Estruturas de Informação

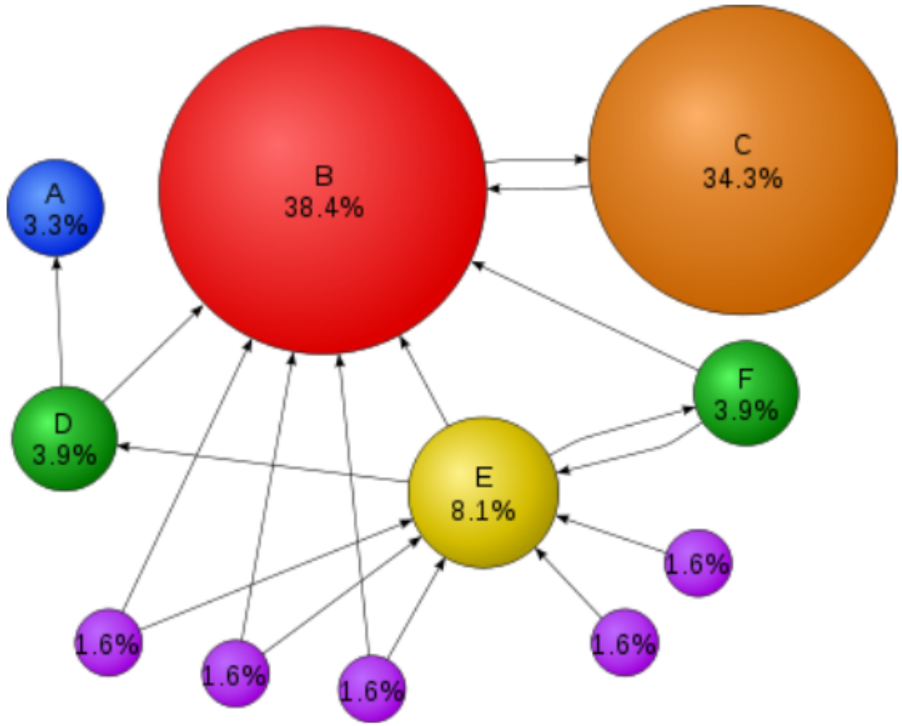
Graphs

Fátima Rodrigues
mfc@isep.ipp.pt
Departamento de Engenharia Informática (DEI/ISEP)

Why do we care about graphs?

Many Applications

- Social Networks – Facebook
- Google – Relevance of webpages
- Delivery Networks/Scheduling/Routing – UPS
- Task Scheduling in Projects
- …



Graphs

Formal definition

A graph is a pair (V, E) where:

- V is a collection of nodes, called Vertices
- E is a collection of pairs of vertices, called Edges

To each graph edge there is associated a pair of graph vertices

$$\forall_{e \in E} \quad e \rightarrow (u,v) \quad u,v \in V$$

Informal definition

Graphs represent general relationships or connections

– Each node may have many predecessors

– There may be multiple paths (or no path) from one node to another

– Can have cycles or loops

Graphs: Vertices and Edges

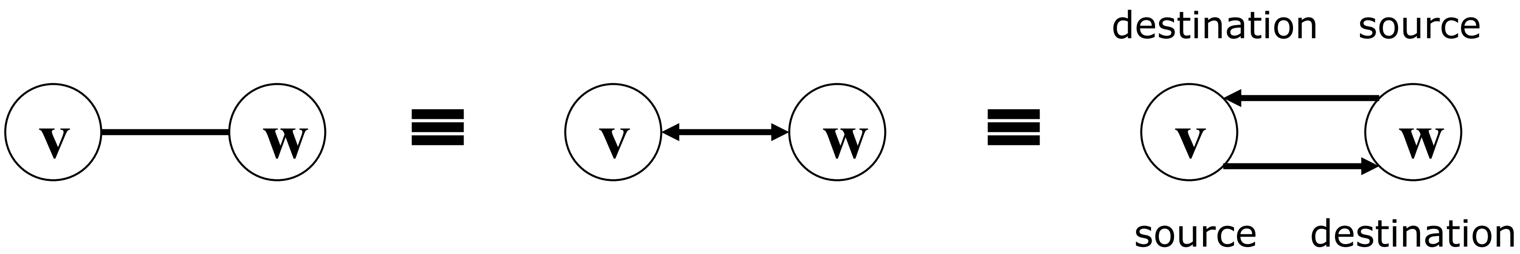
A graph is composed of vertices and edges

- Vertices (nodes):
  - Represent objects, states, positions, place holders
  - Set $\{v_1, v_2, \ldots, v_n\}$
  - Each vertex is **unique** → no two vertices represent the same object/state

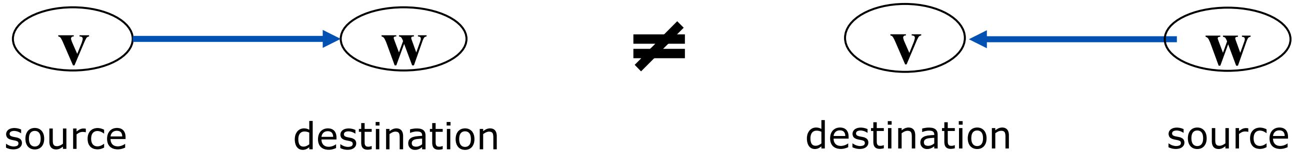
- Edges (arcs):
  - Can be directed or undirected
  - Can be weighted (or labeled) or unweighted

Directed and Undirected Edges

- An undirected edge $e = (v_i, v_j)$ indicates that the relationship, connection, etc. is bi-direction:
  - Can go from vi to vj (i.e., vi is related to vj) and vice-versa

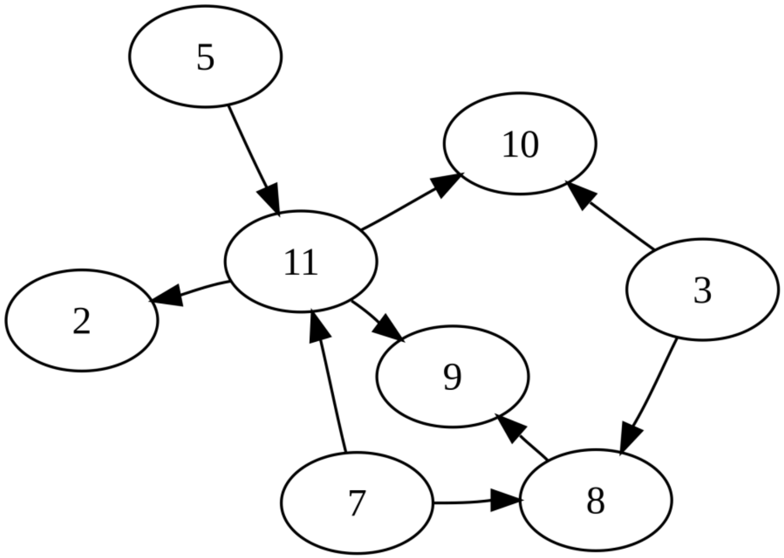


- A directed edge $e = (v_i, v_j)$ specifies a one-directional relationship or connection:
  - Can only go from v_i to v_j

