COMP225: Algorithms and Data Structures

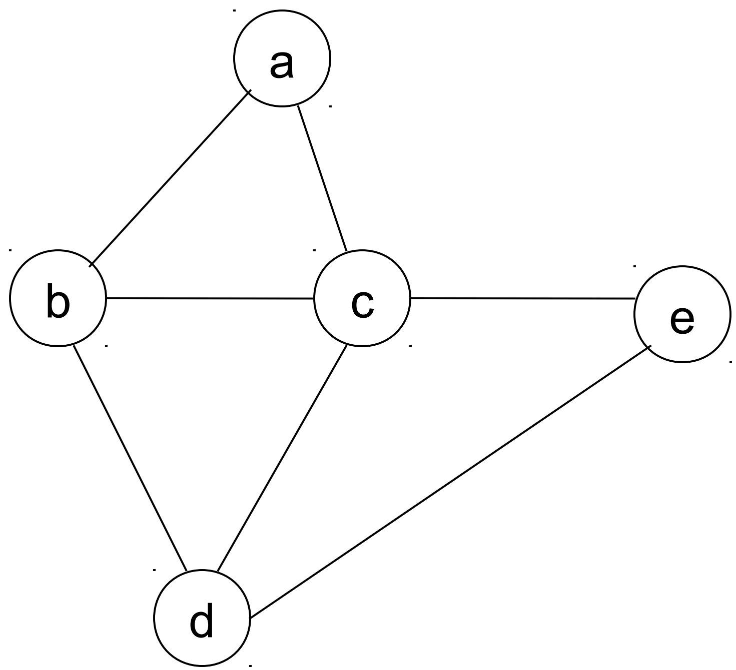
Graphs

Mark Dras
Mark.Dras@mq.edu.au
E6A380

Outline

- Definitions and examples
- Implementations
  - Matrices vs lists
- Searches and traversals
  - Depth-first
  - Breadth-first
- Code

A Graph



Definitions

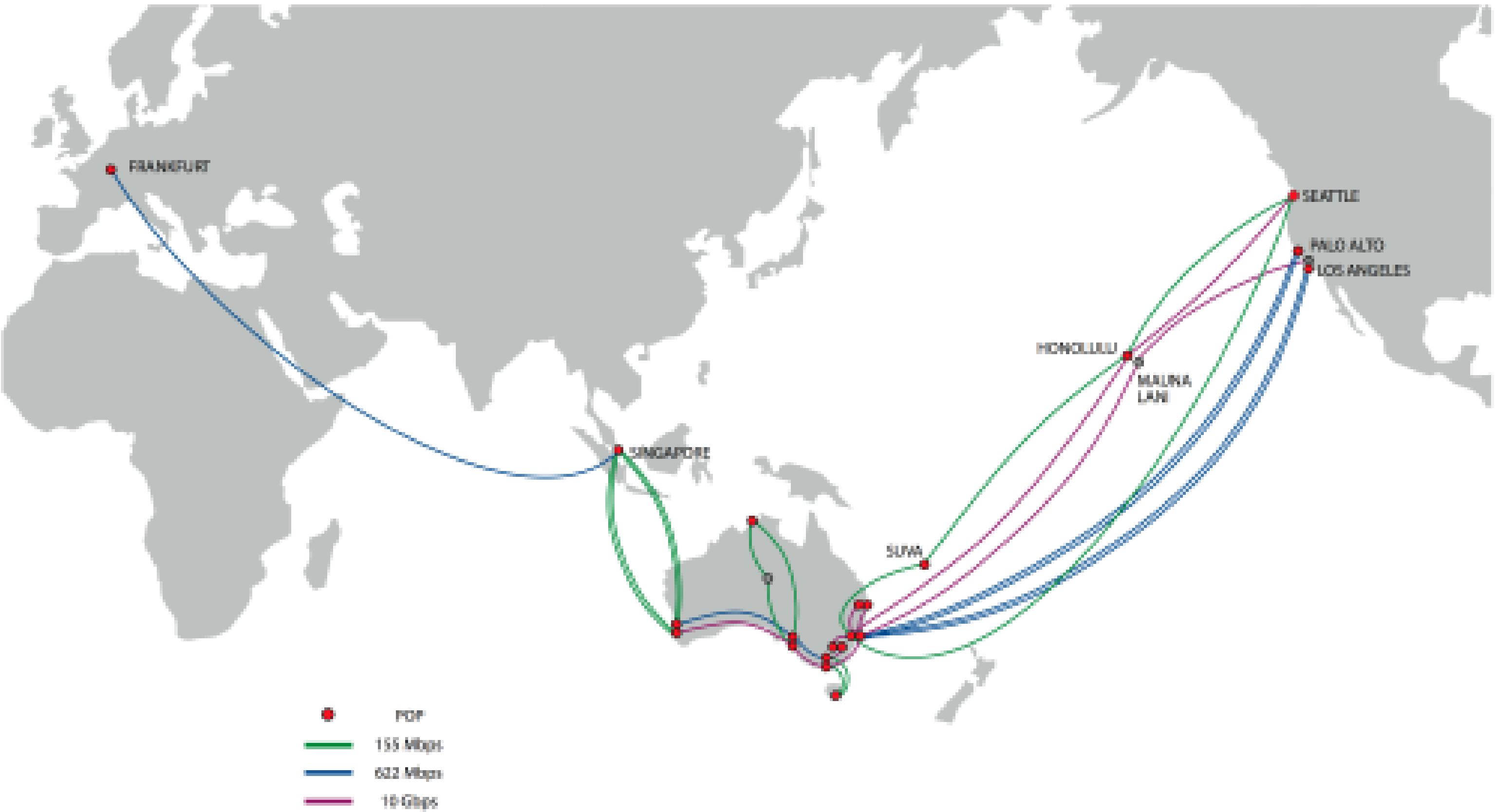
- A simple graph $G=(V,E)$ consists of
  - A set of vertices V
    - in previous graph, $V = \{ a, b, c, d, e \}$
  - A set of edges E
    - often not explicitly labelled
    - can be referred to by a pair of the names of the vertices the edges connect
    - in previous graph,

      $E = \{ (a,b), (a,c), (b,c), (b,d), (c,e), (c,d), (d,e) \}$
  - Only one edge between each pair of vertices

Uses

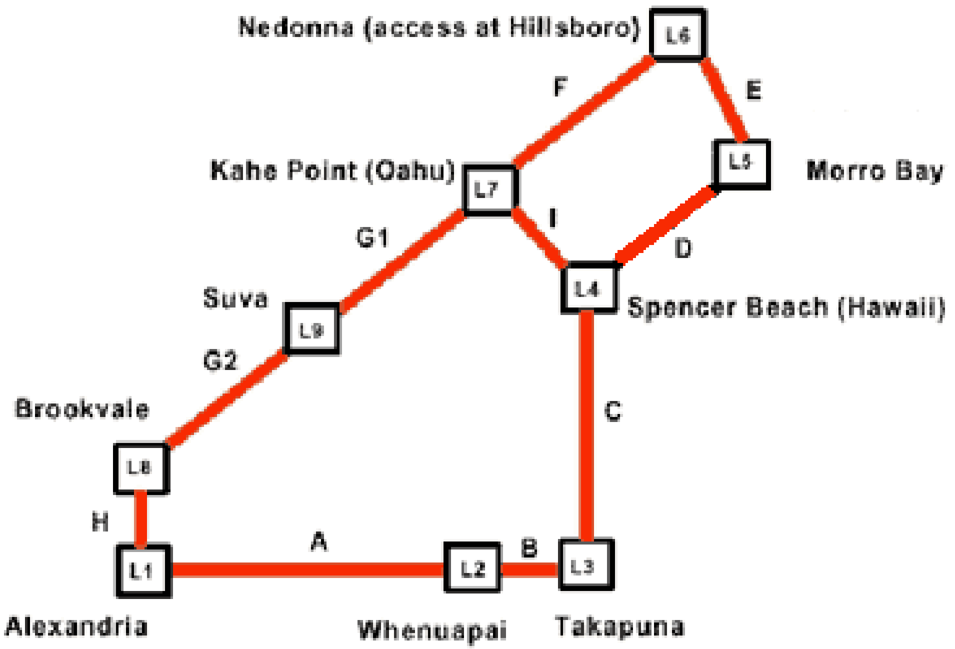
- Is one of the most general structures in computer science
- Can be used to represent physical arrangement of objects and links
  - telephone connections, road networks, routing problems
- Can be used for more abstract relations
  - friendship/acquaintance networks
- Can be used in seemingly unrelated CS applications
  - compression, games, programming language design ...
  - used heavily in AI, or areas where there's a solution space that needs to be searched

