Chapter 6

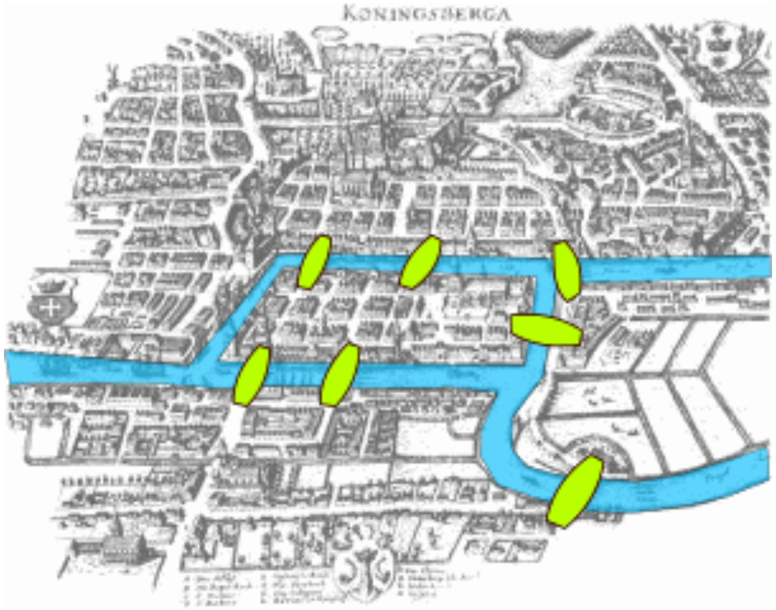
Introduction to Graphs

The Main Topics Covered

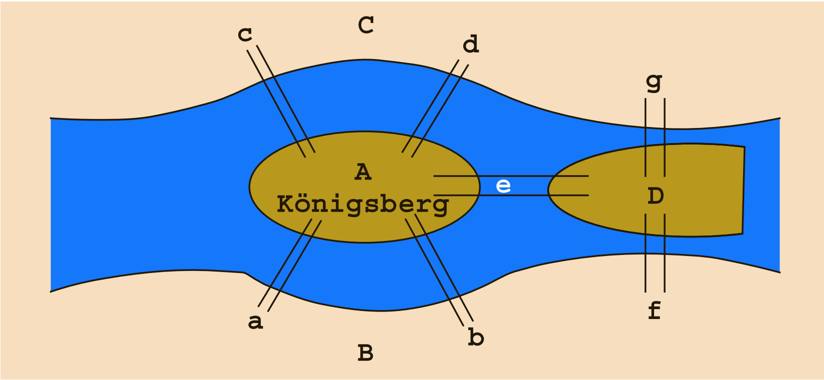
- Learn about graphs
- Become familiar with the basic terminology of graph theory
- Discover how to represent graphs in computer memory
- Examine and implement various graph traversal/search algorithms
- Examine and implement some applications of graphs

Introduction

- In 1736, in the town of Königsberg in Prussia, the river Pregel flows around the island Kneiphof and then divides into two branches



Introduction

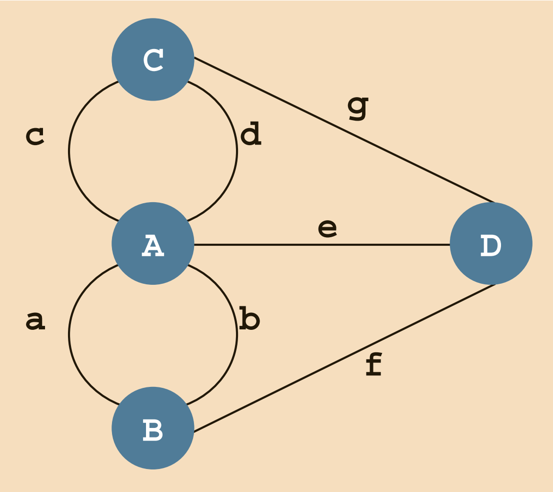


- The river has four land areas: A, B, C, and D
- These land areas are connected using seven bridges that are labeled *a*, *b*, *c*, *d*, *e*, *f*, and *g*
- *The Königsberg bridge problem*

  Starting at one land area, is it possible to walk across all of the bridges exactly once and return to the starting land area?

Introduction

- In 1736, Euler represented the *Königsberg bridge problem* as a *graph* …



… and answered the question in the negative

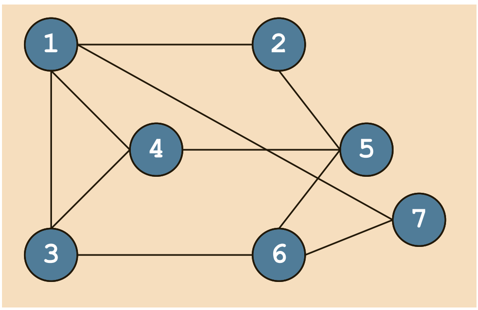
➡️ This marked the birth of *graph theory*

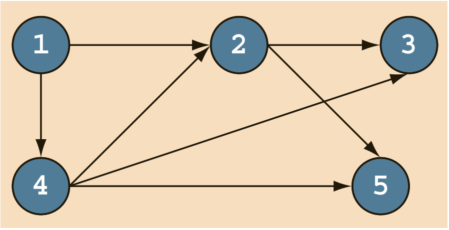
Graph Definitions and Notations

- A *graph* G is a pair, $G = (V, E)$, in which V is the set of vertices of G, and E is the set of edges in G
- If the elements of E are ordered pairs, G is called a *directed graph* (or *digraph*); otherwise, G is called an *undirected graph*
  - In an undirected graph, the pairs (u, v) and (v, u) represent the same edge
  - If (u, v) is an edge in a digraph, the vertex u is called the *origin* of the edge, and the vertex v is called the *destination*

