Estruturas de Informação

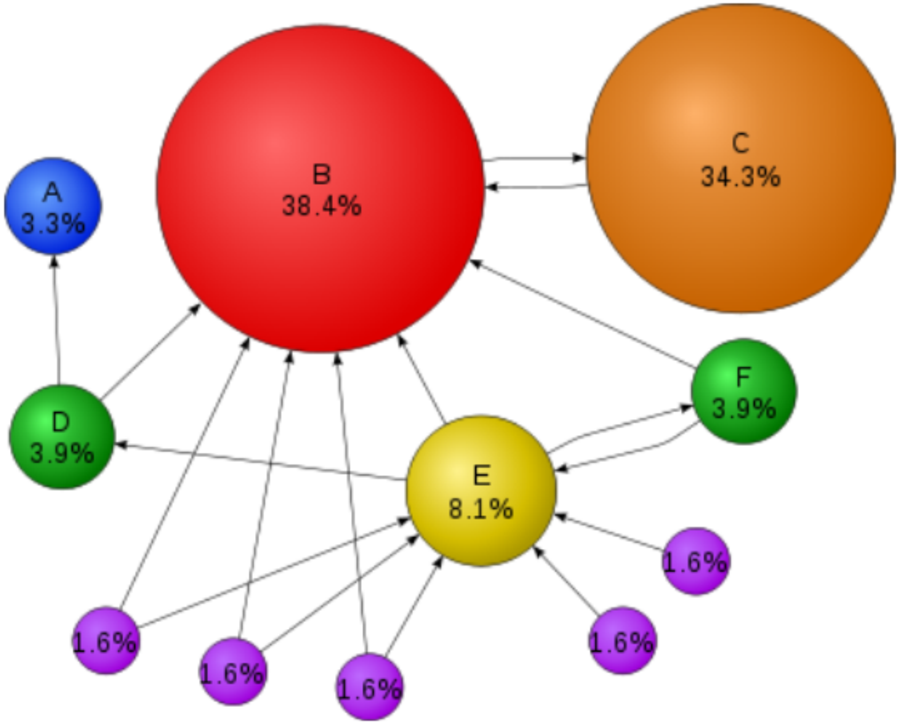
Graphs

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 edges, called Edges

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

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

Informal definition

Graphs represent general relationships or connections

– Each node may have many predecessors and many successors

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

– Can have cycles or loops

Graphs: Vertices and Edges

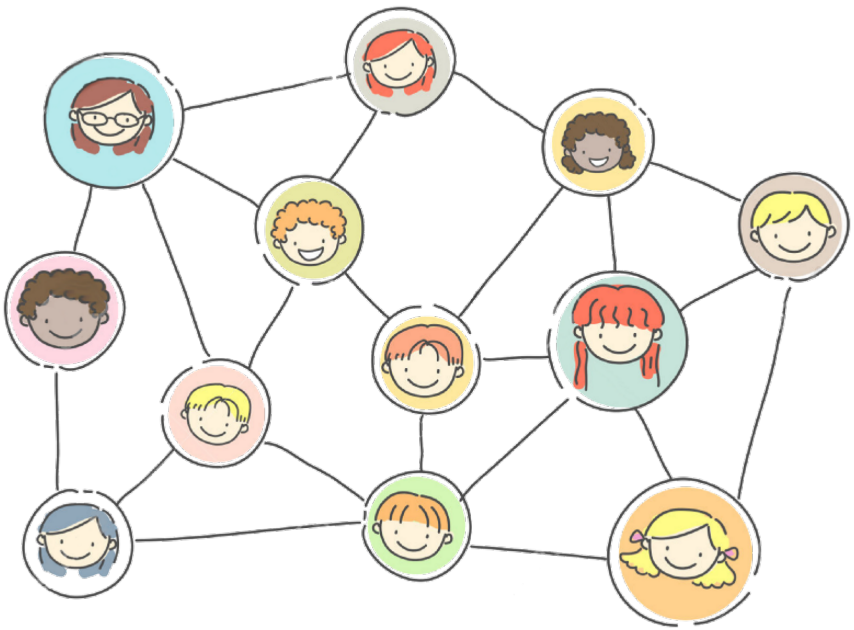
A graph is composed of vertices and edges

- 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

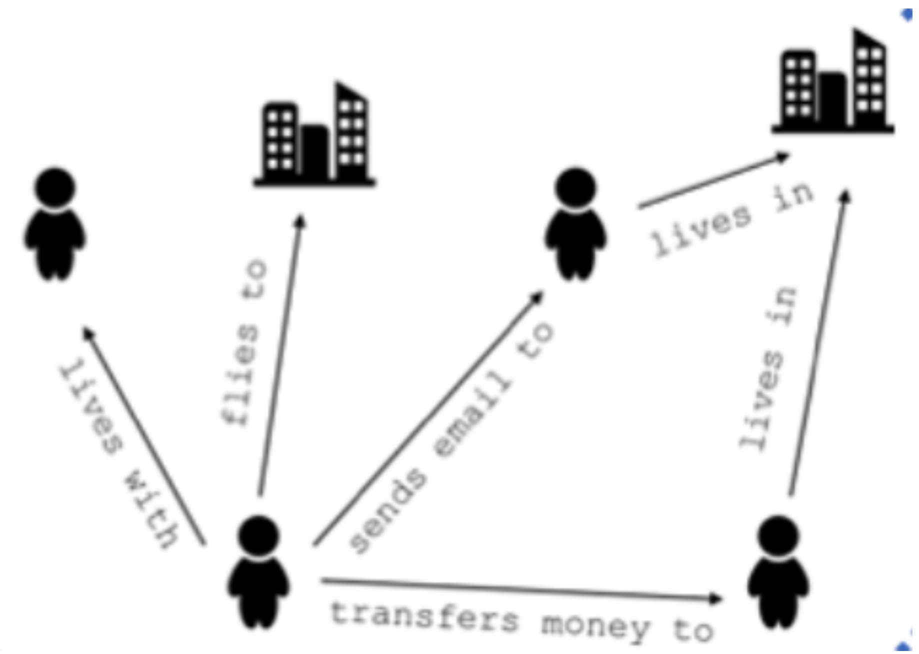
- Edges (arcs):
  - Can be directed or undirected
  - Can be weighted (or labeled) or unweighted

Graphs: Homogeneous and Heterogeneous

- A graph with a single type of node and a single type of edge is called **homogeneous**
- a graph with two or more types of node and/or two or more types of edge is called **heterogeneous**



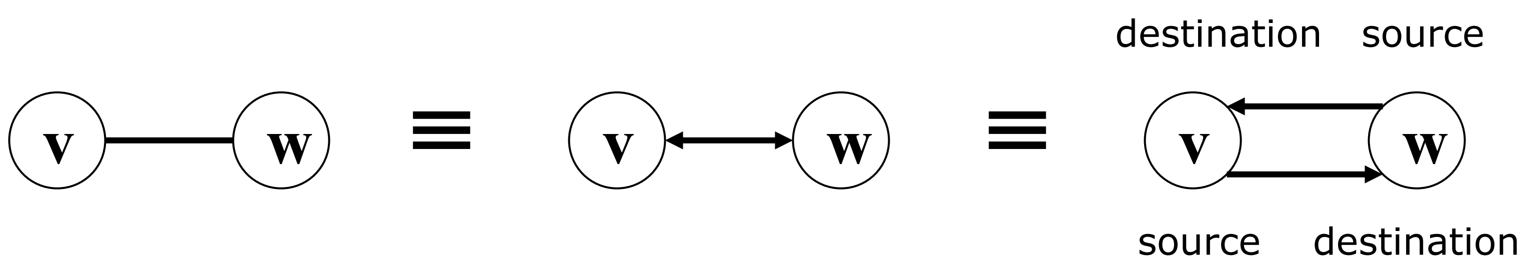
Social network of people friendship connections



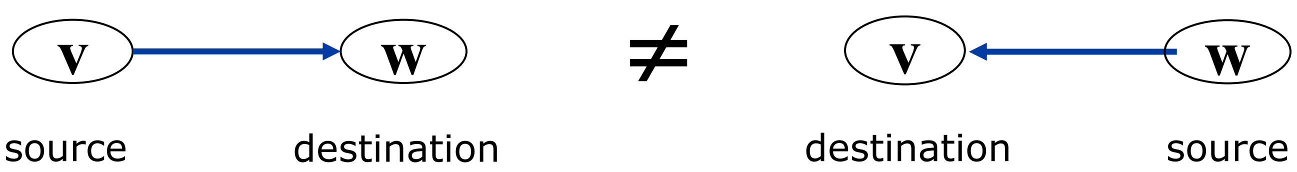
Social network of people and cities, connected by four different types of edges

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

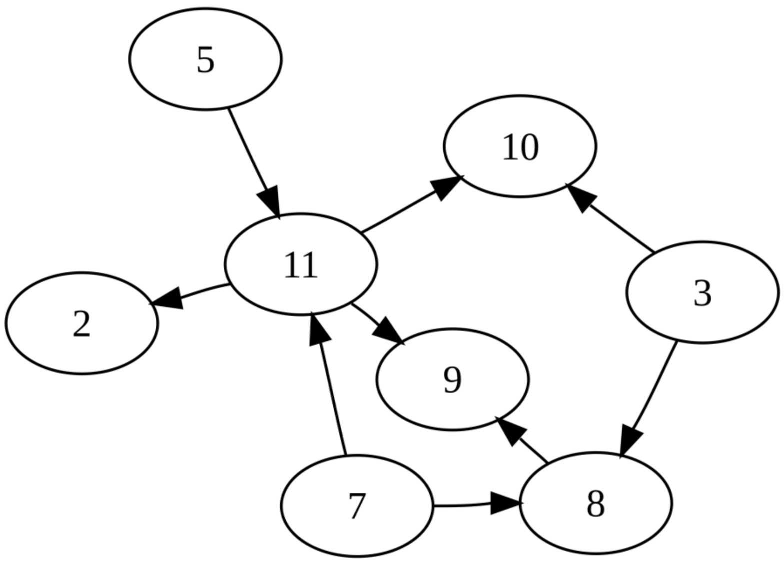


- A directed edge $e = (v_i, v_j)$ specifies a one-directional relationship or connection:
  - Can only go from v_i to v_j

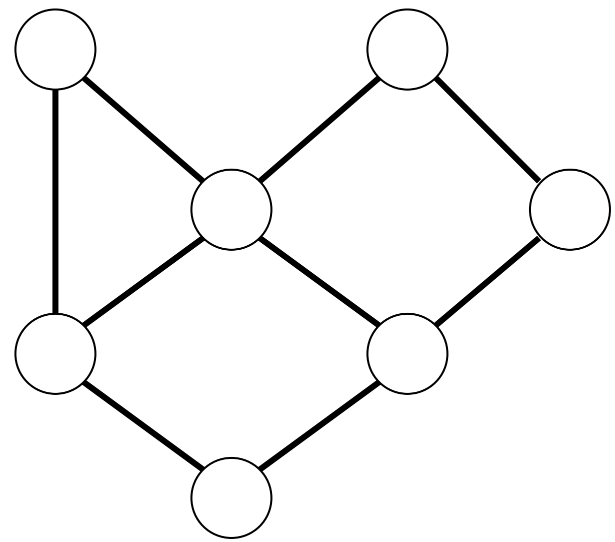


Graphs: Directed and Undirected

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



Ex. route network



Ex. friends network

Graphs: Density

The **density of a graph** is a measure of how many edges between nodes exist compared to how many edges between nodes are possible

Density of an **undirected graph**:

$$\frac{total\ edges}{total\ possible\ edges} = \frac{\mathrm{m}}{n \times (n-1)/2}$$

Density of a **directed graph** there is no need to divide the numerator by two

Friends Network:

$$\frac{9}{(7 \times 6)/2} = 0.428$$

Route network:

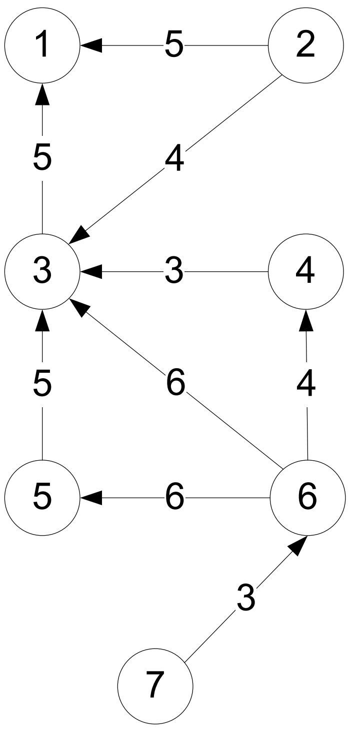
$$\frac{9}{11 \times 10} = 0.08$$

Valorised graph

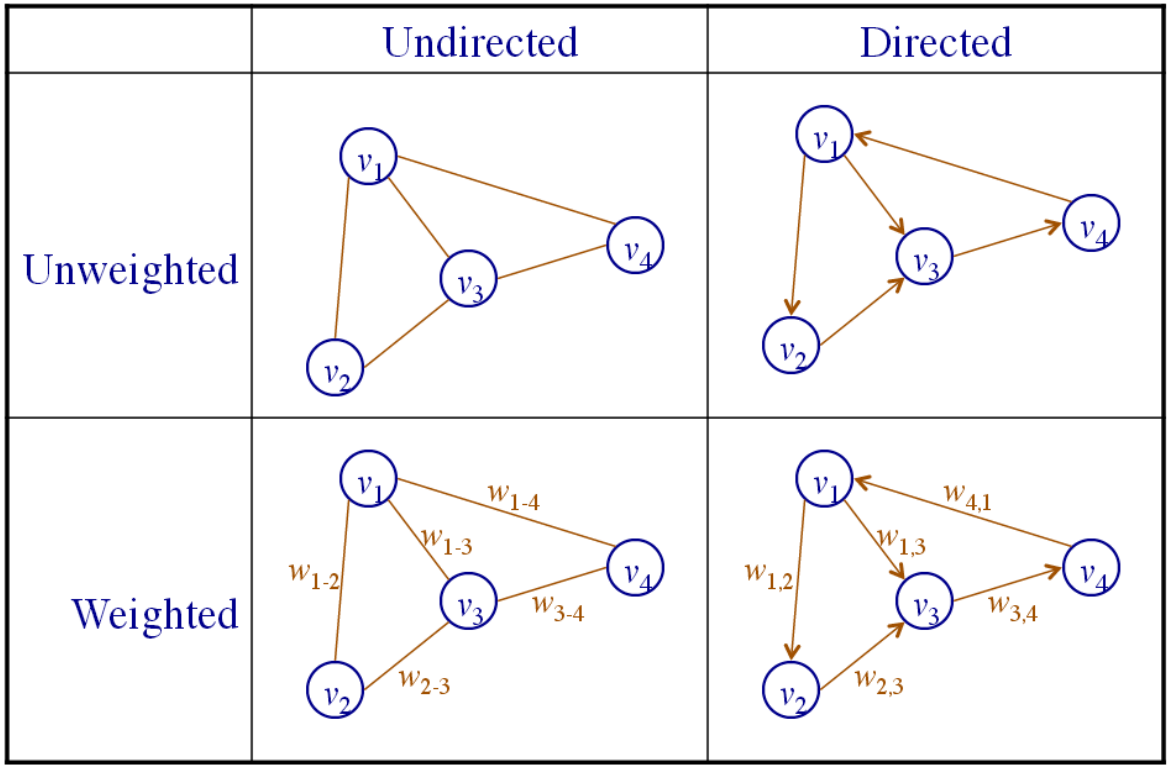
Graph that all its branches have an associated value

