CS302 - Data Structures *using C++*

Topic: Graphs - Introduction

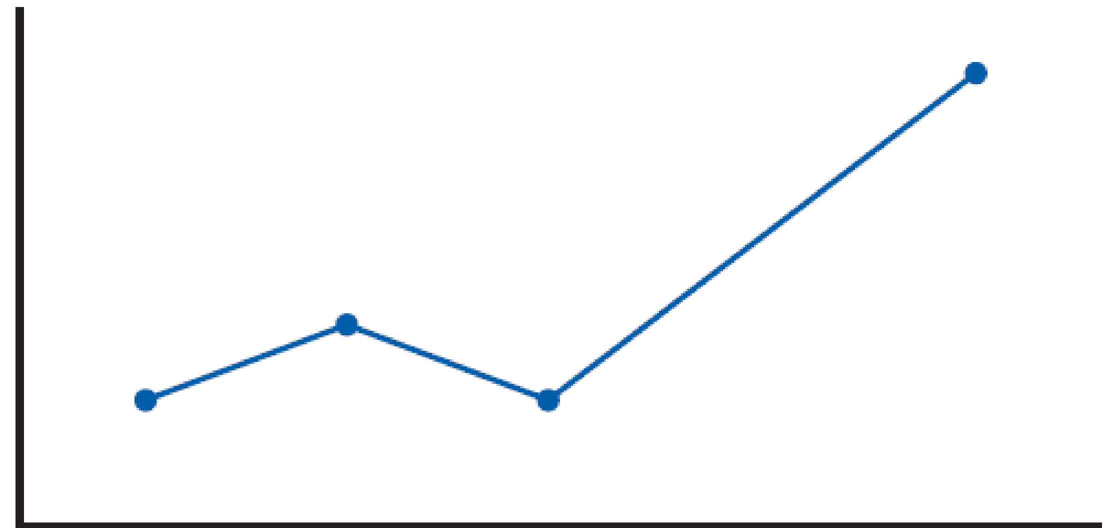
Kostas Alexis

AUTONOMOUS ROBOTS LAB

Terminology

- In the context of our course, graphs represent relations among data items
- G = {V,E}
  - A graph is a set of vertices (nodes) and
  - A set of edges that connect the vertices

A simplified case: a line graph



AUTONOMOUS ROBOTS LAB

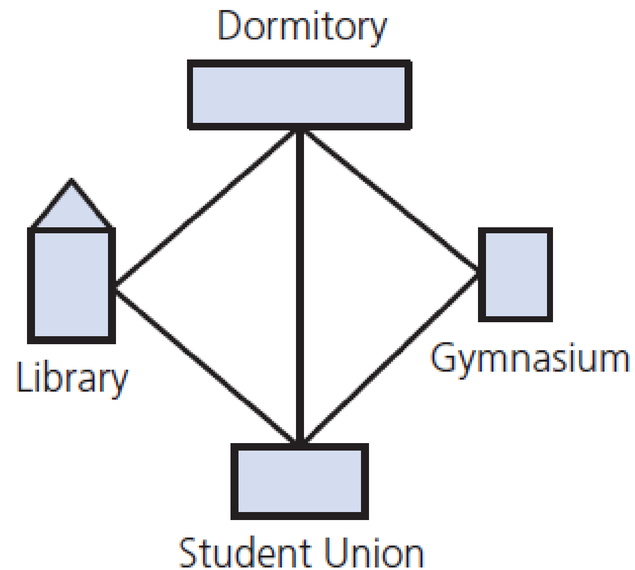
Terminology

- In the context of our course, graphs represent relations among data items
- G = {V,E}
  - A graph is a set of vertices (nodes) and
  - A set of edges that connect the vertices
- A **graph** is an ordered pair **G = (V, E)** comprising a set V of vertices, nodes or points together with a set E of edges, arcs or lines, which are 2-element subsets of V (i.e., an edge is associated with two vertices, and the association takes the form of the unordered pair of the vertices).
  - To avoid ambiguity, this type of graph may be described precisely as undirected and simple.

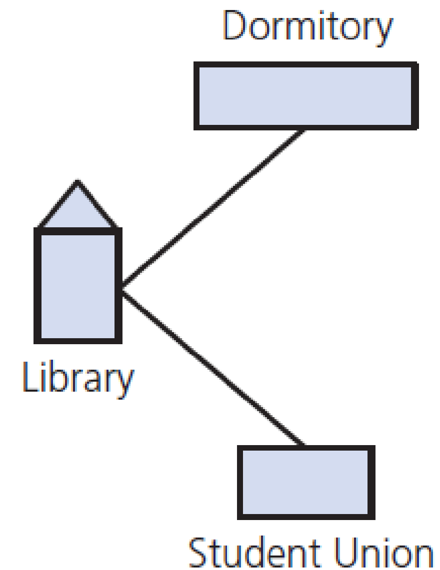
Terminology

- A graph and one of its subgraphs

(a) A campus map as a graph



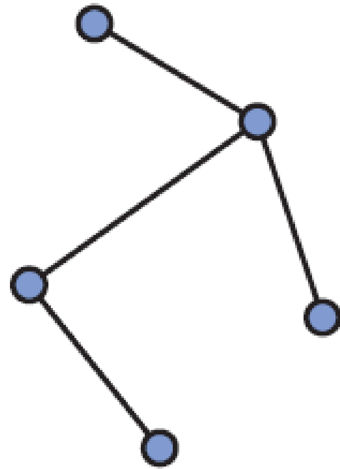
(b) A subgraph of the graph in part a



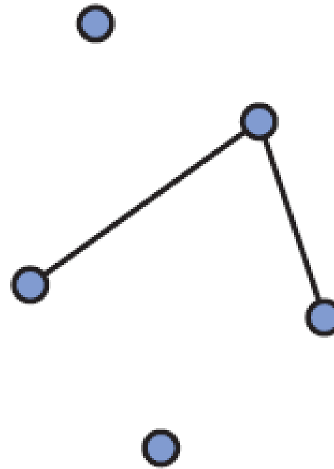
Terminology

- Examples of **graphs** that are either **connected**, **disconnected**, or **complete**