Graph Definitions and Notations

- A graph can be shown pictorially





- A graph is called a *weighted graph* if its edges are labelled with numeric values
- A graph $H = (V_H, E_H)$ is called a *subgraph* of G if

$$V_H \subseteq V, E_H \subseteq E$$

Graph Definitions and Notations

Let G be an undirected graph; u, v be two vertices of G

- u and v are called *adjacent* if there is an edge from one to the other; that is, $(u, v) \in E$
- Let $e = (u, v) \in E$. Edge e is *incident on* the vertices u and v
- An edge incident on a single vertex is called a *loop*
- If two edges, e_1 and e_2, are associated with the same pair of vertices, then e_1 and e_2 are called *parallel edges*

Graph Definitions and Notations

- A graph is called a *simple graph* if it has no loops and no parallel edges
- There is a *path* from u to v if there is a sequence of vertices $u_1, u_2, \ldots, u_n$ such that $u = u_1, v = u_n$, and (u_i, u_{i+1}) is an edge for all $i = 1,2, \ldots, n-1$
- A *simple path* from u to v is a path from u to v with no repeated vertices
- A *cycle* is a simple path in which the first and last vertices are the same

Graph Definitions and Notations

- Vertices u and v are called *connected* if there is a path from u to v
- G is called *connected* if there is a path from any vertex to any other vertex
  - A subset of connected vertices is called a *connected component* of G

Graph Definitions and Notations

Let G be a directed graph; u, v be two vertices in G

- If $(u, v) \in E$ then we say that u is *adjacent to* v and v is *adjacent from* u
- G is called *strongly connected* if any two vertices in G are connected
  - A strongly connected component of G is a maximal strongly connected subgraph
- The *outdegree* of v is the number of directed edges leaving v; the *indegree* of v is the number of directed edges entering v

Graph Representation

- A graph can be represented in several ways
  - How a graph is represented in memory depends on the specific application
- The most two common methods are *adjacency matrices* and *adjacency lists*
- Let $G = (V, E)$ be a graph with $n(= |V|)$ vertices
  - Let $V = \{v_1, v_2, \dots, v_n\}$

Graph Representation: Adjacency Matrices

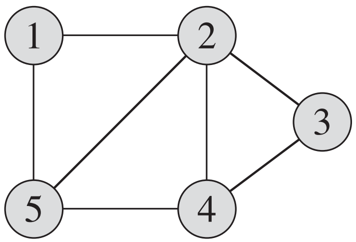
- The *adjacency matrix* A of G is a two-dimensional $n \times n$ matrix such that
  - If there is an edge from v_i to v_j, the $(i, j)^{th}$ entry of A is 1
  - Otherwise, the $(i, j)^{th}$ entry is zero
- In an undirected graph, if $\left(v_i, v_j\right) \in E$ then $\left(v_j, v_i\right) \in E$, so the $(i, j)^{th}$ entry is as same as the $(j, i)^{th}$ entry

➡️ The adjacency matrix of an undirected graph is symmetric

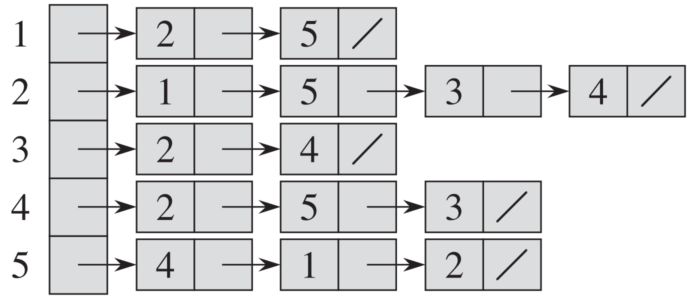
Graph Representation: Adjacency Lists

- With *adjacency lists*, corresponding to each vertex, u, there is a linked list such that each node of the linked list contains the vertex, v, such that $(u, v) \in E$
- Technically, we use an array A of size n, such that $A[i]$ is
  - a representation of the vertex v_i
  - a *pointer* to the first node of the linked list containing the vertices *to which* v_i *is adjacent*

Graph Representation: Examples



(a)

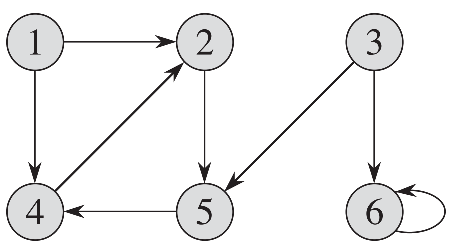


(b)

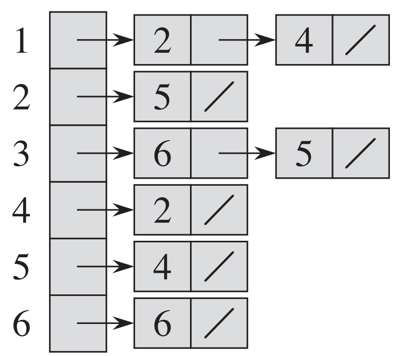
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Two representations of an undirected graph