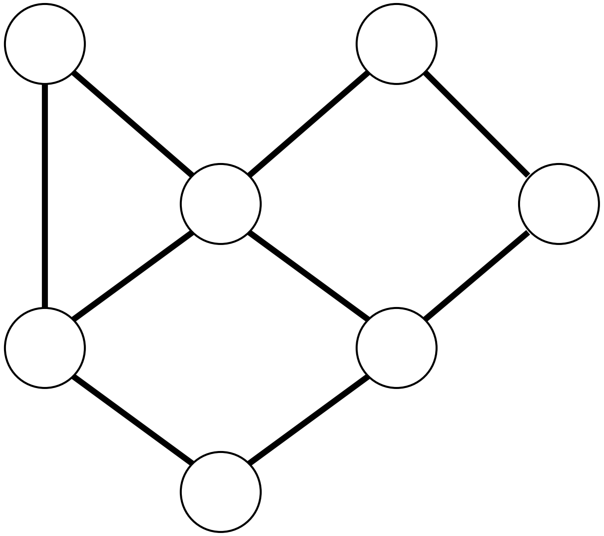


Graphs: Directed and Undirected

- A graph will have either directed or undirected edges, but **not both**



Ex. route network



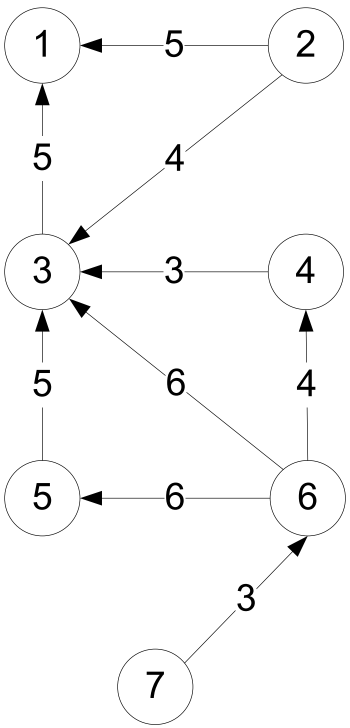
Ex. friends network

Valorised graph

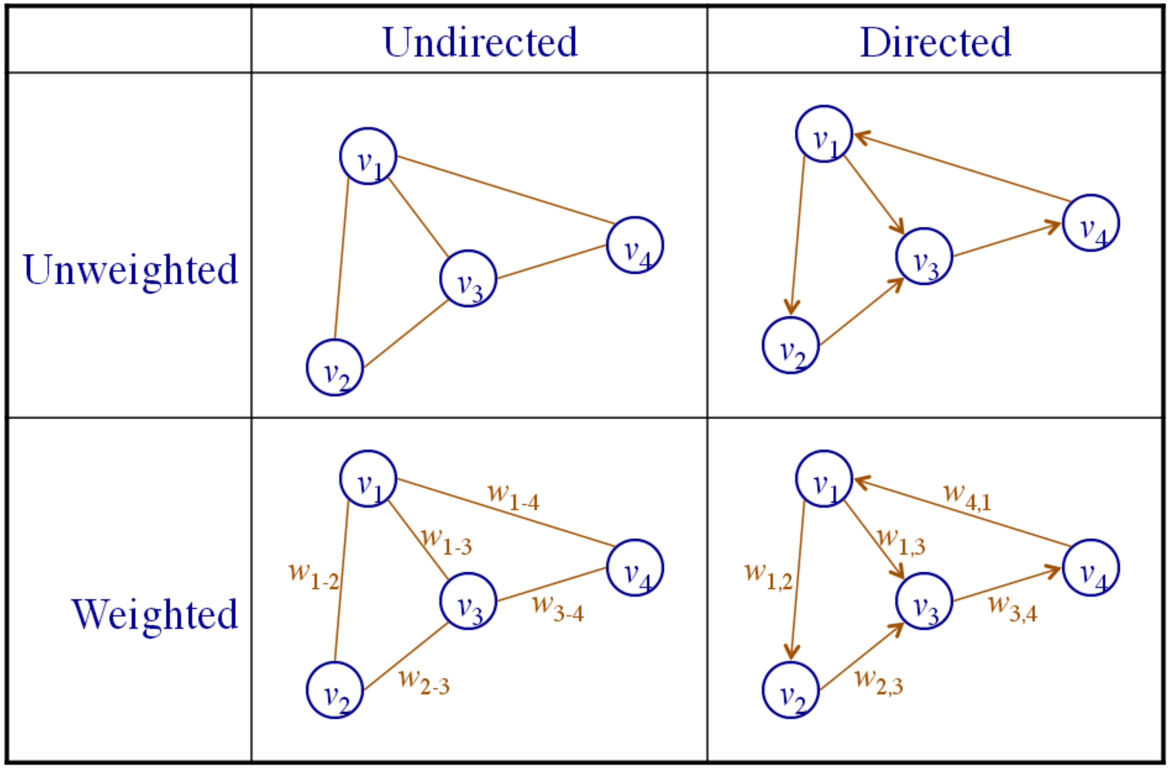
Graph that all its branches have an associated value

These values can represent:

- Costs, distances, or search limitations
- Traffic time
- Waiting time
- Transmission reliability
- Probability of failure occur
- Capacity
- Others