Example



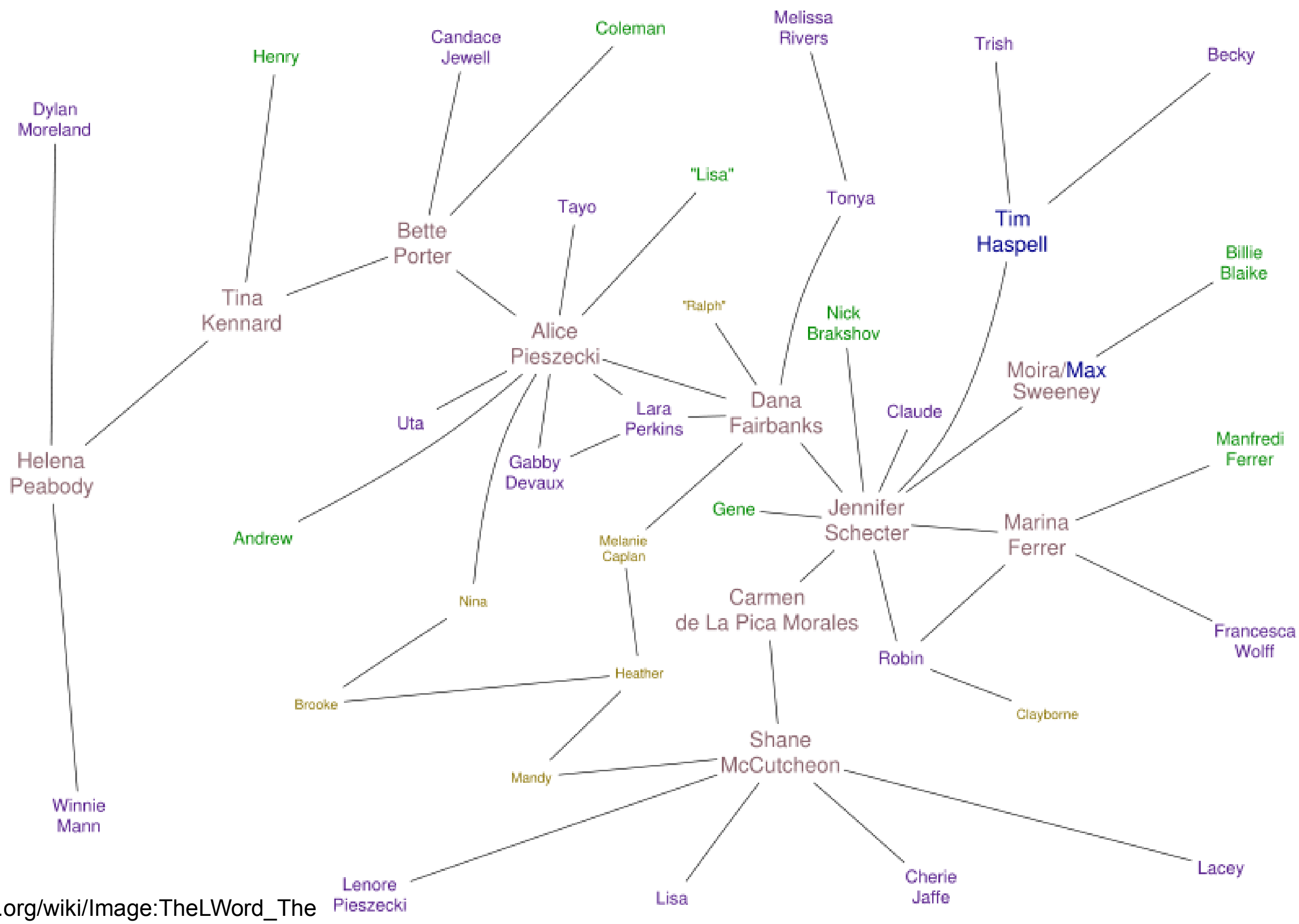
source: AARNet (http://www.aarnet.edu.au/engineering/aarnet3/)

Example



source: AARNet
(http://www.aarnet.edu.au/engineering/networkdesign/sccn/)

Example



source; Wikipedia (http://en.wikipedia.org/wiki/Image:TheLWord_TheChart.svg)