These values can represent:

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



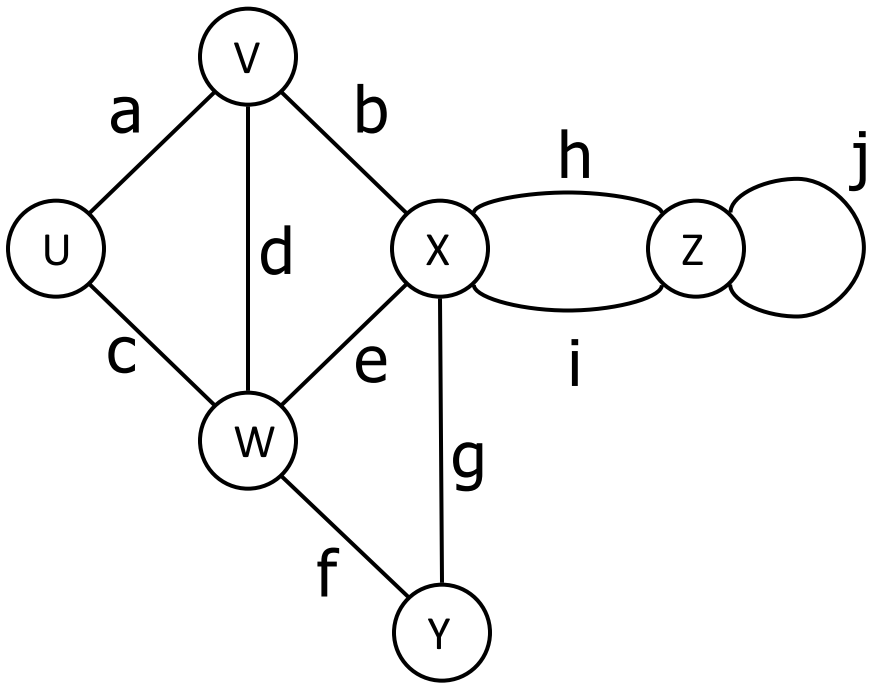
Graphs: Types of Edges



- Unweighted, undirected: social network
- Unweighted, directed: twitter network
- Weighted, undirected: highways network
- Weighted, directed: road network

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



Centrality

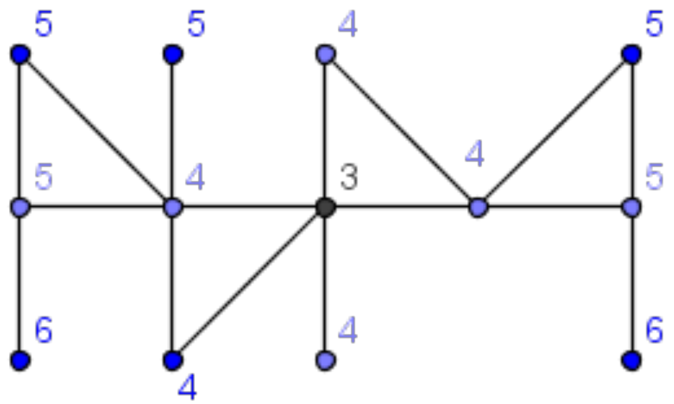
Centrality measures address the question:

"Who is the most important or central person in this network?"

- It depends on what we mean by **importance**
- So, there are a vast number of different centrality measures that have been proposed over the years
  - **Node degree**
  - **Closeness**
  - **Betweenness**
  - **…**
  - **PageRank centrality**

Graph Diameter

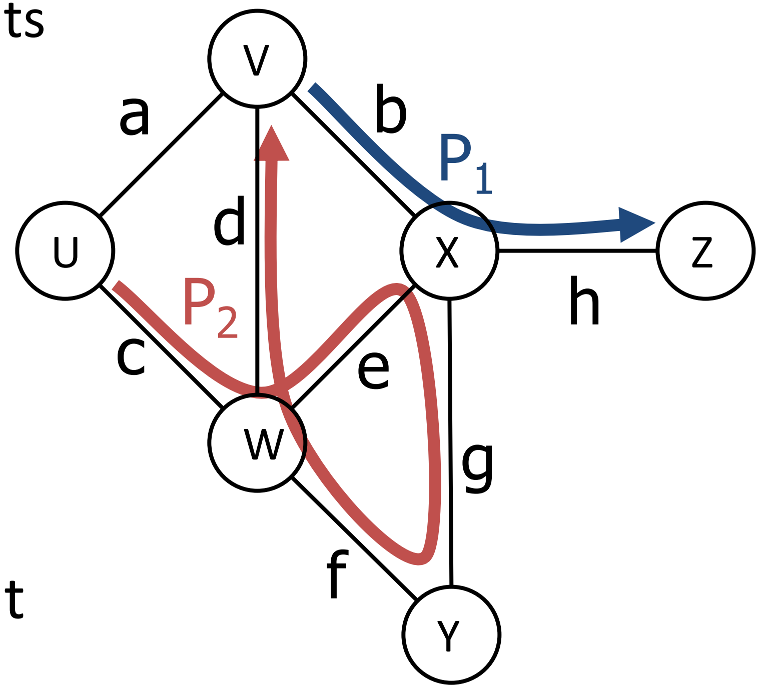
- The diameter of G diam(G) is the longest shortest path between any two nodes in a network



- The diameter indicates how long it will take at most to reach any node in a **connected network**
- Or if unconnected, of the largest component

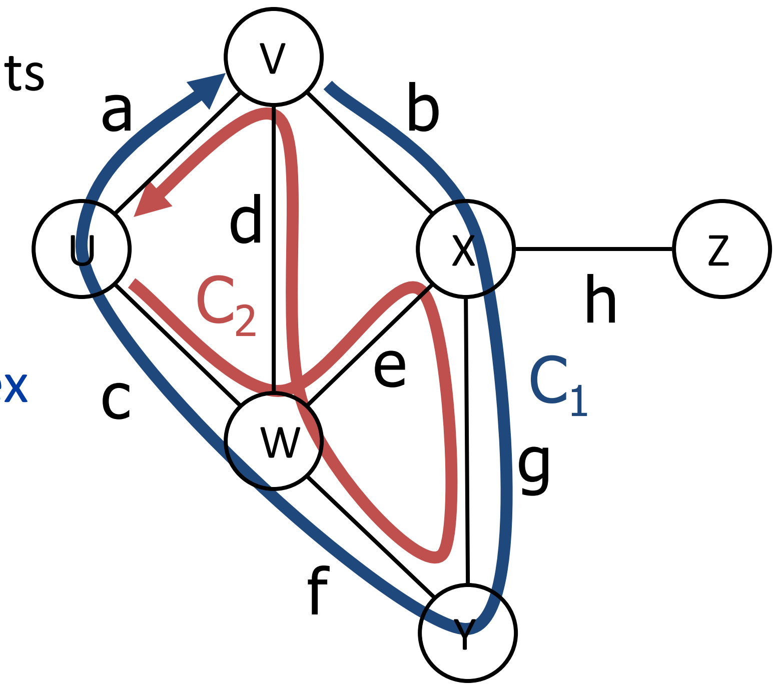
Graph Terminology

- **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



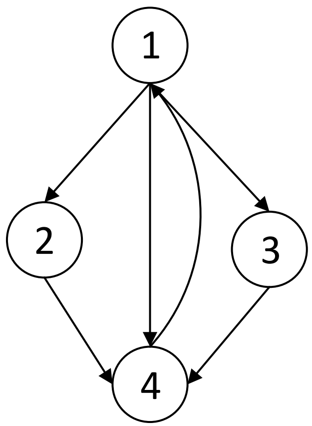
Graph Terminology

- **Cycle**
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- **Simple cycle**
  - cycle such that all its edges and vertices are distinct, except the initial/final vertex
- 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

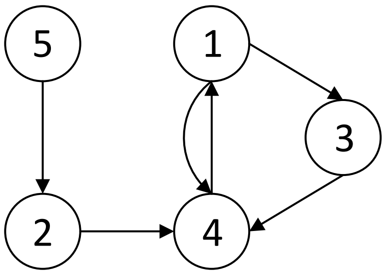


Graph Terminology

- A direct Graph is strongly connected if there is a path between all pair of vertices



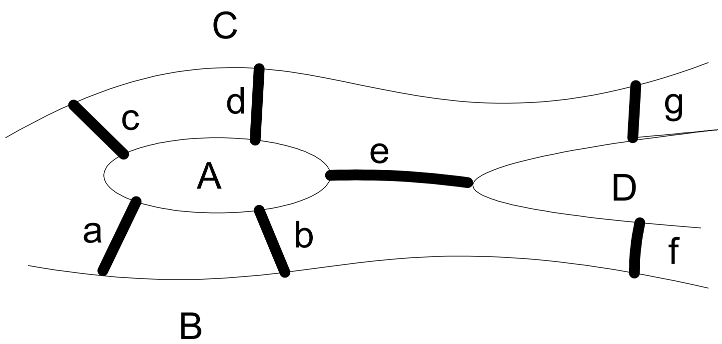
Strong conex graph



Not conex Graph

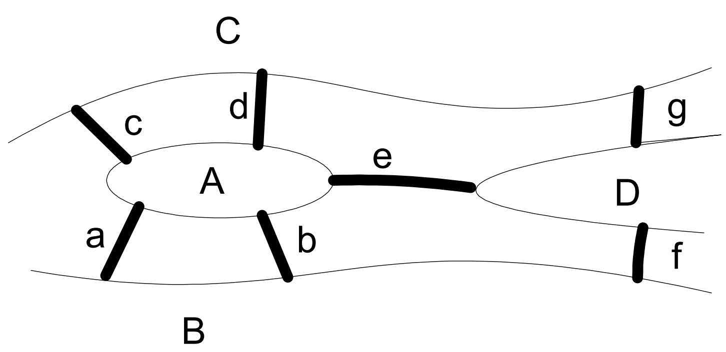
- A Strongly Connected Component (SCC) of a directed graph is a maximal strongly connected subgraph

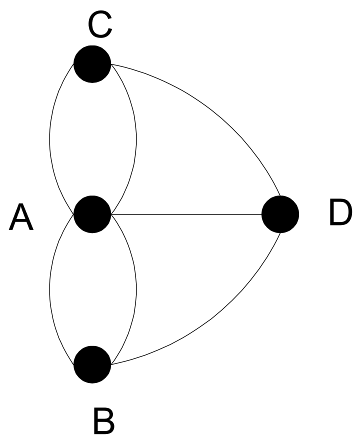
Euler Cycle and the 7 Bridges of Koenigsberg



- 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





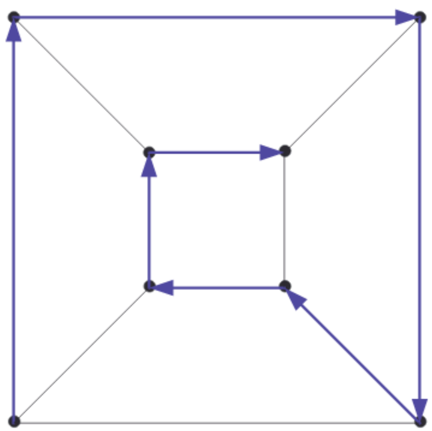
This problem was solved by Euler in 1736 and marks the beginning of Graph Theory

Euler proved

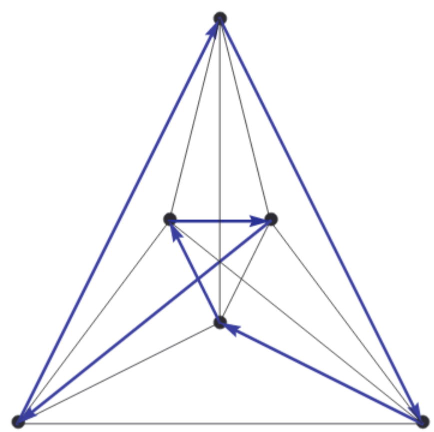
- An undirected and connected graph has an Euler Cycle iff all the vertices have an even degree
- A directed and strongly connected graph has an Euler Cycle iff $d_{in}(V) = d_{out}(V)$ for each vertex V

Hamilton Path/Cycle

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



Hamilton path



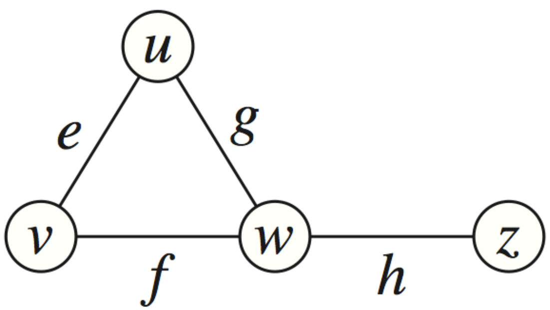
Hamilton cycle

- 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 algorithm 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: $O(V^2)$



		0	1	2	3
$u \rightarrow$	0		e	g	
$v \rightarrow$	1	e		f	
$w \rightarrow$	2	g	f		h
$z \rightarrow$	3			h	

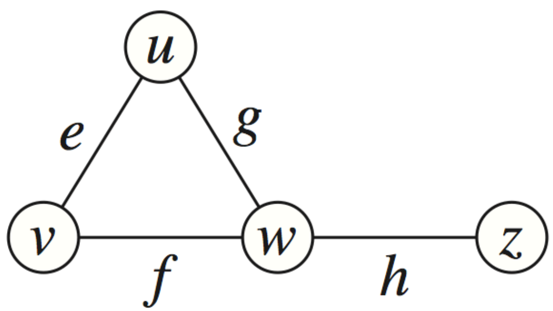
Therefore adjacency matrix should be used only for dense graphs

