


# EECE7205: Fundamentals of Computer Engineering



NP-Completeness and  
Approximation Algorithms



# Introduction

- Almost all the algorithms we have studied thus far have been ***polynomial-time algorithms***: on inputs of size  $n$ , their worst-case running time is  $O(n^k)$  for some constant  $k$ .
- Not *all* problems can be solved in polynomial time.
  - Problems that are solvable by polynomial-time algorithms are tractable.
  - Problems that require exponential time are intractable and they cannot be solved in a reasonable amount of time unless for very small  $n$ .
  - Problems solvable in logarithmic time are solvable in polynomial time as well.
- The status of the “Nondeterministic Polynomial time” or “NP-complete” problems is unknown.
  - No polynomial-time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them.



# NP-Complete Problem Example

- Several NP-complete problems seem on the surface to be similar to problems that we know how to solve in polynomial time.
- Many natural and interesting problems that seem no harder than sorting, graph searching, or network flow are in fact NP-complete.
- Example: longest simple paths problem:
  - We studied that we could find shortest paths from a single source in a directed graph  $G = (V, E)$  in  $O(VE)$  time.
  - Finding a longest simple path between two vertices is difficult. Merely determining whether a graph contains a simple path with at least a given number of edges is NP-complete.



# NP-Complete Class of Problems

- If *any* NP-complete problem can be solved in polynomial time, then *every* problem in the *NP* class can have a polynomial time algorithm.
- As an engineer, you would then do better to spend your time developing an approximation algorithm or solving a tractable special case, rather than searching for a fast algorithm that solves the problem exactly.
- *Conclusion:* you should become familiar with this remarkable class of problems.



# Showing Problems to be NP-Complete

- When we demonstrate that a problem is NP-complete, we are making a statement about how hard it is.
- We are not trying to prove the existence of an efficient algorithm, but instead that no efficient algorithm is likely to exist.
- Three key concepts are used in showing a problem to be NP-complete:
  - Decision problems vs. optimization problems
  - Reductions
  - A first NP-complete problem



# Decision Problems vs. Optimization Problems

- NP-completeness applies directly not to optimization problems, however, but to ***decision problems***, in which the answer is simply “yes” or “no”.
- We can view an abstract decision problem as a function that maps a set  $I$  of problem *instances* to the solution set  $\{0, 1\}$ .
- *Examples:*
  - Optimization problem: given an undirected graph  $G$  and vertices  $u$  and  $v$ , and we wish to find a path from  $u$  to  $v$  that uses the fewest edges.
  - Decision problem: given a directed graph  $G$ , vertices  $u$  and  $v$ , and an integer  $k$ , does a path exist from  $u$  to  $v$  consisting of at most  $k$  edges?



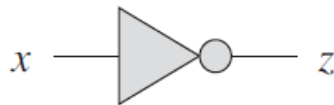
# A First NP-Complete Problem

- The technique of **reduction** relies on having a problem already known to be NP-complete in order to prove a different problem is NP-complete. Therefore, we need a “first” NP-complete problem.
- The problem used in the book is the circuit-satisfiability problem, in which we are given a Boolean combinational circuit composed of AND, OR, and NOT gates, and we wish to know whether there exists some set of Boolean inputs to this circuit that causes its output to be 1.

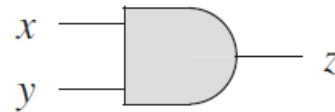


# The Circuit-Satisfiability Problem (1 of 4)

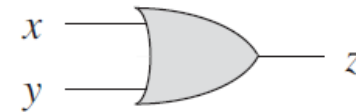
- The formal proof that the circuit-satisfiability problem is NP-complete requires technical detail beyond the scope of this course. Instead, we shall informally describe a proof that relies on a basic understanding of Boolean combinational circuits.
- A **Boolean combinational element** is any circuit element that has a constant number of Boolean inputs and outputs and that performs a well-defined function.
- The following are the Boolean combinational elements that we use in the circuit-satisfiability problem:



