Approximation Algorithm

Approximation

- **Performance ratios for approximation algorithms:** We say that an algorithm for a problem has an ***approximation ratio*** of ρ(n) if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of ρ(n)**-** of the cost C* of an optimal solution:

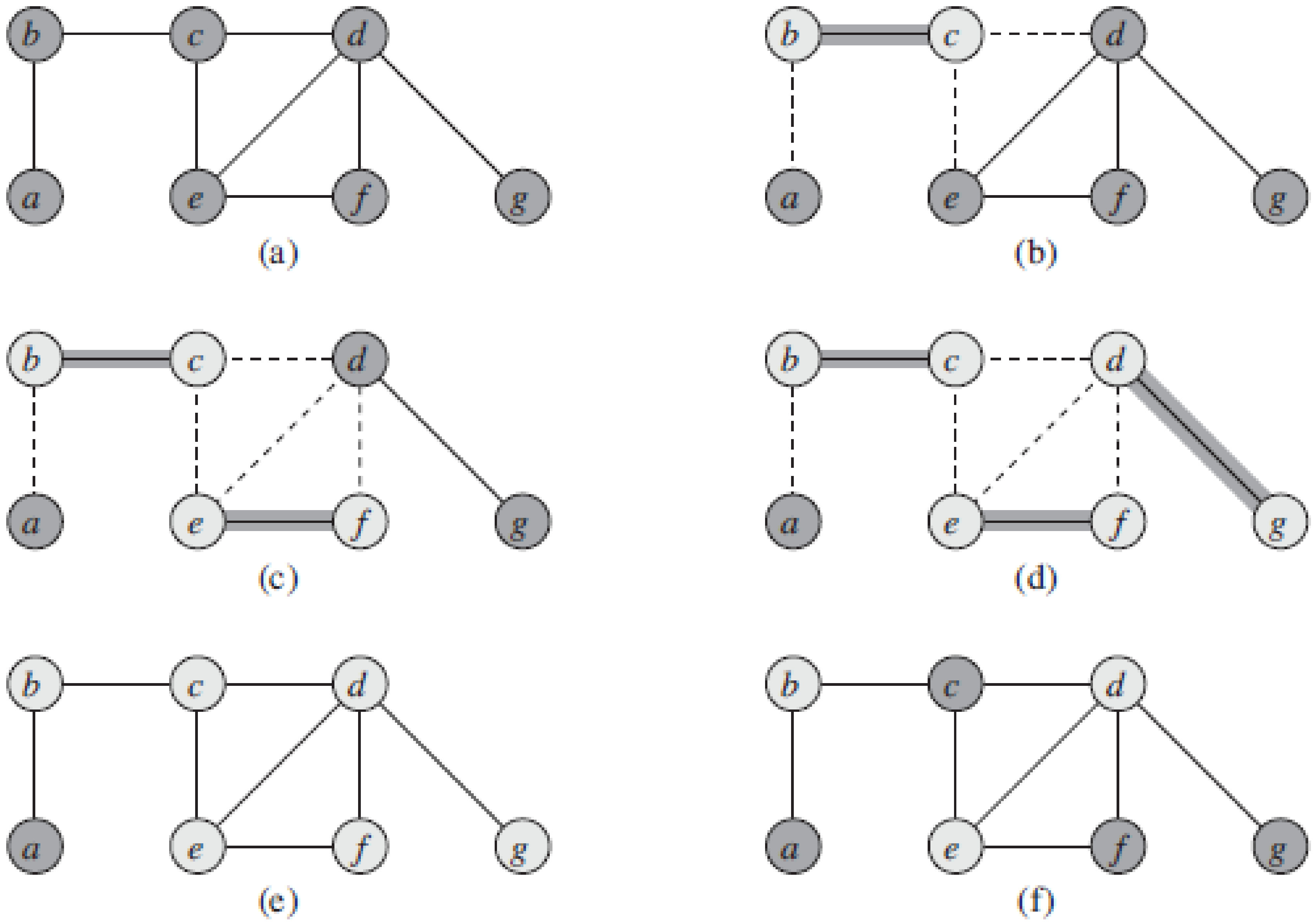
$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n)$$

- If an algorithm achieves an approximation ratio of ρ(n), we call it a ρ(n)***-approximation algorithm***.

