Part V

Selected Topics

§ Selected Topics:

- NP-Completeness
- Approximation Algorithms

- **NP-Completeness**
  - polynomial time
  - NP-completeness and reducibility
  - NP-complete problems

- **Approximation Algorithms**
  - Vertex-cover problem
  - Traveling-salesman problem

NP-Completeness

- **Complexity classes**
  - P: solvable in polynomial time
  - NP: verifiable in polynomial time (**accepting** a certificate, but not **deciding** solutions)
  - NPC (NP-Completeness): in NP and as "hard" as any problem in NP
  - if one NPC problem can be solved in polynomial time, every NPC problem can be solved in polynomial time too

Reduction

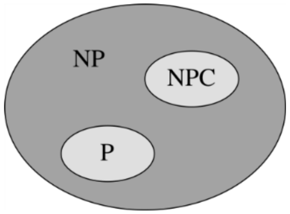
Given (1) problem *A* which has no polynomial-time algorithm and
(2) polynomial-time reduction which transforms instances of *A* to instances of *B*, then
→ no polynomial time algorithm can exist for *B* (can be proved easily by contradiction)

A is polynomial-time reducible to *B*, denoted by $A \leq_P B$; *A* is not harder than *B* by a polynomial-time factor, i.e., *B* is at least as **hard** as *A*.

NP-Completeness (continued)

- **Polynomial time**
  - Language: a set of strings (i.e., certificates) composed of symbols ∈ ∑ (alphabet)
  - Algorithm A accepts a string x if algorithm's output $A(x)$ is 1; otherwise, it rejects x
  - Language L *is decided by algorithm A if every string in L is accepted by A and every string* not in L is rejected by A

    It is harder to decide "every string" in, or not in, *L*.
  - Algorithm A accepts (i.e., verifies) a string in polynomial time, called NP; for deciding a solution, it must accept every certificate ∈ L and reject every certificate ∉ L (i.e., it accepts L) in polynomial time



Complexity class NP: a class of languages **verifiable** by a polynomial-time algorithm

$$L = \{x \in \{0, 1\}^* : \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1 \}.$$

Note:

- **NP** (nondeterministic polynomial time) refers to a set of problems for which their solutions can be **verified** quickly (in polynomial time), whereas **P** can be **solved** in polynomial time and always belongs to NP; P ≠ NP means that there are NP problems, e.g., **NPC**, unable to be solved in polynomial time (i.e., they are outside P).
- Problems of practical interest **all belong to the NP class,** ranging from simple ones which can be solved in polynomial time (i.e., in the P class covered up to Chapter 33) to hardest ones which themselves form a complete subset being NP-hard, the hardest ones in NP. Hence, NP-complete problems are the "hardest NP problems" and they constitute NPC.

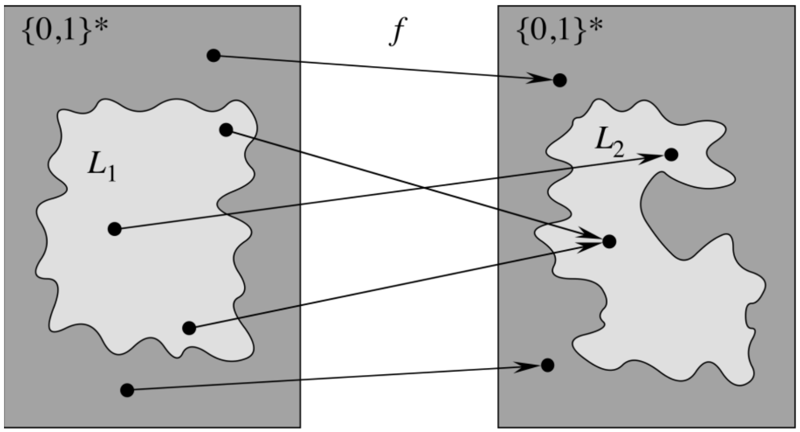
NP-Completeness (continued)

- **NP-Completeness (NPC) and Reducibility**
  - NP-complete languages are the "hardest" language in NP
  - $L_1 \leq_P L_2$: Language L_1 is reducible to L_2 via a polynomial-time computable function f such that for all $x \in \{0, 1\}^*$, $x \in L_1$ if and only if $f(x) \in L_2$
  - Language $L \subseteq \{0, 1\}^*$ is NP-complete (NPC) if
    1. $L \in$ NP and
    2. $L' \leq_P L$ for **every** $L' \in$ NP.