$x$	$\neg x$
0	1
1	0



$x$	$y$	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1



$x$	$y$	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1





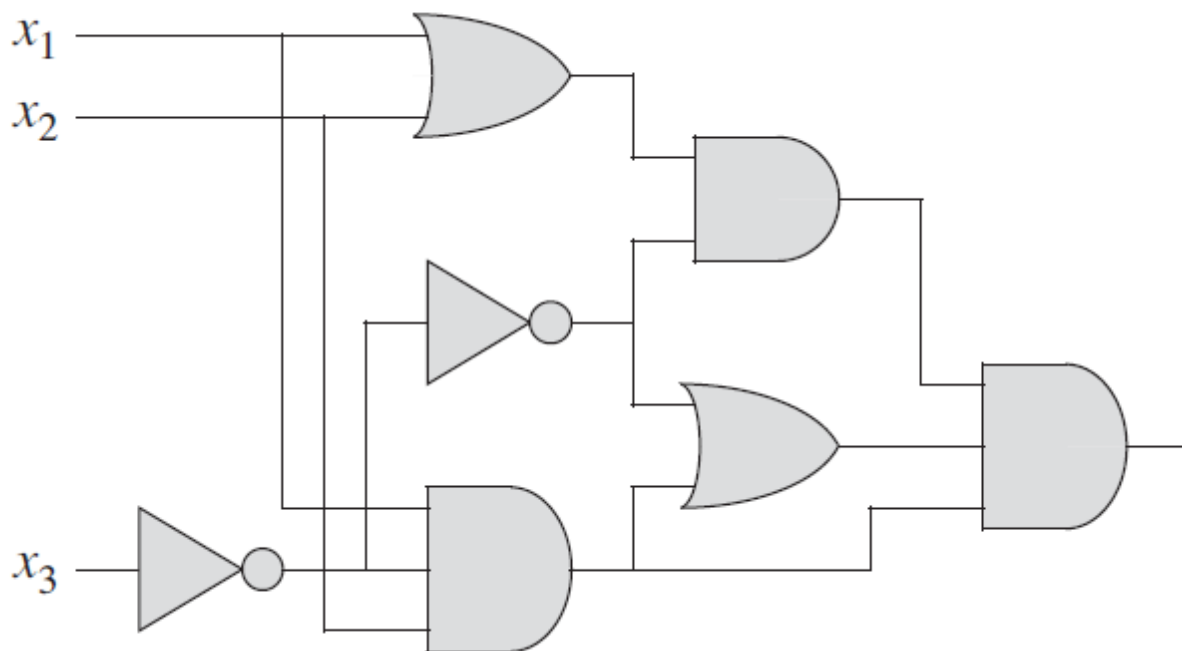
# Circuit Satisfiability (2 of 4)

- The shown circuit is a sample Boolean combinational circuit.
  - Boolean combinational circuits contain no cycles.
  - A **truth assignment** for a Boolean combinational circuit is a set of Boolean input values.
- 
- We say that a one-output Boolean combinational circuit is **satisfiable** if it has a **satisfying assignment**: a truth assignment that causes the output of the circuit to be 1.
    - For example, the shown circuit has the satisfying assignment ( $x_1 = 1$ ,  $x_2 = 1$ ,  $x_3 = 0$ ), and so it is satisfiable.



# Circuit Satisfiability (3 of 4)

- No assignment of values to  $x_1$ ,  $x_2$ , and  $x_3$  causes the following circuit to produce a 1 output as it always produces 0, and so it is unsatisfiable. Prove that!