The vertex-cover problem

- The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an ***optimal vertex cover***. This problem is the optimization version of an NP-complete decision problem.

```
APPROX-VERTEX-COVER$(G)$
1  $C = \emptyset$
2  $E' = G.E$
3  while $E' \neq \emptyset$
4      let $(u, v)$ be an arbitrary edge of $E'$
5      $C = C \cup \{u, v\}$
6      remove from $E'$ every edge incident on either $u$ or $v$
7  return $C$
```


(a)
(b)
(c)
(d)
(e)
(f)

- Theorem: APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm

The set C of vertices that is returned by APPROX-VERTEX-COVER is a vertex cover, since the algorithm loops until every edge in G:*E* has been covered by some vertex in C.

let A denote the set of edges that line 4 of approx-vertex-cover picked. In order to cover the edges in A, any vertex cover—in particular, an optimal cover C*—must include at least one endpoint of each edge in A. 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 $|C^*| \geq |A|$

- on the size of an optimal vertex cover. 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: $|C| = 2|A|$

$$\begin{aligned} |C| &= 2|A| \\ &\leq 2|C^*| , \end{aligned}$$

The traveling-salesman problem

- we are given a complete undirected graph G = (V,E) that has a nonnegative integer cost c(u,v) associated with each edge (u,v)$\in E$, and we must find a hamiltonian cycle (a tour) of G with minimum cost.

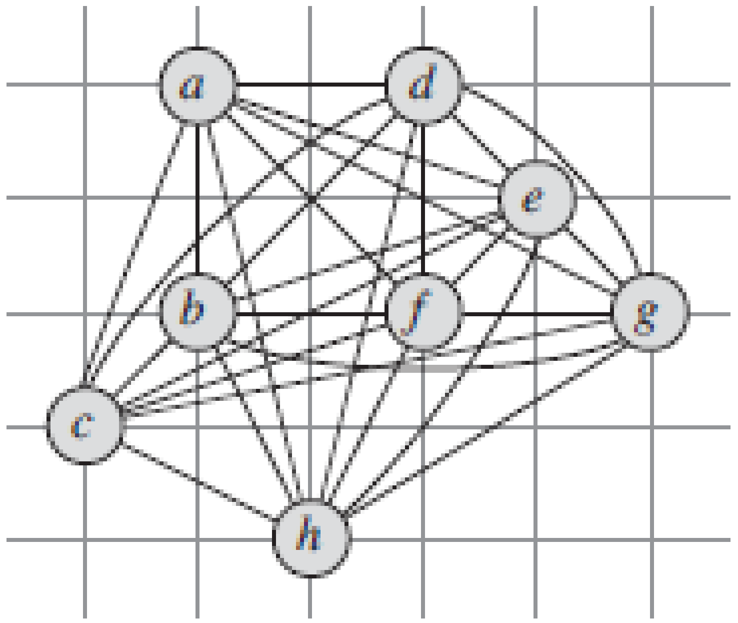
Triangle inequality

- In many practical situations, the least costly way to go from a place u to a place w is to go directly, with no intermediate steps. Put 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 ∈ V