  Any L satisfies only Property 2 alone is NP-hard.

Lemma

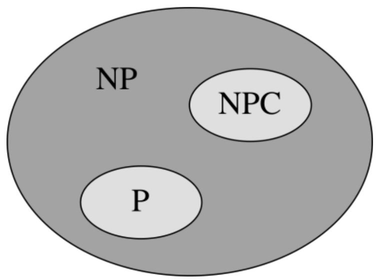
- If $L_1, L_2 \subseteq \{0, 1\}^*$ are languages such that $L_1 \leq_P L_2$ then $L_2 \in$ P implies $L_1 \in$ P.



NP-Completeness (continued)

Theorem

- If any NP-complete problem is polynomial-time solvable, we have P = NP. Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is solvable in polynomial time.



Problems of practical interest **all belong to the NP class** (i.e., nondeterministic polynomial time class), ranging from simple ones solvable in polynomial time (i.e., in the **P class**) to the hardest ones, which form **NPC**.

P $\subset$ NP and NPC $\subset$ NP with P $\cap$ NPC $= \phi$.

Lemmas

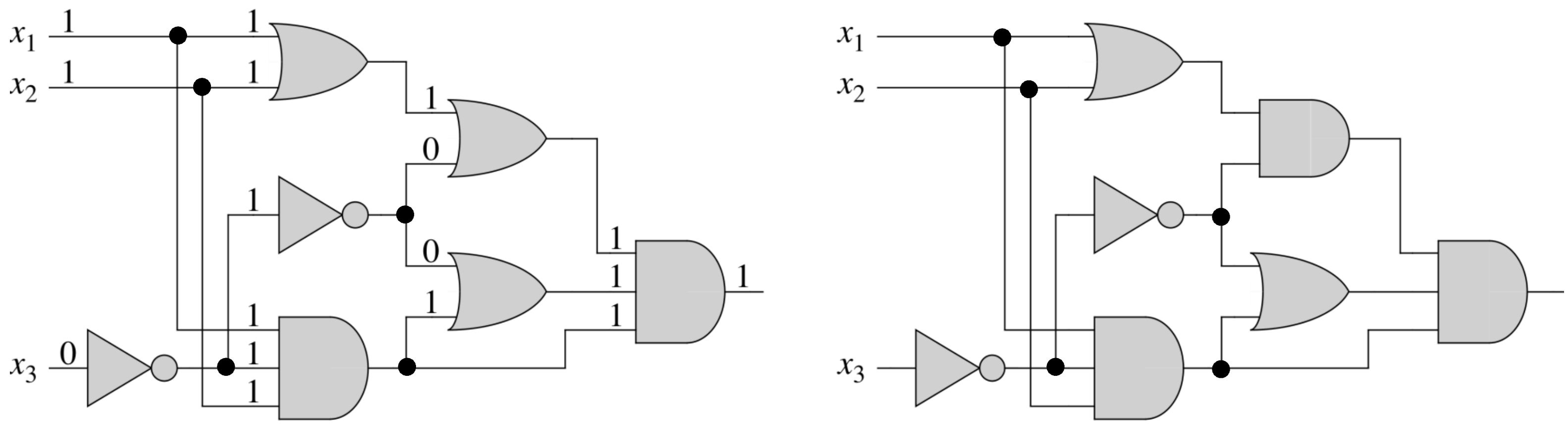
- The circuit-satisfiability (CIRCUIT-SAT) problem belongs to the complexity class of NP.
- The CIRCUIT-SAT problem is NP-hard.

NP-hardness here is proved by reducing every language in NP to CIRCUIT-SAT in polynomial time.

(Not just any language in NPC, because we don't know any NPC language yet!)

Note: the above two lemmas make CIRCUIT-SAT belong to the **NPC class**.

(The very first language proved to be in NPC.)



(a) (b)

Two instances of the <u>circuit-satisfiability</u> (CIRCUIT-SAT) problem.