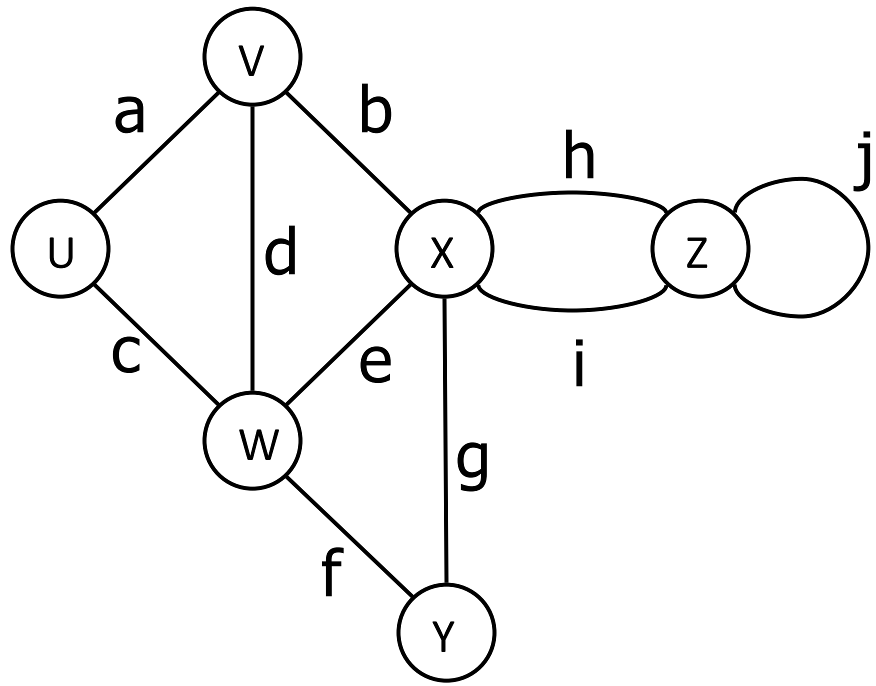
Graphs: Types of Edges



- An edge is said to be **incident** to a vertex if the vertex is one of the edge's endpoints.
- The **outgoing edges** of a vertex are the directed edges whose origin is that vertex.
- The **incoming edges** of a vertex are the directed edges whose

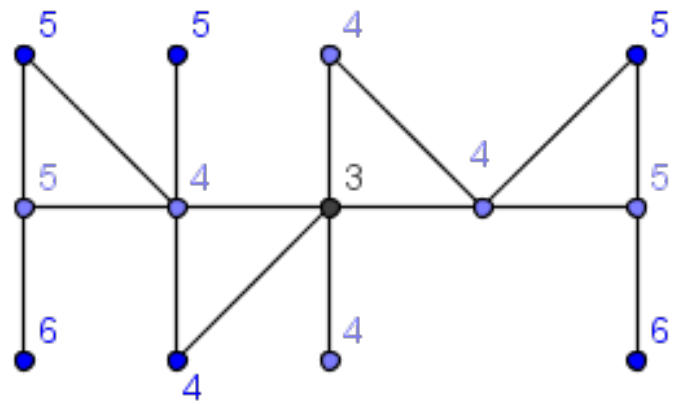
Graph Terminology

- End vertices (or endpoints) of an edge
  - u and v are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on v
- Adjacent vertices
  - u and v are adjacent
- Degree of a vertex
  - x has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



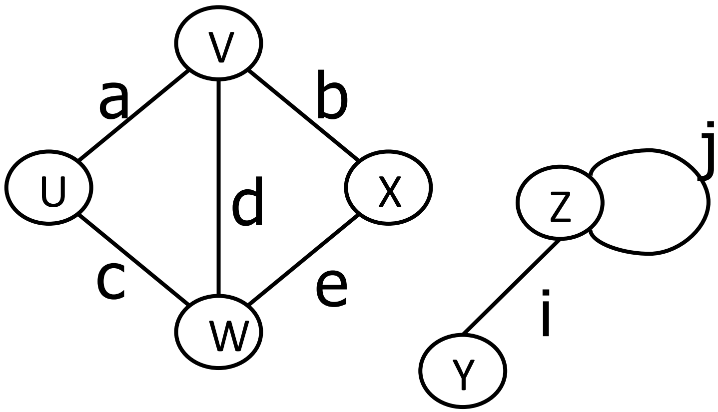
Graph Terminology

- The centrality of a vertex V in G cent(V) is the maximum length among all shortest paths from V
- The radius of G rad(G) is the value of the smallest centrality
- The diameter of G diam(G) is the value of the greatest centrality
- The center of G is the set of vertices V, such that cent(v)=rad(G)

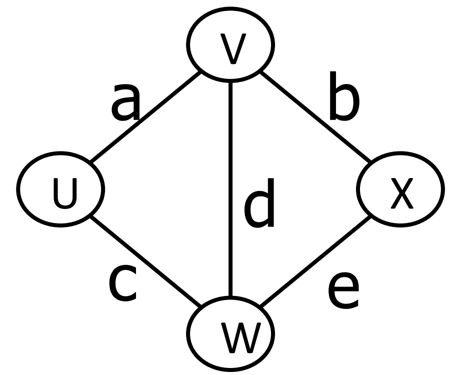


Graph Terminology

- **Connected Graph**
  – Graph where any two vertices are connected by some path



Not connected graph

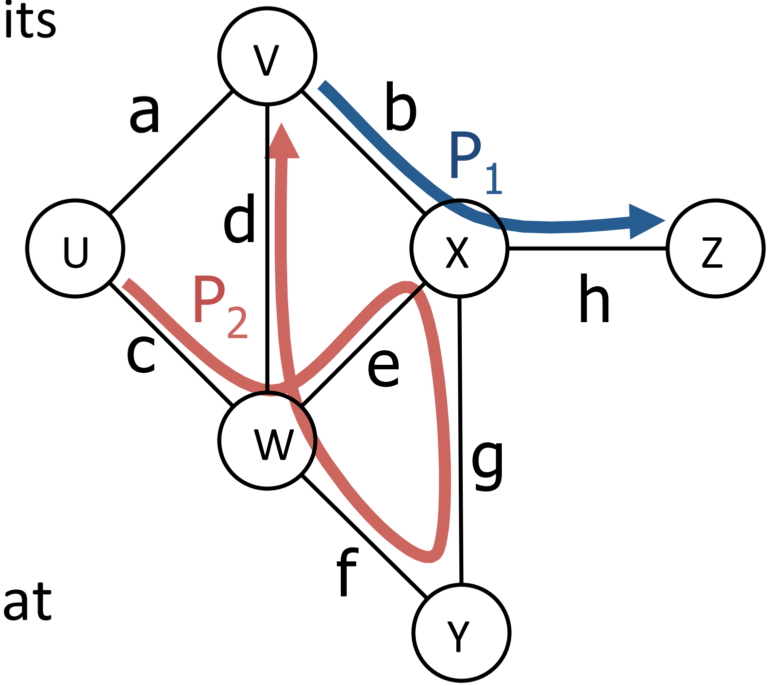


Connected graph

- **Subgraph** (V’,E’) of a Graph (V,E)
  – V’ is a subset of V, E’ is a subset of E
- **Spanning subgraph** of G is a subgraph that contains all vertices of G