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

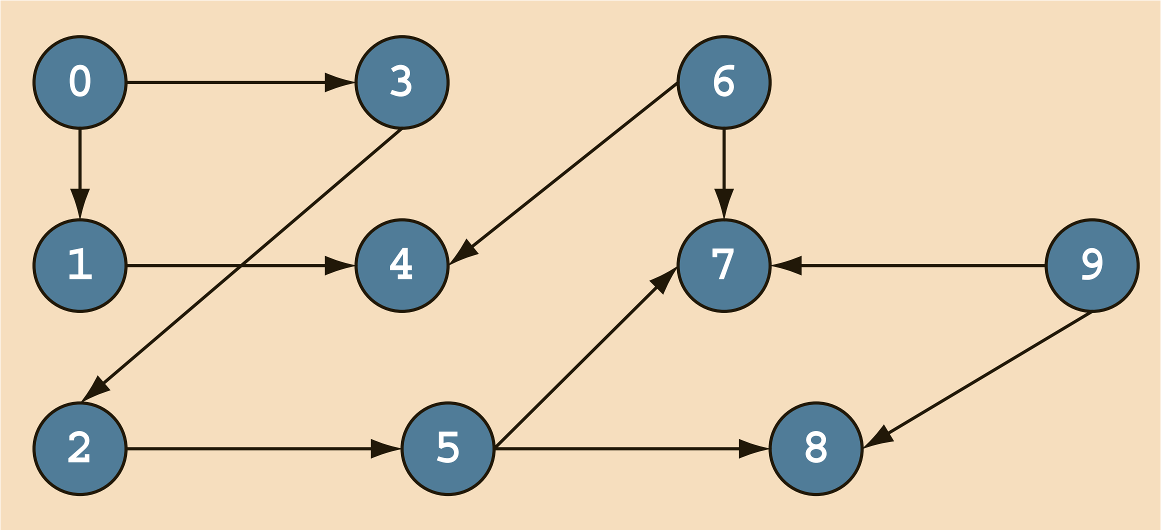
(c)

Two representations of a directed graph

Graph Traversals

- Traversing a graph is a bit more complicated than traversing a binary tree
  - A binary tree has no cycles, but a graph might have cycles
  - We might not be able to traverse the entire graph from a single vertex
- In order to traverse the entire graph, we must ...
  - keep track of the vertices that have been visited
  - traverse the graph from each unvisited vertex

Depth First Traversal/Search



A *depth first ordering* of the vertices of the graph is

0, 1, 4, 3, 2, 5, 7, 8, 6, 9

Depth First Traversal: A Non-recursive Algorithm

// the DFT starts at the vertex v

Mark $\boldsymbol{v}$ as visited

Push $\boldsymbol{v}$ onto the stack

while the stack is not empty

  Pop $\boldsymbol{u}$ off the stack

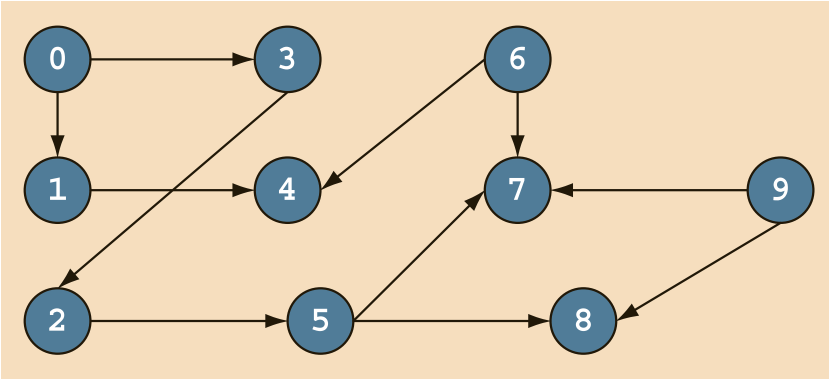
  Visit the vertex

  for each vertex $\boldsymbol{w}$ adjacent to $\boldsymbol{u}$

    if $\boldsymbol{w}$ is not visited

      Mark $\boldsymbol{w}$ as visited

      Push $\boldsymbol{w}$ onto the stack



Depth First Traversal: Pseudo-code

```
dft(graph, vertex, visited) {
  visited[vertex] = true;
  push(S, vertex);
  while (!isEmpty(S)) {
    vertex = pop(S);
    cout << vertex;
    p = graph[vertex];
    while (p) {
      if (!visited[p->vertex]) {
        visited[p->vertex] = true;
        push(S, p->vertex);
      }
      p = p->next;
    }
  }
}
```

Depth First Traversal: A Recursive Algorithm

// the DFT starts at the vertex v

Mark $\boldsymbol{v}$ as visited

Visit the vertex

for each vertex $\boldsymbol{u}$ adjacent to $\boldsymbol{v}$

    if $\boldsymbol{u}$ is not visited

        Start the DFT at $\boldsymbol{u}$

Depth First Traversal: Pseudo-code

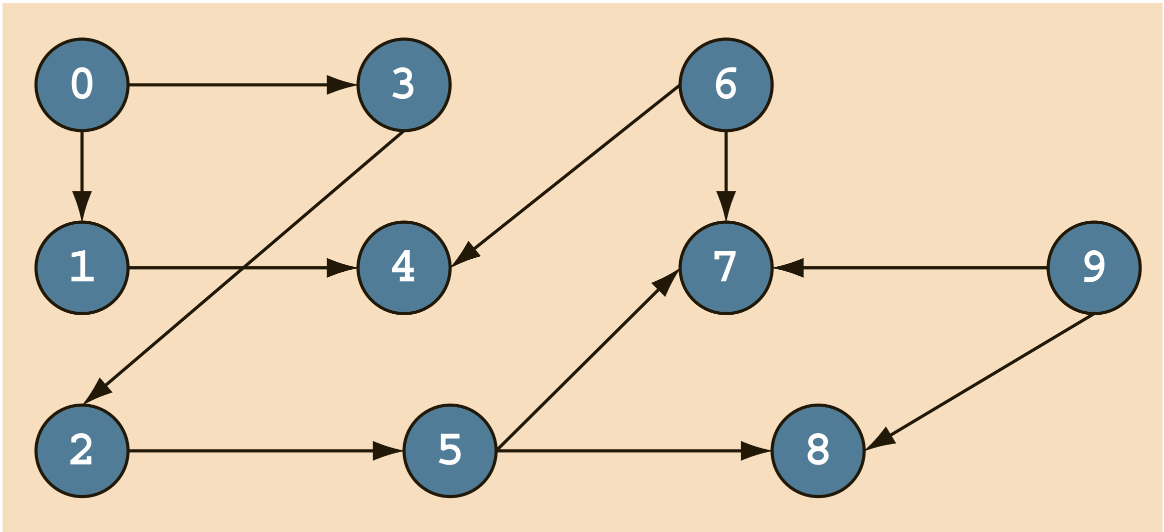
```
dft(graph, vertex, visited) {
  visited[vertex] = true;
  cout << vertex << " ";
  p = graph[vertex];
  while (p) {
    if (!visited[p->vertex])
      dft(graph, p->vertex, visited);
    p = p->next;
  }
}
```