Example

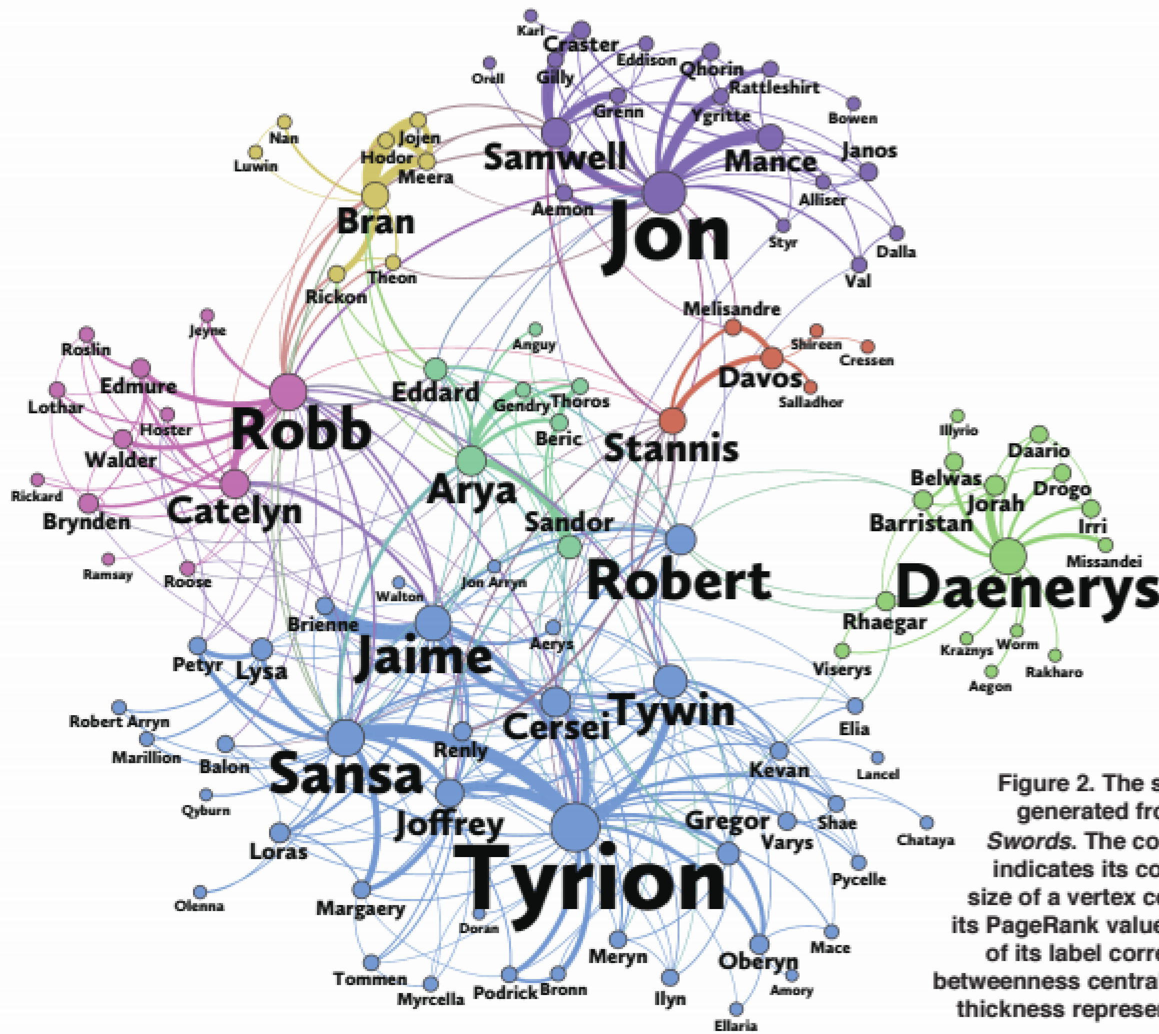


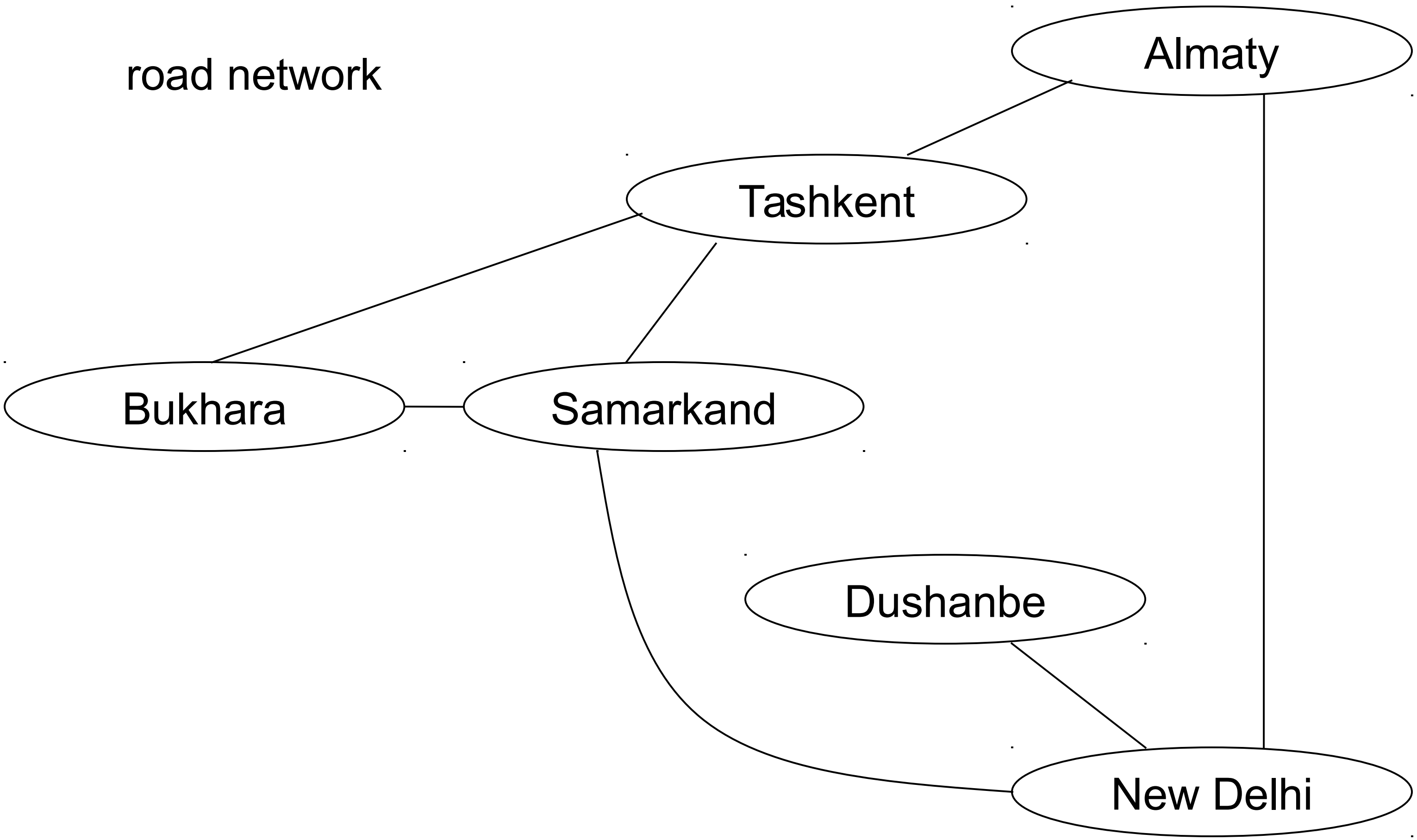
Figure 2. The social network generated from *A Storm of Swords*. The color of a vertex indicates its community. The size of a vertex corresponds to its PageRank value, and the size of its label corresponds to its betweenness centrality. An edge's thickness represents its weight.

http://qz.com/650796/mathematicians-mapped-out-every-game-of-thrones-relationship-to-find-the-main-character/

Definitions

- Degree
  - a vertex v has degree d_v if there are d_v edges connecting v to some other node
- Connectivity
  - a graph is connected if there is a path from any vertex to any other
  - otherwise it's disconnected