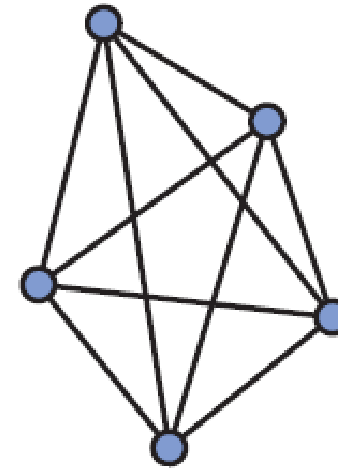
(a) Connected



(b) Disconnected



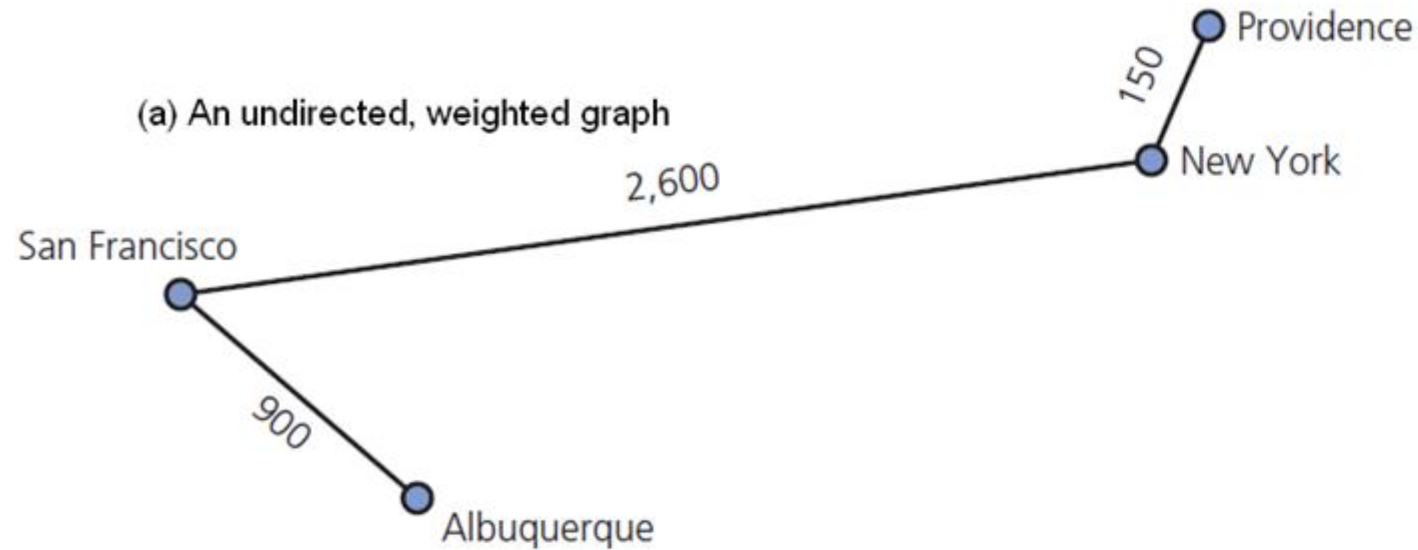
(c) Complete



AUTONOMOUS ROBOTS LAB

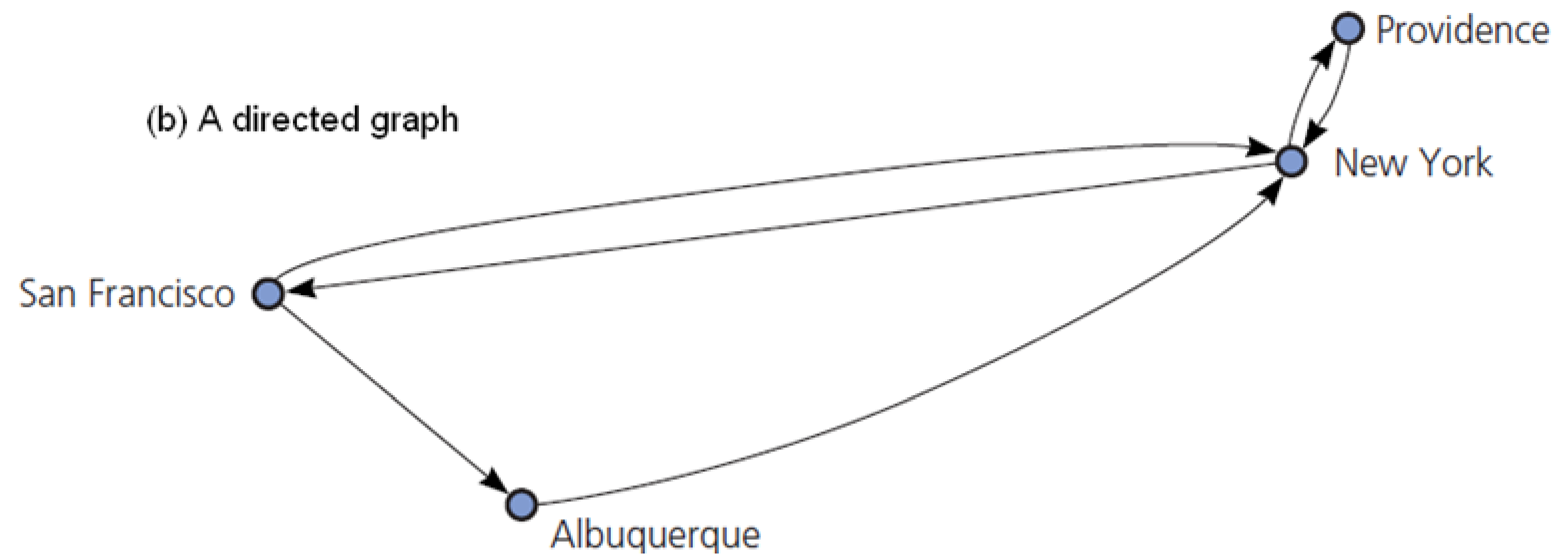
Terminology

- **Undirected** and directed graphs



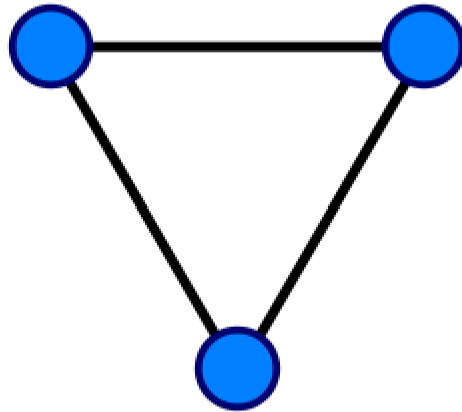
Terminology

- Undirected and **directed** graphs



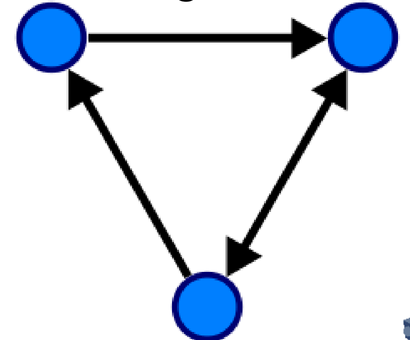
Terminology

- **Undirected Graph:** An undirected graph is a graph in which edges have no orientation. *The edge (x, y) is identical to the edge (y, x).* That is, they are not ordered pairs, but unordered pairs—i.e., sets of two vertices {x, y} (or 2-multisets in the case of loops). The maximum number of edges in an undirected graph without a loop is $n(n - 1)/2$.



Terminology

- **Directed Graph:** A directed graph or digraph is a graph in which *edges have orientations*. It is written as an ordered pair **G = (V, A)** (sometimes G = (V, E)) with
  - **V** a set whose elements are called vertices, nodes, or points;
  - **A** a set of ordered pairs of vertices, called arrows, directed edges (sometimes simply edges with the corresponding set named E instead of A), directed arcs, or directed lines.
  - An **arrow** (x, y) is considered to be directed from x to y; y is called the head and x is called the tail of the arrow; y is said to be a direct successor of x and x is said to be a direct predecessor of y. If a path leads from x to y, then y is said to be a successor of x and reachable from x, and x is said to be a predecessor of y. The arrow (y, x) is called the inverted arrow of (x, y).
  - A directed graph G is called **symmetric** if, for every arrow in G, the corresponding inverted arrow also belongs to G. A symmetric loopless directed graph G = (V, A) is equivalent to a simple undirected graph G′ = (V, E), where the pairs of inverse arrows in A correspond one-to-one with the edges in E; thus the number of edges in G′ is |E| = |A|/2, that is half the number of arrows in G.



Graphs as ADTs

- ADT graph operations

AUTONOMOUS ROBOTS LAB

Graphs as ADTs

- ADT graph operations
  - Test if empty
  - Get number of vertices, edges in a graph
  - See if edge exists between two given vertices
  - Add vertex to graph whose vertices have distinct, different values from new vertex
  - Add/remove an edge between two given vertices
  - Remove vertex, edges to other vertices
  - Retrieve vertex that contains a given value

Graphs as ADTs

- A C++ interface for undirected, connected graphs

```
#ifndef GRAPH_INTERFACE_
#define GRAPH_INTERFACE_

template<class LabelType>
class GraphInterface
{
    public:
    /** Gets the number of vertices in this graph.
    @return The number of vertices in the graph. */
    virtual int getNumVertices() const = 0;
    /** Gets the number of edges in this graph.
    @return The number of edges in the graph. */
    virtual int getNumEdges() const = 0;
```

AUTONOMOUS ROBOTS LAB

Graphs as ADTs

- A C++ interface for undirected, connected graphs

```
/** Creates an undirected edge in this graph between two vertices that have the given labels. If such vertices do not
exist, creates them and adds them to the graph before creating the edge.
@param start A label for the first vertex.
@param end A label for the second vertex.
@param edgeWeight The integer weight of the edge.
@return True if the edge is created, or false otherwise. */
virtual bool add(LabelType start, LabelType end, int edgeWeight) = 0;
/** Removes an edge from this graph. If a vertex is left with no other edges, it is removed from the graph since this
is   a connected graph.
@param start A label for the vertex at the beginning of the edge.
@param end A label for the vertex at the end of the edge.
@return True if the edge is removed, or false otherwise. */
virtual bool remove(LabelType start, LabelType end) = 0;
/** Gets the weight of an edge in this graph.
@param start A label for the vertex at the beginning of the edge.
@param end A label for the vertex at the end of the edge.
@return The weight of the specified edge. If no such edge exists, returns a negative integer. */
virtual int getEdgeWeight(LabelType start, LabelType end) const = 0;
```