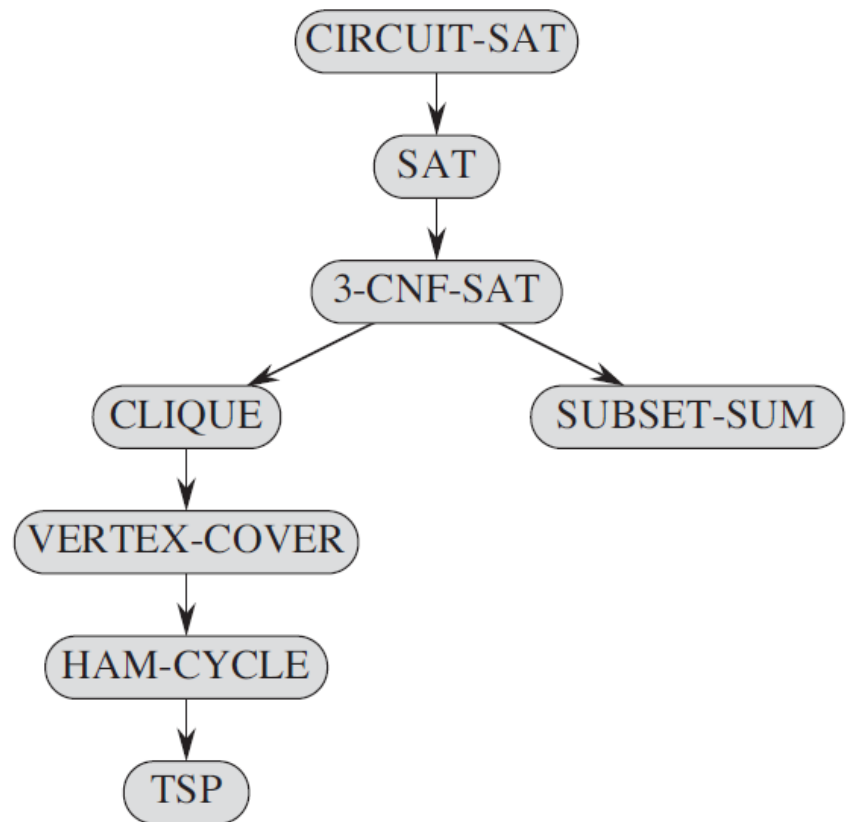
# Circuit Satisfiability (4 of 4)

- The ***circuit-satisfiability problem*** is, “Given a Boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?”
- Given a circuit  $C$ , we might attempt to determine whether it is satisfiable by simply checking all possible assignments to the inputs. Unfortunately, if the circuit has  $k$  inputs, then we would have to check up to  $2^k$  possible assignments.
- If the size of  $C$  is polynomial in  $k$ , checking each one takes  $\Omega(2^k)$  time, which is exponential in the size of the circuit.
- No polynomial-time algorithm exists that solves the circuit satisfiability problem because circuit satisfiability is NP-complete.



# Examples of NP-complete Problems

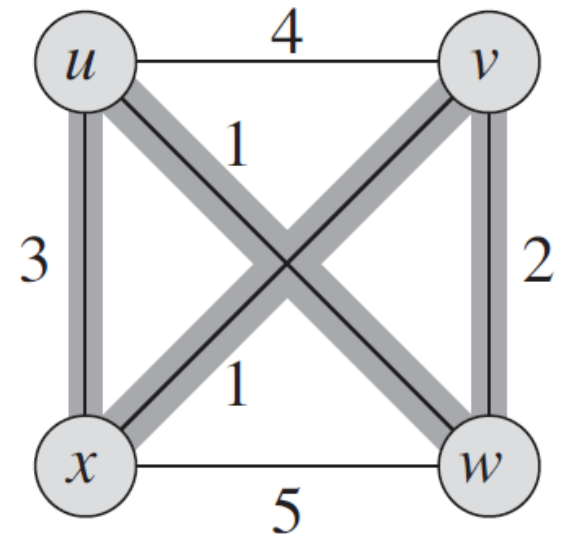
- NP-complete problems arise in diverse domains: Boolean logic, graphs, arithmetic, network design, sets and partitions, storage and retrieval, sequencing and scheduling, mathematical programming, algebra and number theory, games and puzzles, automata and language theory, program optimization, biology, chemistry, physics, and more.
- The figure outlines the structure of the NP-completeness proofs in the book. It shows that the proof of each problem in the figure to be NP-complete by reduction from the problem that points to it. At the root is CIRCUIT-SAT, which we proved NP-complete.