Example

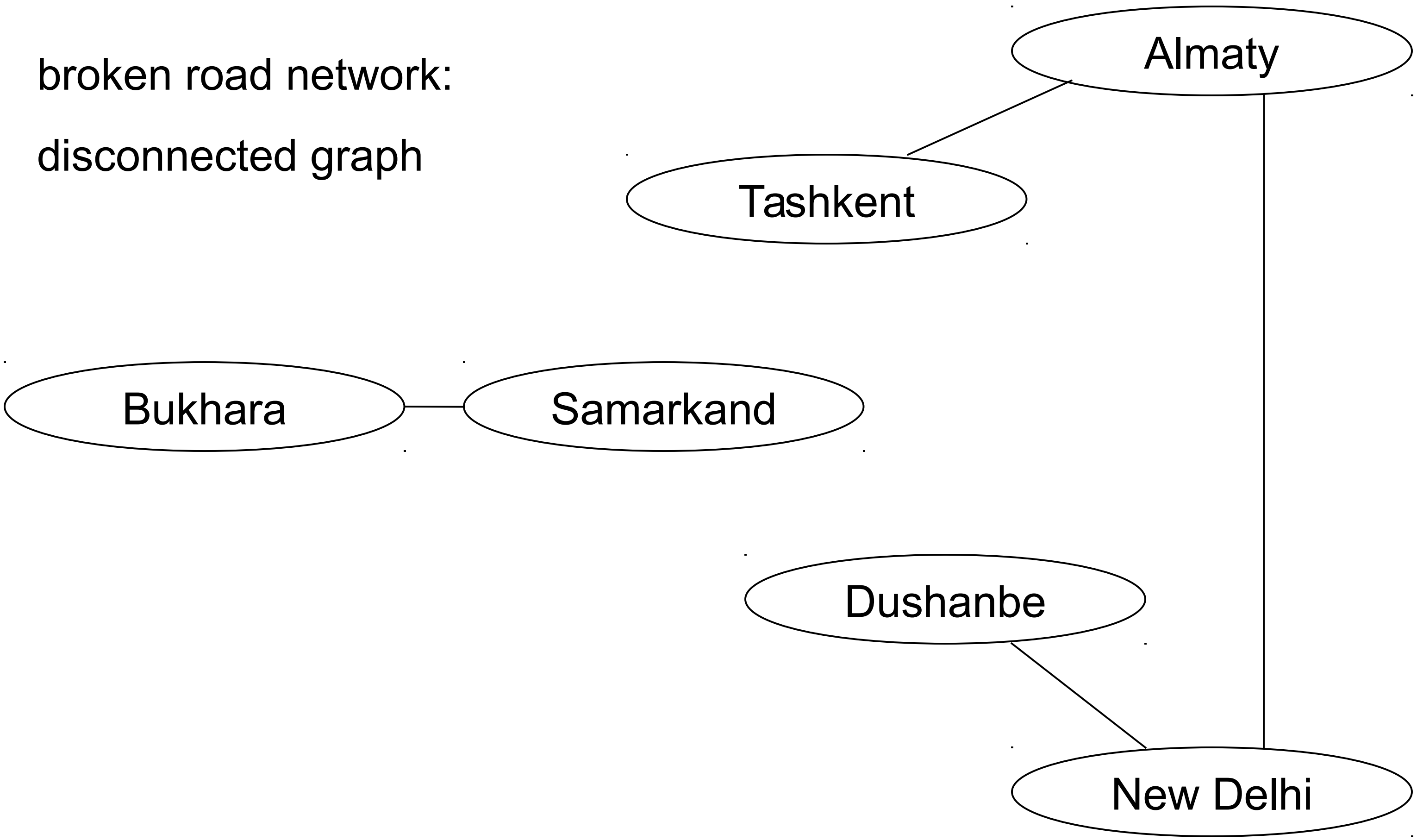


road network
Almaty
Tashkent
Bukhara
Samarkand
Dushanbe
New Delhi

Example

broken road network:

disconnected graph

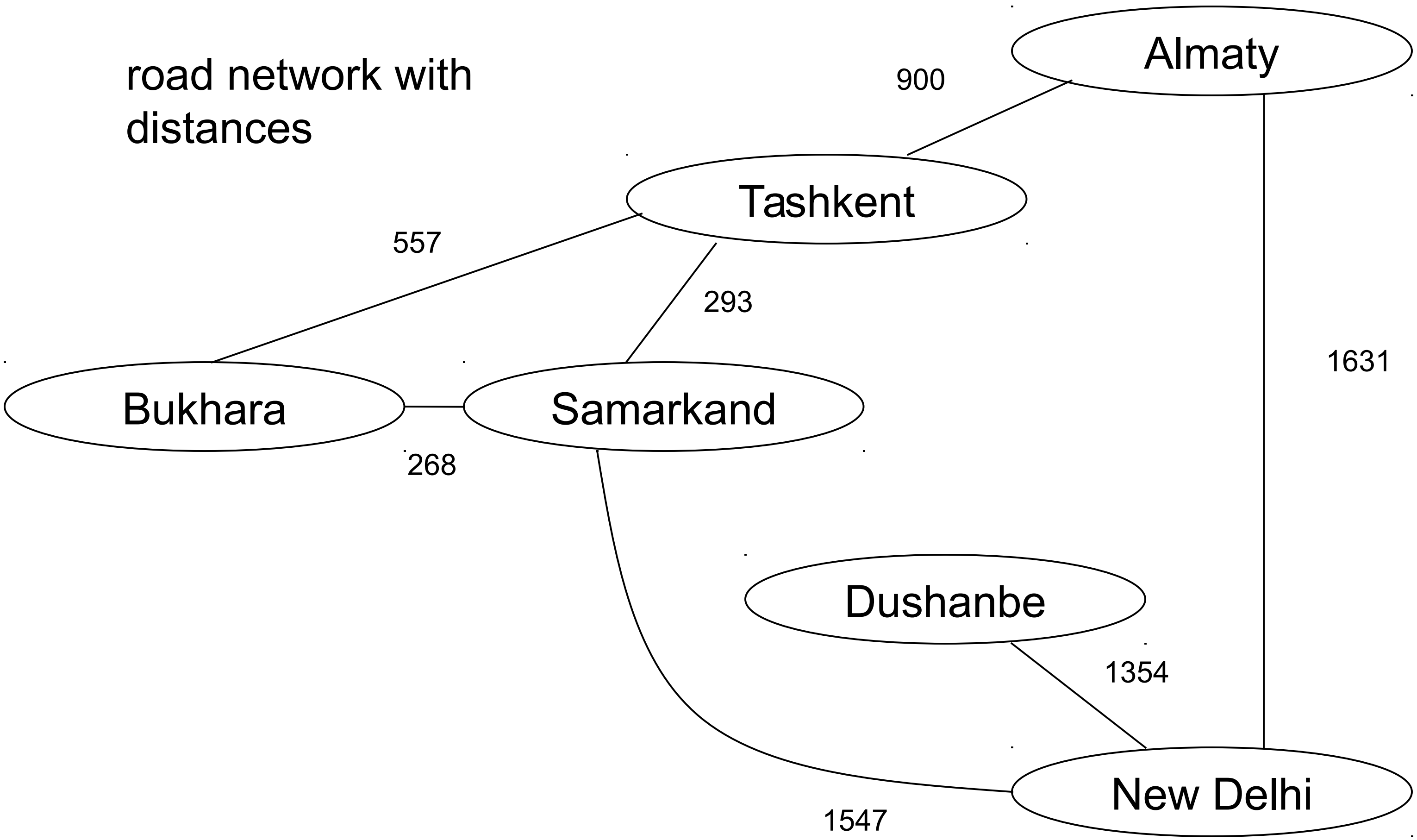


Definitions

- Weights
  - edges can have weights $w(v_1, v_2)$
  - weight function written $w : E \rightarrow R$
  - represent costs associated with traversing the edge
    - physical distance, monetary cost, ...
  - problems are then, e.g.,
    - what's the shortest / least expensive distance from x to y, or path that covers all vertices
    - what's the best graphical morph
    - ...