(a) As the assignment of $<x_1 = 1, x_2 = 1, x_3 = 0>$ to the circuit inputs causes the output to be 1, the circuit is hence satisfiable. (b) No assignment of the three circuit inputs can cause the output to be 1, and hence, it is unsatisfiable.

NP-Completeness (continued)

- **NP-Completeness Proofs**
  - first NPC problem – CIRCUIT-SAT is proved by showing $L \leq_P$ CIRCUIT-SAT for **every language *L* in NP**
  - following lemma then simplifies NP-completeness proofs

Lemma

- If L is a language such that $L' \leq_P L$ for some $L' \in$ NPC, then L is NP-hard. Additionally, if $L \in$ NP, then $L \in$ NPC.

Theorems

- Satisfiability of Boolean formulas is NP-complete.
- Satisfiability of Boolean formulas in 3-conjunctive normal form (3-CNF) is NP-complete.

For example, a Boolean formula and a 3-CNF given below:

$$\mathcal{F}_{\mathrm{B}} = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

$$\mathcal{F}_{\text{3-CNF}} = (\underbrace{(x_1 \vee \neg x_1 \vee \neg x_2)}_{\text{OR clause}} \wedge (x_1 \vee x_3 \vee x_4) \wedge \underbrace{(\neg x_1 \vee \neg x_3 \vee \neg x_4)})$$

AND of "OR clauses", each of which involves exactly 3 literals

NP-Completeness (continued)

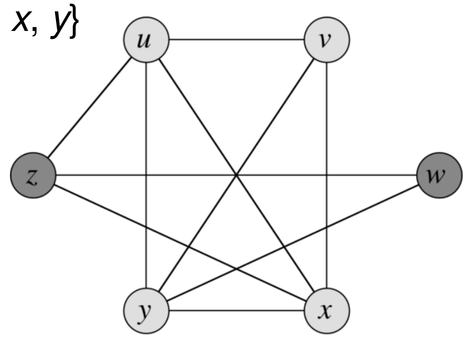
- **NP-Complete Problems**
  – problems in diverse domains
  – reduction methodology followed to provide NP-completeness proofs
  – five problems proved: clique, vertex-cover, Hamiltonian-cycle, traveling-salesman, subset-sum

Problem Definition

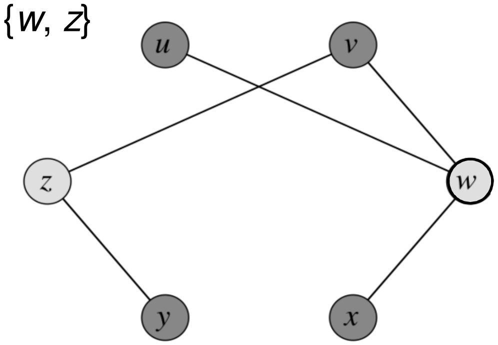
1. clique problem – given an undirected graph $G = (V, E)$, the clique problem is to find the **maximum vertex subset** $V' \subseteq V$ such that every vertex pair in V' is connected by an edge in E. (proof via 3-CNF-SAT $\leq_P$ CLIQUE)

2. vertex-cover problem – given an undirected graph $G = (V, E)$, the vertex-cover problem is to find the **minimum vertex subset** $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both). (proof via CLIQUE $\leq_P$ VERTEX-COVER)

Clique $V' = \{u, v, x, y\}$ for graph G



Vertex-cover $V' = \{w, z\}$ for graph $\overline{G}$



NP-Completeness (continued)