AUTONOMOUS ROBOTS LAB

Graphs as ADTs

- A C++ interface for undirected, connected graphs

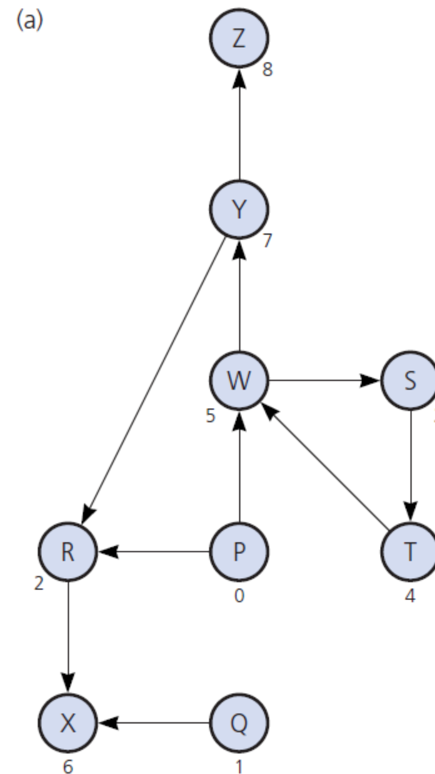
```
    /** Performs a depth-first search of this graph beginning at the given vertex and calls a given function once
          for each vertex visited.
    @param start A label for the beginning vertex.
    @param visit A client-defined function that performs an operation on
    or with each visited vertex. */
    virtual void depthFirstTraversal(LabelType start, void visit(LabelType&)) = 0;

          /** Performs a breadth-first search of this graph beginning at the given
    vertex and calls a given function once for each vertex visited.
    @param start A label for the beginning vertex.
    @param visit A client-defined function that performs an operation on or with each visited vertex. */
    virtual void breadthFirstTraversal(LabelType start, void visit(LabelType&)) = 0;
    /** Destroys this graph and frees its assigned memory. */
    virtual ~GraphInterface() { }
}; // end GraphInterface
#endif
```

AUTONOMOUS ROBOTS LAB

Implementing Graphs

- A directed graph and its **adjacency matrix**

(a)



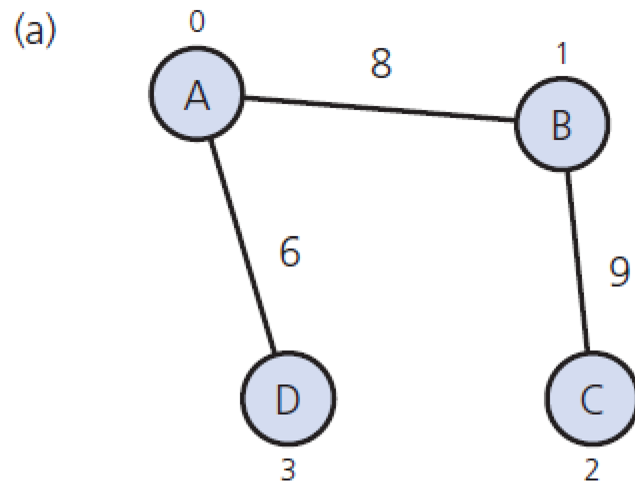
(b)

		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

AUTONOMOUS ROBOTS LAB

Implementing Graphs

- A weighted undirected graph and its **adjacency matrix**



(b)

		0 A	1 B	2 C	3 D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

AUTONOMOUS ROBOTS LAB

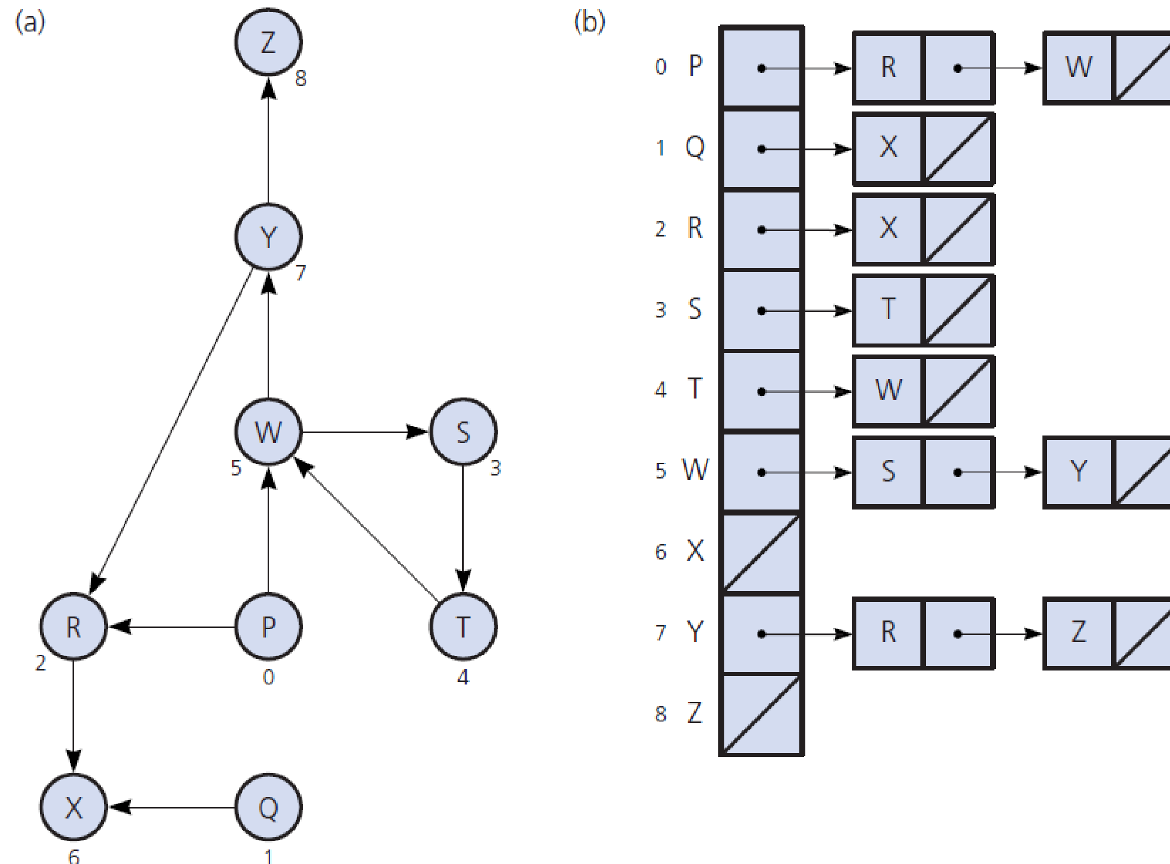
Implementing Graphs

- **Weighted Graph:** A weighted graph is a graph in which a number (the weight) is assigned to each edge. Such weights might represent for example costs, lengths or capacities, depending on the problem at hand.

AUTONOMOUS ROBOTS LAB

Implementing Graphs

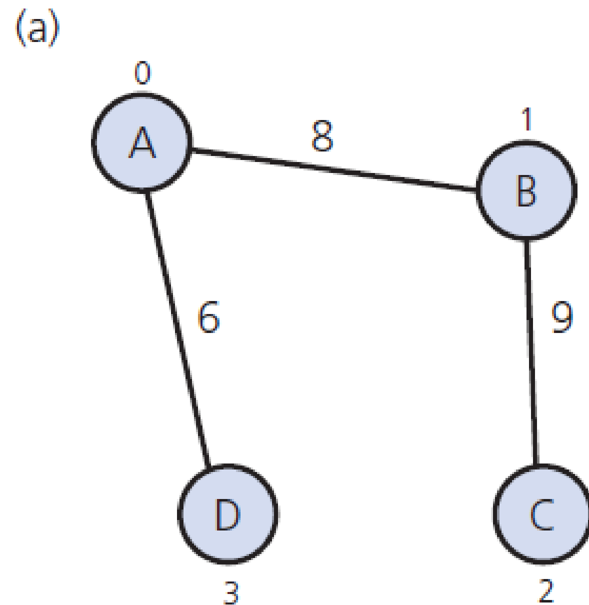
- A directed graph and its **adjacency list**