Depth First Traversal: Pseudo-code

```
dft(graph, vertex, visited) {
  visited[vertex] = true;
  cout << vertex << " ";
  p = graph[vertex];
  while (p) {
    if (!visited[p->vertex])
      dft(graph, p->vertex, visited);
    p = p->next;
  }
}

visited[0 .. n - 1] = false;
for (i = 0; i < n; i++)
  if (!visited[i])
    dft(graph, i, visited);
```

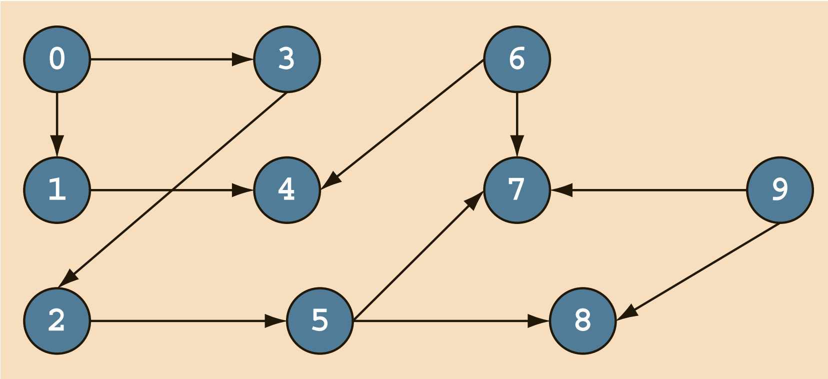
Breadth First Traversal/Search



A *breadth first ordering* of the vertices of the graph is

0, 1, 3, 4, 2, 5, 7, 8, 6, 9

Breadth First Traversal/Search: Algorithm



for each vertex v in the graph
  if v is not visited
    Add v to the queue
    Mark v as visited
    while the queue is not empty
      Extract u from the queue
      Visit the vertex
      for each vertex w that is adjacent to u
        if w is not visited
          Add w to the queue
          Mark w as visited

Breadth First Traversal/Search: Pseudo-code

```
bft(graph) {  visited[0 .. n - 1] = false;
   for (i = 0; i < n; i++)
     if (!visited[i]) {
       enQueue(Q, i); visited[i] = true;
       while (!isEmpty(Q)) {
         vertex = deQueue(Q);
         cout << vertex << " ";
         p = graph[vertex];
         while (p) {
           if (!visited[p->vertex]) {
             enQueue(Q, p->vertex);
             visited[p->vertex] = true;
           }
           p = p->next;
         }
       }
     }
}
```

An Application of Graphs: Topological Sorting

Linear ordering vs. Partial ordering

- A *linear ordering* on a finite set of items is an ordering which is given over *all* pairs of items
- A *partial ordering* on a finite set of items is an ordering which is given over *some* pairs of items but not among all of them

Applications of Graphs: Topological Sorting

Following are examples of partial orderings:

- A task is broken up into subtasks and completion of certain subtasks must usually precede the execution of other subtasks
  - Topological sorting means their arrangement in an order such that upon initiation of each subtask, all its prerequisite subtasks have been completed
  - If a subtask v must precede a subtask w, we write $v \angle w$

Partial Ordering: Example

- In a curriculum of computer science, certain courses must be taken before others
  - Topological sorting means arranging the courses in such an order that no course lists a later course as prerequisite
  - If course v is a prerequisite for course w: $v \angle w$

Applications of Graphs: Topological Sorting

Following are examples of partial orderings:

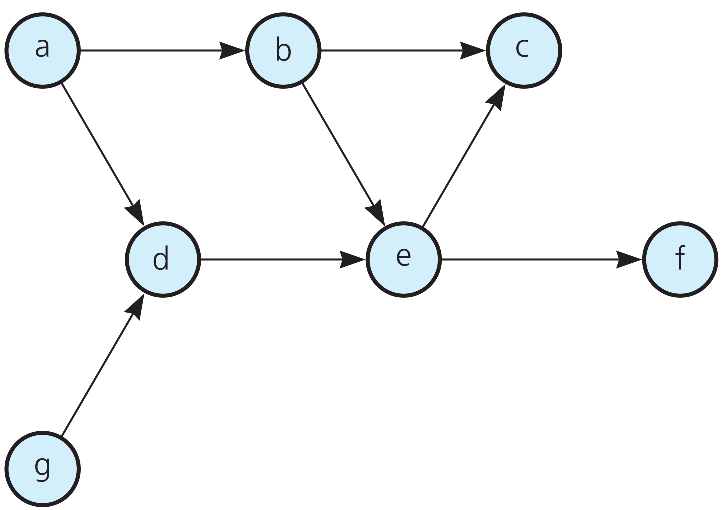
- In a computer program, some procedures may contain calls of other procedures
  - Topological sorting implies the arrangement of procedure declarations in such a way that there are no forward references
  - If a procedure v is called by a procedure w, we write $v \angle w$