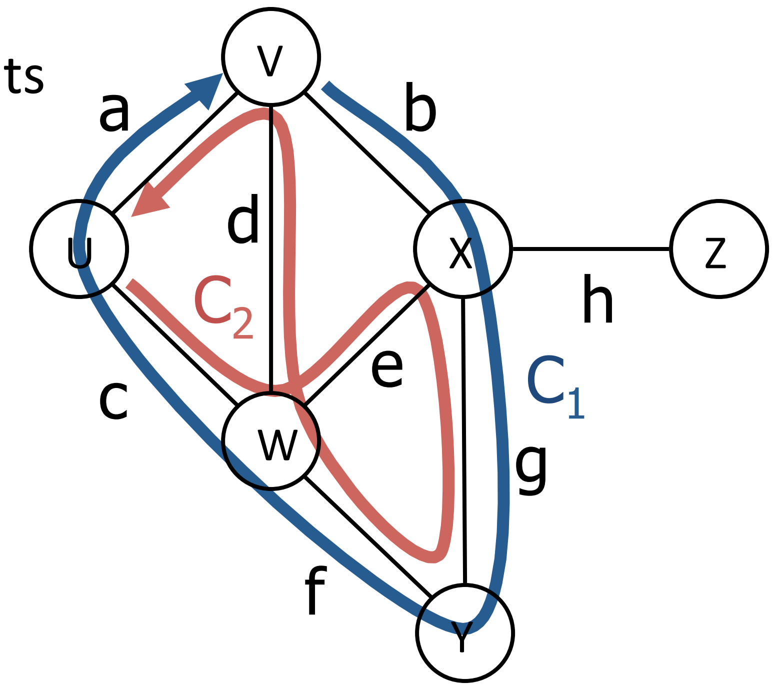
Graph Terminology

- **Path**
  - sequence of alternating vertices and edges
  - begins and ends with a vertex
  - each edge is preceded and followed by its endpoints
- **Simple path**
  - path such that all its vertices and edges are distinct
- Examples
  - P1=(V,b,X,h,Z) is a simple path
  - P2=(U,c,**W**,e,X,g,Y,f,**W**,d,V) is a path that is not simple

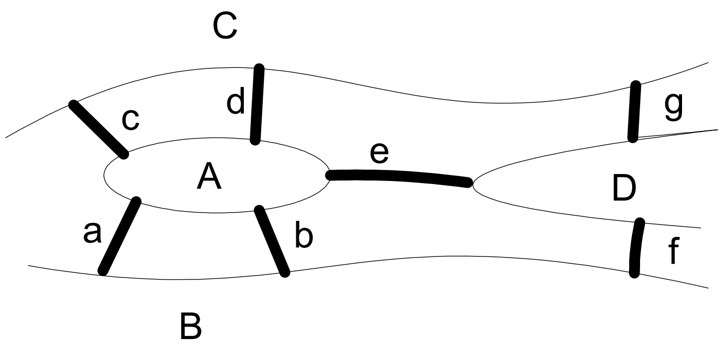


Graph Terminology

- **Cycle**
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- **Simple cycle**
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,V) is a simple cycle
  - C_2=(U,c,**W**,e,X,g,Y,f,**W**,d,V,a,U) is a cycle that is not simple

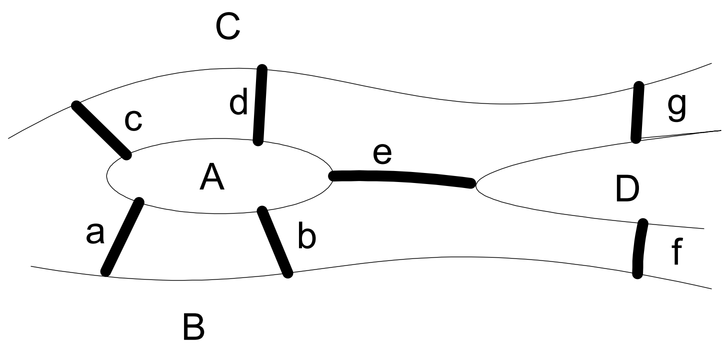


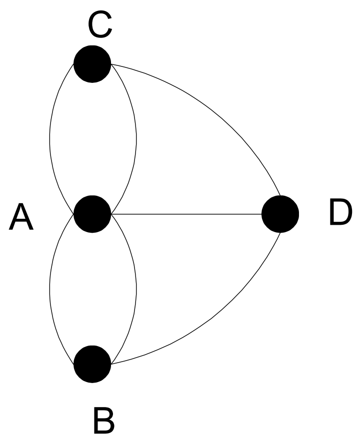
Euler Cycle and the 7 Bridges of Koenigsberg



- The year is 1735. City of Koenigsberg (today Kaliningrado) has a funny layout of 7 bridges across the river
- Citizens of Koenigsberg are wondering if it's possible to walk across each bridge exactly once and return to same starting point?
- They think that it's impossible, but no one can prove it

Euler Cycle





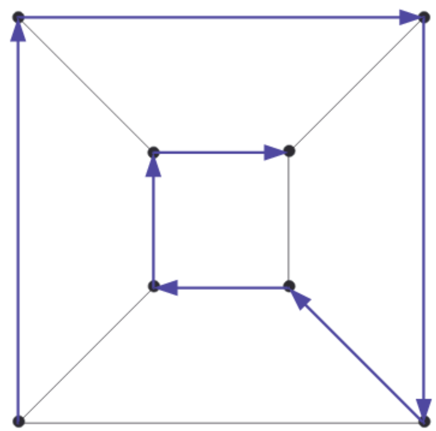
This problem was solved by Euler in 1736 and marks the beginning of Graph Theory

Euler proved

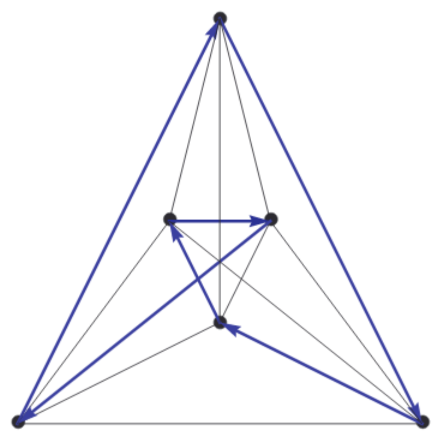
- An undirected and connected graph has an Euler Cycle iff all the vertices have an even degree
- A directed and strongly connected graph has an Euler Cycle iff $d_{in}(V) = d_{out}(V)$ for each vertex V