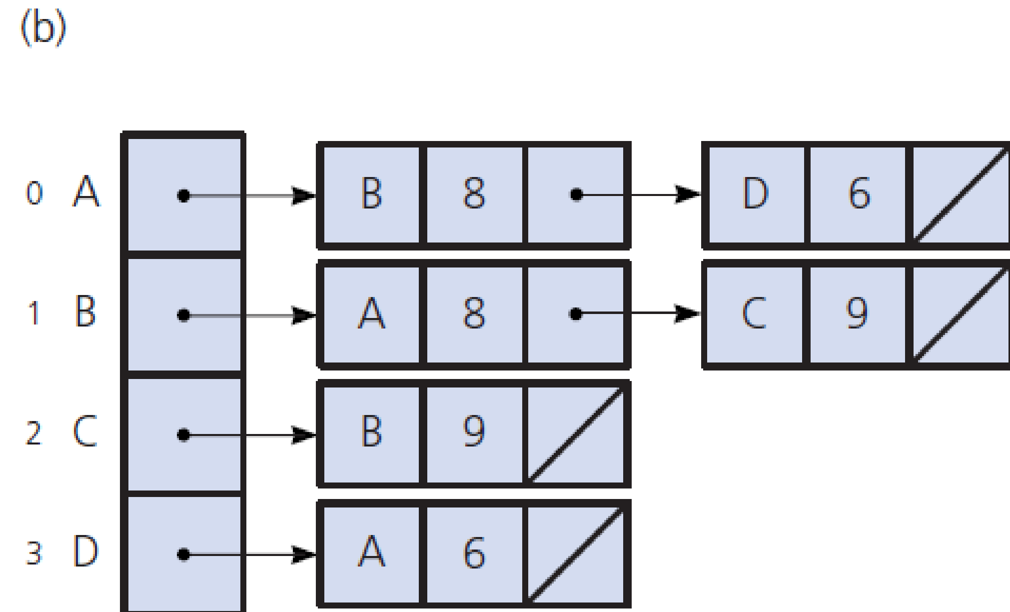


AUTONOMOUS ROBOTS LAB

Implementing Graphs

- A weighted undirected graph and its **adjacency list**



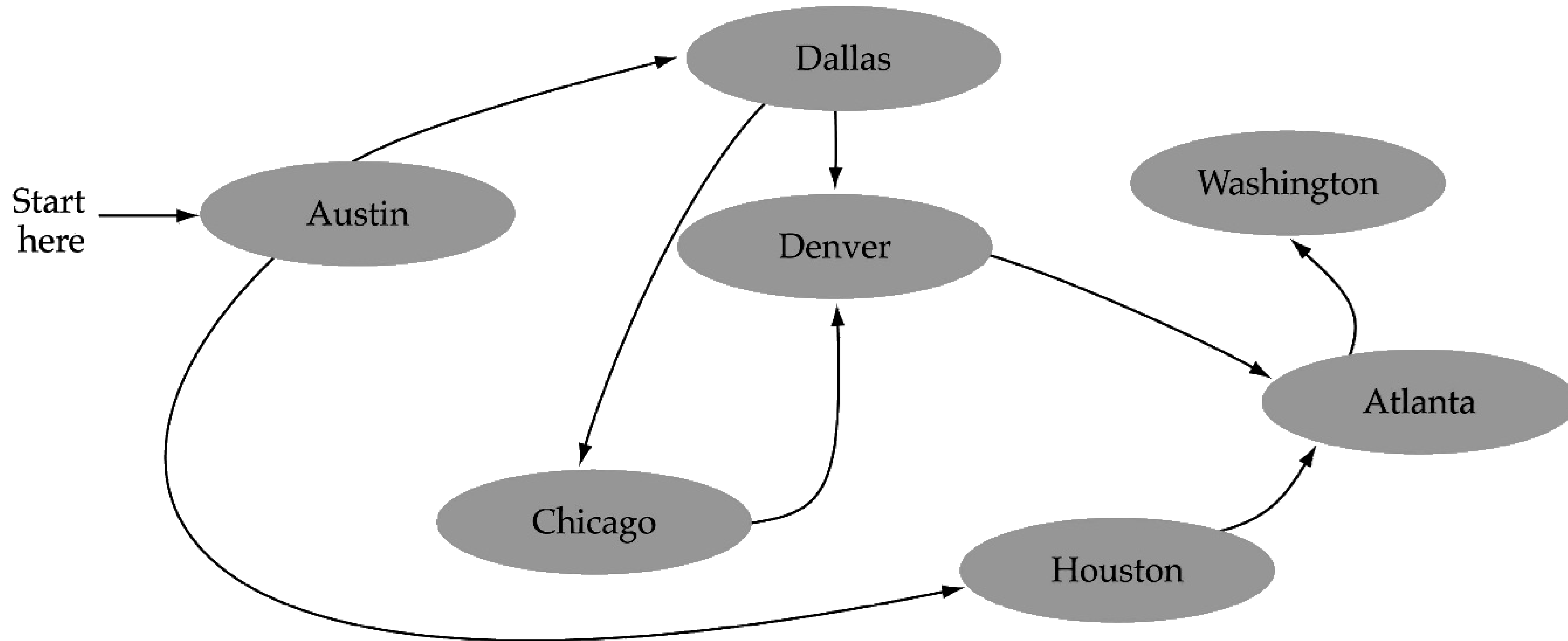


Implementing Graphs

- **Adjacency List**
  - Often requires less space than adjacency matrix
  - Supports finding vertices adjacent to given vertex
- **Adjacency Matrix**
  - Supports process of seeing if there exists from vertex i to vertex j

Graph Traversals

- **Problem:** Find if there is a path between two vertices of the graph
- **Methods:** Depth-First Search (DFS) or Breadth-First Search (BFS)



AUTONOMOUS ROBOTS LAB

Graph Traversals

- Visits all vertices it can reach
- Visits all vertices if and only if the graph is connected
- Connected component
  - Subset of vertices visited during a traversal that belongs at given vertex

AUTONOMOUS ROBOTS LAB

Depth-First Search

- **Depth-First Search (DFS)**
  - Goes as far as possible from a vertex before backing up

Depth-First Search

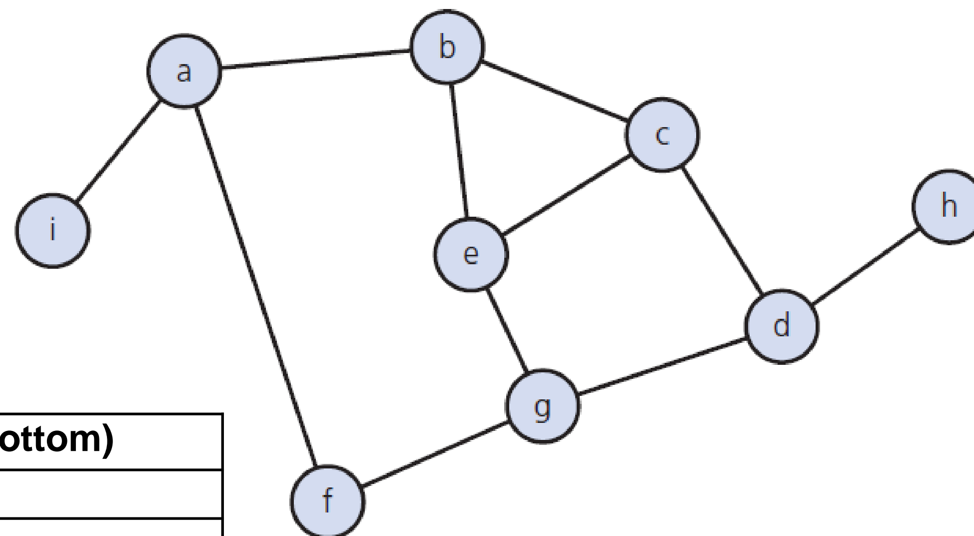
- **Depth-First Search (DFS)**
  - Goes as far as possible from a vertex before backing up
- Recursive traversal algorithm

```
// Traverses a graph beginning at vertex v by using a
// depth-first search. Recursive version
dfs(v: Vertex)
{
    Mark v as visited
    for (each unvisited vertex u adjacent to v)
         dfs(u)
}
```