# The Traveling-Salesperson Problem

- The salesperson wishes to make a **tour** visiting each city exactly once and finishing at the city she/he starts from.
- The salesperson incurs a nonnegative integer cost  $c(i, j)$  to travel from city  $i$  to city  $j$ , and the salesperson wishes to make the tour with a total cost that is minimum.
  - Total cost is the sum of the individual costs along the edges of the tour.
- In the figure a minimum-cost tour is  $(u, w, v, x, u)$ , with cost 7.
- The traveling-salesperson problem (TSP) is NP-complete.
  - Thus, we should not expect to find a polynomial-time algorithm for solving this problem exactly. Instead, we look for good approximation algorithms.





# Approximation Algorithms

- Many problems of practical significance are NP-complete, yet they are too important to abandon merely because we don't know how to find an optimal solution in polynomial time.
- There are at least three ways to get around NP-completeness:
  1. If the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory.
  2. We may be able to isolate important special cases that we can solve in polynomial time.
  3. We might come up with approaches to find near-optimal solutions in polynomial time. In practice, near optimality is often good enough.
- We call an algorithm that returns near-optimal solutions an ***approximation algorithm***.



# The TSP Approximation

- In the traveling-salesperson 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 need to find a minimum cost cycle that begins at vertex  $v$  passes through every vertex exactly once and terminates at  $v$
- The following is a TSP approximation algorithm:

APPROX-TSP-TOUR( $G, c$ )

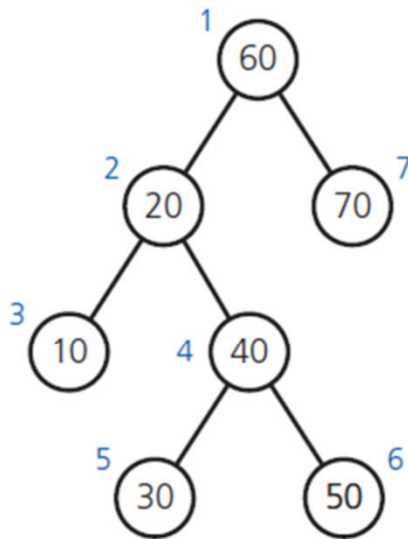
- 1 select a vertex  $r \in G.V$  to be a “root” vertex
- 2 compute a minimum spanning tree  $T$  for  $G$  from root  $r$   
using MST-PRIM( $G, c, r$ )
- 3 let  $H$  be a list of vertices, ordered according to when they are first visited  
in a preorder tree walk of  $T$
- 4 **return** cycle  $H$



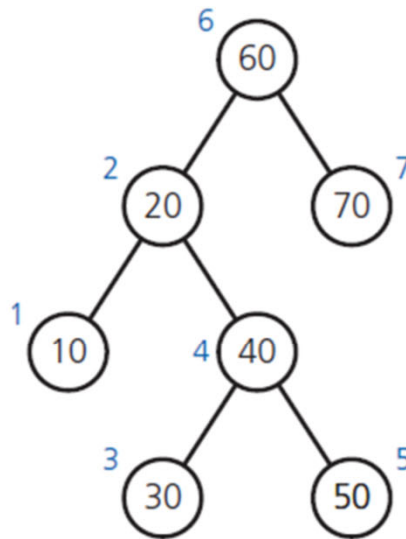


# Trees Traversal Options

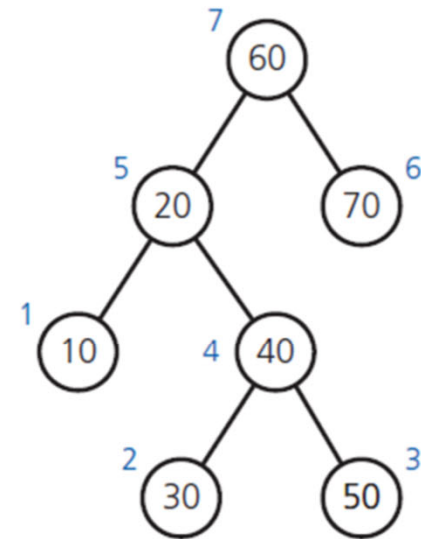
- Options for when to visit the **root**
  - **Preorder**: before it traverses both subtrees
  - **Inorder**: after it traverses left subtree, before it traverses right subtree
  - **Postorder**: after it traverses both subtrees
- Note traversal is  $O(n)$