Hamilton Path/Cycle

- A simple path/cycle that visits all the vertices of the graph exactly once



Hamilton path



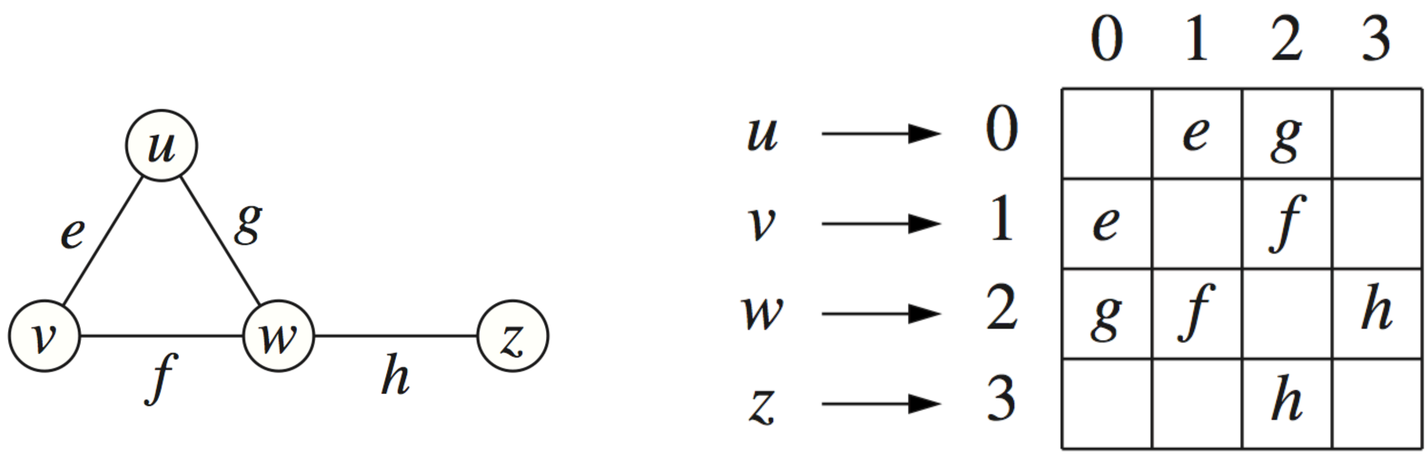
Hamilton cycle

- Unlike the Euler circuit problem, finding Hamilton circuits is hard
- There is no simple set of necessary and sufficient conditions, and no simple algorithm
- The best algorithms known for finding a Hamilton circuit in a graph or determining that no such circuit exists have exponential worst-case time complexity (in the number of vertices of the graph)

Graph Representations

Adjacency Matrix Structure

- Represents a graph as a 2-D matrix
- Vertices are indices for rows and columns of the matrix
- Total Space: O(V^2)



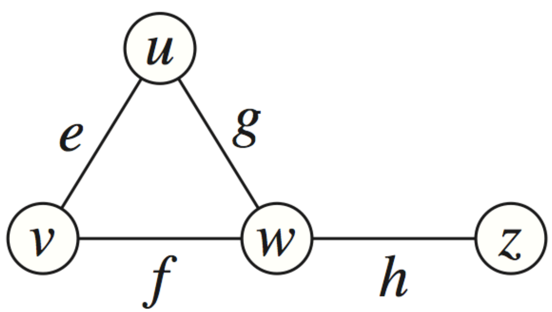
Therefore adjacency matrix should be used only for dense graphs

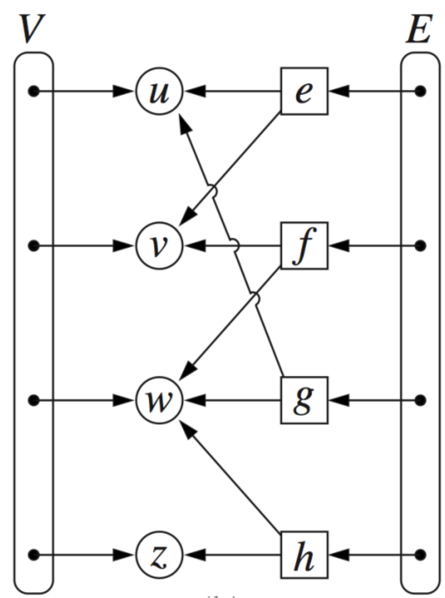
graph is dense if $|E| \approx |V|^2$

graph is sparse if $|E| \approx |V|$

Edge List Structure

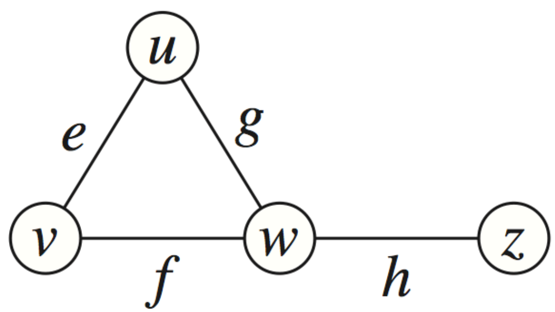
- Vertex objects stored in unsorted sequence
  - Space O(V)
- Edge objects stored in unsorted sequence
  - Space O(E)
- Edge objects has reference to origin and destination vertex object
- Total space: O(V+E)

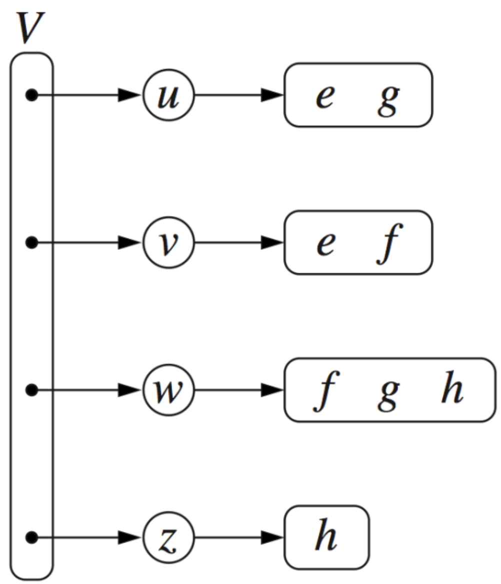




Adjacency List Structure

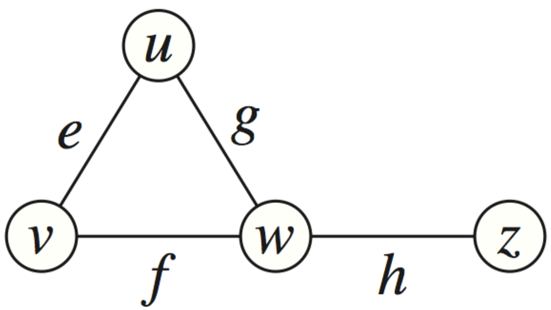
- Each vertex v_i lists the set of its neighbors
  - sequence of references to its adjacent vertices
- More space-efficient for a sparse graph: Total space O(V+E)