AUTONOMOUS ROBOTS LAB

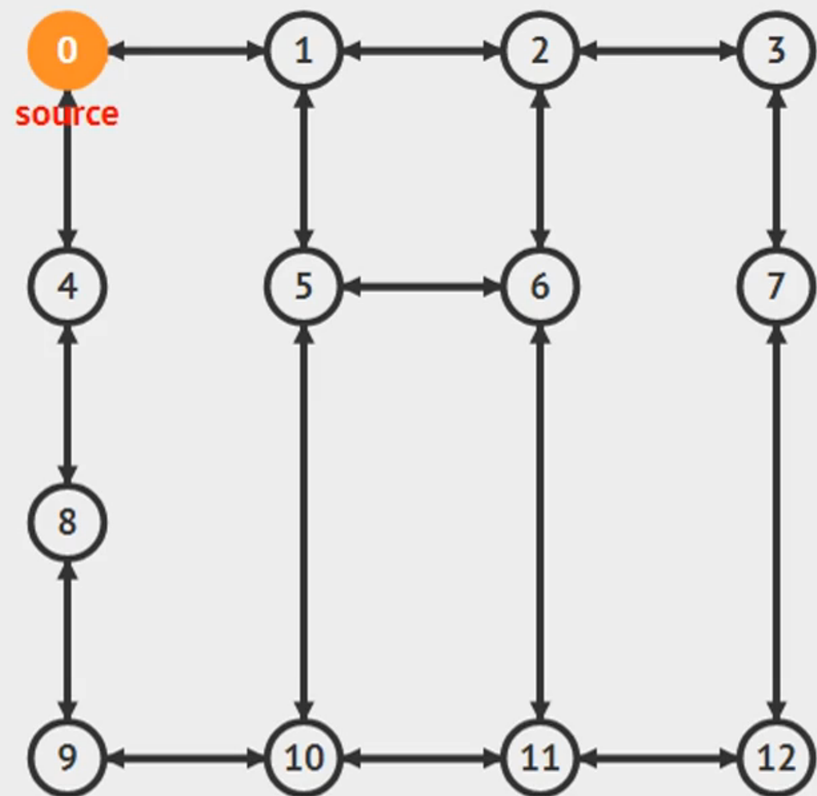
Depth-First Search

- Results of a DFS traversal beginning at vertex a



Node visited	Stack (top to bottom)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(**backtrack**)	a b c d g
f	a b c d g f
(**backtrack**)	a b c d g
(**backtrack**)	a b c d
h	a b c d h
(**backtrack**)	a b c d
(**backtrack**)	a b c
(**backtrack**)	a b
(**backtrack**)	a
i	a i
(**backtrack**)	a
(**backtrack**)	(**empty**)

AUTONOMOUS ROBOTS LAB



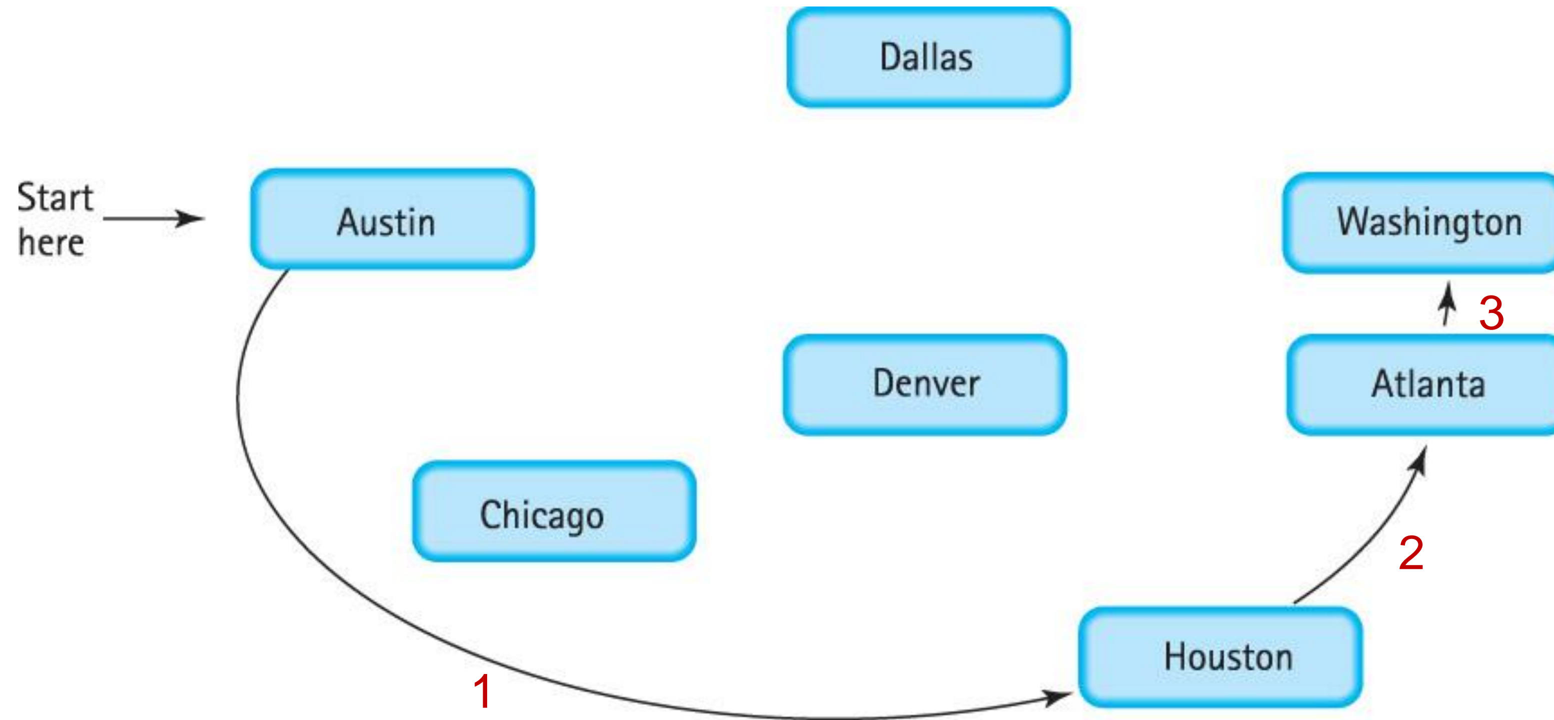
DFS(0)

Try edge 0 → 1

```
DFS(u)
for each neighbor v of u
  if v is unvisited, tree edge, DFS(v)
  else if v is explored, bidirectional/back edge
  else if v is visited, forward/cross edge
// ch4_01_dfs.cpp/java, ch4, CP3
```