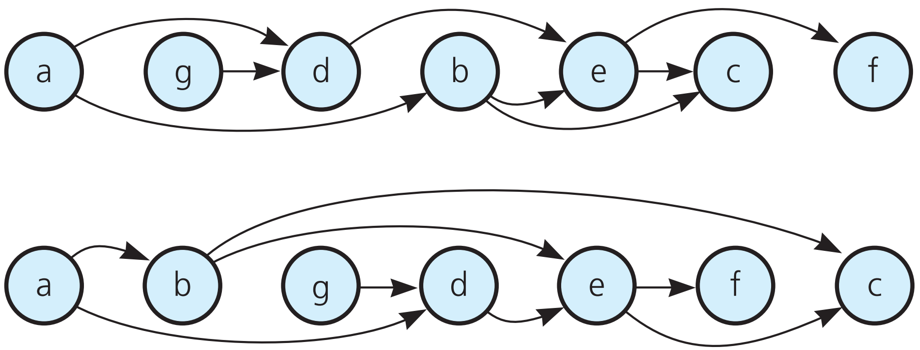
Topological Sorting: Properties

- A partial ordering on a set S satisfies the following properties for any distinct items x, y, and z of S:
  - Transitivity: If $x \angle y$ and $y \angle z$, then $x \angle z$
  - Asymmetry: If $x \angle y$, then not $y \angle x$
  - Irreflexivity: Not $x \angle x$
- Hence, a partial ordering can be illustrated by drawing a *directed acyclic graph* (DAG) in which …
  - the vertices denote the items of S
  - the directed edges represent ordering relationships

Topological Sorting

- The problem of topological sorting is to embed the partial order in a linear order
  - Graphically, if the vertices are arranged linearly and in a topological order, the edges will all point in one direction
- The vertices in a DAG may have several topological orders





Topological Sorting: Algorithm

Step 1: Add all vertices whose *indegree* is 0 to a queue

Step 2: Do the following substeps repeatedly until the queue is empty

Step 2.1: Take a vertex v off the queue and add v to the *end* of the resulting list

Step 2.2: Remove vertex v and the edges that *leave* it from the graph

Step 2.3: Some vertices whose *indegree* is 0 may occur in the graph after *Step 2.2*. Add them to the queue

Topological Sorting: An Alternative Version

Step 1: Add all vertices whose *outdegree* is 0 to a queue

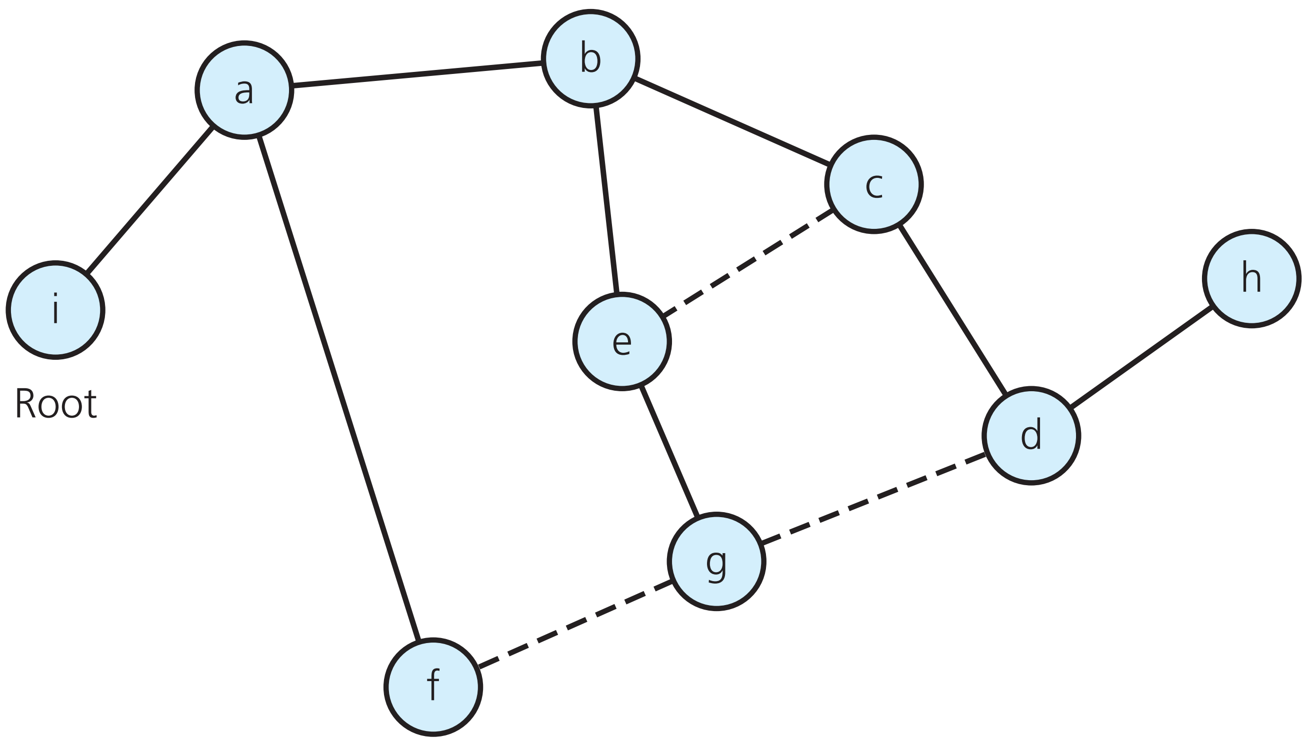
Step 2: Do the following substeps repeatedly until the queue is empty

- *Step 2.1*: Take a vertex v off the queue and add v to the *beginning* of the resulting list
- *Step 2.2*: Remove vertex v and the edges that *enter* it from the graph
- *Step 2.3*: Some vertices whose *outdegree* is 0 may occur in the graph after *Step 2.2*. Add them to the queue