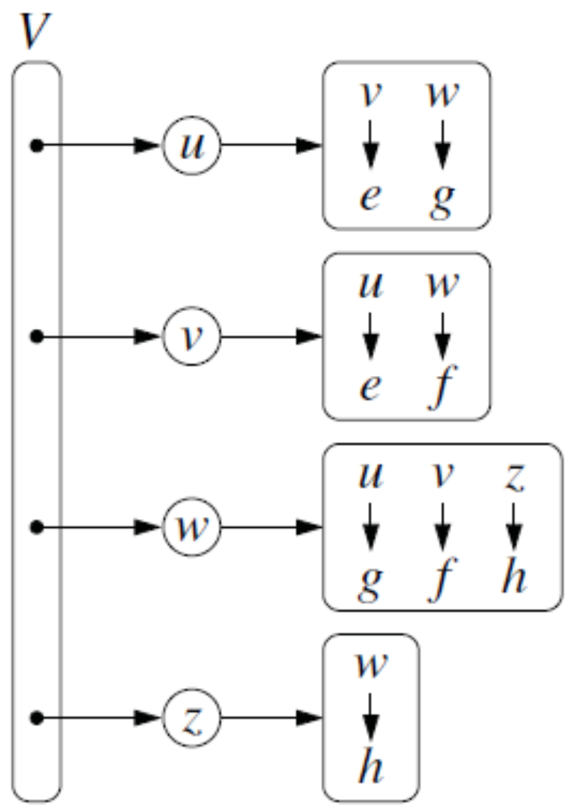




Adjacency Map Structure

- Replaces the neighbour list with a Map:
  - with the adjacent vertex serving as a key: vertex v_j
  - Its value: the edge (i,j)
- This allows more efficient access to a specific edge(*i,j*) in *O*(1) expected time
- Total space: O(V+E)





Graph ADT

```
public interface Graph <V,E> {
  int numVertices();
  Iterable<Vertex<V,E>> vertices();
  int numEdges();
  Iterable<Edge<V,E>> edges();
  Edge<V,E> getEdge(Vertex<V,E> vorig, Vertex<V,E> vdest);
  Vertex<V,E>[] endVertices(Edge<V,E> e);
  Vertex<V,E> opposite(Vertex<V,E> v, Edge<V,E> e);
  int outDegree(Vertex<V,E> v) ;
  int inDegree(Vertex<V,E> v) ;
  Iterable<Edge<V,E>> outgoingEdges (Vertex<V,E> v);
  Iterable<Edge<V,E>> incomingEdges(Vertex<V,E> v);
  Vertex<V,E> insertVertex(V vInf);
  Edge<V,E> insertEdge(V vorigInf, V vdestInf, E eInf, double eWeight);
  void removeVertex(V vInf);
  void removeEdge(Edge<V,E> e);
}
```

Asymptotic performance of graph data structures

	Edge List	Adjacency List	Adjacency Map	Adjacency Matrix
Space	V + E	V + E	V + E	V^2
numVertices(), numEdges()	O(1)	O(1)	O(1)	O(1)
vertices()	O(V)	O(V)	O(V)	O(V)
getEdge(*u*, *v*)	O(E)	$O(\min(d_u,d_v))$	O(1)	O(1)
outDegree(v) inDegree(v)	1	O(1) / O(V×E)	O(1) / O(V×E)	O(V)
outgoingEdges(v) incomingEdges(v)	O(E)	$O(d_v)$ / O(V×E)	$O(d_v)$ / O(V)	O(V)
insertVertex(*x*)	O(1)	O(1)	O(1)	$O(V^2)$
removeVertex(*v*)	O(E)	$O(d_v)$	$O(d_v)$	O(1)
insertEdge(*u*, *v*, *x*) removeEdge(x)	O(1)	O(1)	O(1)	O(1)

Graph Reachability

Reachability

A common question to ask about a graph is reachability:

- Single-source:
  - Which vertices are "reachable" from a given vertex v_i?

- All-pairs:
  - For all pairs of vertices v_i and v_j, is v_j "reachable" from v_i?
  - Solves the single source question for all vertices

All-Pairs Reachability: Adjacency Matrix

To compute all-pairs reachability, it is necessary:

- Start with the adjacency matrix of the graph
  - 1: indicates that there is an edge from v_i to v_j
  - 0: no edge from v_i to v_j

- Calculate the transitive closure of the graph with Floyd Warshall's algorithm
  - transitive closure is a matrix with the same vertices as the original graph and an arc between the pairs of vertices that have a path to join them

Floyd-Warshall algorithm - Basic idea

A path exists between two vertices i, j, iff

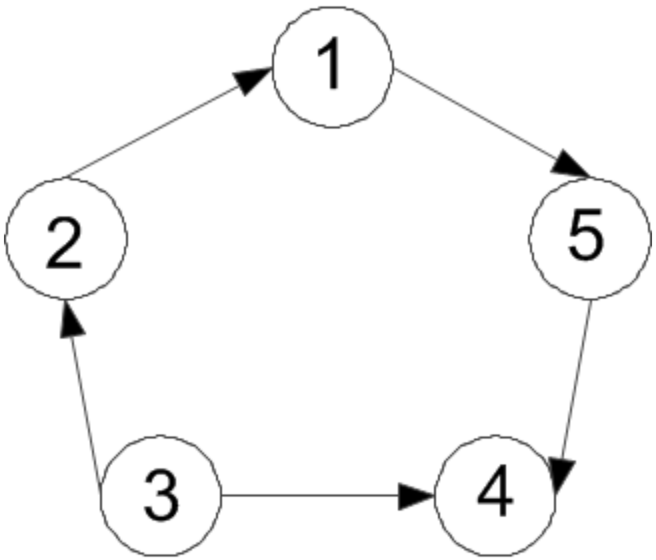
- there is an edge from i to j or
- there is a path from i to j going through vertex 1; or
- there is a path from i to j going through vertex 1 and/or 2; or
- there is a path from i to j going through vertex 1, 2, and/or 3; or
- ...
- there is a path from i to j going through any of the other vertices