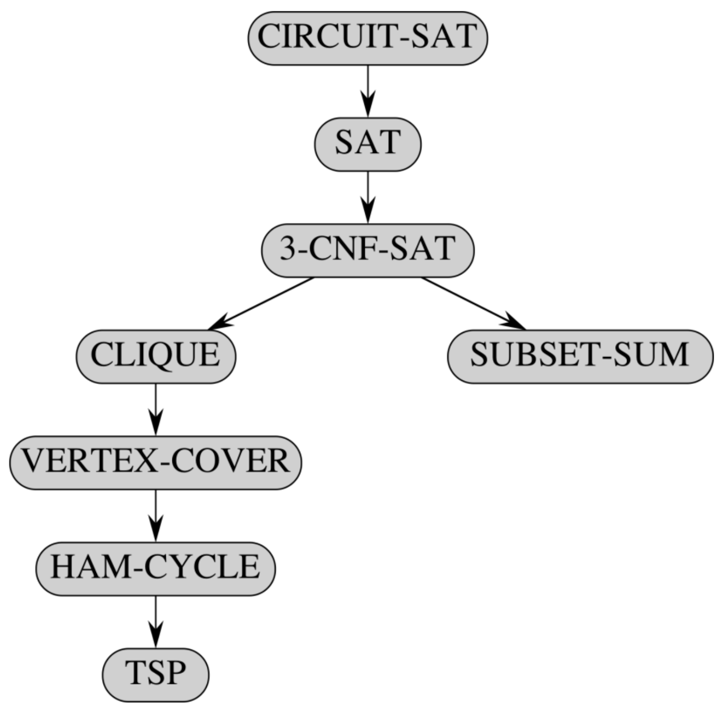
Problem Definition

3. Hamiltonian-cycle problem – given an undirected graph $G = (V, E)$, the HAM-CYCLE problem is to find a simple cycle that contains every vertex in V.

   (proof via VERTEX-COVER $\leq_P$ HAM-CYCLE)

4. traveling-salesman problem – given a complete undirected graph $G = (V, E)$, TSP is to find a minimum tour that visits each node exactly once and finishes at the starting node

5. subset-sum problem – given a finite set of S of positive integers, SUBSET-SUM finds subset of S (say, S') whose elements sum to t (> 0). (proof via 3-CNF-SAT $\leq_P$ SUBSET-SUM)



Approximation Algorithms

- **Ways to get around NP-Completeness**
  – small inputs; or important special cases solvable in polynomial time
  – near-optimal solutions in polynomial time (i.e., approximation algorithms)

Characterize Approximation

+ approximation ratio – $\rho(n)$: given an input sized n with optimal result C^* and approximate result C, we define $\rho(n)$ such that $\max(\frac{C}{C^*}, \frac{C^*}{C}) \leq \rho(n)$