Applications of Graphs: Spanning Trees

- A tree is a special kind of undirected graph, one that is connected but that has no cycles
  - Although all trees are graphs, not all graphs are trees
- A *spanning tree* of a connected undirected graph G is a subgraph of G that contains all of G's vertices and enough of its edges to form a tree
- There may be several spanning trees for a given graph

Spanning Trees: Example



Spanning Trees: Some Observations

- A connected undirected graph that has n vertices …
  - … must have at least $n-1$ edges
  - … and exactly $n-1$ edges cannot contain a cycle
  - … and more than $n-1$ edges must contain at least one cycle

➡️ To obtain the spanning tree of a connected graph of n vertices, we must connect its n vertices with $n-1$ edges

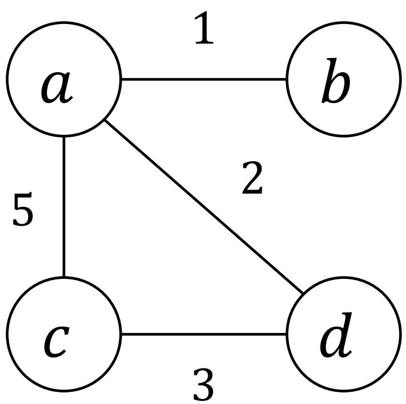
Spanning Trees: Algorithm

- Beginning at a vertex, our computer visits all other vertices in the graph
  - Each vertex will only be visited once
- As our computer traverses the graph, it also marks the edge that it follows
- After the traversal is complete, the graph's vertices and marked edges form a spanning tree
  - The unmarked edges can be removed from the graph

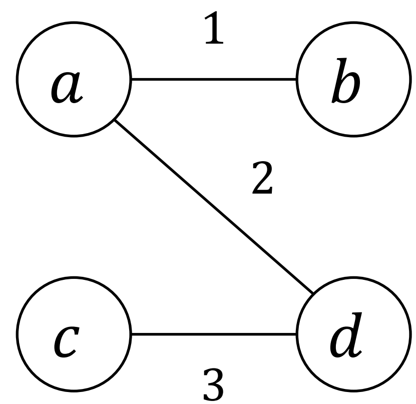
Applications of Graphs: Minimum Spanning Trees

- A *minimum spanning tree* (or *mst*) of an undirected weighted connected graph is its spanning tree of the smallest weight
  - The weight of a tree is defined as the sum of the weights on all its edges
- It has direct applications to the design of all kinds of networks by providing the cheapest way to achieve connectivity

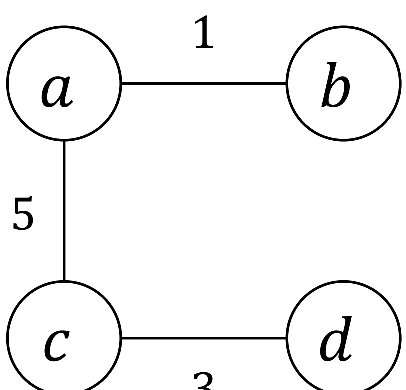
Example



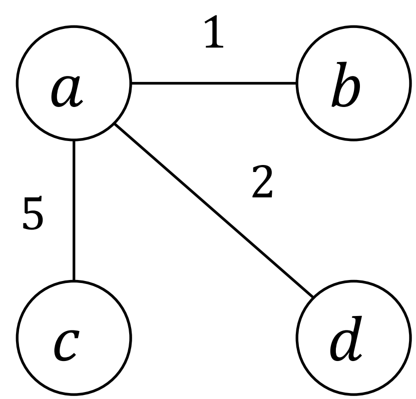
Graph



$w(T_1) = 6$



$w(T_2) = 9$



$w(T_3) = 8$

Minimum Spanning Trees: Prim's Algorithm

- The algorithm constructs a *mst* through a sequence of expanding subtrees:
  - The initial subtree consists of a single vertex selected arbitrarily from the set V
  - On each iteration, the algorithm expands the current tree by attaching to it the *nearest vertex* not in that tree
  - The algorithm stops after all the graph's vertices have been included in the tree being constructed
- The total number of such iterations is $|V| - 1$

An Outline of Prim's Algorithm

Prim($G = (V, T)$)

$V_T = \{v_0\}$

$E_T = \emptyset$

for $i = 1$ to $|V| - 1$

Find a minimum-weight edge $e^* = (u^*, v^*)$ among all edges (u, v) such that $u \in V_T, v \in V \backslash V_T$

