) \Rightarrow |S| \leq |X|) \\ &\leq \ln |X| + 1 \end{aligned}$$

Subset-sum problem

- An instance of the subset-sum problem: A pair (S, t), where S is a set $\{x_1, x_2, ..., x_n\}$ of positive integers and t is a positive integer.
- Decision problem: Asks whether there exists a subset of S that adds up exactly to the target value t.
- It is an NP-complete problem
- Optimization problem: Find a subset of $\{x_1, x_2, ..., x_n\}$ whose sum is as large as possible but not larger than t.

An exponential-time exact algorithm

- Procedure EXACT-SUBSET-SUM
  - Input: Set $S = \{x_1, x_2, ..., x_n\}$ and a target value t.
  - Iteratively computes L_i, the list of sums of all subsets of $\{x_1, ..., x_i\}$ that do not exceed t,
  - Returns the maximum value in L_n.

Notations

- L: List of positive integers ; x: A positive integer
- $L + x$: List of integers derived from L by increasing each element of L by x.
- $L =< 1, 2, 3, 5, 9 >$ then $L + 2 =< 3, 4, 5, 7, 11 >$.
- This notation is also used for sets $S + x = \{s + x : s \in S\}$
- Auxiliary procedure MERGE-LISTS(L, L'): Returns the sorted list that is the merge of its two sorted input lists L and L' with duplicate values removed.
- MERGE-LISTS(L, L') runs in time $O(|L| + |L'|)$

Subset-sum problem

EXACT-SUBSET-SUM(S,t)

```
1  $n \leftarrow |S|$
2  $L_0 \leftarrow < 0 >$
3  **for** $i \leftarrow 1$ **to** $n$
4      **do** $L_i \leftarrow$ MERGE-LISTS($L_{i-1}$,$L_{i-1} + x_i$)
5          remove from $L_i$ every element that is greater than $t$
6  **return** the largest element in $L_n$
```

Example: $S = \{1, 4, 5\}$

$$\begin{aligned} P_1 &= \{0, 1\} \\ P_2 &= \{0, 1, 4, 5\} \\ P_3 &= \{0, 1, 4, 5, 6, 9, 10\} \end{aligned}$$

Subset-sum problem

Given the identity $P_i = P_{i-1} \cup (P_{i-1} + x_i)$

- we can prove by induction on i that the list L_i is a sorted list containing every element of P_i whose value is not more than t.
- Since the length of L_i can be as much as 2^i, EXACT-SUBSET-SUM is an exponential-time algorithm in general.

A fully polynomial-time approximation scheme

- **Idea.** If two values in L are close to each other, then for the purpose of finding an approximate solution, there is no reason to maintain both of them explicitly.
- More precisely, we use a trimming parameter δ such that $0 < \delta < 1$.
- To **trim** a list L by δ means
  - if L' is the result of trimming L
  - then for every element y that was **removed** from L, there is an element z still in L' that approximates y, that is,
  - If $\frac{y}{1+\delta} \leq z \leq y$: Remove y from L.
  - If $y \geq z \times (1+\delta)$: Maintain y in L.

Subset-sum problem

- Example: $\delta = 0.1$ and $L =< 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 >$
- TRIM(L, δ) leads to: $L' =< 10, 12, 15, 20, 23, 29 >$.

TRIM(L, δ)

```
1   $m \leftarrow |L|$
2   $L' \leftarrow < y_1 >$
3   $last \leftarrow y_1$
4   for $i \leftarrow 2$ to $m$
5      do if $y_i > last \times (1 + \delta)$
6         then append $y_i$ onto the end of $L'$
7              $last \leftarrow y_i$
8   return $L'$
```

Subset-sum problem

- The elements of L are scanned in monotonically increasing order
- A number is put into the returned list L' **only if** it is the first element of L **or if** it cannot be represented by the most recent number placed into L'
- Given the procedure TRIM, we can construct our approximation scheme as follows.
- **Input:** A set $S = x_1, x_2, ..., x_n$ of n integers (in arbitrary order), a target integer t, and an "approximation parameter" ε where $0 < \varepsilon < 1$
- **Output:** A value z whose value is within a $1 + \varepsilon$ factor of the optimal solution.