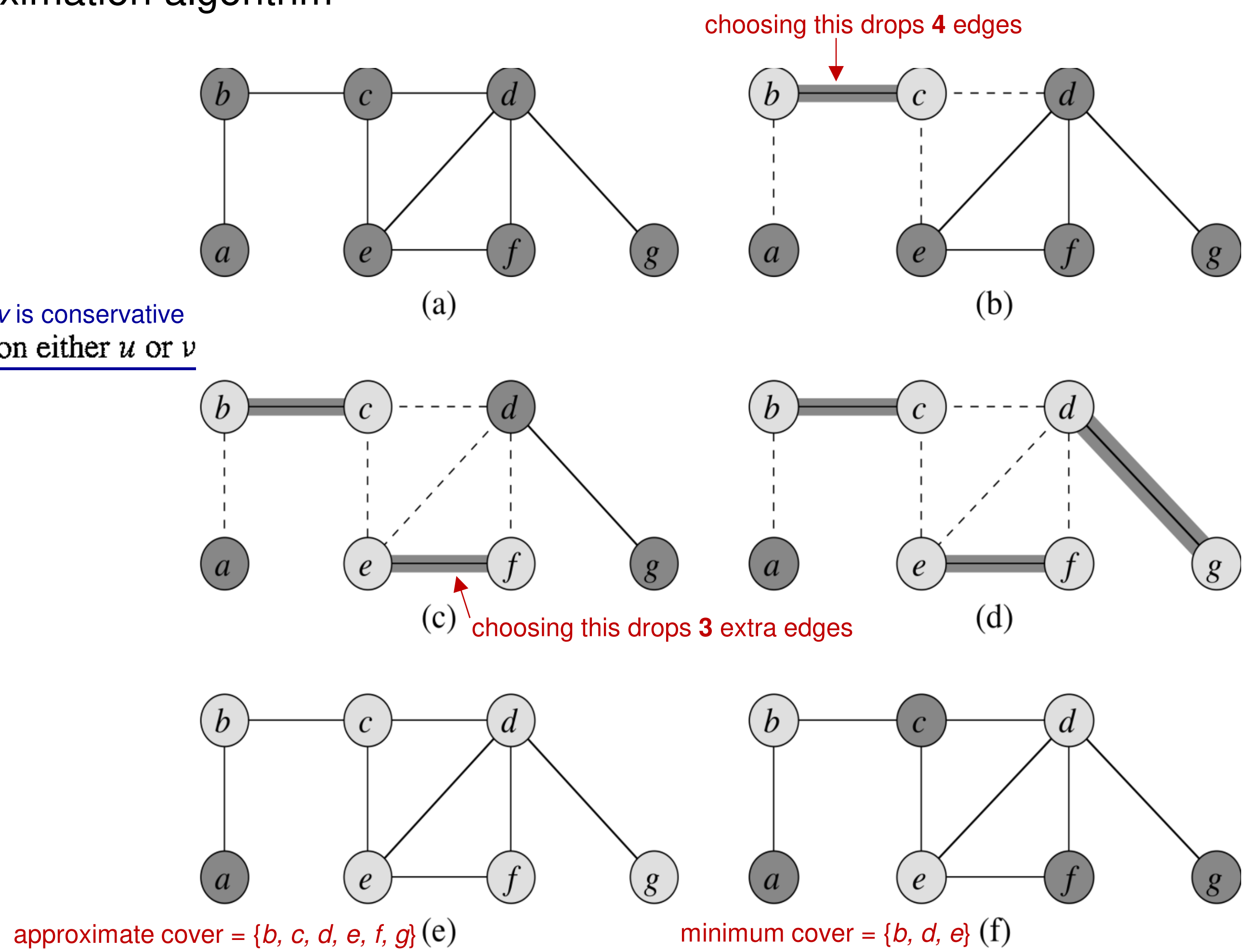
+ polynomial-time approximation algorithm with $\rho(n) = (1+\epsilon)$ for fixed $\epsilon > 0$ under input sized n

Approximation Algorithms (continued)

- **Vertex-cover problem**
  - find a minimum subset of vertices of an undirected graph $G = (V, E)$
  - a polynomial-time 2-approximation algorithm

APPROX-VERTEX-COVER(G)

$C = \emptyset$
$E' = G.E$
while $E' \neq \emptyset$
  let (u, v) be an arbitrary edge of E'
  $C = C \cup \{u, v\}$ // adding both u and v is conservative
  remove from E' every edge incident on either u or v
return C



Approximation Algorithms (continued)

Theorem

- APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm for an undirected graph $G = (V, E)$.

```
APPROX-VERTEX-COVER(G)
1  C = ∅
2  E′ = G.E
3  while E′ ≠ ∅
4      let (u, v) be an arbitrary edge of E′
5      C = C ∪ {u, v}
6      remove from E′ every edge incident on either u or v
7  return C
```

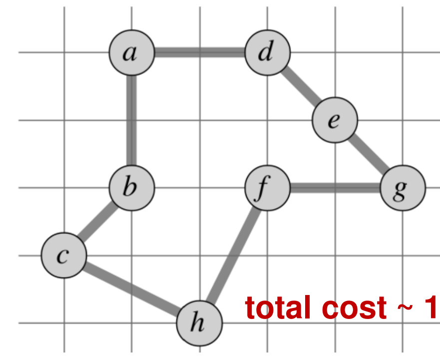
Proof sketch:

For A being the set of edges chosen by Statement 4, we have $|C| = 2|A|$ as each edge in A involves *two unique vertexes* not shared with any other edge in A.

For C^* being an optimal cover (of vertexes), we have $|C^*| \geq |A|$, since *no two edges* in A share one common vertex. Hence, $|C| \leq 2|C^*|$, the proof.