(a) Preorder: 60, 20, 10, 40, 30, 50, 70



(b) Inorder: 10, 20, 30, 40, 50, 60, 70



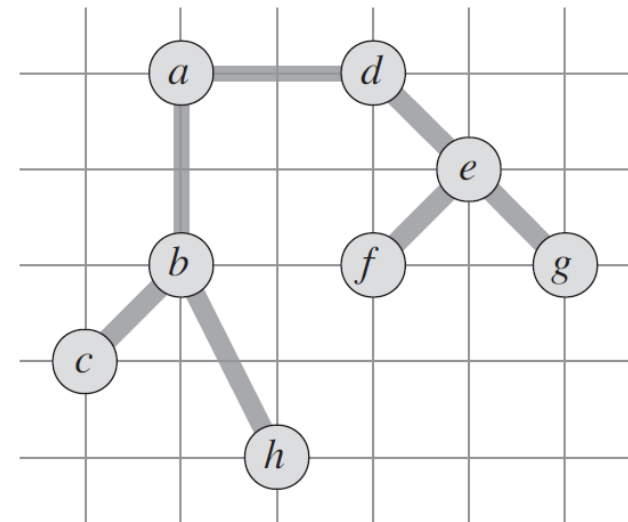
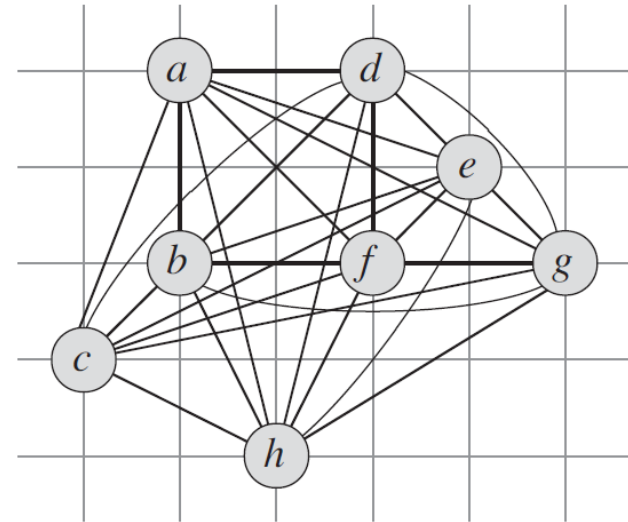
(c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)



# APPROX-TSP-TOUR Example (1 of 3)

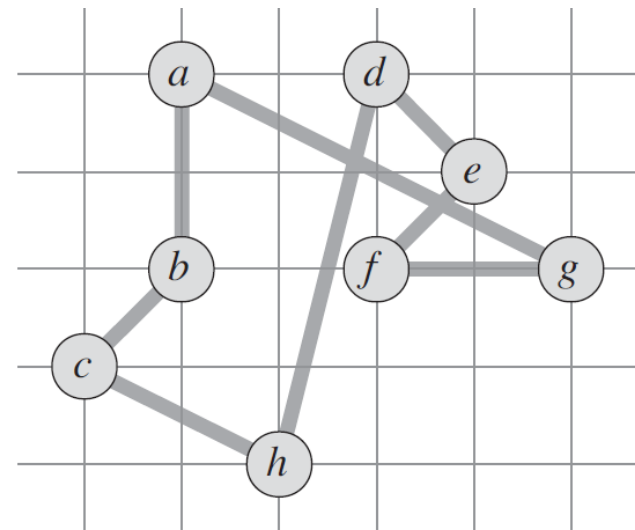
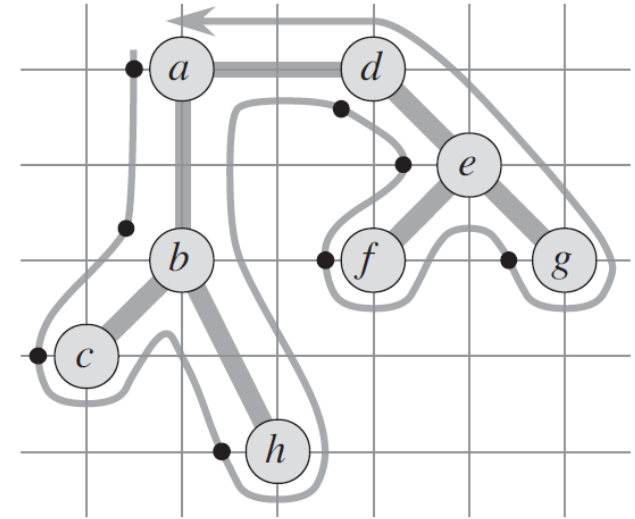
- A complete undirected graph. Vertices lie on intersections of integer grid lines.
- A minimum spanning tree  $T$  of the complete graph, as computed by MST-PRIM. Vertex  $a$  is the root vertex.
- The weight of this MST gives a lower bound on the length of an optimal traveling-salesperson tour.