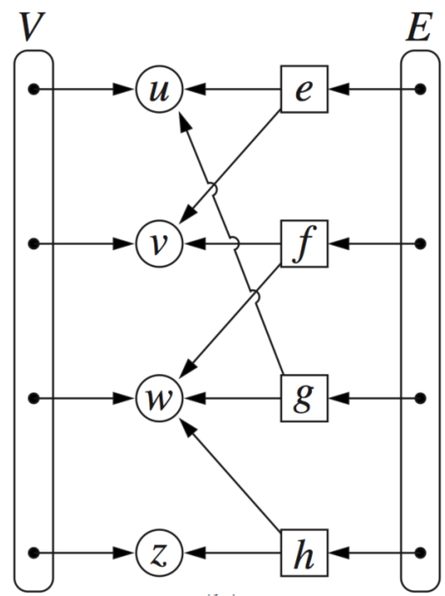
graph is dense if $|E| \approx |V|^2$

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

Edge List Structure

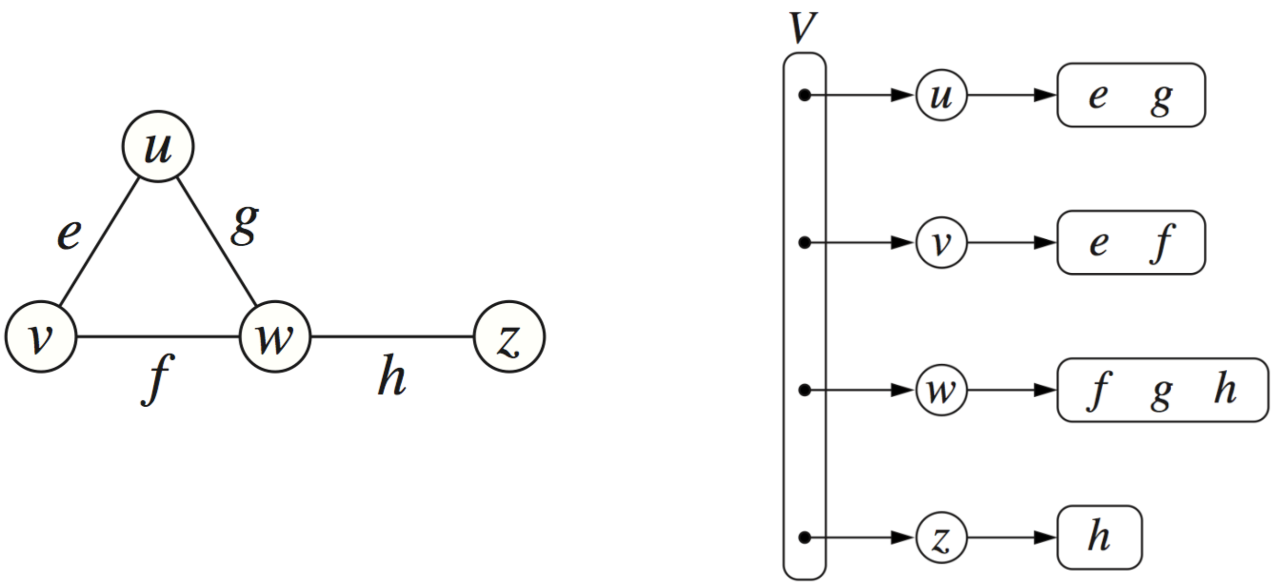
- Vertex objects stored in unsorted sequence
  - Space O(V)
- Edge objects stored in unsorted sequence
  - Space O(E)
- Edge objects has reference to origin and destination vertex object
- Total space: O(V+E)





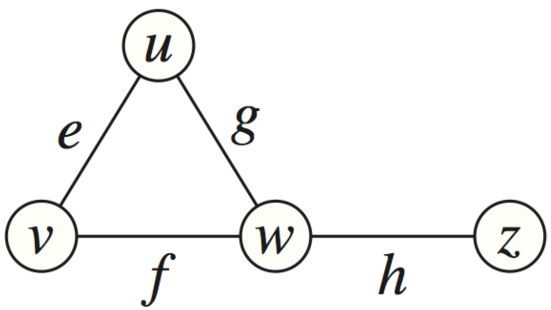
Adjacency List Structure

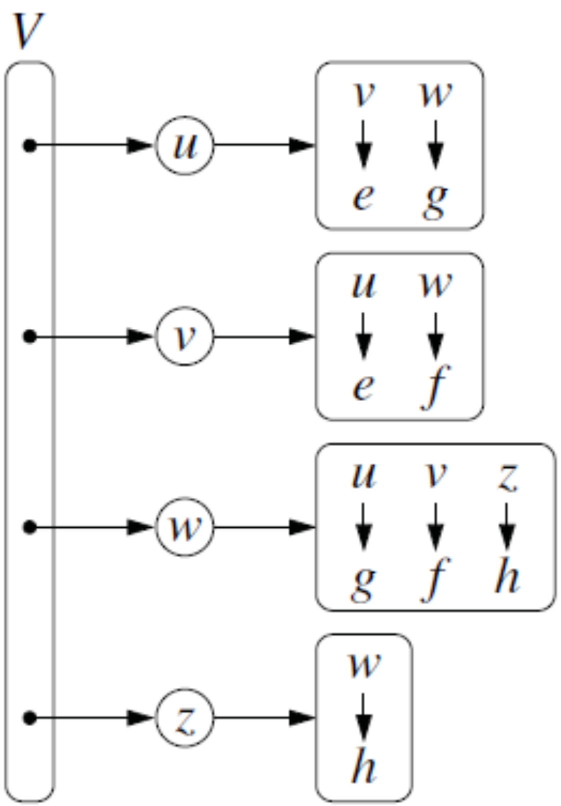
- Each vertex v_i lists the set of its neighbors
  - sequence of references to its adjacent vertices
- More space-efficient for a sparse graph: Total space O(V+E)



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 Interface

```
public interface Graph<V, E> extends Cloneable {

    boolean isDirected();
    int numVertices();
    ArrayList<V> vertices();
    boolean validVertex(V vert);
    int key(V vert);
    V vertex(int key);
    V vertex(Predicate<V> p);
    Collection<V> adjVertices(V vert);
    int numEdges();
    Collection<Edge<V, E>> edges();
    Edge<V, E> edge(V vOrig, V vDest);
    Edge<V, E> edge(int vOrigKey, int vDestKey);
    int outDegree(V vert);
    int inDegree(V vert);
```

Graph Interface

```
    Collection<Edge<V, E>> outgoingEdges(V vert);
    Collection<Edge<V, E>> incomingEdges(V vert);
    boolean addVertex(V vert);
    boolean addEdge(V vOrig, V vDest, E weight);
    boolean removeVertex(V vert);
    boolean removeEdge(V vOrig, V vDest);
    Graph<V, E> clone();
}
```

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(VxE)	O(V)
outgoingEdges(v) incomingEdges(v)	O(E)	$O(d_v)$ / O(V×E)	$O(d_v)$ / O(VxE)	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

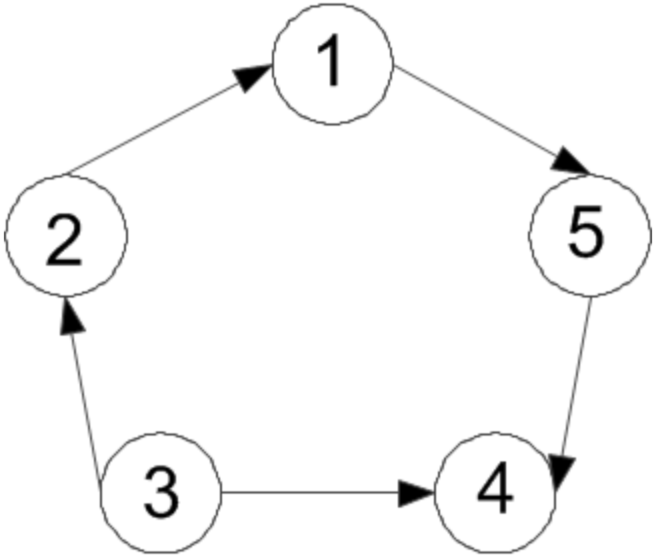
- there is an edge from i to j or
- there is a path from i to j going through vertex 1; or
- there is a path from i to j going through vertex 1 and/or 2; or
- there is a path from i to j going through vertex 1, 2, and/or 3; or
- ...
- there is a path from i to j going through any of the other vertices

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

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

Transitive Closure

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

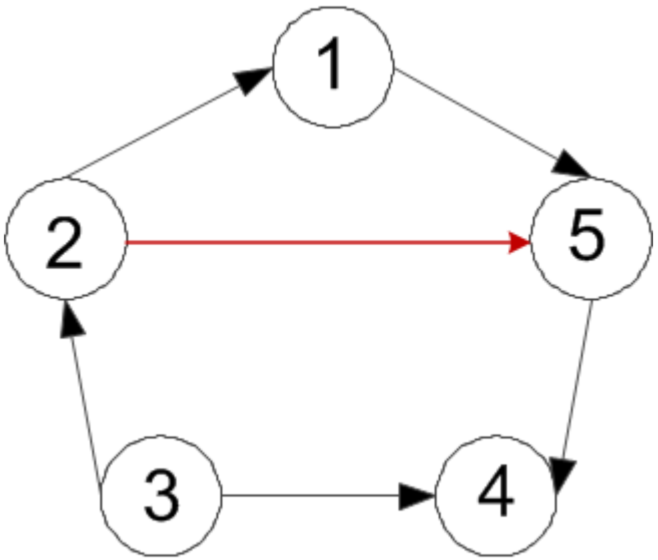


T^0	1	2	3	4	5
1	0	0	0	0	1
2	1	0	0	0	0
3	0	1	0	1	0
4	0	0	0	0	0
5	0	0	0	1	0

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

Floyd-Warshall algorithm: Transitive Closure

T^1	**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



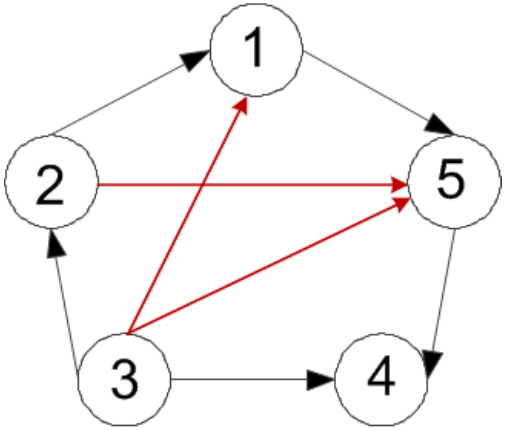
Floyd-Warshall algorithm: Transitive Closure

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

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

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

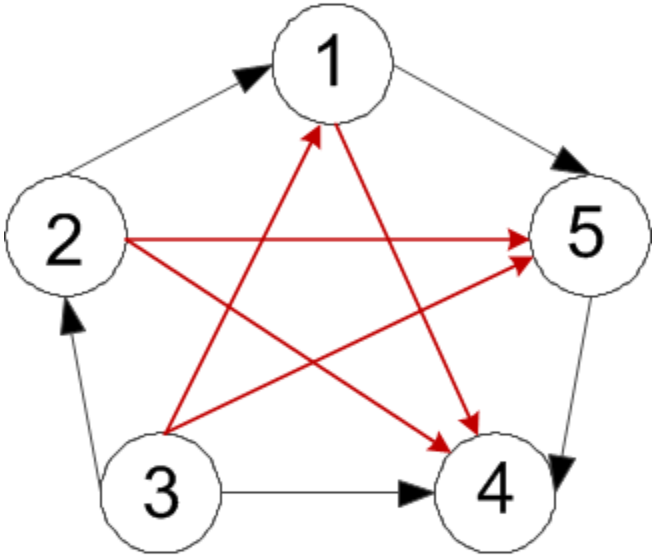


The addition of the vertex 3 and 4 doesn't add new paths, so $T^2 = T^3 = T^4$

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 minimum distances: Weighted Graph

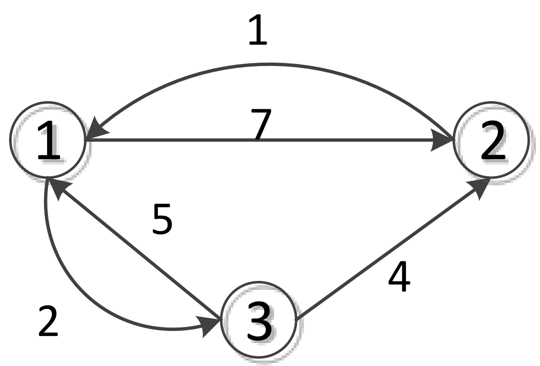
- Key difference: adjacency graph now has weights instead of binary values
- In place of logical operations (AND, OR) use arithmetic operations (addition)

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

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

Exercise

Show how the Floyd-Warshall algorithm determines the length of the shortest path between all pairs of vertices in the following graph



	1	2	3
1	0	7	2
2	1	0	∞
3	5	4	0

Vertex 1

	1	2	3
1	0	7	2
2	1	0	3
3	5	4	0

Vertex 2

	1	2	3
1	0	7	2
2	1	0	3
3	5	4	0

Vertex 3

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

Shortest path between any two nodes - Floyd Warshall

- It is necessary to use a matrix (2D array) to keep track of the next node to point if the shortest path changes for any pair of nodes

- Initially, the shortest path between any two nodes i and j is j (if exists a direct edge from i -> j)
  - *Next[i][j] = j*
- *If i and j can be connected through an intermediate node k:*
  - *Next[i][j] = Next[i][k]*
    (that means we found the shortest path between i, j through an intermediate node k)

Graph Traversals
Breadth-First / Depth-First

Graph Traversals

- For solving most problems on graphs
  - We need to systematically visit all the vertices and edges of a graph in an efficient way: without explore anything twice

- 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 vs. Depth-First

- Breadth-First
  - Queue (iterative method)
  - It produces a ‘breadth first tree’ with root s that contains all the vertices reachable from s
  - For any vertex v reachable from s, the path in the breadth first tree corresponds to the shortest path in graph G from s to v, if all edges have length 1, or the same length

- Depth-First
  - Stack (recursive)
  - Starts at the selected node and explores as far as possible along each branch before backtracking

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

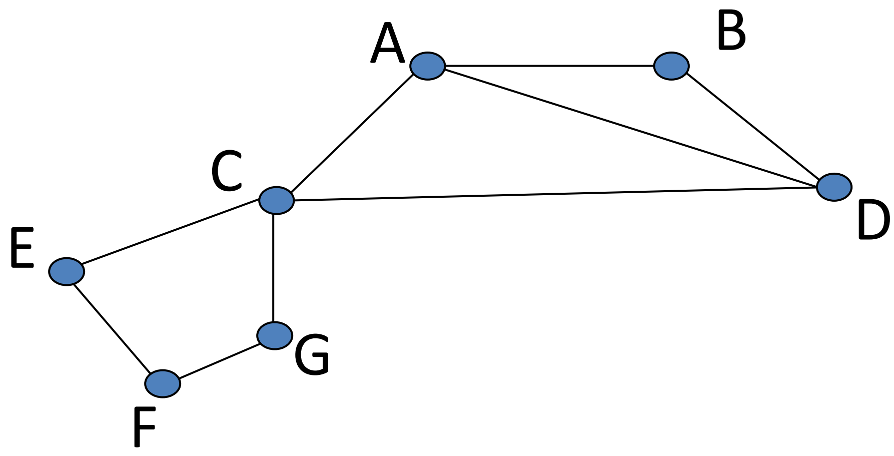
3. Repeat step 2 until no more nodes can be visited

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 to qbfs
 Add vOrig to qaux
 visited[vOrigKey] = true

 while (!qaux is Empty){
   vOrig ← Remove first vertex from qaux
   for (each vAdj of vOrig)
      if (vAdj is not visited){
         Add vAdj to qbfs
         Add vAdj to qaux
         visited[vAdjKey] = true
      }
   }
 }
 return qbfs
}
```

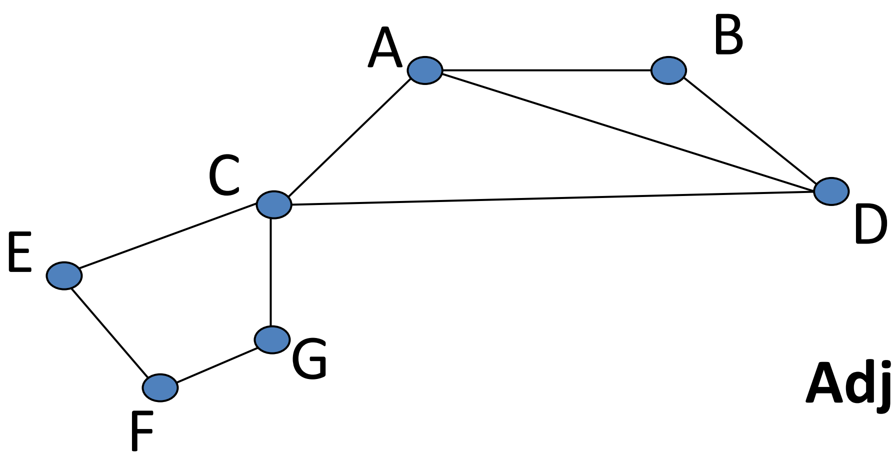
Time Complexity: O(?)

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 adjacent vertices, 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 - Algorithm

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

   if (visited[vOrigKey])
       return

    Push vOrig-element to qdfs
    visited[vOrigKey]=true

    for (each vAdj of vOrig) {
          Recursively call DFS(G, vAdj, visited, qdfs)
    }
 }
```

Time Complexity: O(?)