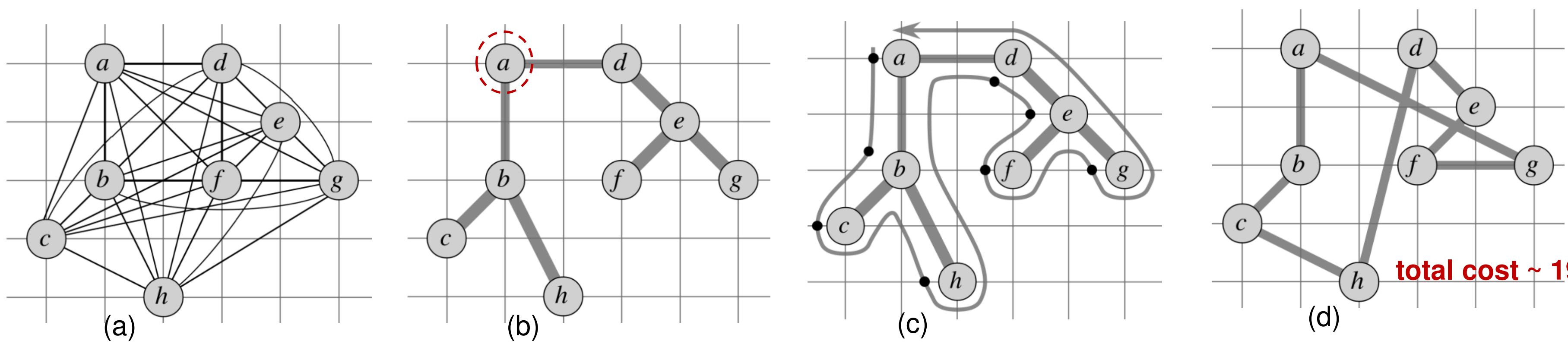
Approximation Algorithms (continued)

- **Traveling-salesman problem**
  - $c(A)$ be total cost of the edges in $A \subseteq E$ under complete undirected weighted graph $G = (V, E)$, i.e., $c(A) = \sum_{(u,v)\in A} c(u, v)$
  - if the cost function c satisfies the triangle inequality (i.e., $c(u, w) \le c(u, v)+c(v, w)$), a polynomial-time 2-approximation algorithm based on MST-PRIM exists




```
APPROX-TSP-TOUR(G, c)
select a vertex r ∈ G.V to be a "root" vertex
compute a minimum spanning tree T for G from root r
    using MST-PRIM(G, c, r)          // see Ch. 23.2
let H be a list of vertices, ordered according to when they are first visited
    in a preorder tree walk of T
return the hamiltonian cycle H
```

preorder tree walk (over each edge **twice**): $\{a, b, c, b, h, b, a, d, e, f, e, g, e, d, a\}$ → drop all re-visited vertexes ≠ a to get $\{a, b, c, h, d, e, f, g, a\}$