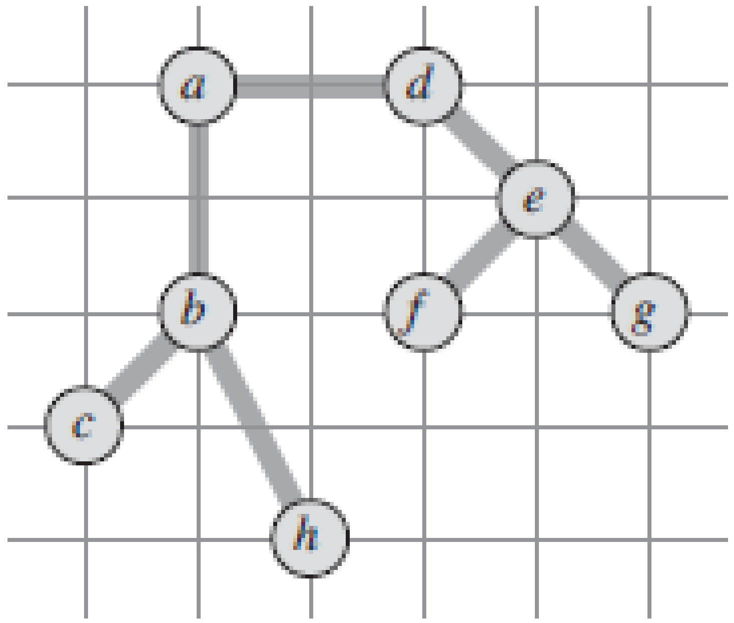
$$c(u, w) \le c(u, v) + c(v, w) .$$

Approximation Algorithm

```
APPROX-TSP-TOUR($G, c$)
1  select a vertex $r \in G.V$ 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 a list of vertices, ordered according to when they are first visited
       in a preorder tree walk of $T$
4  return the hamiltonian cycle $H$
```



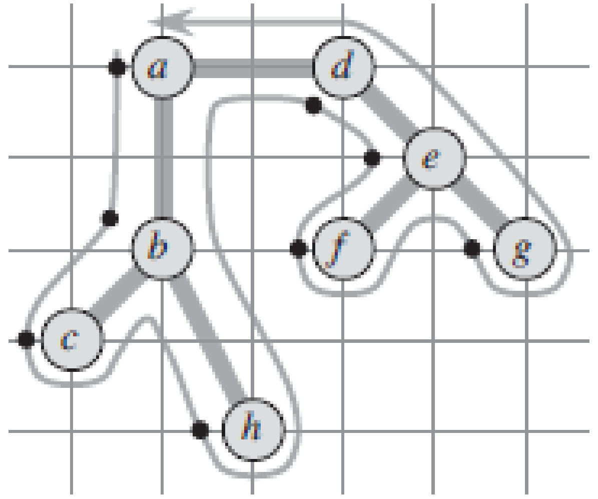
(a)



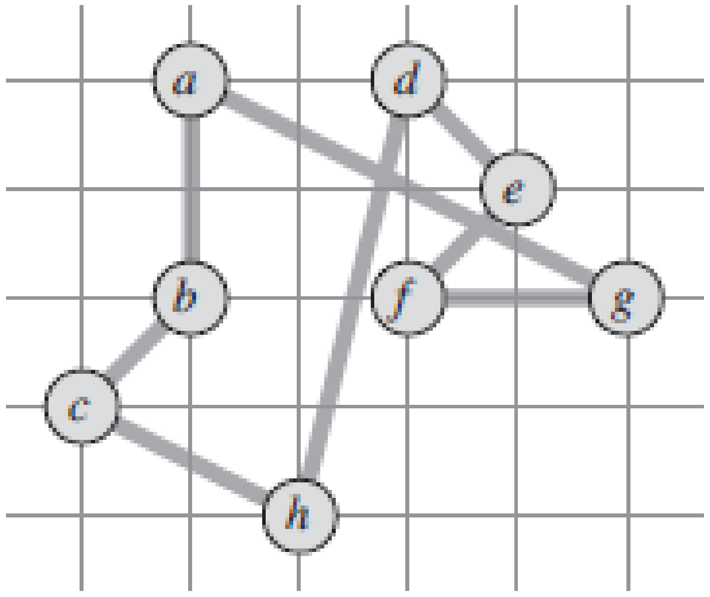
(b)

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

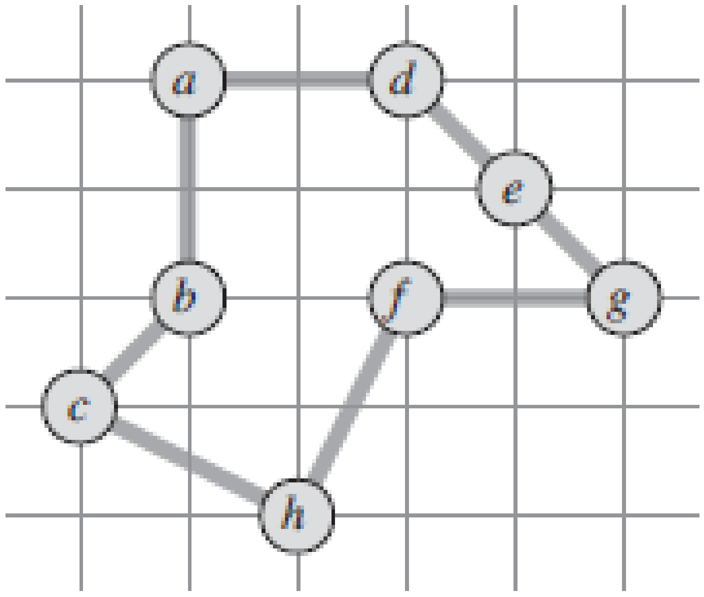
a,b,c,h,d,e,f,g



(c)