Depth-First Search

- **DFS uses a stack**

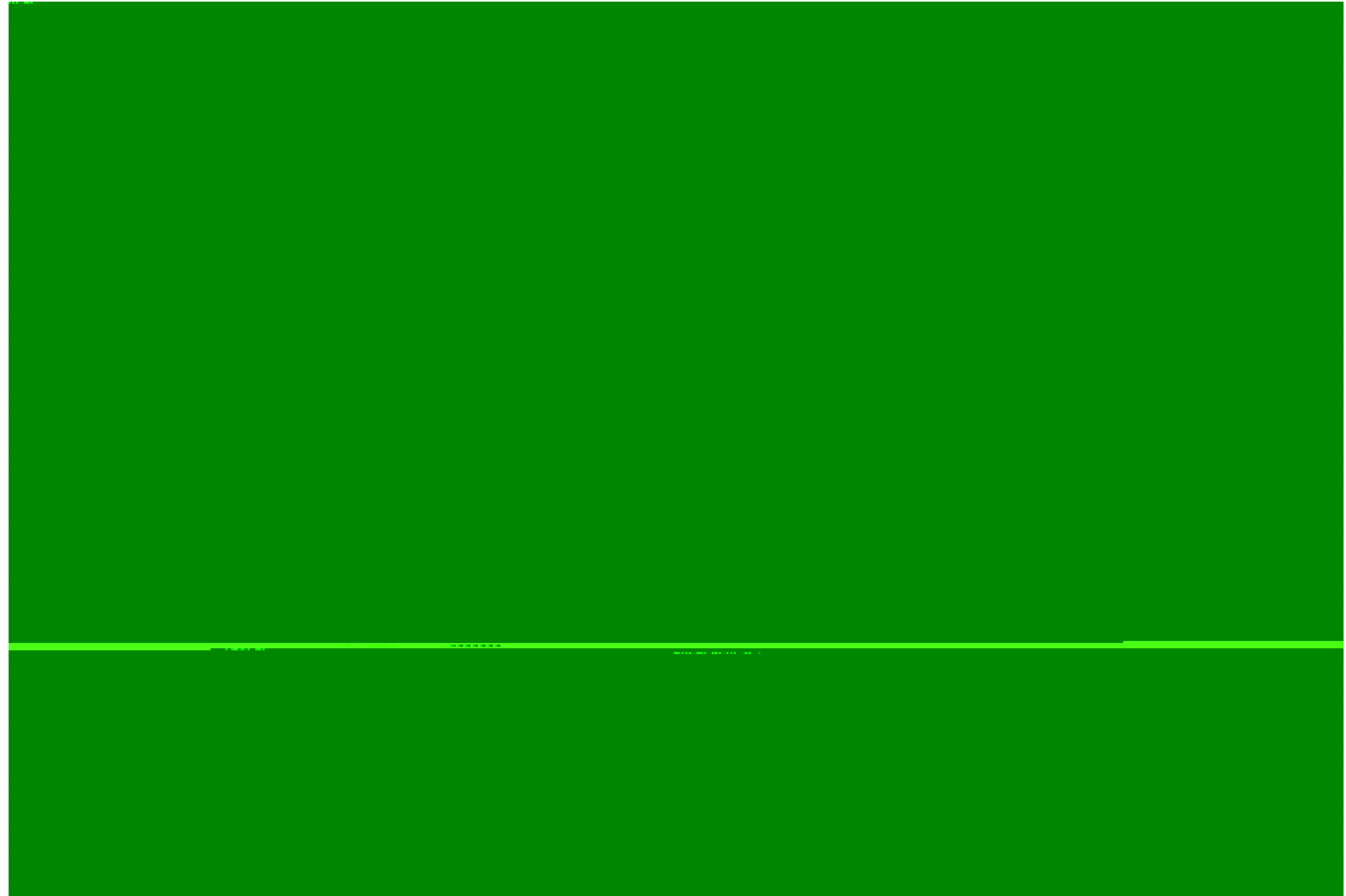


Depth-First Search

- Applications:
  - Finding connected components.
  - Topological sorting.
  - Finding 2-(edge or vertex)-connected components.
  - Finding 3-(edge or vertex)-connected components.
  - Finding the bridges of a graph.
  - Generating words in order to plot the Limit Set of a Group.
  - Finding strongly connected components.
  - Planarity testing[7][8]
  - Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)
  - Maze generation may use a randomized depth-first search.
  - Finding biconnectivity in graphs.

Depth-First Search

- Maze generation



Breadth-First Search

- **Breadth-First Search (BFS) traversal**
  - Visits all vertices adjacent to a vertex before going forward.
- BFS is a first visited, first explored strategy
  - Contrast to DFS as last visited, first explored

AUTONOMOUS ROBOTS LAB

Breadth-First Search

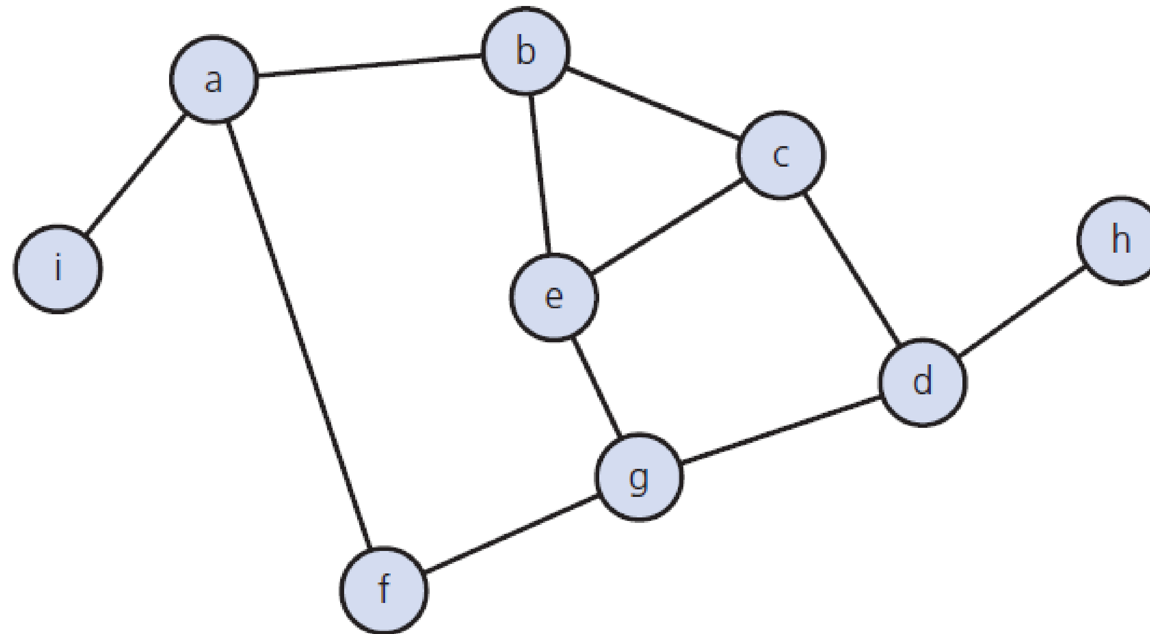
- **Breadth-First Search (BFS) traversal**
  - Visits all vertices adjacent to a vertex before going forward.
- BFS is a first visited, first explored strategy
  - Contrast to DFS as last visited, first explored
- Main Idea:
  - Look at all possible paths at the same depth before you go at a deeper level
  - Backup as far as possible when you reach a “dead end”
    - i.e., next vertex has been “marked” or there is no next vertex

Breadth-First Search

```
// Traverses a graph beginning at vertex v by using a
// breadth-first search. Iterative version
bfs(v: Vertex)
{
    q = a new empty queue
    // Add v to queue and mark it
    q.enqueue(v)
    Mark v as visited
    while (!q.isEmpty())
    {
        q.dequeue(w)
        // Loop invariant: there is a path from vertex w to every vertex in the queue q
        for (each unvisited vertex u adjacent to w)
        {
            Mark u as visited
            q.enqueue(u)
        }
    }
}
```