# Example (2 of 3)

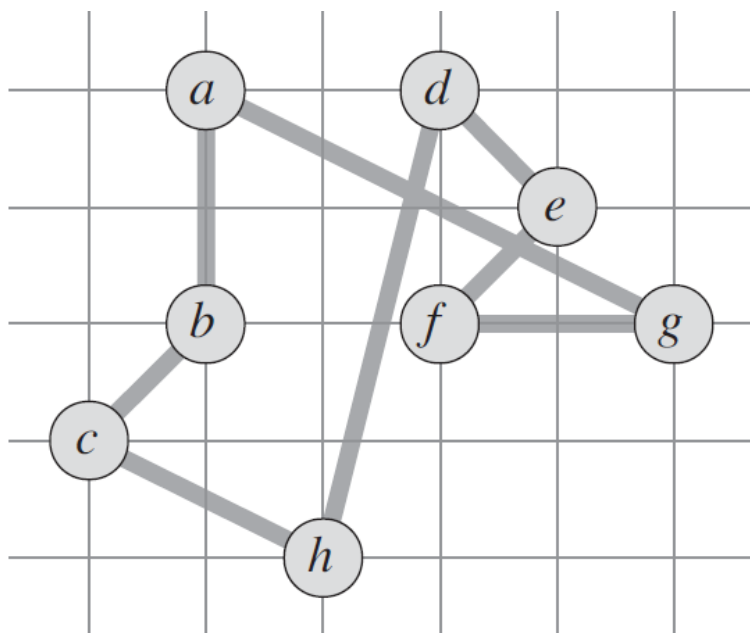
- A walk of  $T$ , starting at  $a$ . A full walk of the tree visits the vertices in the order  $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$ . A **preorder** walk of  $T$  lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering  $a, b, c, h, d, e, f, g$ .
- A tour obtained by visiting the vertices in the order given by the preorder walk, which is the tour  $H$  returned by APPROX-TSP-TOUR. Its total cost is approximately 19.



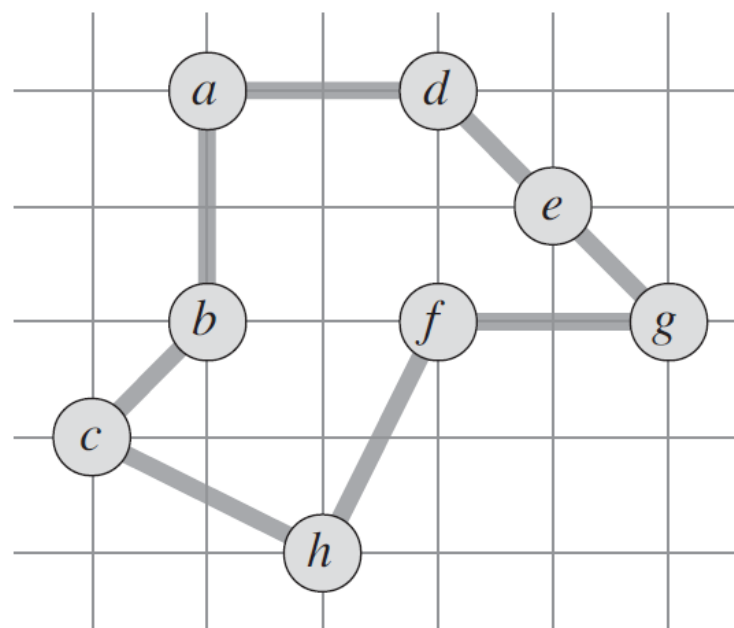


# Example (3 of 3)

- An optimal tour  $H^*$  for the original complete graph. Its total cost is approximately 14.7, which is about 23% shorter than the approximate solution.