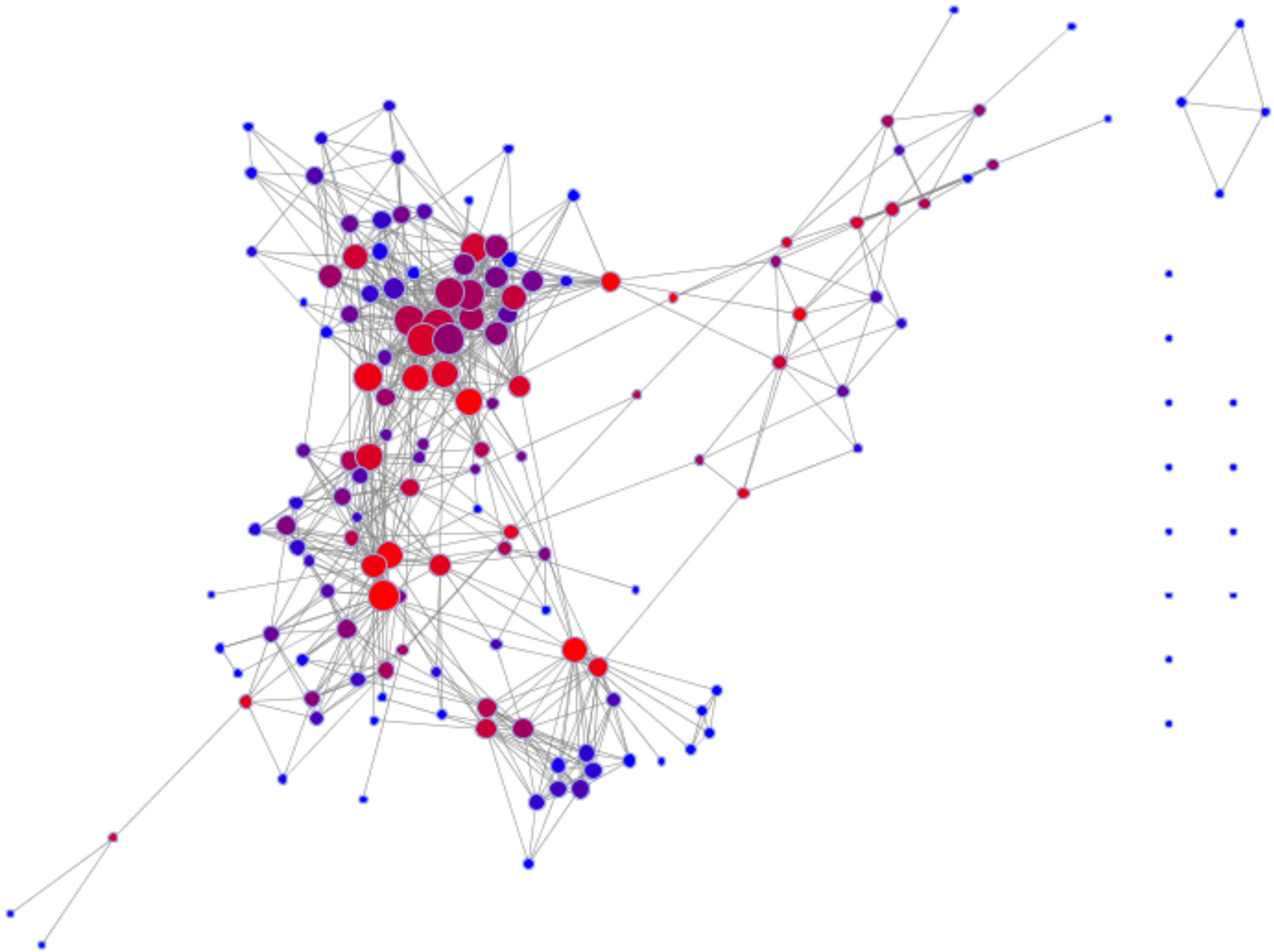
Breadth-First / Depth-First

- Breadth-First
  - Explores nodes in “layers” using a queue (FIFO)
  - Can compute shortest paths
  - Can compute connected components of an undirected graph

- Depth-First
  - Explores nodes ”deeper”, backtracks when necessary, uses a stack
  - Can compute topological ordering of a directed acyclic graph
  - Can compute strongly connected components of a directed graph

Graphs
Connected Components

Connected Components



Connected Components via BFS

- Let G = (V, E) be an **undirected graph**, the **connected components** are the “pieces” of G
- **Formal definition**: equivalence classes of the relation u <-> v, there exists a u-v path in G

Goal: compute all connected components

Why:

- check if the network is disconnected
- Graph visualization
- clustering

Connected Components - Algorithm

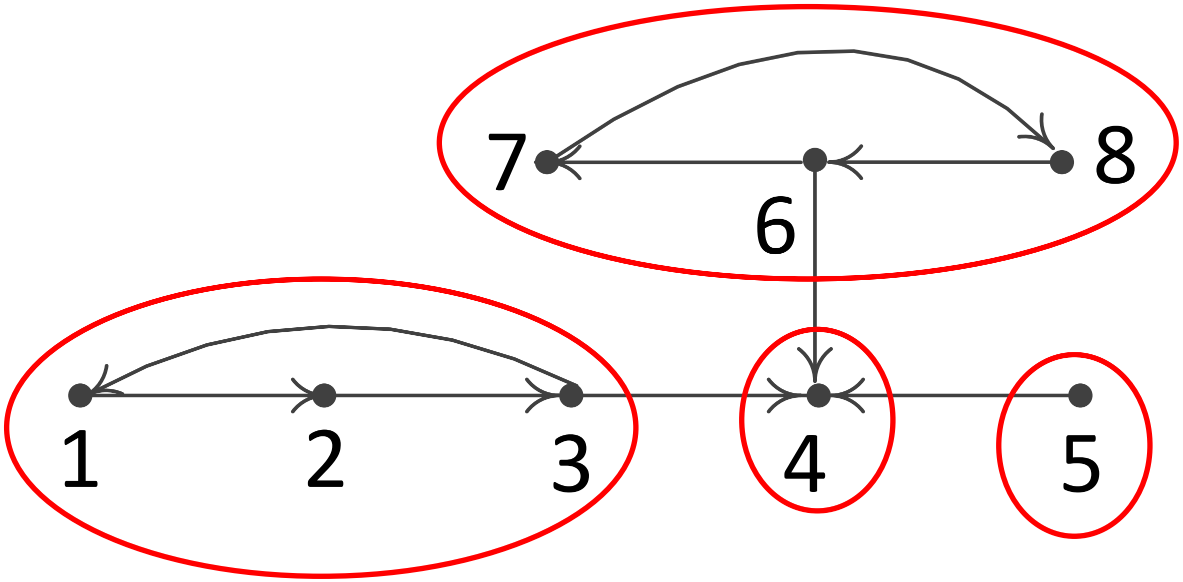
```
Algorithm void connectComps (Graph<V,E> G,
                                    ArrayList<LinkedList<V>> ccs){

    initialize all vertices as unvisited

    for (all Vertex V in G){
         if (V has not been visited)            //in some previous BFS
            conComp = BFS(G, V)                 //discovers precisely V's
            add conComp ccs                     //connected component
            for (all Vertex V in conComp)
              make them visited
    }
}
```

Time Complexity: O(?)

Strongly Connected Components

Strongly Connected Components

A **strongly connected component (SCC)** of a **directed graph** G = (V, E) is the **largest subset of vertices** C ⊆ V such that for every pair of vertices v, w ⊆ C there is a path from v to w and from w to v



Kosaraju's Algorithm[1]

[1]Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

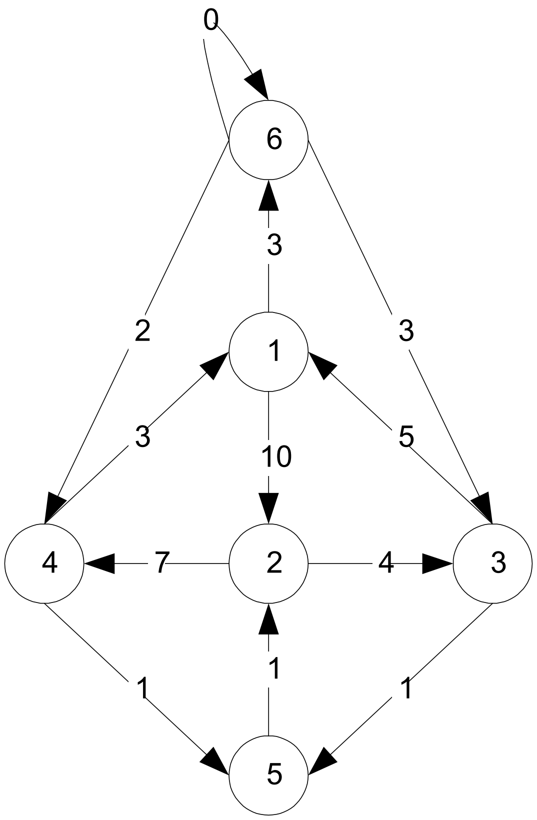
Graphs
Paths

Paths and Circuits

- Path in a graph is an alternating sequence of adjacent vertices and edges, that does not contain a repeated edge
- Simple path is a path that does not contain a repeated vertex

- Circuit is a closed path that does not contain a repeated edge
  – Simple circuit is a circuit which does not have a repeated vertex except for the first and last
  – Euler circuit is a circuit that contains every vertex and every edge of a graph. Every edge is traversed exactly once
  – Hamiltonian circuit is a simple circuit that contains all vertices of the graph (and each exactly once)

All Simple paths between two vertices



Adjacency List

1 → 2 , 6
2 → 3 , 4
3 → 1 , 5
4 → 5
5 → 2
6 → 3, 4, 6

All paths between Vertices 1 - 5:

- 1, 2, 3, 5
- 1, 2, 4, 5
- 1, 6, 3, 5
- 1, 6, 4, 5

All paths between two vertices - Algorithm

```
Algorithm void allPaths (Graph<V,E> G, V vOrig, V vDst,
                                    boolean[] visited,
                                     LinkedList<V> path,
                                    ArrayList<LinkedList<V>> paths){
  push vOrig onto path
  visited[key_vOrig]=true
  for (each vAdj of vOrig){
     if (vAdj == Vdst){
        push vDst onto path
        add path to paths
        pop last Vertex from path }
     else
        if (!visited[key_vAdj])
           recursively call allPaths(G,vAdj,vDst,visited,path,paths)
  }
  pop last Vertex from path
}
```

Time Complexity: O(?)

Graphs
Shortest Paths

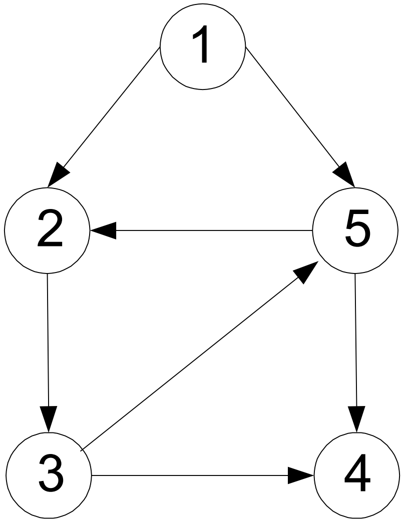
Shortest path problem

Given a directed graph G(V,E) find the shortest path from a given start vertex *S* to all other vertices (*Single Source Shortest Paths*) where the length of a path is the sum of its edge weights

- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex *S*
- Assumptions:
  - the graph is connected
  - the edge weights are **nonnegative**

Single Source Shortest Paths on unweighted Graph

Given a graph G = (V, E) directed, unweighted and an initial vertex **s**, find all shortest paths between this and any other vertex of the graph



Shortest paths starting at vertex 1:	**Weight path**
– 1-2	1
– 1-5	1
– 1-2-3	2
– 1-5-4	2

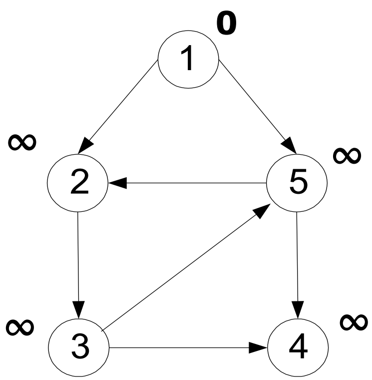
Dijkstra’s Algorithm for an unweighted Graph

- The main idea is to perform a breadth-first search starting at the source vertex *S*

- The algorithm uses two vectors which records for the others vertices:
  - the distance from each vertex to the initial vertex (dist)
  - the predecessor on the shortest path (path)

- The algorithm starts to mark the initial vertex *S* with length 0
- then processes its adjacent vertices that are marked with a path of length 1, and continues making a breadth-first search

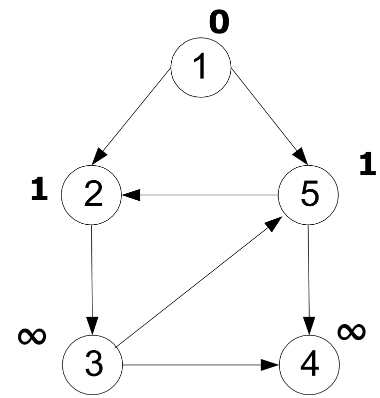
Dijkstra's Algorithm exemplification



dist	0	∞	∞	∞	∞
	1	2	3	4	5

path	0	0	0	0	0
	1	2	3	4	5

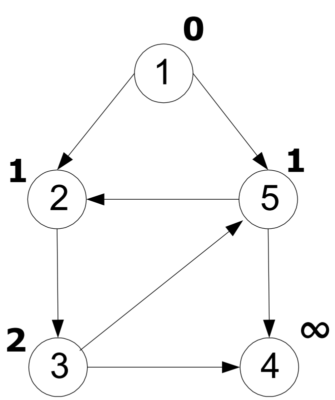
queue		1			



dist	0	1	∞	∞	1
	1	2	3	4	5

path	0	1	0	0	1
	1	2	3	4	5

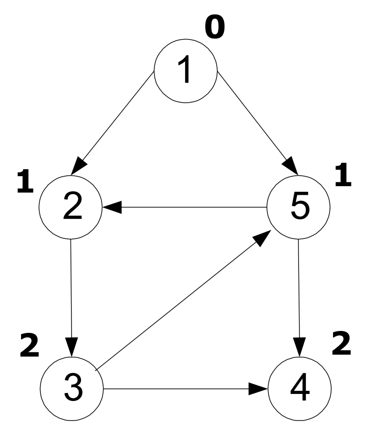
queue		2	5		



dist	0	1	2	∞	1
	1	2	3	4	5

path	0	1	2	0	1
	1	2	3	4	5

queue		5	3		



dist	0	1	2	2	1
	1	2	3	4	5

path	0	1	2	5	1
	1	2	3	4	5

queue		3	4		

.... ends

queue					

Dijkstra's Algorithm for an unweighted Graph

```
Algorithm void shortestPathEdges(Graph<V,E> g, V vOrig) {

  for (all V vertices in g){ dist[Vkey]=∞ path[Vkey]=-1 }
  add vOrig to queue-aux
  dist[VOrigKey]=0
  while (!queue-aux is Empty){
    vOrig ← Remove first vertex from queue-aux
    for (each vAdj of vOrig)
       if (dist[vAdjKey] = ∞) {
          dist[vAdjkey] = dist[vOrigkey] + 1
          path[vAdjkey] = vOrigkey
          Add vAdj to queue-aux  }
  }
}
```

Time Complexity: O(?)