$V_T = V_T \cup \{v^*\}$

$E_T = E_T \cup \{e^*\}$

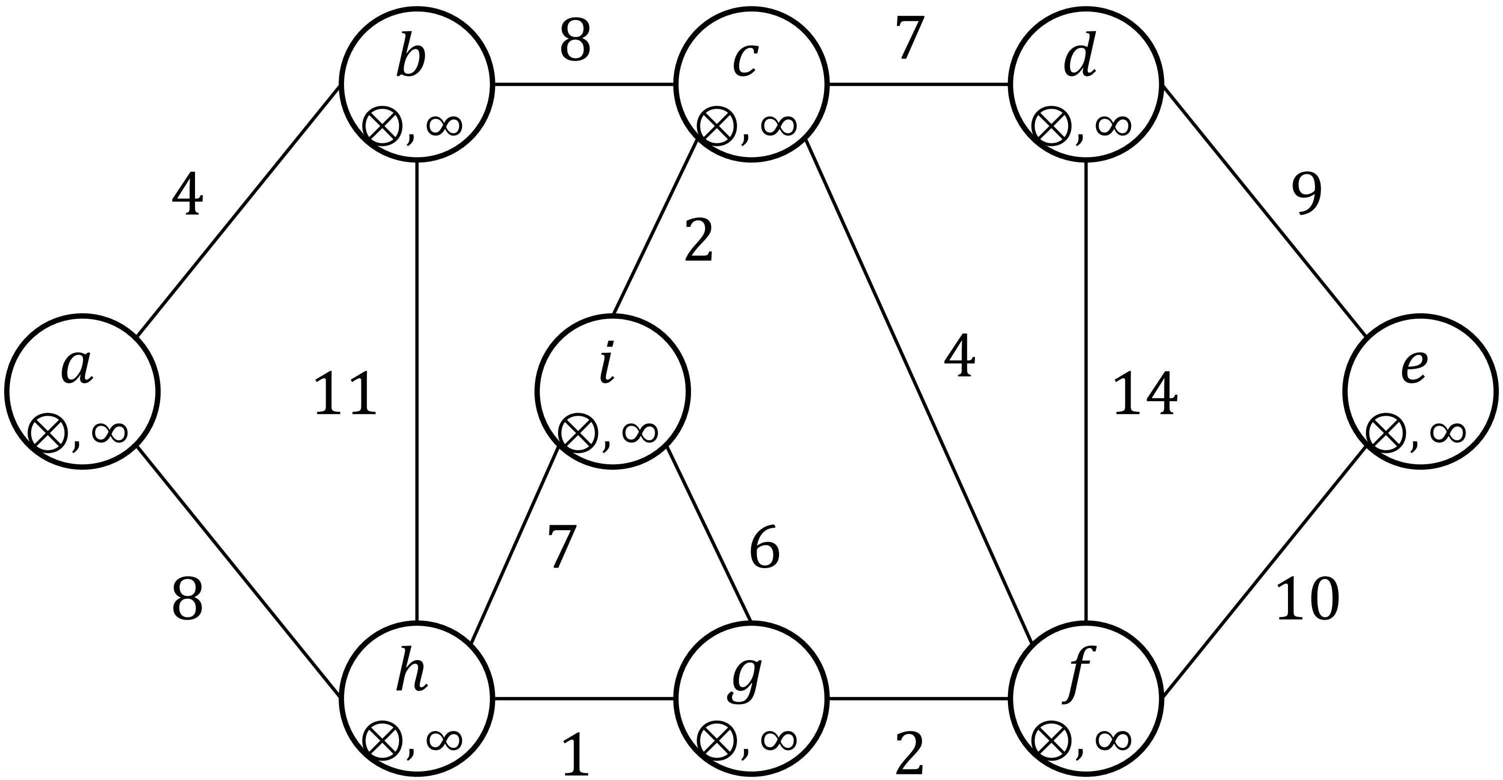
return $G_T = (V_T, E_T)$

An Implementation of Prim's Algorithm

- Each vertex in $V \backslash V_T$ needs to remember the shortest edge connecting the vertex to a vertex in V_T
- Each vertex has two labels:
  - "parent" label: The name of the nearest tree vertex ($\in V_T$)
  - "distance" label: The weight of the corresponding edge
- A vertex that is not adjacent to any of the tree vertices:
  - "parent" label is NIL
  - "distance" label is ∞

An Implementation of Prim's Algorithm

- A vertex ($\in V \backslash V_T$) with the smallest distance label is the next one to be added to the tree being constructed G_T
  - Ties can be broken arbitrarily
- Let v^* be the next vertex to be added to G_T
  - Move v^* from the set $V \backslash V_T$ to the set V_T
  - For each remaining vertex v in $V \backslash V_T$ that is connected to v^* by a shorter edge than the v's current distance label, update its labels by v^* and $G[v^*][v]$, respectively



a
b
c
d
e
f
g
h
i
⊗, ∞
⊗, ∞
⊗, ∞
⊗, ∞
⊗, ∞
⊗, ∞
⊗, ∞
⊗, ∞
⊗, ∞
4
8
8
11
7
2
7
6
1
2
4
14
9
10

```
Prim(G, root) {
  for (each vertex v ∈ V) {
    v.dist = ∞;
    v.parent = NIL;
  }
  root.dist = 0;    V_T = ∅;
  createQueue(Q, V);
  while (!isEmpty(Q)) {
    v* = extractQueue(Q);
    add v* to V_T;
    for (each v ∈ Q that is adjacent to v*)
      if (G[v*][v] < v.dist) {
        v.dist = G[v*][v];
        v.parent = v*;
        updateQueue(Q, v);
      }
  }
}
```

Single-Source Shortest-Paths Problem

- For a given vertex called the *source* in a weighted connected graph, find shortest paths to all its other vertices
- The most widely applications of the problem are transportation planning and packet routing in communication networks, including the Internet
- Dijkstra’s algorithm is the best-known algorithm for the problem

Dijkstra's Algorithm: The General Idea

- The algorithm finds the shortest path from the *source* to a vertex nearest to it, then to a second nearest, …

➡️ Before the i^{th} iteration starts, the algorithm has already identified the shortest paths to $i-1$ other vertices nearest to the *source*

  – These vertices, the *source*, and the edges of the shortest paths leading to them from the *source* form a subtree T_i

- The next vertex nearest to the *source* can be found among the vertices adjacent to the vertices of T_i

An Outline of Dijkstra's Algorithm

// d_u the length of the shortest path from the *source* to u

Dijkstra($G = (V, T)$)

$V_T = \{v_0\}$ // v_0 is the *source*

$E_T = \emptyset$

for $i = 1$ to $|V| - 1$

Find an edge $e^* = (u^*, v^*)$ among all edges (u, v) such that $u \in V_T, v \in V \backslash V_T$ and $d_{u^*} + G[u^*][v^*] \leq d_u + G[u][v]$