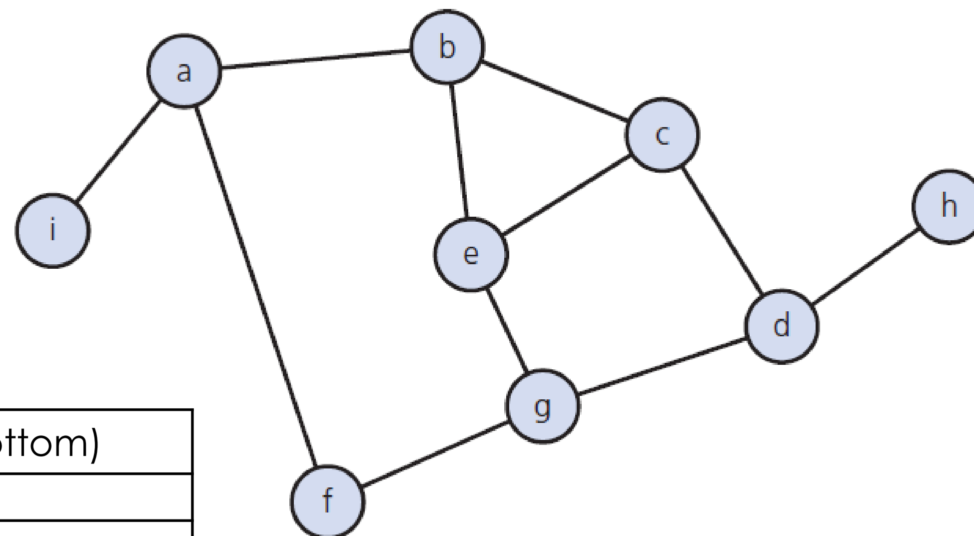
Breadth-First Search

- Consider the graph



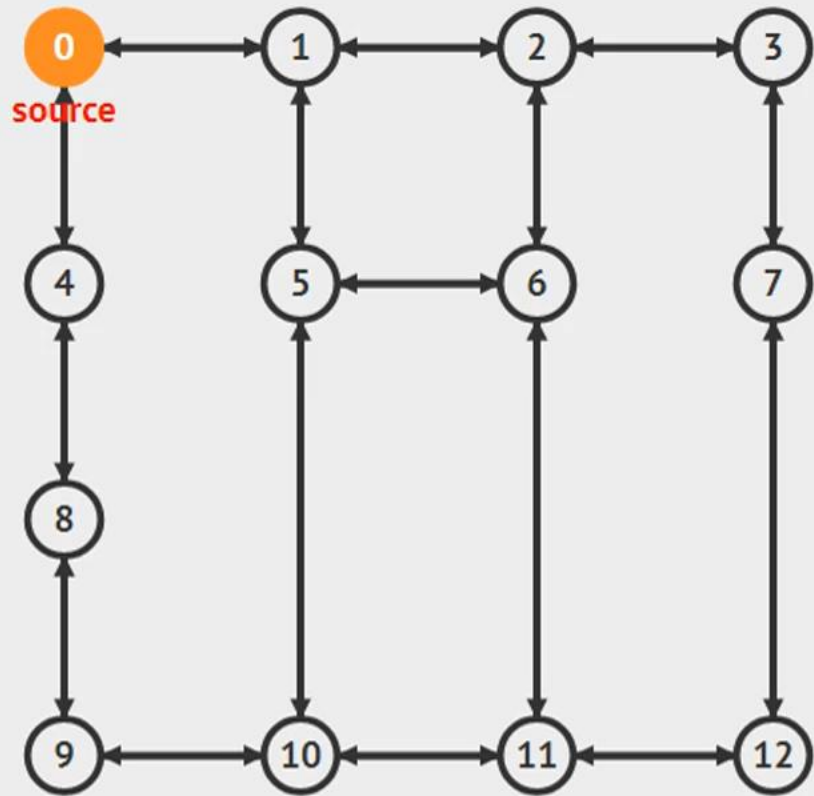
Breadth-First Search

- Results of a BFS traversal beginning at vertex a



Node visited	Stack (top to bottom)
a	a
	(**empty**)
b	b
f	b f
i	b f i
	f i
c	f i c
e	f i c e
	i c e
g	i c e g
	c e g
	e g
d	e g d
	g d
	d
	(**empty**)
h	h
	(**empty**)

AUTONOMOUS ROBOTS LAB



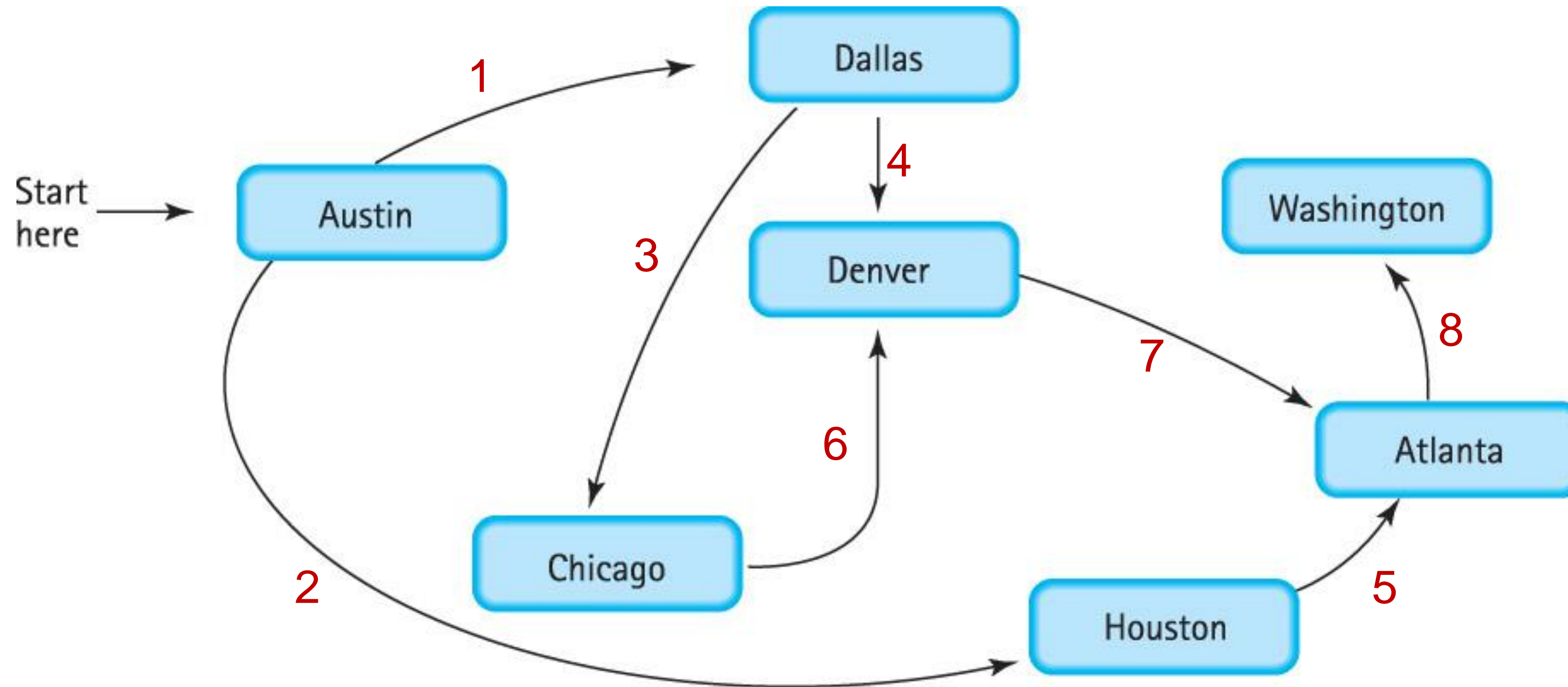
BFS(0)

The queue is now {0}.
Exploring neighbors of vertex u = 0.

```
BFS(u), Q = {u}
while !Q.empty // Q is a normal queue
  for each neighbor v of u = Q.front, Q.pop
    if v is unvisited, tree edge, Q.push(v)
    else if v is visited, we ignore this edge
// ch4_04_bfs.cpp/java, ch4, CP3
```

Breadth-First Search

- **BFS uses a Queue**



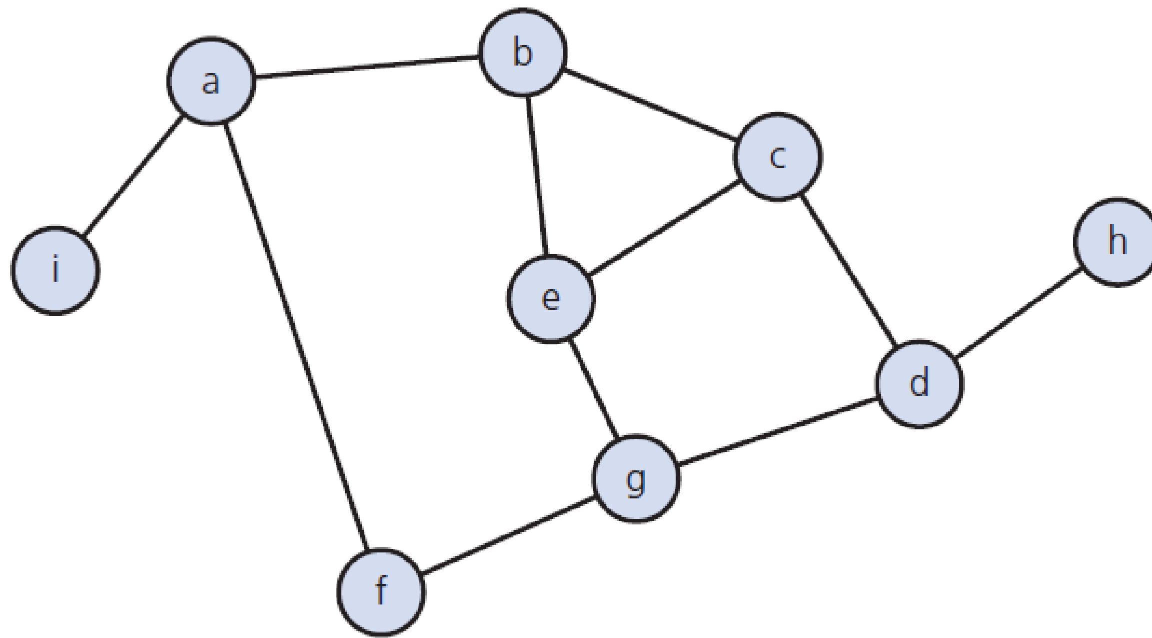
Applications of Graphs

- Topological Sorting
- Spanning Trees
- Minimum Spanning Trees
- Shortest Paths
- Circuits
- Crucial mathematical/engineering problems