Example

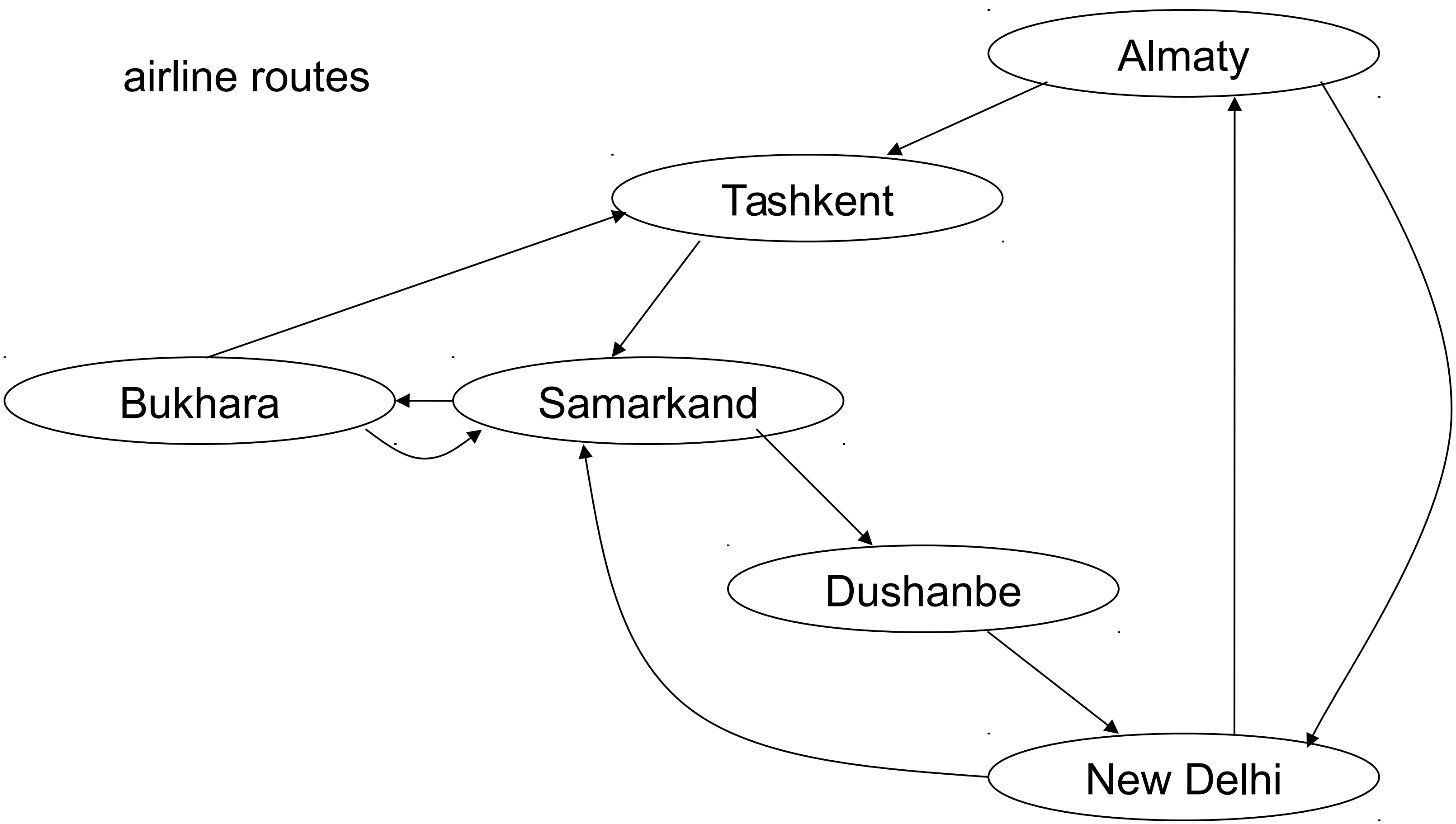
road network with distances



Definitions

- The previous graph was undirected
  - that is, the edges connect node v_1 to v_2 and vice versa
- Can have directed graphs as well
  - each edge only connects v_1 to v_2 in one direction

Example



airline routes
Almaty
Tashkent
Bukhara
Samarkand
Dushanbe
New Delhi

Some Special Graphs

- Of graphs with *m* edges, *n* vertices:
- Trees
  - a tree is a graph with minimal edges given full connectivity: $m = n - 1$
  - all tree stuff we've done is just a special case
  - difference between rooted trees and free trees
- Complete
  - every vertex is connected to every other:

    $m = n * (n - 1) / 2$
- Very often more edges than vertices

Outline

- Definitions and examples
- Implementations
  - Matrices vs lists
- Searches and traversals
  - Depth-first
  - Breadth-first
- Code

Graphs as ADTs

Operations (i.e. public members of a class Graph):

create an empty graph
destroy a graph
determine whether a graph is empty
determine the number of vertices
determine the number of edges
determine whether an edge exists between two vertices
insert a vertex
insert an edge between two given vertices
delete a vertex (plus associated edges)
delete the edge between two given vertices
retrieve a given vertex
determine the degree of a vertex v
retrieve the set of vertices adjacent to v
...

Graph Implementations

- Adjacency matrix
  - for *n* vertices, an *n x n* matrix *aMatrix*
  - unweighted:

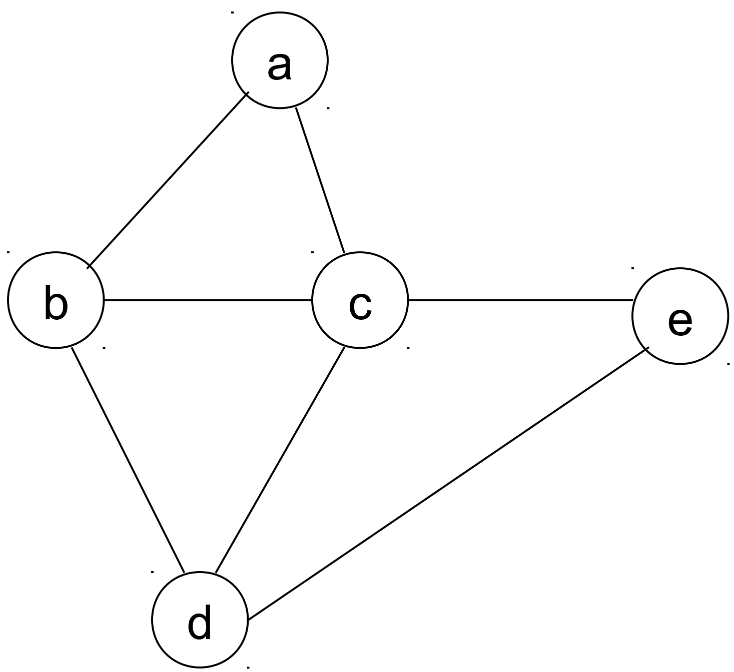
    aMatrix[i][j] = 1 if *i* and *j* are connected by an edge