$V_T = V_T \cup \{v^*\}$

$E_T = E_T \cup \{e^*\}$

return $G_T = (V_T, E_T)$

An Implementation of Dijkstra's Algorithm

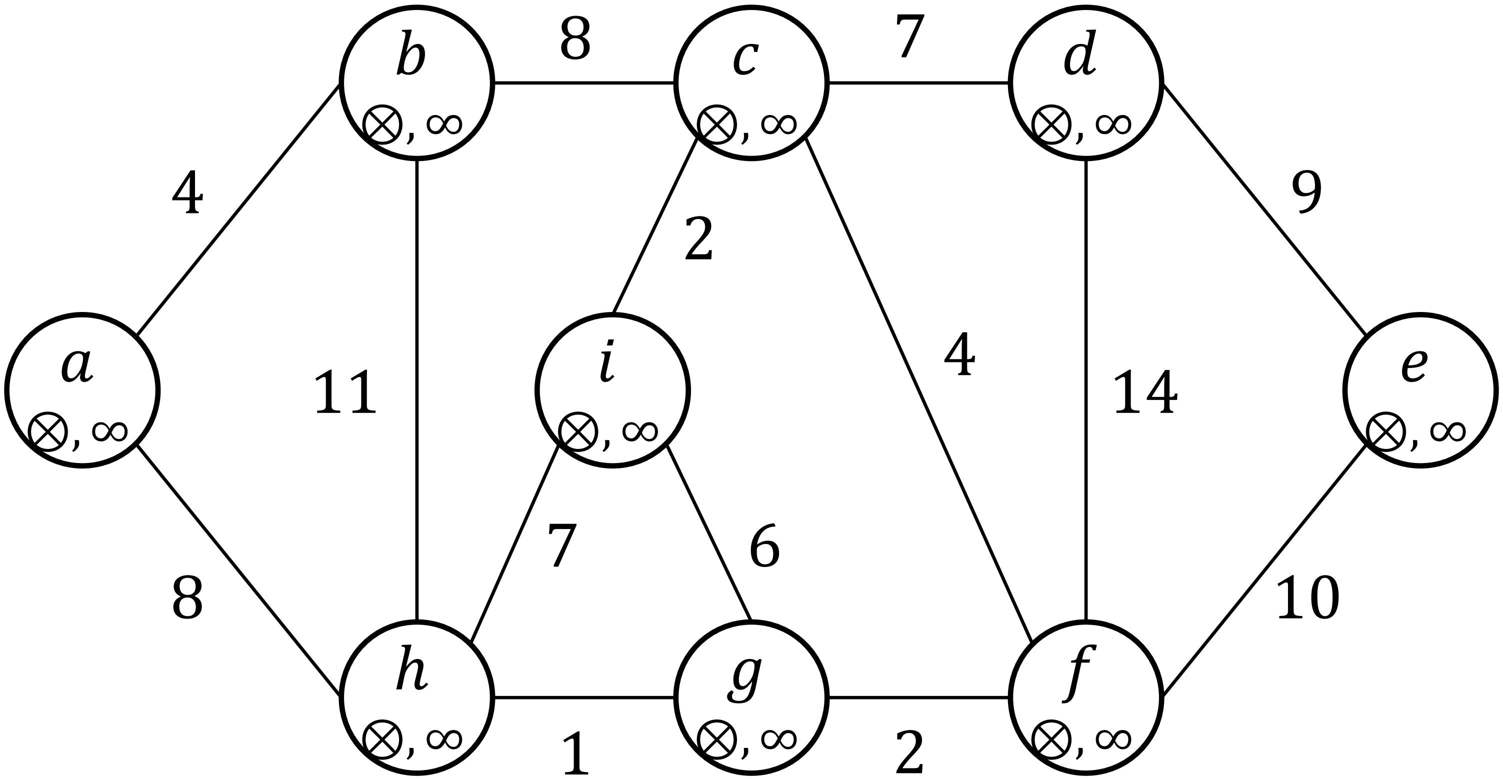
- Each vertex has two labels:
  - "parent" label: The name of the nearest tree vertex ($\in V_T$)
  - "distance" label: The length of the shortest path from the *source* to this vertex found by the algorithm so far

  ➡️ When a vertex is added to V_T, this label indicates the length of the shortest path from the *source* to that vertex

- A vertex that is not adjacent to any of the tree vertices:
  - "parent" label is NIL
  - "distance" label is ∞

An Implementation of Dijkstra's Algorithm

- Let v^* be the next vertex to be added to G_T
  - Move v^* from the set $V \backslash V_T$ to the set V_T
  - For each remaining vertex v in $V \backslash V_T$ that is connected to v^* by an edge of weight $G[v^*][v]$ such that $d_{v^*} + G[v^*][v] < d_v$, update the labels of v by v^* and $d_{v^*} + G[v^*][v]$, respectively



8
7
4
2
9
4
11
14
7
6
8
10
1
2
a
b
c
d
e
f
g
h
i
$\otimes, \infty$
$\otimes, \infty$
$\otimes, \infty$
$\otimes, \infty$
$\otimes, \infty$
$\otimes, \infty$
$\otimes, \infty$
$\otimes, \infty$
$\otimes, \infty$

```
Dijkstra(G(V, E), source) {
  for (each vertex v ∈ V) {
    v.dist = ∞;
    v.parent = NIL;
  }
  source.dist = 0;  V_T = ∅;
  createQueue(Q, V);
  while (!isEmpty(Q)) {
    v* = extractQueue(Q);
    add v* to V_T;
    for (each v ∈ Q that is adjacent to v*)
      if (v*.dist + G[v*][v] < v.dist) {
        v.parent = v*;
        v.dist = v*.dist + G[v*][v];
        updateQueue(Q, v);
      }
  }
}
```