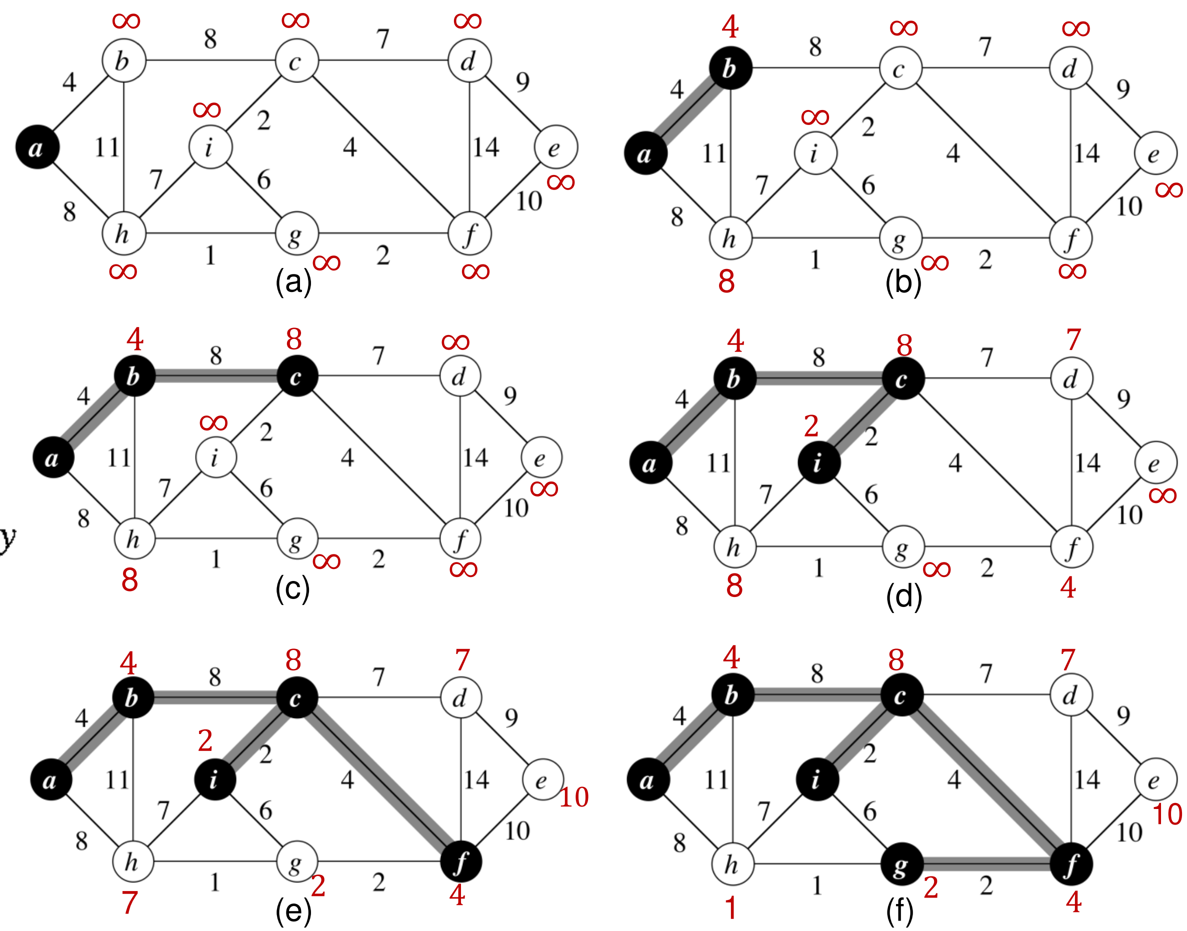


Approximation Algorithms (continued)

- **Prim's algorithm for MST** (minimum spanning trees)
  - *greedy algorithm* by selecting examined edge with smallest weight to add to the tree
  - *v.key* denotes the minimum weight from *v* to the tree established so far, with *v.key* values of all vertices (other than the root) initialized to ∞

MST-PRIM(G, w, r)

```
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for one v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

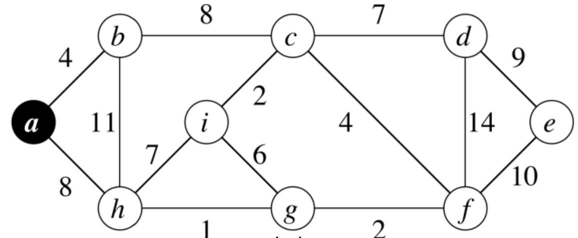


Approximation Algorithms (continued)

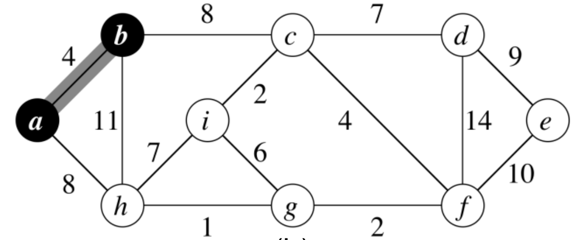
- **Prim's algorithm for MST** (minimum spanning trees)