    aMatrix[i][j] = 0 otherwise
  - weighted

    aMatrix[i][j] = w(i,j) if *i* and *j* are connected by an edge

    aMatrix[i][j] = ∞ otherwise

A Graph and Its Matrix

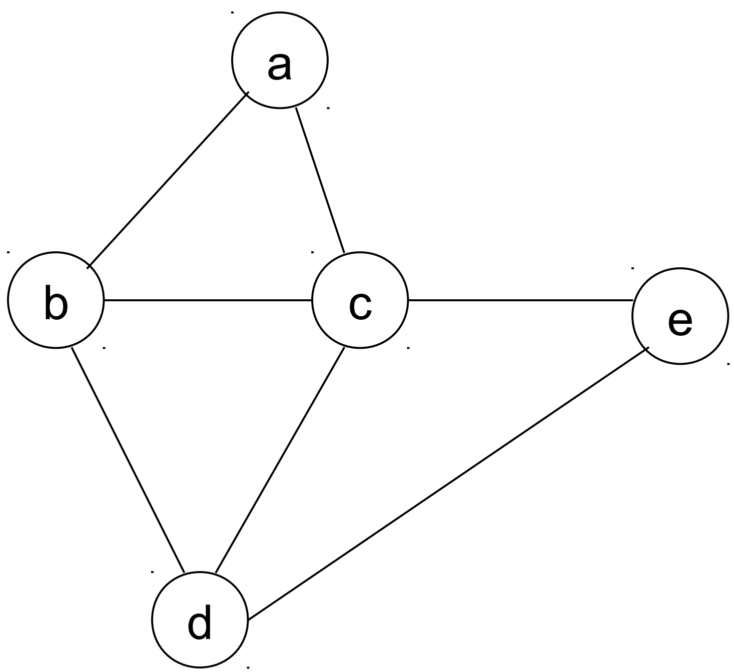


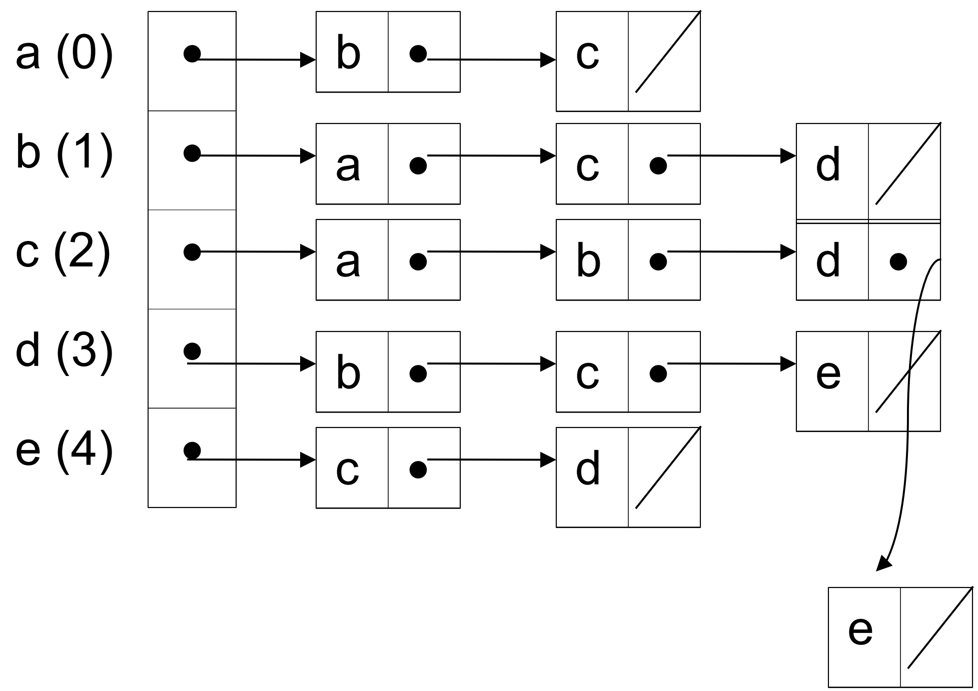
	a (0)	b (1)	c (2)	d (3)	e (4)
a (0)	0	1	1	0	0
b (1)	1	0	1	1	0
c (2)	1	1	0	1	1
d (3)	0	1	1	0	1
e (4)	0	0	1	1	0

Graph Implementations

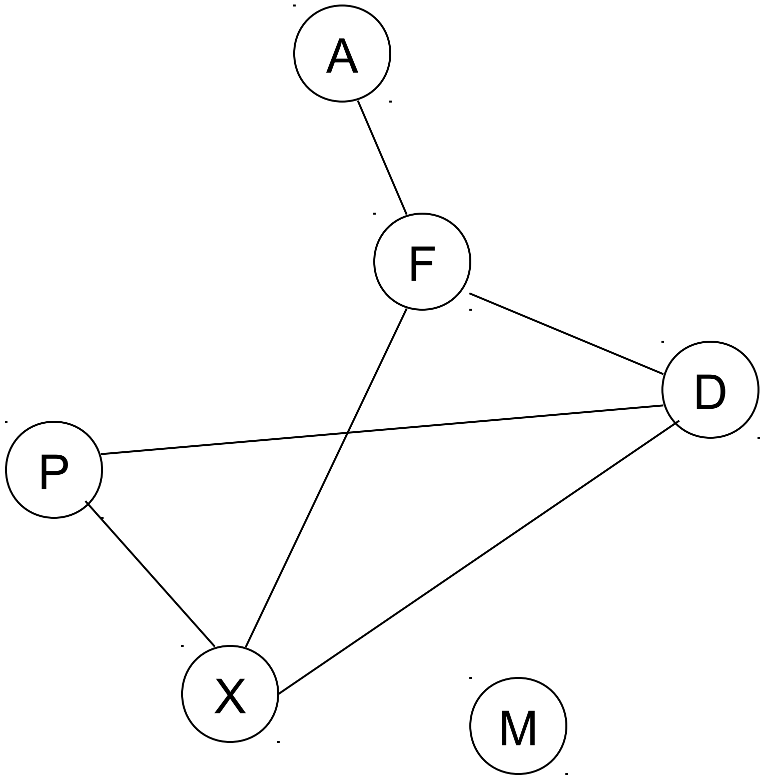
- Adjacency list
  - for n vertices, an n-length array of linked lists
- Unweighted
  - the list items are just vertex indices plus pointers
- Weighted
  - list items also include weights

A Graph and Its List





Exercise: Graph representation



Which Is Better?

- For time complexity, depends on purpose
  - determine whether there is an edge between v_i and v_j : ?
  - find all vertices adjacent to v_i : ?
- For space requirements, depends on graph

Converting Implementations

- How about if you wanted to convert one implementation to another
  - to convert an adjacency list to an adjacency matrix, you just traverse the list, and for each item add it to the matrix (an O(1) operation)
  - therefore, an O(E) operation (equal to the number of edges; can also write O(m))
  - what about the reverse?

Outline

- Definitions and examples
- Implementations
  – Matrices vs lists
- Searches and traversals
  – Depth-first
  – Breadth-first
- Code

Graph Traversals

- Difference between a search and a traversal
  - search usually refers to moving from vertex to vertex until a target is found
  - traversal refers to visiting every reachable vertex
- In a graph, not every vertex is reachable if the graph isn't connected
  - different from trees

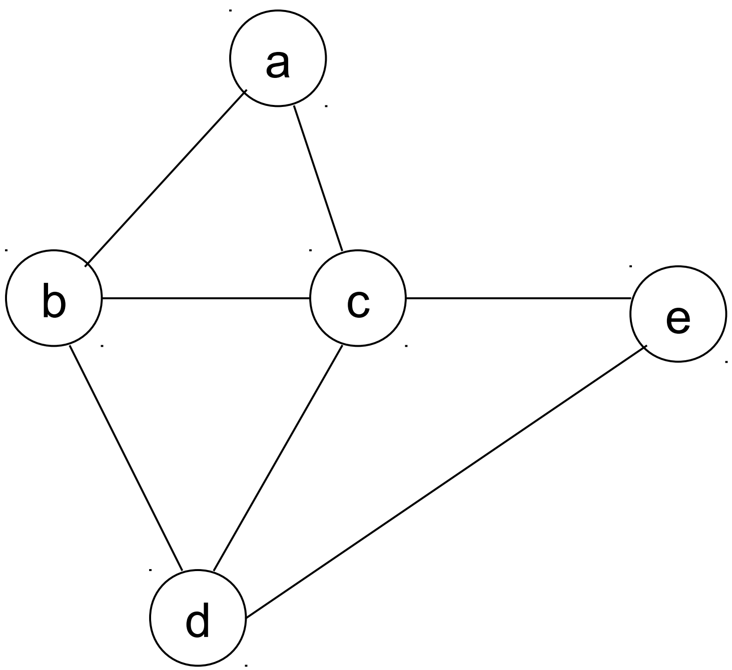
Graph Traversals

- What they're for
  - they're a fundamental process underlying all the others
    - need to traverse graph to work out e.g. lowest cost telephone network connectivity, or best alignment of pixels
- Potential problem: loops

Definitions

- Path
  - a path is a sequence of edges from one vertex v_i to another v_j
- Cycle
  - a cycle is a path beginning and ending at the same vertex v_i
- In a traversal, want to avoid cycles

A Graph: Paths and Cycles



paths: a-b-d

c-e-d-b-a

e-c

...

cycles: a-b-c-a

a-c-e-d-b-a

...