On the k^{th} iteration, the algorithm determine if a path exists, between two vertices i, j using just vertices among 1,..., k allowed as intermediate

$$T_{i,j}^{(k)} = \begin{cases} T_{i,j} & if\ k = 0 \\ T_{i,j}^{(k-1)} \vee (T_{i,k}^{(k-1)} \wedge T_{k,j}^{(k-1)}) & if\ k \geq 1 \end{cases}$$

Floyd-Warshall algorithm: Transitive Closure

T^0 matrix is equal to the adjacency matrix – matrix with a path of length 1

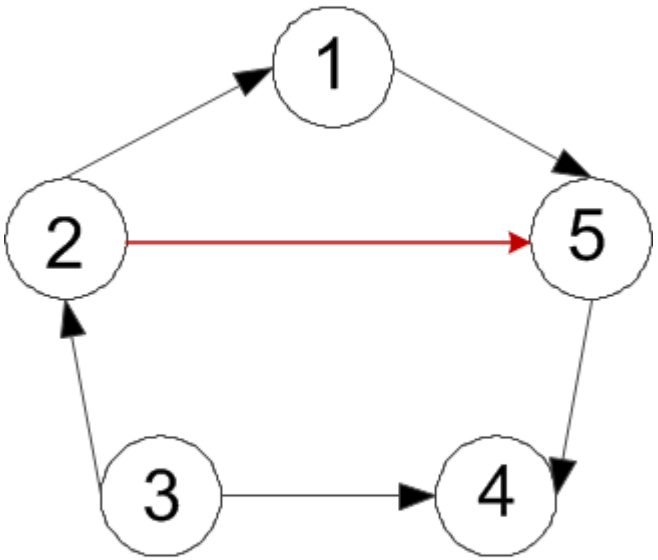


$$T^0 = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 0 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 \end{array}$$

$$T_{2,5}^{(1)} = T_{2,5}^{0} \vee (T_{2,1}^{0} \wedge T_{1,5}^{0}) = 0 \vee (1 \wedge 1) = 1$$

Floyd-Warshall algorithm: Transitive Closure

$$T^1 = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 0 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 0 & \mathbf{1} \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 1 & 0 \end{array}$$



Floyd-Warshall algorithm: Transitive Closure

T^1=

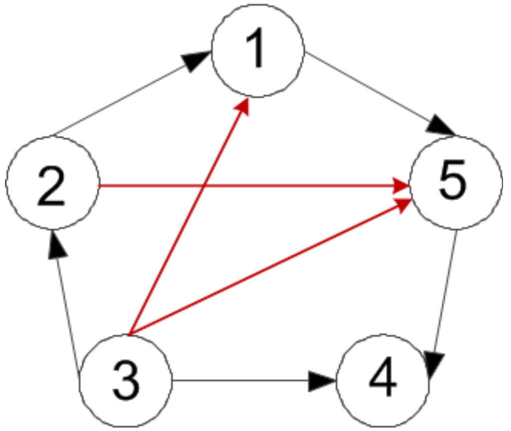
	1	2	3	4	5
1	0	0	0	0	1
2	1	0	0	0	**1**
3	0	1	0	1	0
4	0	0	0	0	0
5	0	0	0	1	0

T^2 =

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

$$T_{3,1}^{(2)} = T_{3,1}^{1} \vee (T_{3,2}^{1} \wedge T_{2,1}^{1}) = 0 \vee (1 \wedge 1) = 1$$

$$T_{3,5}^{(2)} = T_{3,5}^{1} \vee (T_{3,2}^{1} \wedge T_{2,5}^{1}) = 0 \vee (1 \wedge 1) = 1$$



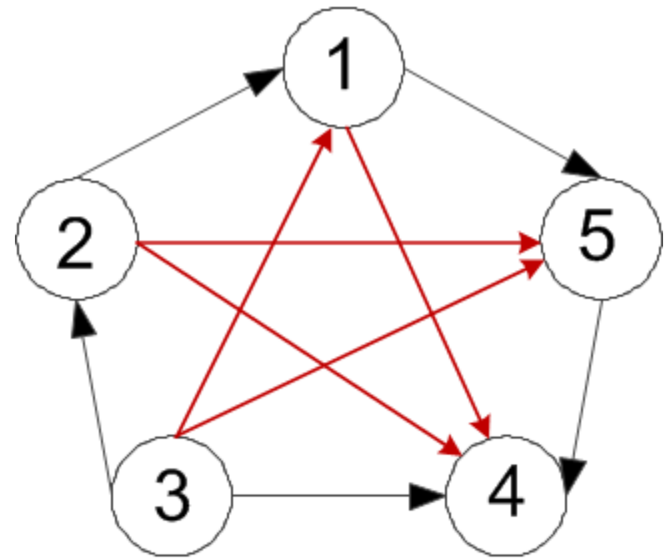
The addition of the vertex 3 and 4 doesn't add new paths, so $T^2 = T^3 = T^4$

Floyd-Warshall algorithm: Transitive Closure

The addition of vertex 5 allows to add edges (1,4) e (2,4)

T^5 =

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



The final matrix has a 1 in row i and column j, if vertex v_j is reachable from vertex v_i via some path

Floyd-Warshall algorithm

```
Algorithm void transitiveClosure (Graph<V,E> g) {
  for (k ← 0; k < n; k++)
     for (i ← 0; i < n; i++) {
          if (i != k && T[i,k] = 1)
            for (j ← 0; j < n; j++)
               if (i != j && k != j && T[k,j] = 1 )
                  T[i,j] = 1
      }
}
```

Time Complexity: O(?)