Cost  $\approx 19$



Cost  $\approx 14.7$



# TSP and the Circuits Problem

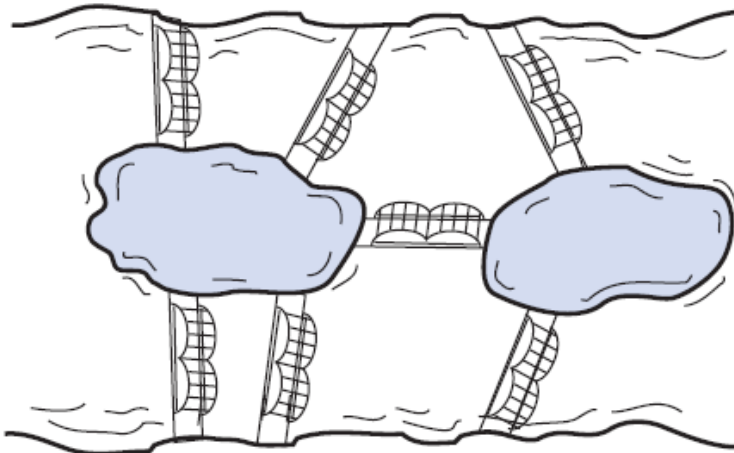
- The TSP problem belongs to the general category of problems to find **circuit** or cycles in a graph.
- A cycle in a graph is a path that **begins and ends at the same vertex**.
- Two types of circuits:
  1. Hamilton Circuit: visit every vertex once
  2. Euler Circuit: visit every edge once.
- The TSP problem is a special case of the Hamilton Circuit as it requires a circuit with a minimum cost.



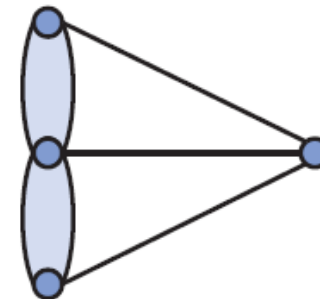
# Euler Circuit (1 of 2)

- In 1700s Euler proposed a bridge problem.
- Two islands in a river are joined to each other and to the riverbanks by several bridges, as shown in figure (a)
- The bridges correspond to the edges in the graph in figure (b) and the land masses correspond to the vertices.
- The problem asked whether you could **begin at any vertex  $v$ , pass through every edge exactly once, and terminate at  $v$** .
- Euler demonstrated that no solution exists for this particular configuration of edges and vertices.

(a)



(b)

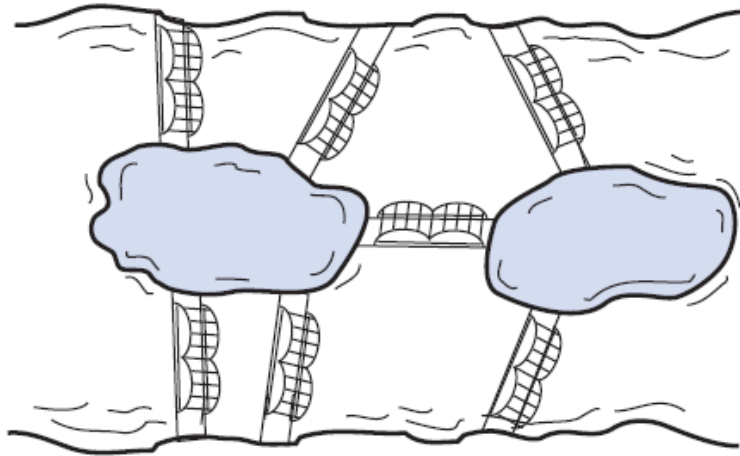




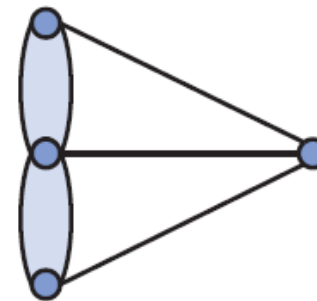
# Euler Circuit (2 of 2)

- A path in an undirected graph that begins at any vertex  $v$ , passes through every edge in the graph exactly once, and terminates at  $v$  is called an **Euler circuit**.
  - It can be used to check for the graph's connectivity.
- Euler showed that an Euler circuit exists if and only if each vertex touches an **even number** of edges.