Graph Traversal

- Need some way of marking whether nodes have been visited, to avoid looping
- Then, two choices about how to search outwards
  - depth-first: when a vertex v_i is encountered, explore its neighbours before exploring the vertices that were encountered at the same time as v_i
  - breadth-first: the reverse

Graph Traversal

- At most vertices, there's a choice of edges
  - a lot of graphs have some ordering on the vertices (e.g. numerical, alphabetical)
  - this is important in dynamic programming

Depth-First Traversal

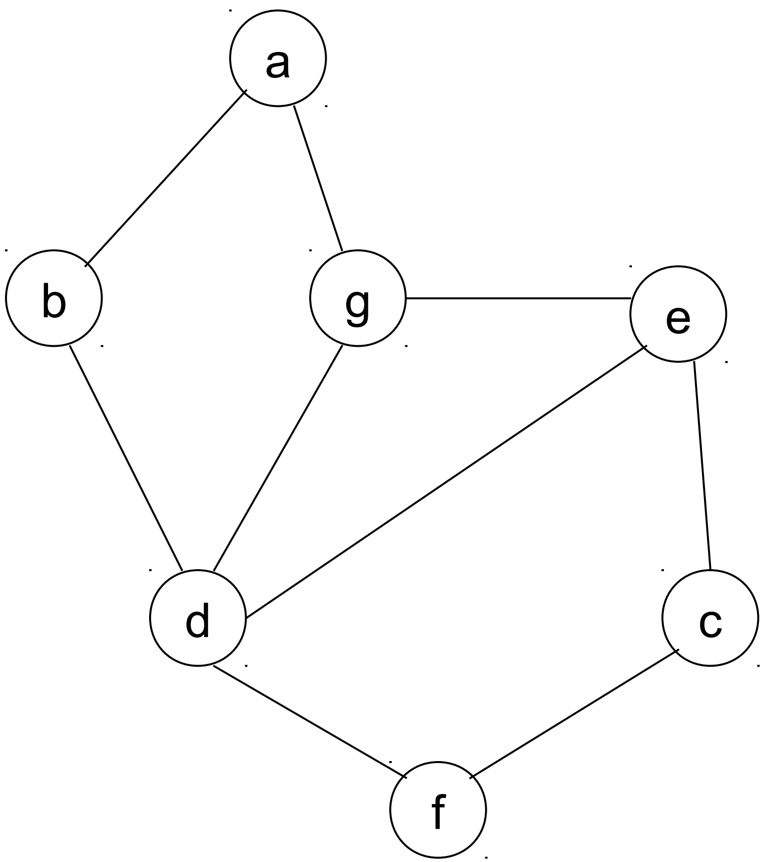
```
depthFirstSearch()
    for all vertices v
        num(v) = 0;
    edges = null;
    i = 1;
    while there is a vertex v such that num(v) == 0
        DFS(v);
    output edges;

dfs (v)
    num(v) = i++;   // mark v as visited
    for (all vertices u adjacent to v)
        if num(u) == 0    // u is unvisited
            attach edge(uv) to edges   // mark (u,v) as a discovery edge
            dfs(u)
```

Depth-First Traversal

```
dfs ()
   for all vertices v
      num(v) = 0;
   edges = null;
   i = 1;
   s.createStack()
   while there is a vertex v such that num(v) == 0
      num(v) = i++;    // mark v as visited
      s.push(v)
      // loop invariant: there is a path from vertex at bottom of stack to vertex at top of stack
      while (!s.isEmpty()) {
         u = s.top()
         if (no unvisited vertices adjacent to u)
            s.pop()                               // backtrack
         else {
            select an unvisited vertex w adjacent to vertex u on top of stack
            s.push(w)
            num(w) = i++;    // mark w as visited
            attach edge(uw) to edges    // mark (u,w) as discovery edge
         }
      }
   }
   output edges
}
```

Example: Depth-First



depth-first traversal (starting from a):

a-b-d-e-c-f-g

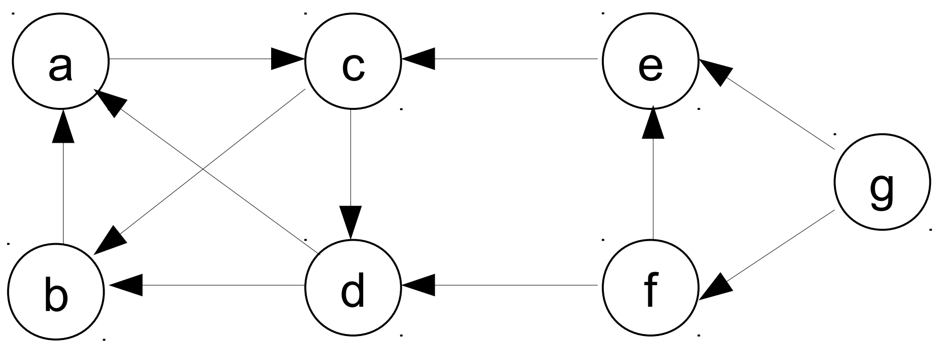
Traversal in a Directed Graph

- Concepts are slightly different wrt undirected graphs—assume you have an edge (v_i, v_j) but not an edge (v_j, v_i)
  - v_j is a neighbour of v_i but not vice versa
  - v_j is adjacent to v_i but not vice versa
- Otherwise algorithms are the same

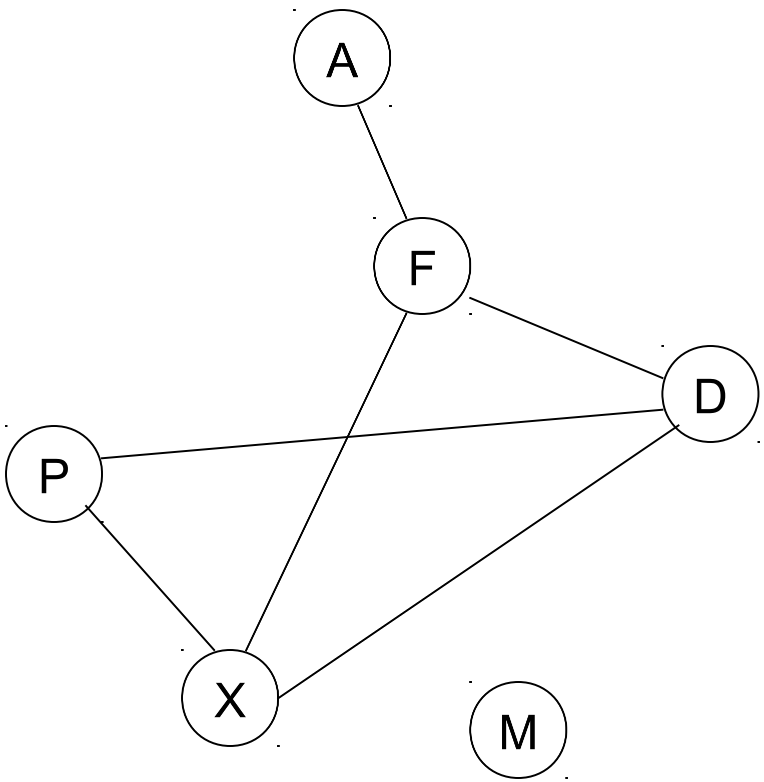
Example: Depth-First

depth-first traversal (starting from a, choosing next letter for outer while loop):