AUTONOMOUS ROBOTS LAB

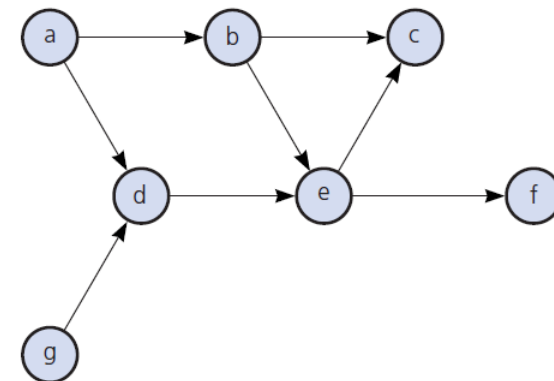
Example Graph

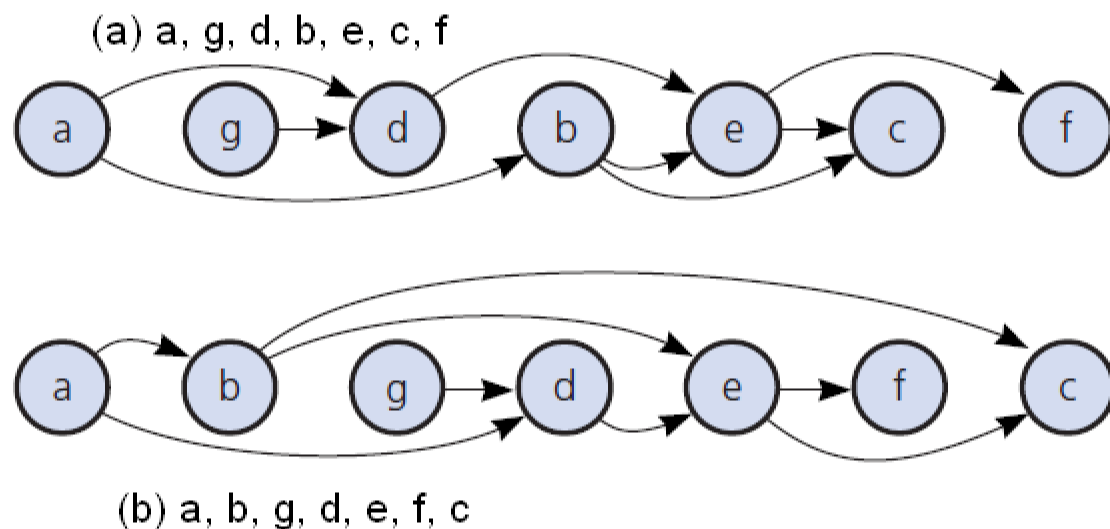
- A connected graph with cycles



Topological Sorting

- The graph in the previous Figure arranged according to two topological orders





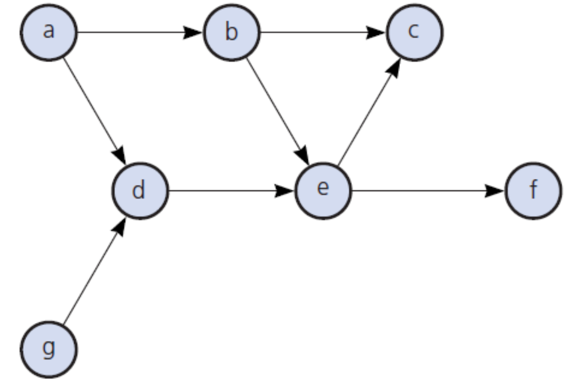
Topological Sorting

- **topSort1** algorithm

```
// Arranges the vertices in graph theGraph into a
// topological order and places them in list aList.
topSort1(theGraph: Graph, aList: List)
{
    n = number of vertices in theGraph
    for (step = 1 through n)
    {
        Select a vertex v that has no successors
        aList.insert(1,v)
        Remove from theGraph vertex v and its edges
    }
}
```

AUTONOMOUS ROBOTS LAB

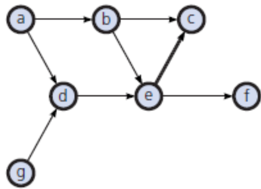
Topological Sorting



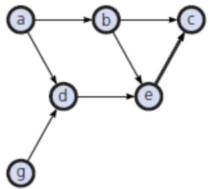
- A trace of **topSort1** for the previous figure.

Graph theGraph

List aList



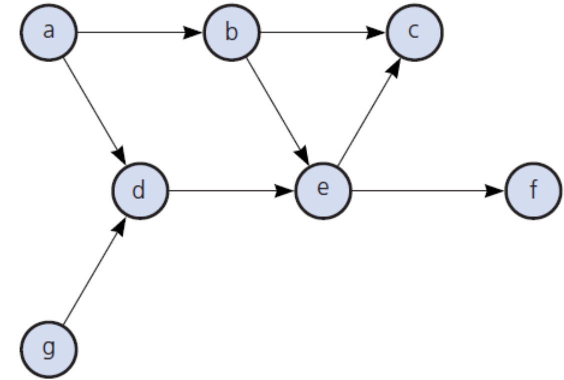
Remove f from theGraph;
add it to aList



f

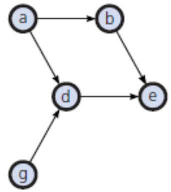
Remove c from theGraph;
add it to aList

Topological Sorting



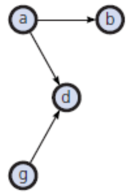
- (continued)

Remove c from `theGraph;`
add it to `aList`



c f

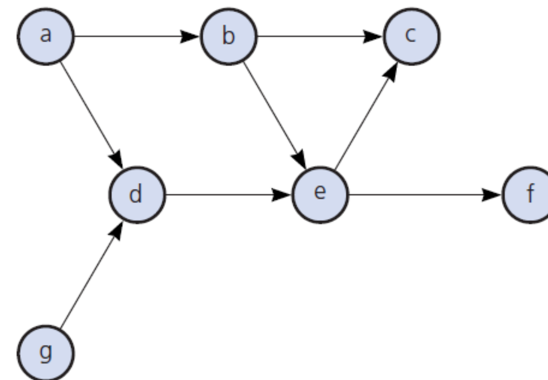
Remove e from `theGraph;`
add it to `aList`



e c f

AUTONOMOUS ROBOTS LAB

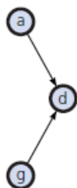
Topological Sorting



- A trace of **topSort1** for the previous figure.

Graph `theGraph` | List `aList`

Remove b from `theGraph`;
add it to `aList`



b e c f

Remove d from `theGraph`;
add it to `aList`

a

d b e c f

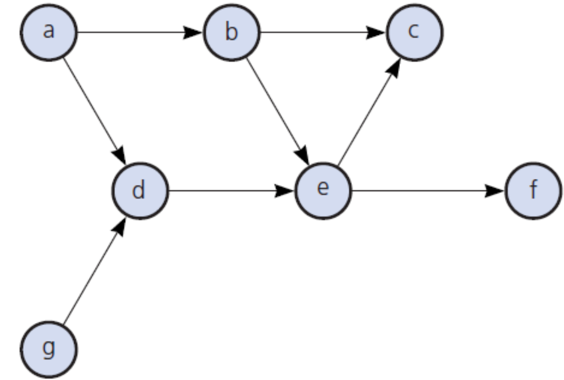
g

Remove g from `theGraph`;
add it to `aList`

a

AUTONOMOUS ROBOTS LAB

Topological Sorting



- (continued)

g

Remove g from theGraph;
add it to aList

a

g d b e c f

Remove a from theGraph;
add it to aList

a g d b e c f

AUTONOMOUS ROBOTS LAB

Topological Sorting

- **topSort2**

```
// Arranges the vertices in graph theGraph into a
// topological order and places them in list aList.
topSort2(theGraph: Graph, aList: List)
{
    s = a new empty stack
    for (all vertices v in the graph)
    {
        if (v has no predecessors)
        {
            s.push(v)
            Mark v as visited
        }
    }
    while (!s.isEmpty())
```

AUTONOMOUS ROBOTS LAB

Topological Sorting

- **topSort2**

```
    while (!s.isEmpty())
    {
        if (all vertices adjacent to the vertex on the top of the stack have been visited)
        {
            s.pop(v)
            aList.insert(1,v)
        }
        else
        {
            Select an unvisited vertex u adjacent to the vertex on the top of the stack
            s.push(u)
            Mark us as visited
        }
    }
}
```

Thank you