Subset-sum problem

APPROX-SUBSET-SUM(S, t, ε)

```
1. $n \leftarrow |S|$
2. $L_0 \leftarrow < 0 >$
3. **for** $i \leftarrow 1$ **to** $n$
4.     **do** $L_i \leftarrow$ MERGE-LISTS($L_{i-1}$,$L_{i-1} + x_i$)
5.         $L_i \leftarrow TRIM(L_i, \varepsilon/(2n))$
6.         remove from $L_i$ every element that is greater than $t$
7. Let $z^\star$ be the largest value in $L_n$
8. **return** $z^\star$
```

- Line 2: Initializes the list L_0 to be the list containing just the element 0.
- **for** loop in lines 3-6: Has the effect of computing L_i as a sorted list containing a suitably trimmed version of the set P_i, with all elements larger than t removed.

Subset-sum problem

APPROX-SUBSET-SUM(S,t,ε)

```
1  $n \leftarrow |S|$
2  $L_0 \leftarrow < 0 >$
3  **for** $i \leftarrow 1$ **to** $n$
4     **do** $L_i \leftarrow$ MERGE-LISTS($L_{i-1}$,$L_{i-1} + x_i$)
5        $L_i \leftarrow TRIM(L_i, \varepsilon/(2n))$
6        remove from $L_i$ every element that is greater than $t$
7  let $z^\star$ be the largest value in $L_n$
8  **return** $z^\star$
```

- $S =< 104, 102, 201, 101 >$, with $t = 308$ and $\varepsilon = 0.40$.
- Trimming parameter: $\delta = \varepsilon/8 = 0.05$
- APPROX-SUBSET-SUM computes the following values:

line 2: $L_0 =< 0 >$	line 4: $L_2 =< 0, 102, 104, 206 >$
line 4: $L_1 =< 0, 104 >$	line 5: $L_2 =< 0, 102, 206 >$
line 5: $L_1 =< 0, 104 >$	line 6: $L_2 =< 0, 102, 206 >$
line 6: $L_1 =< 0, 104 >$	

Subset-sum problem

APPROX-SUBSET-SUM(S,t,ε)

```
1  $n \leftarrow |S|$
2  $L_0 \leftarrow < 0 >$
3  for $i \leftarrow 1$ to $n$
4     do $L_i \leftarrow$ MERGE-LISTS($L_{i-1}$,$L_{i-1} + x_i$)
5        $L_i \leftarrow TRIM(L_i, \varepsilon/(2n))$
6           remove from $L_i$ every element that is greater than $t$
7  let $z^\star$ be the largest value in $L_n$
8  return $z^\star$
```

line 4: $L_3 =< 0, 102, 201, 206, 303, 407 >$

line 5: $L_3 =< 0, 102, 201, 303, 407 >$

line 6: $L_3 =< 0, 102, 201, 303 >$

Subset-sum problem

APPROX-SUBSET-SUM(S,t,ε)

```
1  $n \leftarrow |S|$
2  $L_0 \leftarrow < 0 >$
3  for $i \leftarrow 1$ to $n$
4     do $L_i \leftarrow$ MERGE-LISTS($L_{i-1}$,$L_{i-1} + x_i$)
5        $L_i \leftarrow TRIM(L_i, \varepsilon/(2n))$
6           remove from $L_i$ every element that is greater than $t$
7  let $z^\star$ be the largest value in $L_n$
8  return $z^\star$
```

line 4: $L_4 =< 0, 101, 102, 201, 203, 302, 303, 404 >$

line 5: $L_4 =< 0, 101, 201, 302, 404 >$

line 6: $L_4 =< 0, 102, 201, 302 >$

The algo returns $z^\star = 302$ which is well within $\varepsilon = 40\%$ of the optimal solution $104 + 102 + 101 = 307$, it is within 2%.

Subset-sum problem

Theorem

APPROX-SUBSET-SUM is a fully polynomial-time approximation scheme for the subset-sum problem.