a-c-b-d-e-f-g



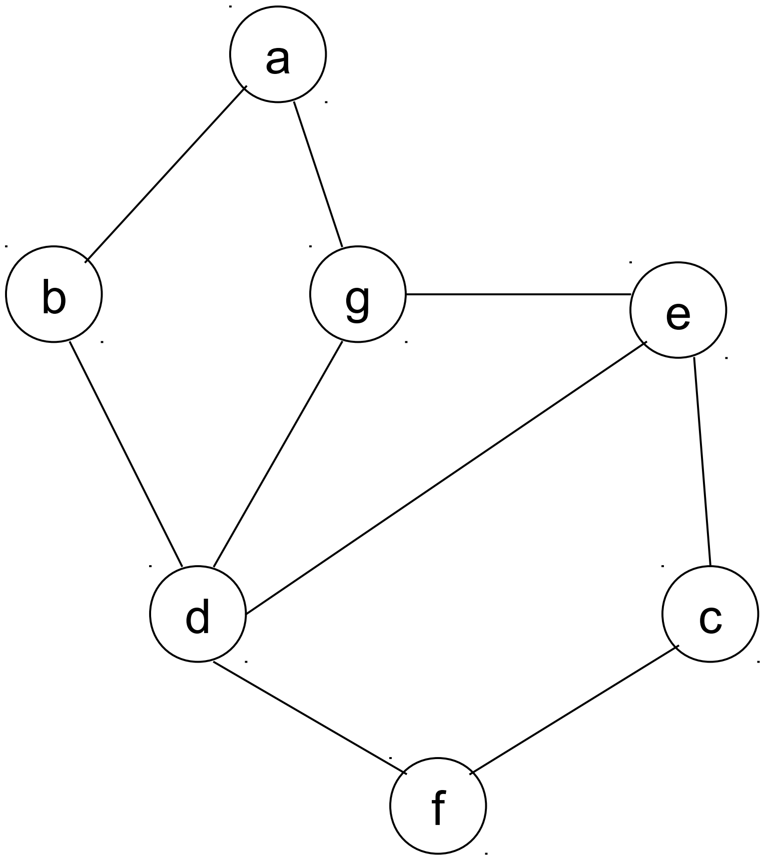
Exercise: DFT



Breadth-First Traversal

```
bfs ()
    for all vertices v
        num(v) = 0;
    edges = null;
    i = 1;
    q.createQueue();
    while there is a vertex v such that num(v) == 0
        num(v) = i++;      // mark v as visited
        q.push(v);
        while (!q.isEmpty()) {
            v = q.pop()
            // loop invariant: there is a path from former front of queue to every vertex in queue
           for (all unvisited vertices u adjacent to v) {
                num(u) = i++;    // mark u as visited
                attach edge(vu) to edges    // mark (v,u) as discovery edge
                q.push(u)
            }
        }
    }
    output edges
}
```

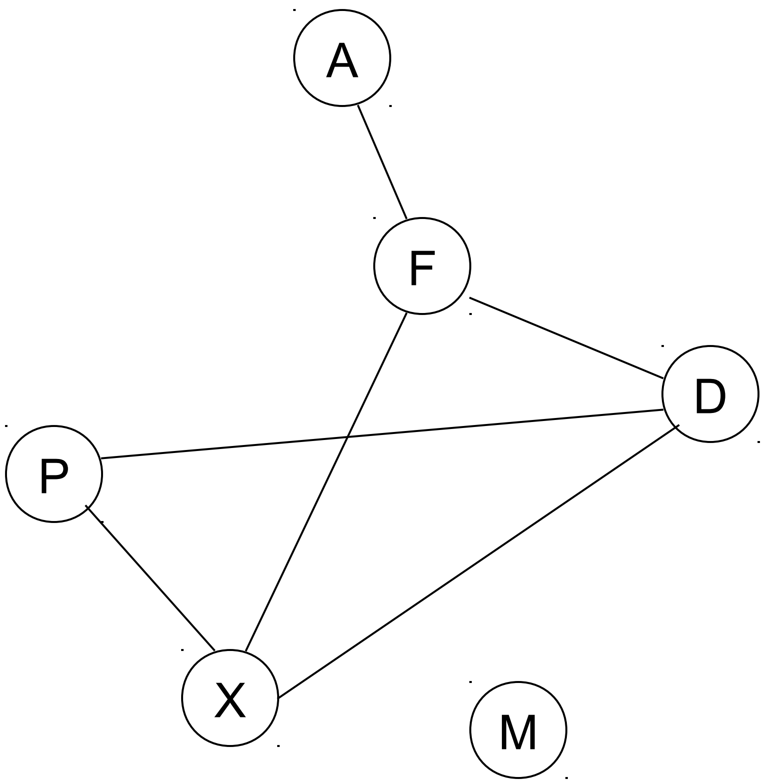
Example: Breadth-First



breadth-first traversal (starting from a):

a-b-g-d-e-f-c

Exercise: BFT

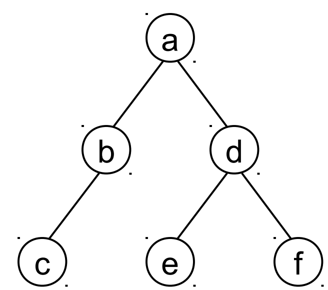


Comparison

- For a graph *G=(V,E)*, for both DFT and BFT there are O(V+E) algorithms for
  - the traversal itself
  - testing whether G is connected
  - computing a spanning tree for G, if G is connected
  - computing a cycle in G

Comparison

- Data structures
  - DFS uses stack (most recently added is first explored)
  - BFS uses queue (first added is first explored)
- BFS has no easy recursive version
  - consider trees
  - DF recursive is context-free



Outline

- Definitions and examples
- Implementations
  - Matrices vs lists
- Searches and traversals
  - Depth-first
  - Breadth-first
- Code

Code

Edge

- represents edges as pairs of integers

VertexIDList

- a representation of the adjacency list

Vertex

Graph

- a structure containing vertices and edges, with basic operations

GraphApplic

- extensions of basic operations

DF Traversal Code

```
public void depthFirstTraversalRec1(Integer v) {
// PRE: v is the id of a vertex in the graph
// POST: Prints out a depth-first traversal of a graph
//         starting from v
//         (for just the connected component containing v)

// Recursive version of DFT
  System.out.print(" " + v);
  getVertex(v).setMarked(); // get vertex object with id v,
                            // indicate visited by setting marked
  VertexIDList adjList = getVertex(v).getAdjs();
                    // get adjacency list representing neighbours
  Iterator<Integer> vIt = adjList.iterator();
  while (vIt.hasNext()) {   // iterate over neighbours
    Integer nextVertex = vIt.next();
    if (!getVertex(nextVertex).isMarked())
                            // if neighbour hasn't been visited
      depthFirstTraversalRec1(nextVertex); // visit it
  }
}
```

Detect Cycle Code

```
public boolean detectCycle() {
  return detectCycleAux(this.getFirstVertexID());
}

public boolean detectCycleAux(Integer v) {
// Only for undirected graphs

  getVertex(v).setMarked();
  VertexIDList adjList = getVertex(v).getAdjs();
  boolean foundCycle = false;
  Integer numAdjMarked = 0;

  Iterator<Integer> vIt = adjList.iterator();
  while (!foundCycle && vIt.hasNext()) {
    Integer nextVertex = vIt.next();
    if (!getVertex(nextVertex).isMarked())
      foundCycle = foundCycle || detectCycleAux(nextVertex);
    else
      numAdjMarked++;
  }
  foundCycle = foundCycle || (numAdjMarked > 1);

  return foundCycle;
}
```