```
MST-PRIM(G, w, r)
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
  r.key = 0
  Q = G.V
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for one v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              v.key = w(u, v)
```

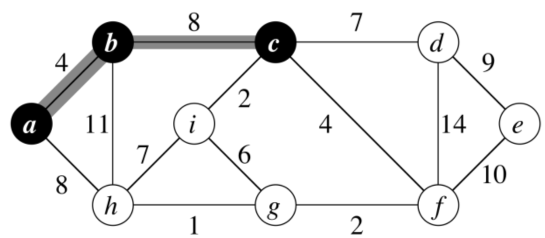
vertices in *G.V–Q* already included in the tree



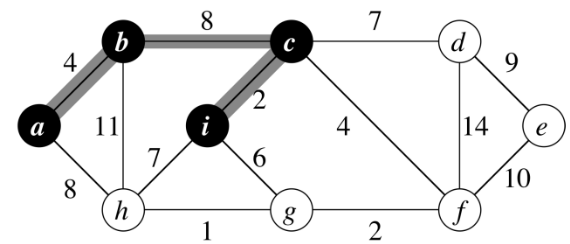
(a)



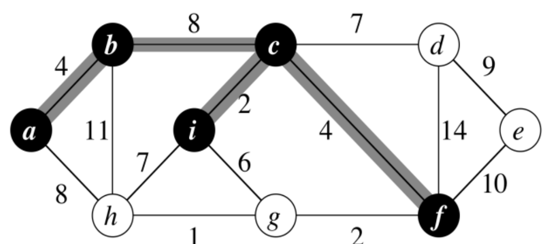
(b)



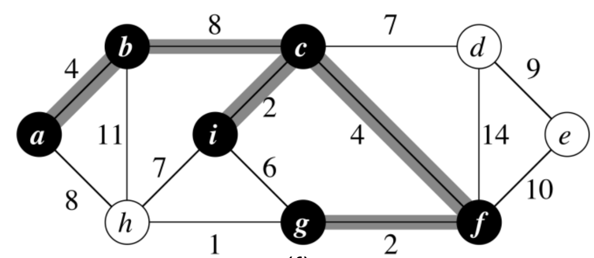
(c)



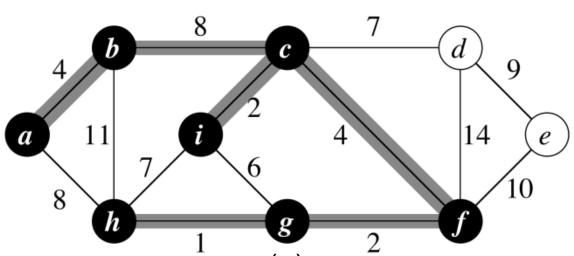
(d)



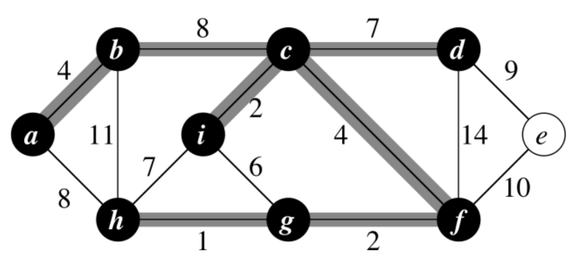
(e)



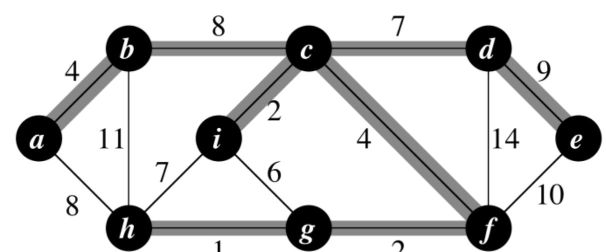
(f)



(g)



(h)



(i)

Approximation Algorithms (continued)

- **Traveling-salesman problem**

Theorem

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

Proof sketch:

Let T and H^* be the minimum spanning tree obtained and an optimal tour, respectively; we have $c(T) \le c(H^*)$.

Let W be a full tree walk (which traverses **each edge twice** exactly), we have $c(W) = 2c(T)$, implying $c(W) \le 2c(H^*)$.

Since a cycle H can be obtained from any full walk by dropping revisited vertexes, we have $c(H) \le c(W)$ according to the triangle inequality. Hence, $c(H) \le 2c(H^*)$, the proof.

Theorem

- For general TSP, no polynomial-time approximation algorithm exists with a fixed approximation ratio ρ (≥ 1).

```
APPROX-TSP-TOUR(G, c)
select a vertex r ∈ G.V to be a "root" vertex
compute a minimum spanning tree T for G from root r
    using MST-PRIM(G, c, r)
let H be a list of vertices, ordered according to when they are first visited
    in a preorder tree walk of T
return the hamiltonian cycle H
```