(d)



(e)

T’, $c(T') \leq C(H^*)$

=> $C(T) \leq C(T')$

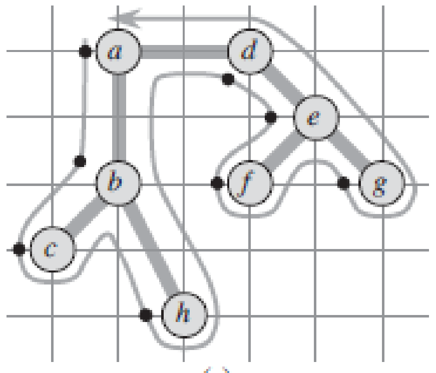
=> $C(T) \leq C(H^*)$

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

- We have already seen that APPROX-TSP-TOUR runs in polynomial time. Let H* denote an optimal tour for the given set of vertices. We obtain a spanning tree by deleting any edge from a tour, and each edge cost is nonnegative. Therefore, the weight of the minimum spanning tree T computed in line 2 of APPROX-TSPTOUR provides a lower bound on the cost of an optimal tour:
- $c(T) \leq c(H^*)$

- A ***full walk*** of T lists the vertices when they are first visited and also whenever they are returned to after a visit to a subtree. Let us call this full walk W . The full walk of our example gives the order

  a; b; c; b; h; b; a; d; e; f; e; g; e; d; a :

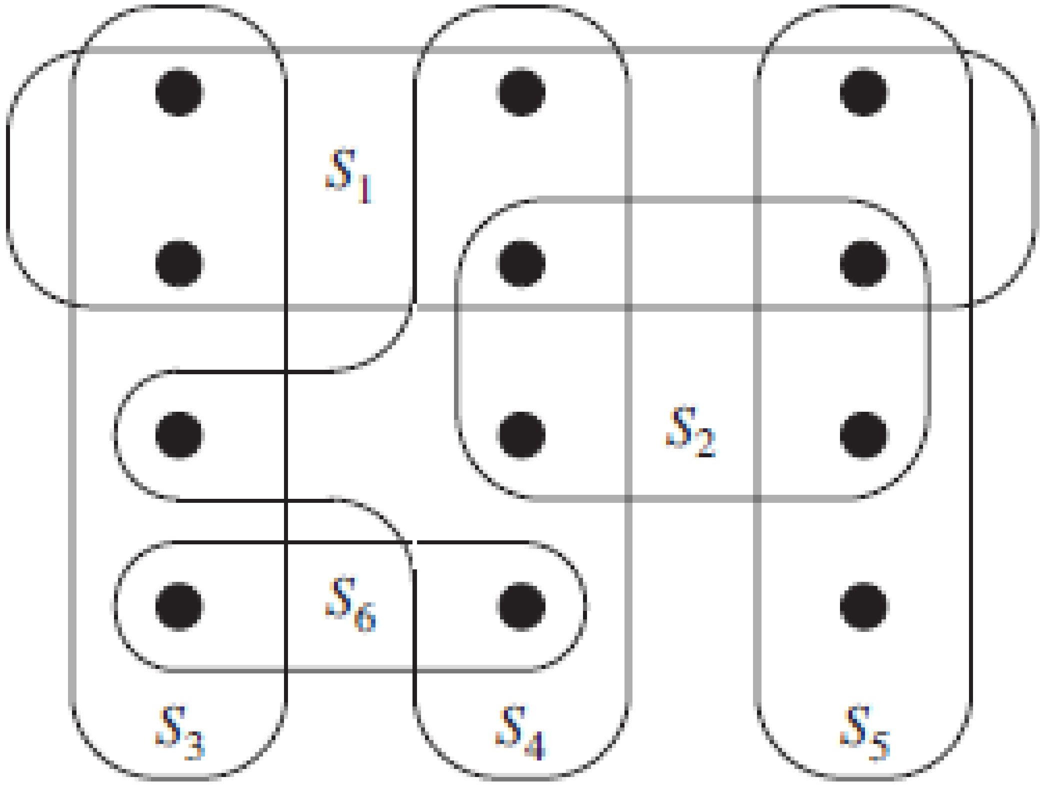


(c)

- Since the full walk traverses every edge of T exactly twice, we have
- $c(W) = 2c(T)$
- $c(W) \leq 2c(H^*)$
- Unfortunately, the full walk W is generally not a tour, since it visits some vertices more than once. By the triangle inequality, however, we can delete a visit to any vertex from W and the cost does not increase.