(a)



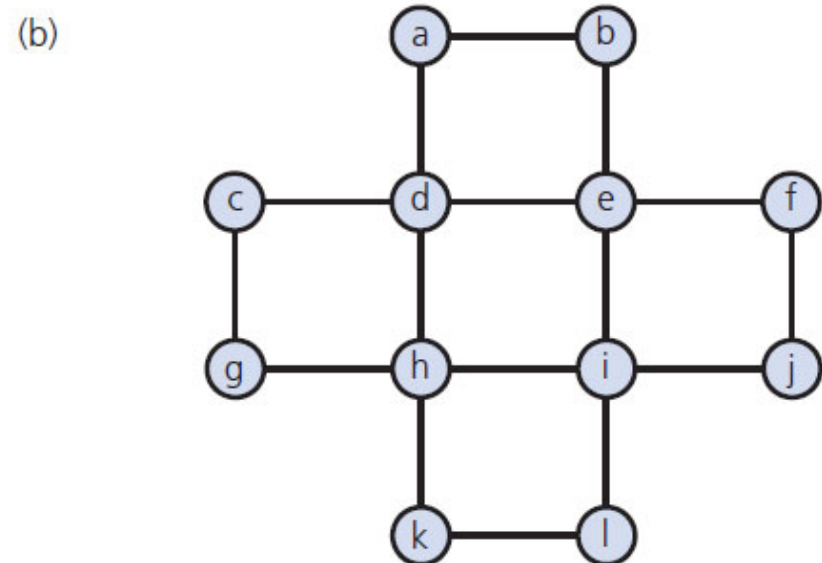
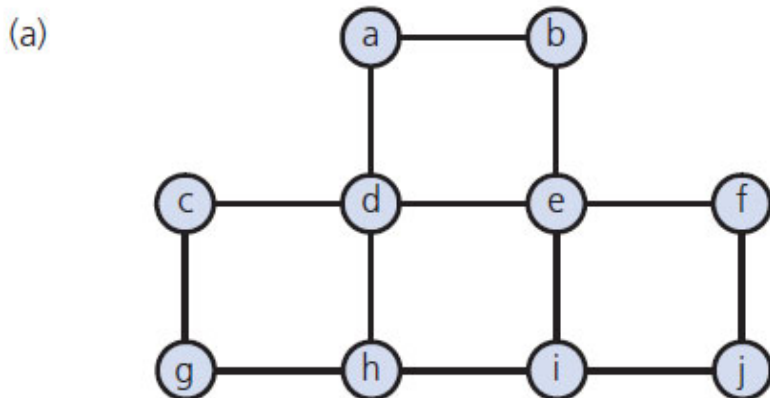
(b)





# Euler Circuit Example (1 of 2)

- Finding an Euler circuit is like drawing the following diagrams without lifting your pencil or redrawing a line and ending at your starting point.
- For diagram (a), vertices *h* and *i* each touch an odd number of edges (three), so no Euler circuit is possible.
  - Note that there are sequences where you can pass through every edge once but not ending at the starting vertex. Example: *h g c d a b e d h i e f j i*
- On the other hand, each vertex in diagram (b) touches an even number of edges, making an Euler circuit feasible.





# Euler Circuit Example (2 of 2)

1. The strategy is to use a depth-first search that marks edges (visited/unvisited) instead of vertices. Break ties using alphabetical order.
2. Stop when you have a cycle. To continue, find the first vertex along the cycle that touches an unvisited edge.
3. Repeat applying depth-first search from that vertex and repeat step 2.

■ Euler circuit is:

```
a b e f j i h d
c g h k l i e d a
```