Dijkstra's Algorithm for a weighted Graph

The problem of computing the shortest path in a weighted graph is solved by making minor modifications to the previous algorithm

- As in that algorithm, a **distance vector** that maps vertices to their distances from the source vertex *S* is kept
- Instead of adding 1 to the distance, it is added the weight of the edge traversed
- **Crucial modification**: **at each iteration choose the vertex with the smallest distance to the source vertex**

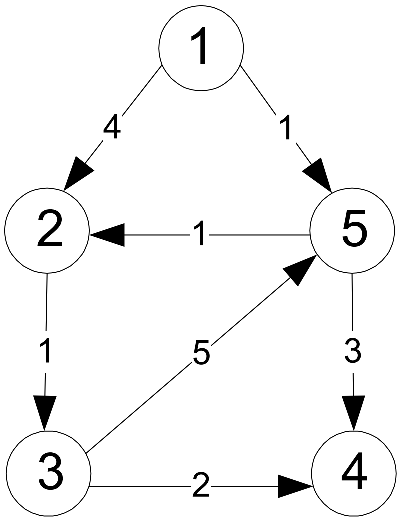
This approach is simple but, nevertheless powerful is an example of the **greedy-method design pattern**

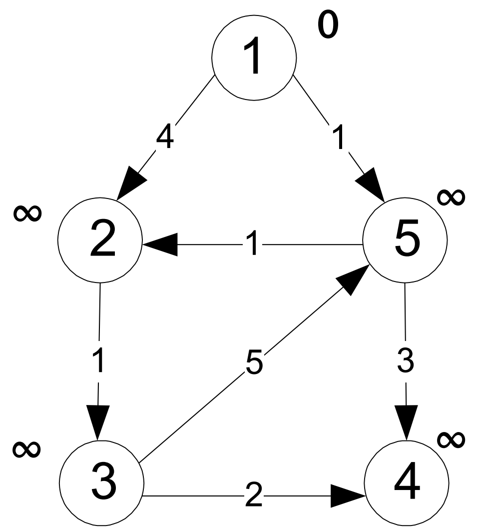
Restriction: is only valid if there are no branches with negative costs

Design pattern: greedy method

- The greedy method solves a given optimization problem using a **sequence of choices**:
  - The sequence starts from some well-understood starting condition and computes the cost for that initial condition
  - Next, the pattern makes additional choices by identifying the decision that achieves the **best cost improvement from all of the choices** that are currently possible
- This approach **does not always lead to an optimal solution**
- On average this approach produces, in linear time, solutions **25% more expensive** than the optimal solution
- But there are several problems that it does work for, and such problems are said to possess the ***greedy-choice property***

Dijkstra's Algorithm for a weighted graph exemplification

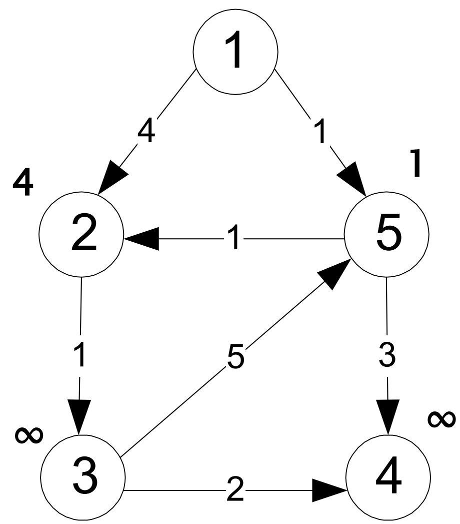




path	0	0	0	0	0
	1	2	3	4	5

dist	0	∞	∞	∞	∞
	1	2	3	4	5

visited	0	0	0	0	0
	1	2	3	4	5

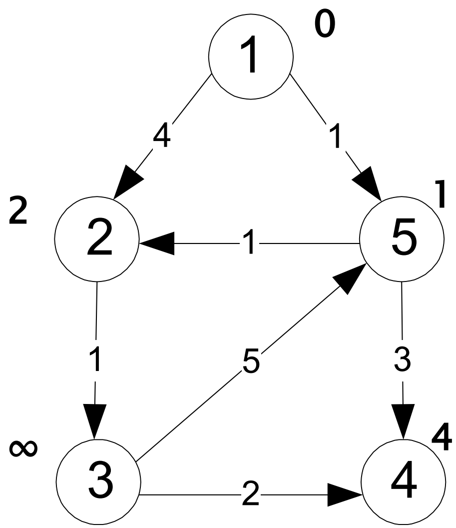


path	0	1	0	0	1
	1	2	3	4	5

dist	0	4	∞	∞	1
	1	2	3	4	5

visited	1	0	0	0	0
	1	2	3	4	5

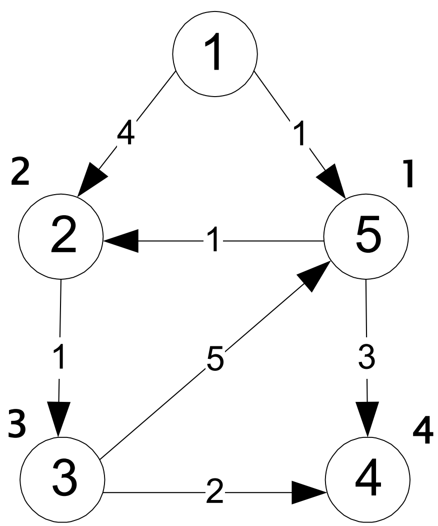
Dijkstra's Algorithm for a weighted graph exemplification



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

dist	**0**	**2**	**∞**	**4**	**1**
	1	**2**	**3**	**4**	**5**

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



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

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

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

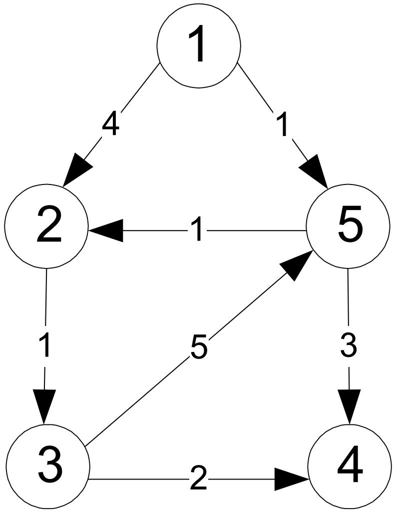
...

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

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

visited	**1**	**1**	**1**	**1**	**1**
	1	**2**	**3**	**4**	**5**

Single Source Shortest Paths on a weighted Graph



path	0	5	2	5	1
	1	2	3	4	5

dist	0	2	3	4	1
	1	2	3	4	5

Shortest paths

Source Vertex 1 :	**Weight**
– 1-5-2	2
– 1-5-2-3	3
– 1-5-4	4
– 1-5	1

Dijkstra’s Algorithm for a weighted Graph

```
Algorithm void shortestPathLength(Graph<V,E> g, V vOrig,
                      boolean[] visited, int[] path, double[] dist) {
  for (all V vertices in g) { dist[V]=∞ path[V]=-1 visited[V]=false }
  dist[vOrig]=0
  while (vOrig != -1){
    visited[vOrig]=true
    for (each vAdj of vOrig){
      get edge between vOrig e vAdj
      if (!visited[vAdj] && dist[vAdj]>dist[vOrig]+edge.getWeight()){
          dist[vAdj] = dist[vOrig]+edge.getWeight()
          path[vAdj] = vOrig
       }
    }
    vOrig = getVertMinDist(dist, visited)
  }
}
```

Time Complexity: O(?)

Graphs with negative weight edges

Applications:

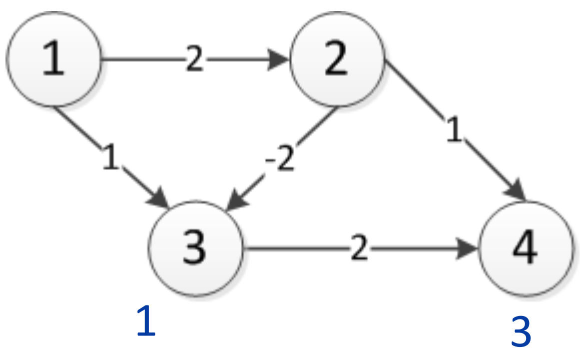
- Distance vector routing protocols in networking
- Currency Exchange
- Chemical processes

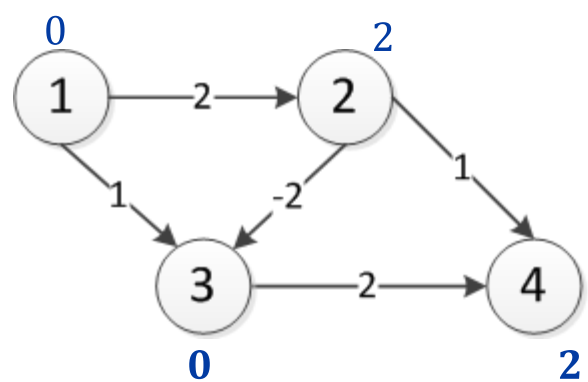
Dijkstra's algorithm fails for graphs with negative weight edge

Negative weight edges

Why Dijkstra's algorithm doesn´t work for Negative-Weight Edges?

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance
- When a vertex with a negative incident edge is selected late, it could alter distances of vertices already processed

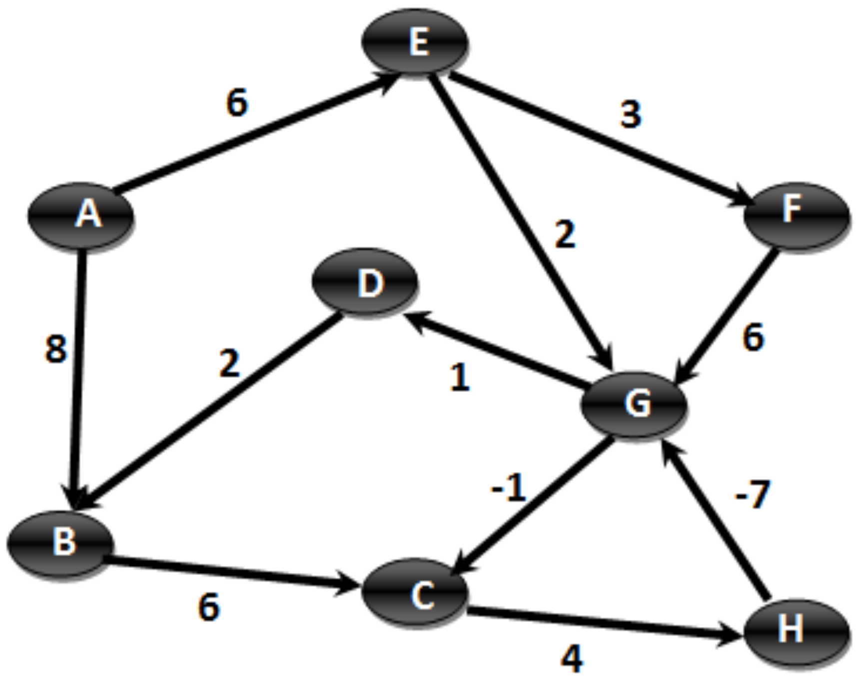




- Dijkstra's algorithm would visit vertice 3, then 4 and leave vertices 3 and 4 with a wrong distance

Bellman-Ford Algorithm

Bellman-Ford algorithm computes the *Single Source Shortest Path* in graphs with negative weight edge but with no negative cycles



Graph with a negative weight cycle (C, H, G, C) with weight -4

Bellman-Ford algorithm will either give a valid shortest path, or will indicate that there is a negative weight cycle

Bellman-Ford Algorithm

The algorithm shares the notion of edge relaxation from Dijkstra's algorithm, but does not use it in conjunction with the greedy method

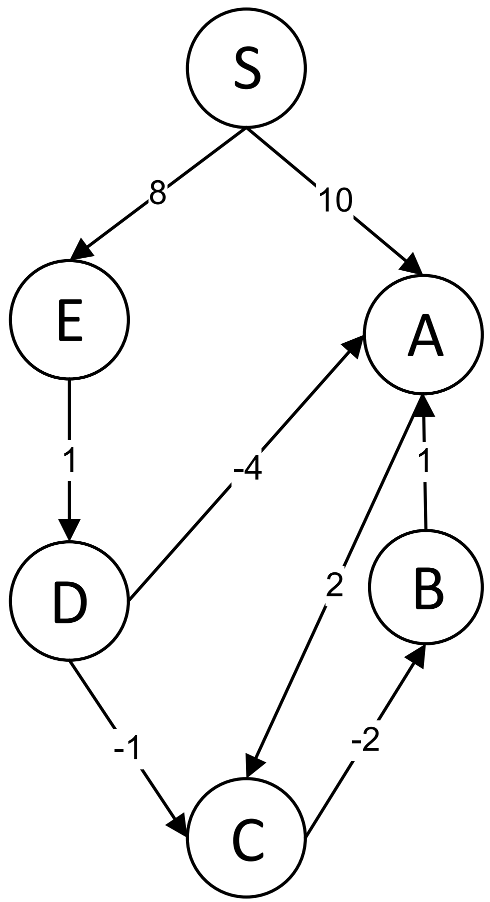
Basic Idea

- Assuming there are no negative cycles, every shortest path is Simple and contains at most n-1 edges
- Therefore the Bellman-Ford algorithm correctly identifies all shortest paths from a source vertex *S* in at most **V-1 iterations**

Bellman-Ford Algorithm – Basic Idea

- Iterate over the number of vertices
- Keep track of the current shortest path (distance and parent) for each vertex
- For each iteration, "relax" all edges
  - Consider whether this edge can be used to improve the current shortest path of the vertex at its endpoint
  - Because every shortest path is simple, after at most n-1 iterations, every shortest path is determined
- To check for negative cycles, after completing n-1 iterations, simply scan all edges one more time to see if there is a vertex that could still be improved
  - If so, that means there exists a path longer than n-1 edges to achieve the shortest path, which implies a negative cycle