All-Pairs Reachability: Weighted Graph

- Key difference: adjacency graph now has weights instead of binary values
- In place of logical operations (AND, OR) use arithmetic operations (addition)

$$D_{i,j}^{(k)} = \begin{cases} w_{i,j} & if\ k = 0 \\ \min\left(D_{i,j}^{(k-1)}, D_{i,k}^{(k-1)} + D_{k,j}^{(k-1)}\right) & if\ k \geq 1 \end{cases}$$

The final matrix gives, in row i and column j, the length of the minimum path between vertices i, j, if vertex v_j is reachable from vertex v_i via some path

Graph Traversals

- For solving most problems on graphs
  - We need to systematically visit all the vertices and edges of a graph

- There are two standard graph traversal techniques that provide an efficient way to “visit” each vertex and edge exactly once:
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

- Graph Traversals (BFS, DFS):
  - Starts at some source vertex *S*
  - Discover every vertex that is reachable from *S*

Breadth-First Search – Basic Idea

1. Choose a starting vertex, its level is called the current level

2. From each node N in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbours of N. The newly visited nodes from this level form a new level that becomes the next current level

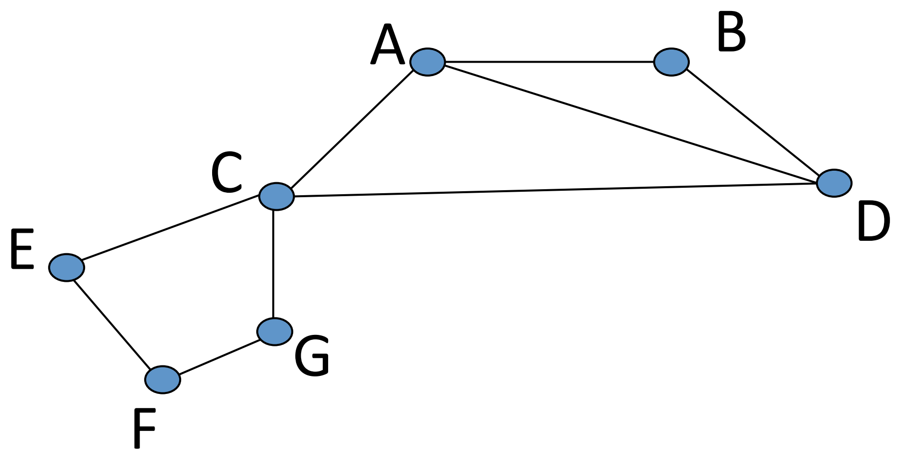
3. Repeat step 2 until no more nodes can be visited

4. If there are still unvisited nodes, repeat from Step 1

BFS → For each vertex visit all its edges (neighbours)

Breadth-First Search – Example

BFS starting at vertex D



Adjacency List:

A → B, C, D

B → A, D

C → A, D, E, G

D → A, B, C

E → C, F

F → E, G

G → C, F

> BFS: D, A, B, C, E, G, F

Breadth-First Search - Algorithm

```
Algorithm LinkedList<V> BFS(Graph<V,E> G, V vOrig){

 Add vOrig-element to qbfs
 Add vOrig to qaux
 Make vOrig as visited
 while (!qaux is Empty){
   vOrig ← Remove first vertex from qaux
   for (each vAdj of vOrig){
      if (vAdj has not been visited){
         Add vAdj to qbfs;
         Add vAdj to qaux;
         Make vAdj as visited;
      }
   }
 }
 return qbfs;
}
```

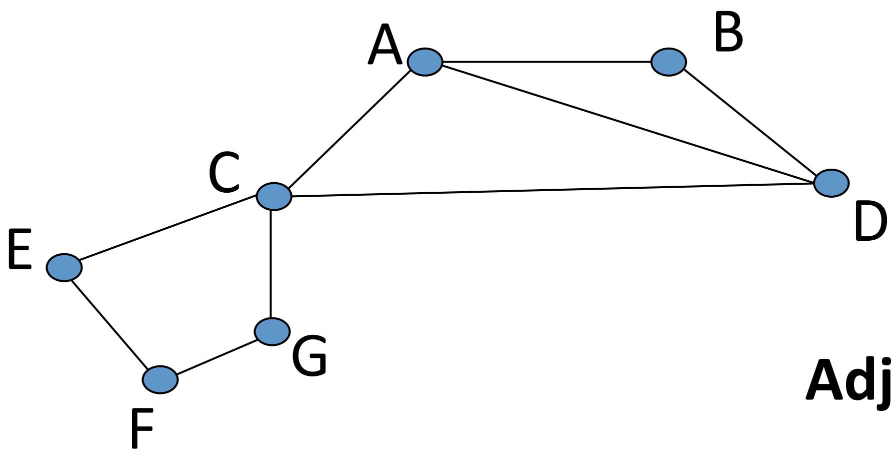
Time Complexity: O(?)

Depth-First Search - Basic Idea

1. choose a starting vertex, distance d = 0
2. Examine One edge leading from vertex (at distance d) to adjacent vertices (at distance d+1)
3. Then, examine One edge leading from vertices at distance d+1 to distance d+2, and so on,
4. until no new vertex is discovered, or dead end
5. Then, backtrack one distance back up, and try other edges, and so on
6. Until finally backtrack to starting vertex, with no more new vertex to be discovered

Depth-First Search – Example

DFS starting at vertex G



Adjacency List:

A → B, C, D
B → A, D
C → A, D, E, G
D → A, B, C
E → C, F
F → E, G
G → C, F

> DFS: G, C, A, B, D, E, F

Depth-First Search

- It searches ‘deeper’ the graph when possible
- Starts at the selected node and explores as far as possible along each branch before backtracking
- The Depth-First Search can go far way from the right path by exploring a branch that is never close to the goal

Depth-First Search - Algorithm

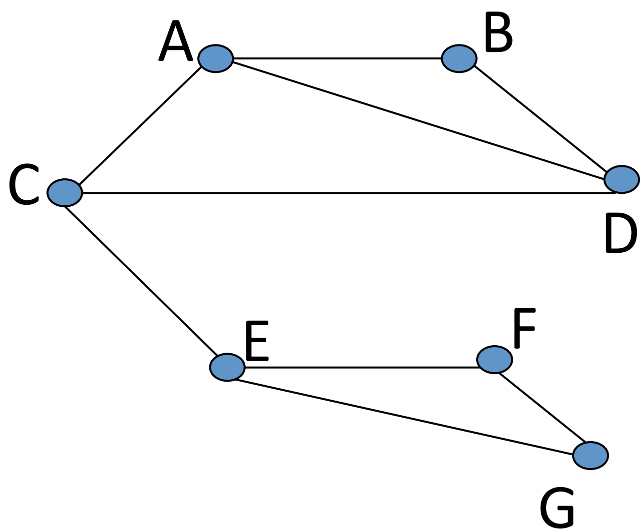
```
Algorithm void DFS(Graph<V,E> G, V vOrig, LinkedList<V> qdfs){

    Push vOrig-element to qdfs
    Make vOrig as visited
    for (each vAdj of vOrig) {
       if (vAdj has not been visited)
          Recursively call DFS(G, vAdj, qdfs);
    }
 }
```

Time Complexity: O(?)

Exercise

Present the BFS starting at vertex B
Present the DFS starting at vertex G



Adjacency List:

A → B, C, D
B → A, D
C → A, D, E
D → A, B, C
E → C, F, G
F → E, G
G → E, F