- $c(H) \leq c(W)$
- $c(H) \leq 2c(H^*)$

The set-covering problem



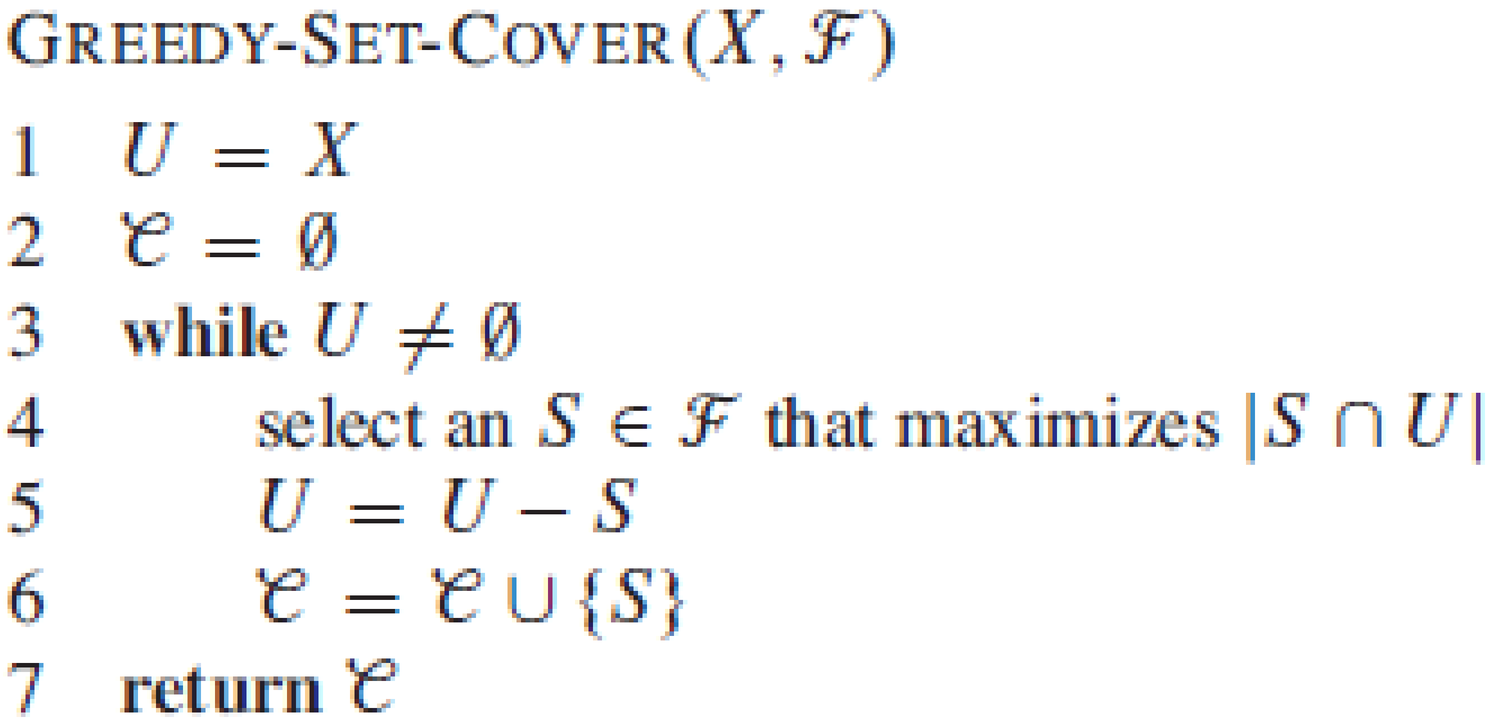
An instance $(X, \mathcal{F})$ of the ***set-covering problem*** consists of a finite set X and a family $\mathcal{F}$ of subsets of X, such that every element of X belongs to at least one subset in $\mathcal{F}$:

$$X = \bigcup_{S \in \mathcal{F}} S .$$

We say that a subset $S \in \mathcal{F}$ ***covers*** its elements. The problem is to find a minimum-size subset $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of X:

$$X = \bigcup_{S \in \mathcal{C}} S .$$

Approximation Algorithm



```
GREEDY-SET-COVER($X$, $\mathcal{F}$)
$U = X$
$\mathcal{C} = \emptyset$
while $U \neq \emptyset$
    select an $S \in \mathcal{F}$ that maximizes $|S \cap U|$
    $U = U - S$
    $\mathcal{C} = \mathcal{C} \cup \{S\}$
return $\mathcal{C}$
```

- To show that GREEDY-SET-COVER is a ρ(n)-approximation algorithm, we assign a cost of 1 to each set selected by the algorithm, distribute this cost over the elements covered for the first time, and then use these costs to derive the desired relationship between the size of an optimal set cover $\mathcal{C}^*$ and the size of the set cover $\mathcal{C}$ returned by the algorithm.

$$c_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|} .$$

Each step of the algorithm assigns 1 unit of cost, and so

$$|\mathcal{C}| = \sum_{x \in X} c_x .$$

Each element $x \in X$ is in at least one set in the optimal cover $\mathcal{C}^*$, and so we have

$$\sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x .$$

$$|\mathcal{C}| \leq \sum_{S \in \mathcal{C}^*} \sum_{x \in S} c_x .$$

$$\sum_{x \in S} c_x \leq H(|S|) .$$

$$H_d = \sum_{i=1}^{d} 1/i$$

$$\begin{aligned} |\mathcal{C}| &\leq \sum_{S \in \mathcal{C}^*} H(|S|) \\ &\leq |\mathcal{C}^*| \cdot H(\max\{|S| : S \in \mathcal{F}\}) , \end{aligned}$$

A randomized approximation algorithm for MAX-3-CNF satisfiability

- We say that a randomized algorithm for a problem has an ***approximation ratio*** of ρ(n), if, for any input of size n, the *expected* cost C of the solution produced by the randomized algorithm is within a factor of ρ(n) of the cost C* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n)$$

We call a randomized algorithm that achieves an approximation ratio of ρ(n) a randomized ρ(n)-approximation algorithm.

Optimization version of 3-CNF

- A particular instance of 3-CNF satisfiability may or may not be satisfiable. In order to be satisfiable, there must exist an assignment of the variables so that every clause evaluates to 1. If an instance is not satisfiable, we may want to compute how “close” to satisfiable it is, that is, we may wish to find an assignment of the variables that satisfies as many clauses as possible. We call the resulting maximization problem ***MAX-3-CNF satisfiability***

Cont..

- The input to MAX-3-CNF satisfiability is the same as for 3-CNF satisfiability, and the goal is to return an assignment of the variables that maximizes the number of clauses evaluating to 1.

Theorem: Given an instance of MAX-3-CNF satisfiability with n variables x1, x2, … ,xn and m clauses, the randomized algorithm that independently sets each variable to 1 with probability 1/2 and to 0 with probability 1/2 is a randomized 8/7-approximation algorithm.

- Proof: Suppose that we have independently set each variable to 1 with probability 1/2 and to 0 with probability 1/2. For i = 1,2, … ,m, we define the indicator random variable
- Yi = 1 {clause i is satisfied} ;
- so that Yi = 1 as long as we have set at least one of the literals in the i th clause to 1. Since no literal appears more than once in the same clause, and since we have assumed that no variable and its negation appear in the same clause, the settings of the three literals in each clause are independent.

- A clause is not satisfied only if all three of its literals are set to 0, and so Pr {clause i is not satisfied} = $(1/2)^3$ = 1/8.
- Thus, we have Pr {clause i is satisfied} = 1 – 1/8 = 7/8, Let Y be the number of satisfied clauses overall, so that Y = Y1 + Y2 +…+ Ym. Then, we have

$$\begin{aligned} \mathrm{E}[Y] &= \mathrm{E}\left[\sum_{i=1}^{m} Y_i\right] \\ &= \sum_{i=1}^{m} \mathrm{E}[Y_i] \qquad \text{(by linearity of expectation)} \\ &= \sum_{i=1}^{m} 7/8 \\ &= 7m/8\,. \end{aligned}$$

- Clearly, m is an upper bound on the number of satisfied clauses, and hence the approximation ratio is at most m/(7m/8) = 8/7.

Approximating weighted vertex cover using linear programming

In the ***minimum-weight vertex-cover problem***, we are given an undirected graph $G = (V, E)$ in which each vertex $v \in V$ has an associated positive weight $w(v)$. For any vertex cover $V' \subseteq V$, we define the weight of the vertex cover $w(V') = \sum_{v \in V'} w(v)$. The goal is to find a vertex cover of minimum weight.

Suppose that we associate a variable $x(v)$ with each vertex $v \in V$, and let us require that $x(v)$ equals either 0 or 1 for each $v \in V$. We put v into the vertex cover if and only if $x(v) = 1$. Then, we can write the constraint that for any edge (u, v), at least one of u and v must be in the vertex cover as $x(u) + x(v) \geq 1$. This view gives rise to the following ***0-1 integer program*** for finding a minimum-weight vertex cover:

$$
\begin{array}{llcll}
\text{minimize} & \displaystyle\sum_{v \in V} w(v)\,x(v) & & & \\
\text{subject to} & & & & \\
& x(u) + x(v) & \ge & 1 & \text{for each } (u,v) \in E \\
& x(v) & \in & \{0,1\} & \text{for each } v \in V\,.
\end{array}
$$

In the special case in which all the weights $w(v)$ are equal to 1, this formulation is the optimization version of the NP-hard vertex-cover problem. Suppose, however, that we remove the constraint that $x(v) \in \{0,1\}$ and replace it by $0 \le x(v) \le 1$. We then obtain the following linear program, which is known as the ***linear-programming relaxation***:

$$
\begin{array}{ll}
\text{minimize} & \displaystyle\sum_{v \in V} w(v)\, x(v) \\
\text{subject to} & \\
& \begin{array}{rcll}
x(u) + x(v) & \geq & 1 & \text{for each } (u, v) \in E \\
x(v) & \leq & 1 & \text{for each } v \in V \\
x(v) & \geq & 0 & \text{for each } v \in V\,.
\end{array}
\end{array}
$$

```
APPROX-MIN-WEIGHT-VC$(G, w)$
$C = \emptyset$
compute $\bar{x}$, an optimal solution to the linear program   with relaxation
for each $v \in V$
    if $\bar{x}(v) \geq 1/2$
        $C = C \cup \{v\}$
return $C$
```

Algorithm APPROX-MIN-WEIGHT-VC is a polynomial-time 2-approximation algorithm for the minimum-weight vertex-cover problem.

Proof Because there is a polynomial-time algorithm to solve the linear program in line 2, and because the **for** loop of lines 3–5 runs in polynomial time, APPROX-MIN-WEIGHT-VC is a polynomial-time algorithm.

Let C* be an optimal solution to the minimum-weight vertex-cover problem, and let z* be the value of an optimal solution to the linear program with relaxation. Since an optimal vertex cover is a feasible solution to the linear program with relaxation, z* must be a lower bound on w(C*), that is,

z* $\leq w(C\,*)$

- Next, we claim that by rounding the fractional values of the variables $\bar{x}(v)$, we produce a set C that is a vertex cover and satisfies $w(C) \leq 2z^*$. To see that C is a vertex cover, consider any edge $(u,v) \in E$. we know that $\bar{x}(u) + \bar{x}(v) \geq 1$, which implies that at least one of $\bar{x}(u)$ and $\bar{x}(v)$ is at least 1/2. Therefore, at least one of u and v is included in the vertex cover, and so every edge is covered.

$$
\begin{aligned}
z^* &= \sum_{v \in V} w(v)\,\bar{x}(v) \\
&\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v)\,\bar{x}(v) \\
&\geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} \\
&= \sum_{v \in C} w(v) \cdot \frac{1}{2} \\
&= \frac{1}{2} \sum_{v \in C} w(v) \\
&= \frac{1}{2} w(C)\,.
\end{aligned}
$$

$$w(C) \leq 2z^* \leq 2w(C^*)$$