Bellman-Ford Algorithm – Exemplification

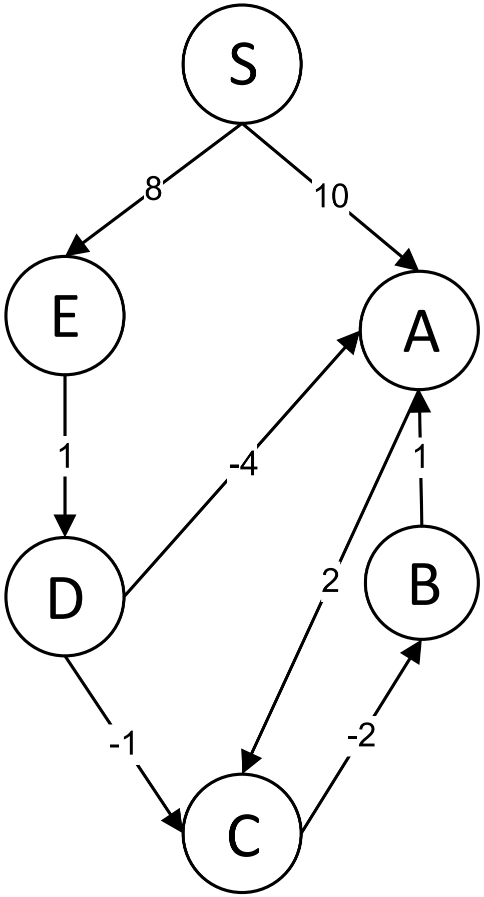


Step1: Considering S as the source, assign it the cost zero and all other vertices assign an infinity cost

0	∞	∞	∞	∞	∞
S	A	B	C	D	E

Step2: Take one vertex at a time and relax all the edges in the graph

Bellman-Ford Algorithm – 1st Iteration



Vertex S

0	10	∞	∞	∞	8
S	A	B	C	D	E

Vertex A

0	10	∞	12	∞	8
S	A	B	C	D	E

Vertex B: not reached yet

Vertex C

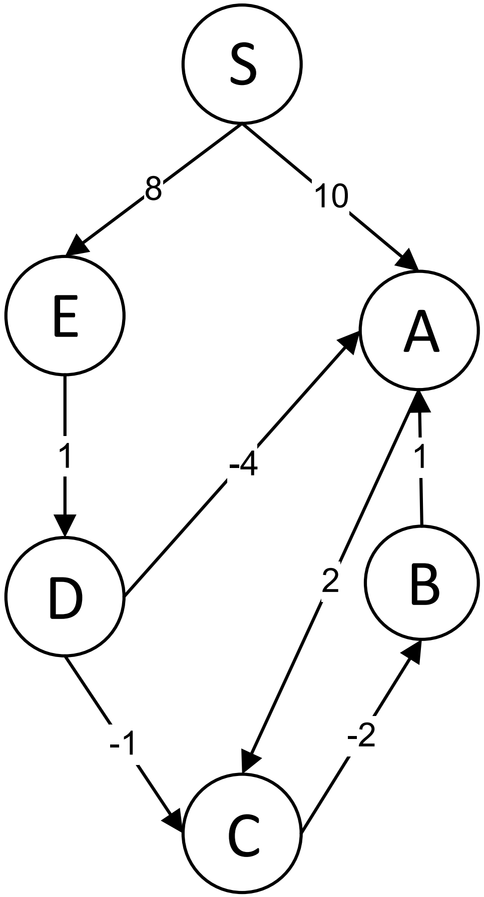
0	10	10	12	∞	8
S	A	B	C	D	E

Vertex D: not reached yet

Vertex E

0	10	10	12	9	8
S	A	B	C	D	E

Bellman-Ford Algorithm – 2nd Iteration



Vertex S

0	10	10	12	9	8
S	A	B	C	D	E

Vertex A

Vertex B

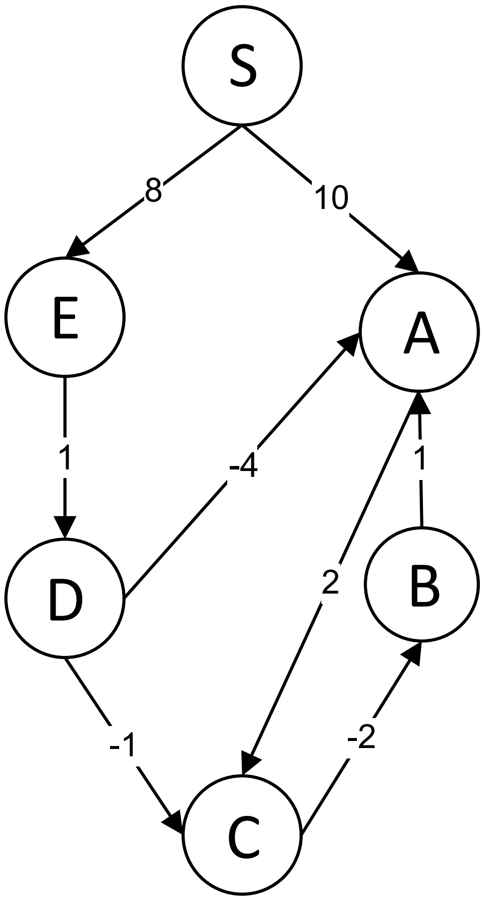
Vertex C

Vertex D

0	5	10	8	9	8
S	A	B	C	D	E

Vertex E

Bellman-Ford Algorithm – Exemplification



Step3: Take one vertex at a time and relax all the edges in the graph

2nd Iteration

0	5	10	8	9	8
S	A	B	C	D	E

3th Iteration

0	5	5	7	9	8
S	A	B	C	D	E

4th Iteration

0	5	5	7	9	8
S	A	B	C	D	E

The fourth iteration doesn’t improve the distances - the algorithm stops previously to V-1 iterations

Bellman-Ford Algorithm

```
Algorithm boolean BellmanFord (Graph<V,E> g, V vOrig, int[] path,
                                                     double[] dist){

  for (all V vertices in g) { dist[V]=∞ path[V]=-1 }
  dist[vOrig]=0
  boolean negativeCycle = false
  for (all VOrig vertices in g) {
     for (each vAdj of vOrig){
        get edge between vOrig and vAdj
        if (dist[vAdj] > dist [vOrig]+edge.getWeight ) { //relaxation
           dist[vAdj] = dist [vOrig]+ edge.getWeight
           path[vAdj] = vOrig  }
     }

  for (every edge(VOrig,vAdj) in g) //determine a negative cycle
     if (dist[vAdj] > dist[vOrig]+edge.getWeight)
        negativeCycle = true

  return negativeCycle
}
```

Time Complexity (?)

Constrained Shortest Path

Constrained Shortest Path (CSP) Problem

Problem

Given a beginning vertex Vb, an ending vertex Ve, and a subset V$s \subseteq V$, find a path with the minimum length among all the paths passing through every V$i \in$ Vs from Vb to Ve. The **subset $\boldsymbol{Vs}$** is called **vertex constraint**; that is, the shortest path must pass through every vertex in the subset Vs

Problems of Shortest path with constraints:

- definition of tourist itineraries
- industrial robot circuit boards
- planning of new telecommunication networks
- carpooling with the development of sharing economy

CSP – Brute Force Solution

A brute force solution involves determining all possible routes (applying successively Dijkstra algorithm) and choose the shortest one

Is this solution computational feasible?

CSP – Brute Force Solution

1. Generate all permutations
2. Find a path with minimum cost

Number of Intermediate nodes	Number of Paths	6 paths/sec.
A,B	X, A, B, Y X, B, A, Y	
A,B,C	X,A,B,C,Y X,A,C,B,Y X,B,A,C,Y X,B,C,A,Y X,C,A,B,Y X,C,B,A,Y	
A,B,C,D	24 paths	144 sec.
13 intermediate nodes	**6 227 020 800 paths**	**33 years**
n intermediate nodes	n! paths	

Constrained Shortest Path - Heuristics

Search Heuristics (Informed Search) use domain-specific information to aid decision making

- Nearest-neighbor method (greedy method)
- Branch&Bound
- A*

Graphs
Circuits

Circuits

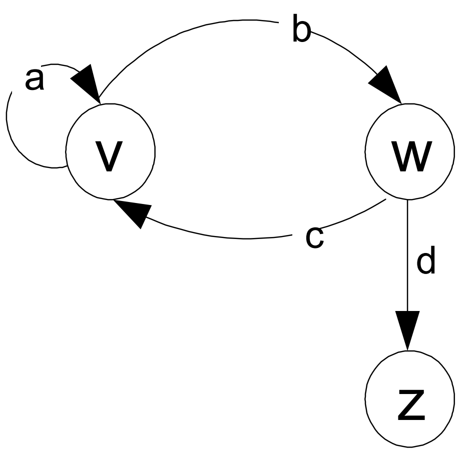
Circuit (or cycle) is a closed path that does not contain a repeated edge

Simple circuit is a circuit which does not have a repeated vertex (v_0, v_1, v_2, ..., v_k) except for the first and last $v_k = v_0$

If (u, v, w, ..., z, u) is a cycle, (v, w, ..., z, u, v) is also a cycle

Any cyclic permutation of a cycle is an original equivalent cycle

Example:



The paths:

- (v,v) is a cycle length 1
- (v,w,v) is a cycle length 2
- (w,v,w) is an equivalent cycle to the above
- (w,v,v,w) is not a cycle - Vertex v is repeated

Cycle Detection - Basic Idea

- In a DFS of an acyclic graph, a vertex whose adjacent vertices have all been visited, can be seen again without implying a cycle
- However, if a vertex is seen a second time before all of its adjacent vertices have been visited, then there must be a cycle
- Suppose there is a cycle containing vertex A. Then this means that A must be reachable from one of its adjacent vertices. So, when the DFS is visiting one adjacent vertex, it will see A again, before it has finished visiting all of A's adjacent - **So there is a circuit**

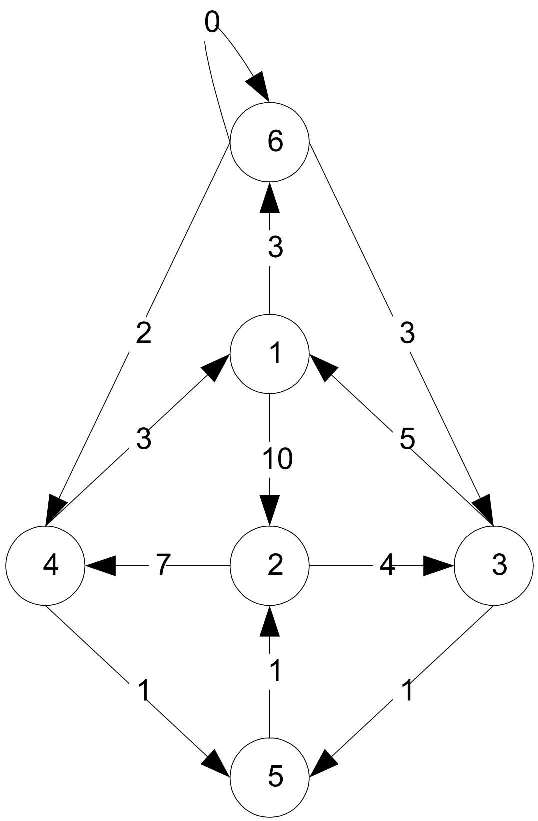
- In order to detect cycles, we use a modified DFS called a colored DFS
  - All nodes are initially marked white
  - When a node is encountered, it is marked grey
  - and when its adjacents are completely visited, it is marked black
- If a grey node is ever encountered, then there is a cycle

Circuit Detection - Colored DFS Algorithm

```
Algorithm boolean colorDFS(Graph<V,E> G, V vOrig, int[] color) {
    make vOrig gray
    for (each vAdj of vOrig) {
       if (vertex vAdj is gray)
          return true
       if (vertex vAdj is white)
          return colorDFS(G, vAdj, color)
    }
    make vOrig black
    return false
 }
```

Time Complexity (?)

All circuits in a Graph



Adjancency List

1 → 2, 6

2 → 3, 4

3 → 1, 5

4 → 1, 5

5 → 2

6 → 3, 4, 6

All cycles (– 9 –) :

- 1-2-3-1
- 1-2-4-1
- 1-6-3-1
- 1-6-3-5-2-4-1
- 1-6-4-1
- 1-6-4-5-2-3-1
- 2-3-5-2
- 2-4-5-2
- 6-6

All circuits in a Graph – Algorithm

```
Algorithm ArrayList<LinkedList<V>> allCycles (Graph<V,E> g){

   for (all Vertex vOrig in g) {
      for (all V Vertices previous to vOrig) {
         visited[V]=true
      }
      allPaths(g, vOrig, vOrig, visited, path, paths);
   }
   return paths;
 }
```

Inefficient algorithm: the number of steps needed to carry it out grows disproportionally with the number of vertices

Euler Circuit

An Euler path is a path that uses every edge of a graph exactly once. An Euler path starts and ends at different vertices

An Euler circuit is a circuit that uses every edge of a graph exactly once. An Euler circuit starts and ends at the same vertex

Accordingly to Euler theorem:

- 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

There are two algorithms to find an Euler circuit:

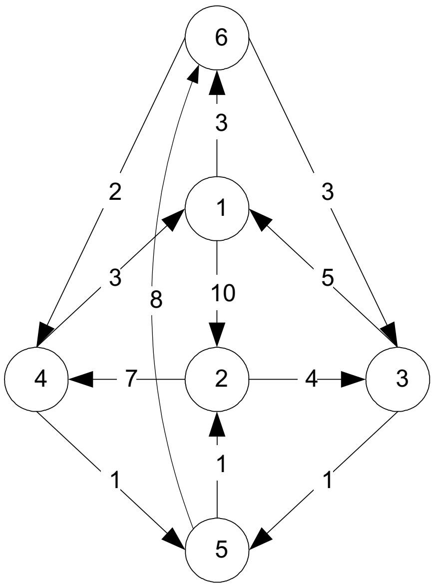
- Hierholzer Algorithm
- Fleury Algorithm

Hierholzer Algorithm – Basic Idea

- Every vertex of G has degree ≥ 2 so, G necessarily has some simple cycle C_1
  - if C_1 contains all edges of G - C_1 is the Euler circuit
  - if not, remove from G all the edges of C_1
  - The resulting graph has all vertices still with even degree
  - So, it is possible to determine new cycle C_i
  - ...... and repeat this process until no more edges in G

- At the end, the edges of G are partitioned into simple cycles
- Because G is connected, each cycle C_i has at least one vertex in common which permits to make the union of all cycles and obtain the Euler cycle that contains all edges of G exactly once

Hierholzer Algorithm – Exemplification

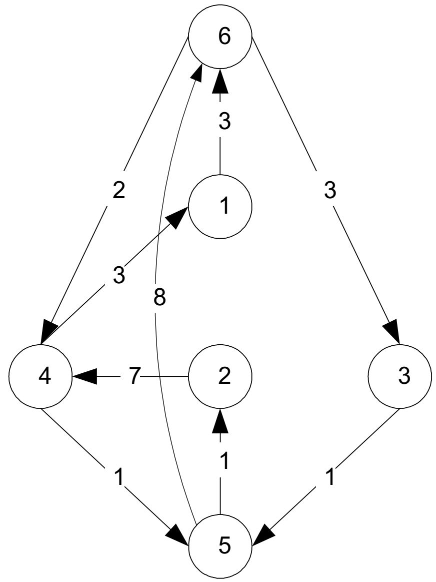


Adjacency List

1 → 2, 6
2 → 3, 4
3 → 1, 5
4 → 1, 5
5 → 2, 6
6 → 3, 4

1st iteration:

1 – 2 – 3 – 1

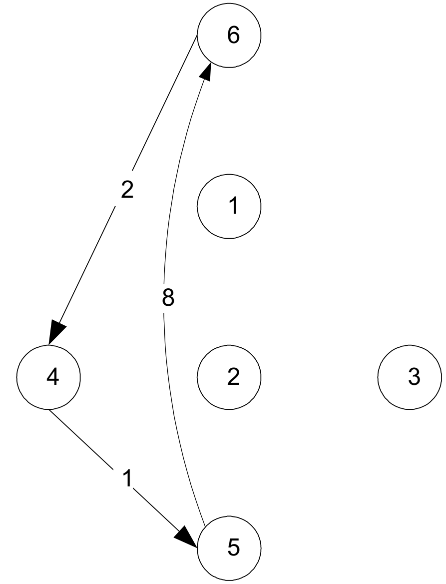


Adjacency List

1 → 6
2 → 4
3 → 5
4 → 1, 5
5 → 2, 6
6 → 3, 4

2nd iteration:

1 – 6 – 3 – 5 – 2 – 4 – 1



Lista de adjacências

1 →
2 →
3 →
4 → 5
5 → 6
6 → 4

3th iteration:

4 – 5 – 6 – 4

Hierholzer Algorithm – Exemplification

Found cycles

1 – 2 – 3 – 1

1 – 6 – 3 – 5 – 2 – 4 – 1

4 – 5 – 6 – 4

Euler Circuit = Union of all cycles

1 – 2 – 3 – 1

1 – 6 – 3 – 5 – 2 – 4 – 1

4 – 5 – 6 – 4

Euler Circuit:

1 – 2 – 3 – 1 – 6 – 3 – 5 – 2 – 4 – 5 – 6 – 4 – 1

Graphs
Topological Sort

Topological Sort

Topological sort of a direct acyclic graph G(V,E) is a linear ordering of all the vertices in the graph $V_1, V_2, \ldots, V_n$ such that vertex V_i comes before vertex V_j if there is an edge $(V_i, V_j) \in G$

- Topological Sort is only possible in a **Direct Acyclic Graph (DAG)**
- There can be multiple topological sorts of a graph G

Applications of Topological Sort include the following:

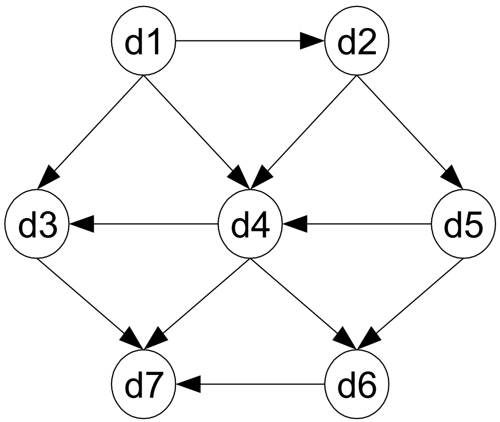
- Prerequisites between courses of an academic program
- Inheritance between classes of an object-oriented program
- Scheduling constraints between the tasks of a project

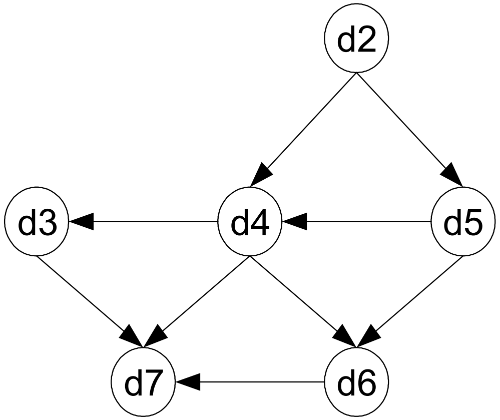
Topological Sort – Basic Idea

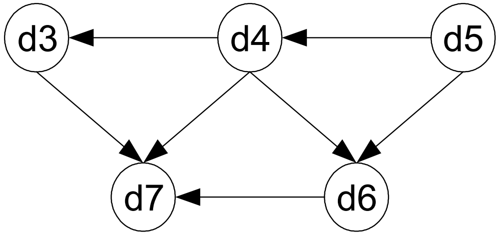
Lemma: All Directed Acyclic Graph (DAG) has at least one vertex with input degree zero

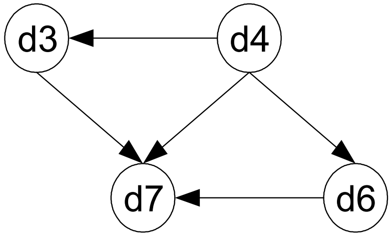
- Starts to process all vertices with the input degree zero
- After these vertices are processed the input degree of its adjacent vertices are decremented
- The adjacent vertices which input degree becomes zero are the next vertices to be processed
- And so on…

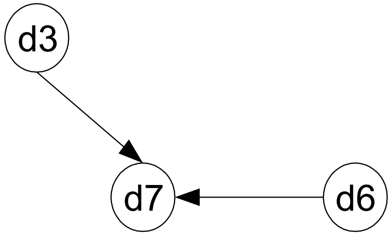
Topological Sort – Exemplification











The graph has two Topological Sorts:

$d_1, d_2, d_5, d_4, d_3, d_6, d_7$

$d_1, d_2, d_5, d_4, d_6, d_3, d_7$

Topological Sort – First Algorithm

```
Algorithm boolean topologicalSort (Graph<V,E> g, List<V> topsort) {
  for (all V vertices in g) {
     degreeIn[idxV] = g.inDegree(V)
     if (degreeIn[idxV] == 0) Add vertex V to queue-aux }
  numVerts=0
  while (!queue-aux is Empty){
    vOrig ← remove first vertex from queue-aux
    add vOrig to topsort
    numVerts++
    for (each vAdj of vOrig){
       degreeIn[vAdj]--
       if (degreeIn[vAdj] == 0) add vertex VAdj to queue-aux
    }
  }
  if (numVerts < g.numVertices()) //Graph has a cycle
     return false
  return true }
```

Time Complexity (?)

Topological Sort – Version more simple

- Use the DFS algorithm to process the vertices not yet visited
- As each vertex ends (has no adjacent vertices) it is placed in a stack (which guarantees that it appears after all its predecessors)
- At the end of the algorithm, the stack contains the topological sort of the graph

Note

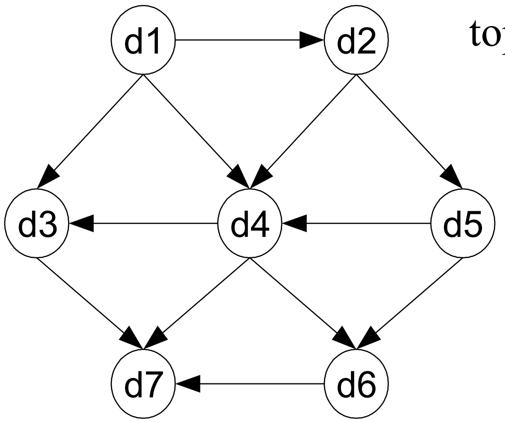
- This algorithm can be initiated by any vertex, regardless of its Input degree
- Therefore, it is possible to find different correct Topological Sorts for the same Directed Acyclic Graph

Topological Sort – Algorithm

```
Algorithm Queue<V> topologSort(Graph<V,E> g) {

 for (all V Vertices in g) { visited[V]=false }
 for (each of vOrig of graph g)
    if (vertex vOrig has not been visited)
       topologSort(G, vOrig, visited, topsort);
 return topsort
}

Algorithm void topologSort(Graph<V,E> g, V vOrig, boolean[] visited,
                                        LinkedList<V> topsort) {
  make vOrig as visited
  for (each vAdj of vOrig)
     if (!visited[vAdj])
        topologSort(g,vAdj,visited,topsort)
  push vOrig into topsort
}
```

Time Complexity: O(?)

Topological Sort – Exemplification



Adjacency List

1 → 2, 3, 4
2 → 4, 5
3 → 7
4 → 3, 6, 7
5 → 4, 6
6 → 7
7 →

topologSort (1, vector, topsort)
topologSort (2, vector, topsort)
topologSort (4, vector, topsort)
topologSort (3, vector, topsort)
topologSort (7, vector, topsort)

visited	1	1	1	1	0	0	1
	1	2	3	4	5	6	7

topsort (7)
topsort (7, 3)

topologSort (6, vector, topsort)

visited	1	1	1	1	0	1	1
	1	2	3	4	5	6	7

topsort (7, 3, 6)
topsort (7, 3, 6, 4)

topologSort (5, vector, topsort)

visited	1	1	1	1	1	1	1
	1	2	3	4	5	6	7

topsort (7, 3, 6, 4, 5)
topsort (7, 3, 6, 4, 5, 2)
topsort (7, 3, 6, 4, 5, 2, 1)

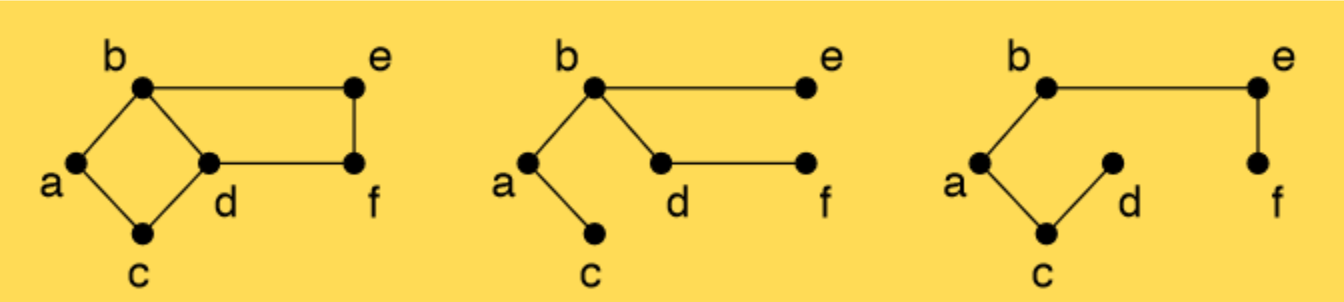
Graphs
Minimum Spanning Trees

Minimum Spanning Tree (MST)

Given an undirected, connected graph G=(V,E) with positive edge weights, a minimum spanning tree T=(V,E') is a tree that connects all the vertices of the graph G with the minimum cost (weights)

|E'| = |V|-1 → The number of edges of the MST is one less than the number of vertices, so that there are no cycles

A graph can have many spanning trees, but all have V vertices and |V|-1 edges



Minimum Spanning Tree

MST is a fundamental problem with diverse applications
- network design: telephone, electrical, hydraulic, computer
- Cluster analysis

Two algorithms to find the MST of an undirected, connected graph:
Kruskal's algorithm
- Consider edges in ascending order of cost
- Add the next edge to T unless doing so would create a cycle

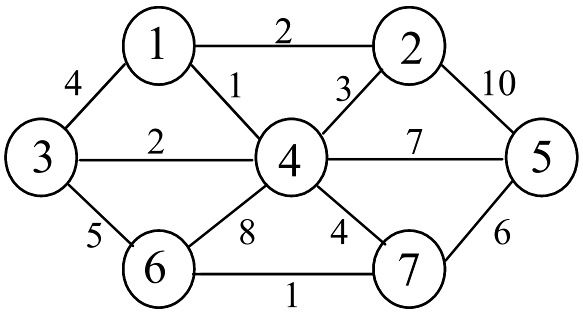
Prim's algorithm
- Start with any vertex S and greedily grow a tree T from S
- At each step, add the cheapest edge to T that has exactly one endpoint in T

Kruskal's algorithm – Basic Idea

Given an undirected, connected graph G = (V, E) with positive edge weights

- At the beginning each vertex is an isolated vertex
- All edges of the graph are inserted in a vector in order of increasing weight
- each edge is removed from the vector and if it joins two vertices that are not connected, it is added to the MST
- this operation is repeated until all the vertices are linked
- When the algorithm terminates there is only one tree – the **minimum spanning tree** with V-1 edges

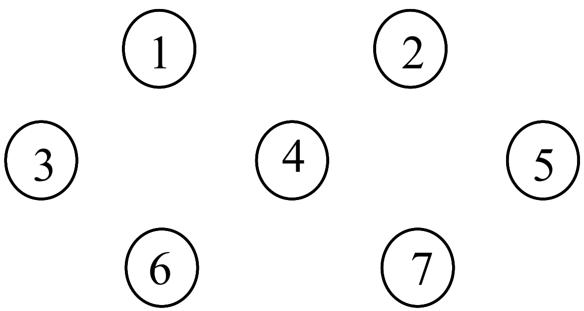
Kruskal's algorithm – Exemplification



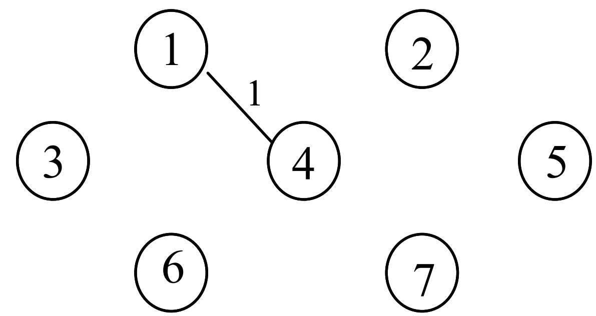
Ordered set of edges:

{ (1,4), (6,7), (1,2), (3,4), (2,4), (1,3), (4,7), (3,6), (5,7), (4,5), (4,6), (2,5)}

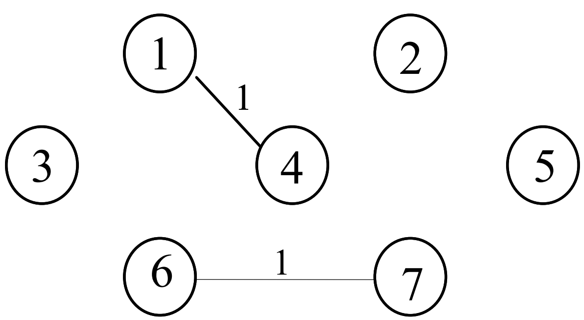
Q = {}



Q = {{1},{2},{3},{4},{5},{6},{7}}

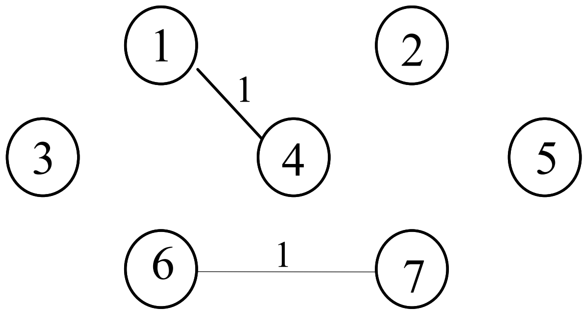


Q = {{1,4},{2},{3},{5},{6},{7}}



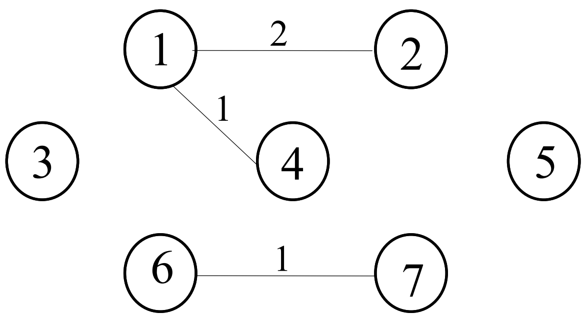
Q = {{1,4},{6,7},{2},{3},{5}}

Kruskal's algorithm – Exemplification

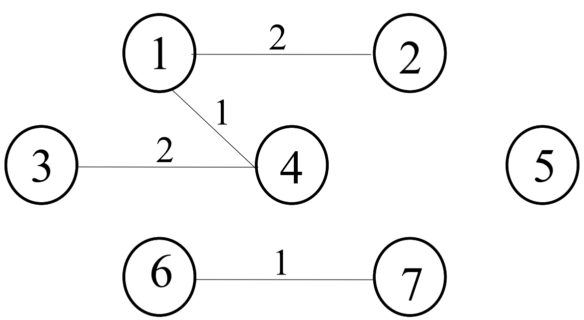


Ordered set of edges:

{ (1,4), (6,7), (1,2), (3,4), (2,4), (1,3), (4,7), (3,6), (5,7), (4,5), (4,6), (2,5)}

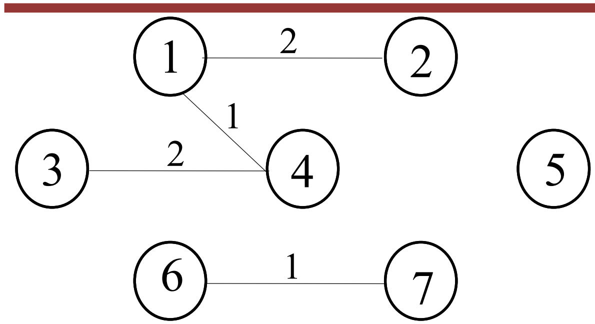


Q = {{1,4,2},{6,7},{3},{5}}



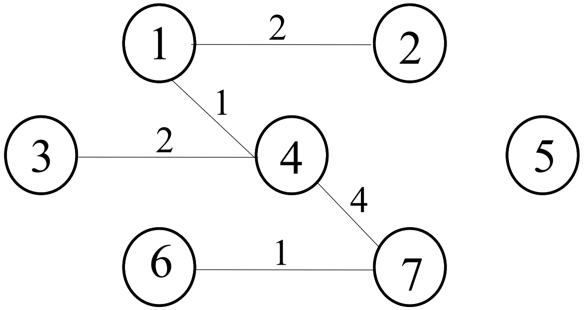
Q = {{1,4,2,3},{6,7},{5}}

Kruskal's algorithm – Exemplification

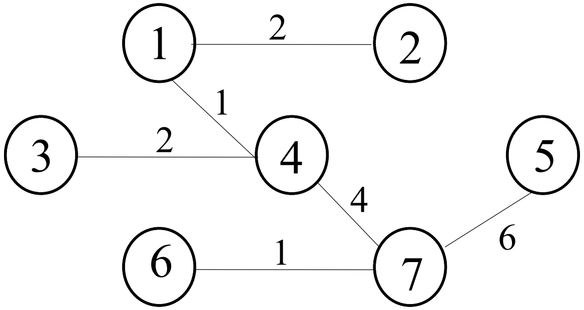


Ordered set of edges:

{ (1,4), (6,7), (1,2), (3,4), (2,4), (1,3),
(4,7), (3,6), (5,7), (4,5), (4,6), (2,5)}

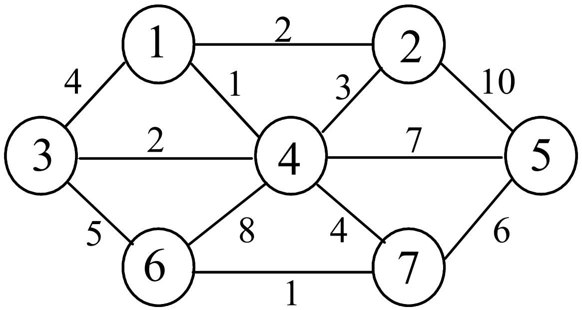


Q = {{1,4,2,3,6,7},{5}}



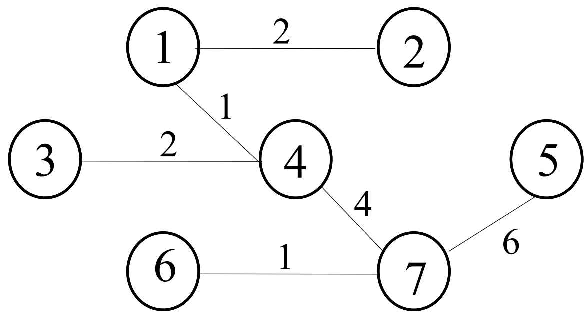
Q = {{1,4,2,3,6,7,5}}

Kruskal's algorithm – Exemplification



Ordered set of edges:

{ (1,4), (6,7), (1,2), (3,4), (2,4), (1,3), (4,7), (3,6), (5,7), (4,5), (4,6), (2,5)}



Minimum spanning tree

A = {(1,4), (7,6), (1,2), (4,3), (4,7), (5,7)}

The weight of the tree is the sum of its edges W (A) = 16

Kruskal's – Algorithm

```
Algorithm Graph<V,E> kruskall (Graph<V,E> g) {
  for (all V vertices in g)
       Add vertex V to mst
  for (all E edges in g)
       Add edge into a lstEdges
  sort lstEdges              // in ascending order of weight
  for (each Edge e=(VOrig,vDst) of lstEdges)
      connectedVerts=DepthFirstSearch(mst, vOrig);
      if (!connectedVerts.contains(vDst))
        add Edge to mst

  return mst;
}
```

Time Complexity (?)

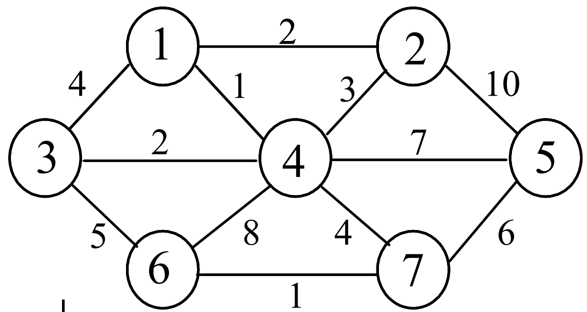
Prim’s algorithm – Basic Idea

- The main idea is to perform a breadth-first search starting at any source vertex *S*
- The algorithm uses three vectors which records for the vertices:
  - the distance from each vertex to its predecessor vertex (dist)
  - the predecessor vertex (path)
  - If the vertex has been processed (visited)

- The algorithm starts to mark the initial vertex S with length 0
- Then, update the weights of all its adjacent vertices not yet visited
- Next, choose the vertex not yet processed with less weight

Repeat the process until all vertices have been processed

Prim’s algorithm – Exemplification



Initial Vertex: 7

	1	2	3	4	5	6	7
path	0	0	0	7	7	7	0
dist	∞	∞	∞	4	6	1	0
visited	0	0	0	0	0	0	1

minimum(dist)=1, V = 6

	1	2	3	4	5	6	7
path	∞	∞	6	7	7	7	0
dist	∞	∞	5	4	6	1	0
visited	0	0	0	0	0	1	1

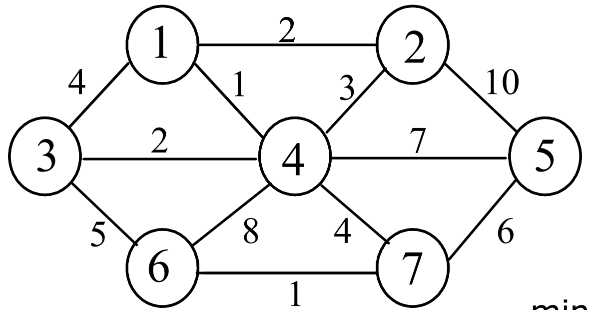
minimum(dist)=4, V = 4

	1	2	3	4	5	6	7
path	4	4	4	7	7	7	0
dist	1	3	2	4	6	1	0
visited	0	0	0	1	0	1	1

minimum(dist)=1, V = 1

	1	2	3	4	5	6	7
path	4	1	4	1	7	7	0
dist	1	2	2	1	6	1	0
visited	1	0	0	1	0	1	1

Prim's algorithm – Exemplification



minimum(dist)=2, V = 3
Adj[3] = {1, 4, 6} → already processed
→ doesn't change dist vector

minimum(dist)=2, V = 2
V = 2 → doesn't change dist vector

path	4	1	4	1	7	7	0
	1	2	3	4	5	6	7

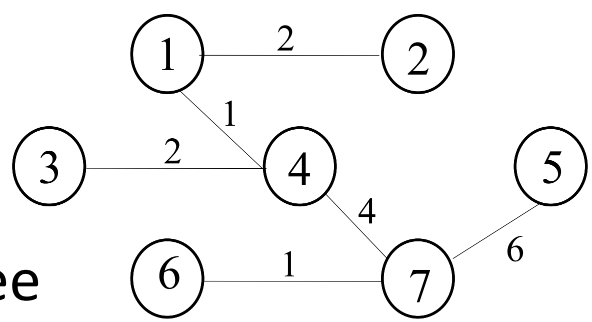
visited	1	1	1	1	0	1	1
	1	2	3	4	5	6	7

dist	1	2	2	1	6	1	0
	1	2	3	4	5	6	7

minimum(dist)=6, V = 5
Adj[5] = {2, 4, 7} → already processed
→ doesn't change dist vector

visited	1	1	0	1	0	1	1
	1	2	3	4	5	6	7

visited	1	1	1	1	1	1	1
	1	2	3	4	5	6	7



Minimum Spanning Tree

Prim's Algorithm

```
Algorithm void prim(Graph<V,E> g, Vertex<V,E> vOrig, Graph<V,E> mst){

  for (all V vertices in g) {dist[V]=∞ path[V]=-1 visited[v]=false }
  dist[vOrig]=0
  while (vOrig != -1){
    make vOrig as visited
    for (each vAdj of vOrig){
      get edge between vOrig and vAdj
      if (!visited[vAdj] && dist[vAdj] > edge.getWeight()){
          dist[vAdj] = edge.getWeight()
          path[vAdj] = vOrig  }
     }
     vOrig = getVertMinDist(dist, visited)
  }
  mst=buildMst(path,dist)
}
```

Time Complexity: O(?)

Graphs
Maximum Network Flow

Network Flow Problem

A type of network optimization problem

Arise in many different contexts:

- Point-to-point liquid supply
- Traffic between two points
- Irrigation systems
- rail traffic
- computer network: routing as many packets as possible on a given network
- Transport problems
  - Shipment of finished products from the producer to the distributor
  - Shipment of letters

Network Flow Problem

Settings:

A flow graph is a directed graph G=(V,E), where each edge E is associated with its capacity c(E) > 0

Two special nodes: source s and sink t are given (s ≠ t)

Source Node (S): in degree is zero

Destination Node (T): out degree is zero

Problem:

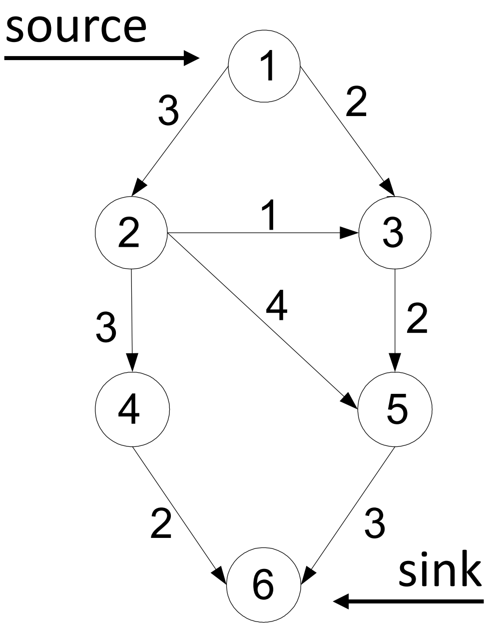
Maximize the total amount of flow from s to t subject to two constraints:

– Flow on edge E doesn't exceed c(E)

– For every node v ≠ s, t, **incoming flow is equal to outgoing flow**

=> There is no conservation of flow, or loss of flow

Maximum flow



All paths between the source and the sink nodes:

$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6$	Flow = 1
$1 \rightarrow 2 \rightarrow 4 \rightarrow 6$	Flow = 2
$1 \rightarrow 2 \rightarrow 5 \rightarrow 6$	Flow = 3
$1 \rightarrow 3 \rightarrow 5 \rightarrow 6$	Flow = 2

Ensures the Maximum Flow in the network?

Ford-Fulkerson Algorithm

A simple and practical max-flow algorithm

Main idea:

Find valid flow paths until there is none left, and add them up

Back Edges

If f amount of flow goes through u → v, then:

- Decrease c(u → v) by f
- Increase c(v → u) by f

Why do we need to do this?

– Sending flow to both directions is equivalent to canceling flow

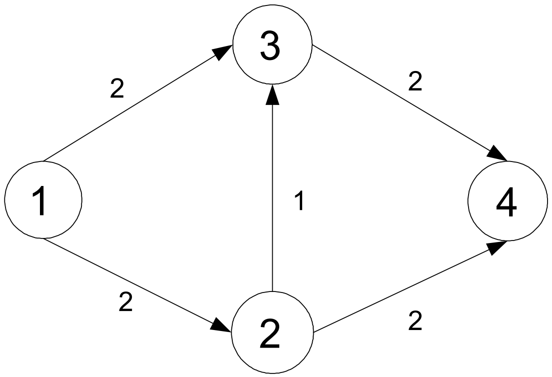
Ford-Fulkerson Pseudocode

Set $f_{total} = 0$

Repeat until there is no path from s to t:

- – Run DFS from s to find a flow path to t
- – Let f be the minimum capacity value on the path – Add f to f_{total}
- – For each edge u→v on the path:
  - Decrease c(u → v) by f
  - Increase c(v → u) by f

Maximum Flow

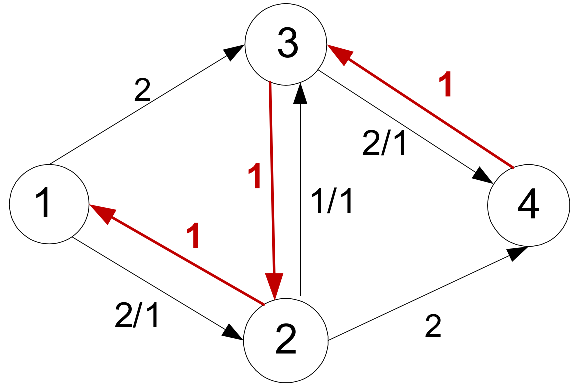


$1 \rightarrow 2, 3$
$2 \rightarrow 3, 4$
$3 \rightarrow 4$

1st Iteration

Path: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$

$Cap_{min} = 1 \quad \Rightarrow \quad$ Flow = 1

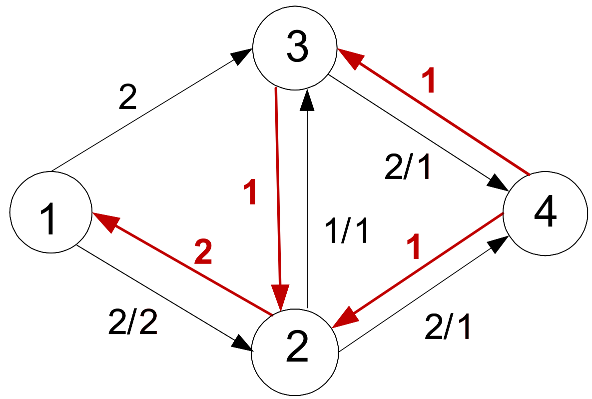


$1 \rightarrow 2, 3$
$2 \rightarrow 3, 4, 1$
$3 \rightarrow 4, 2$
$4 \rightarrow 3$

2nd Iteration

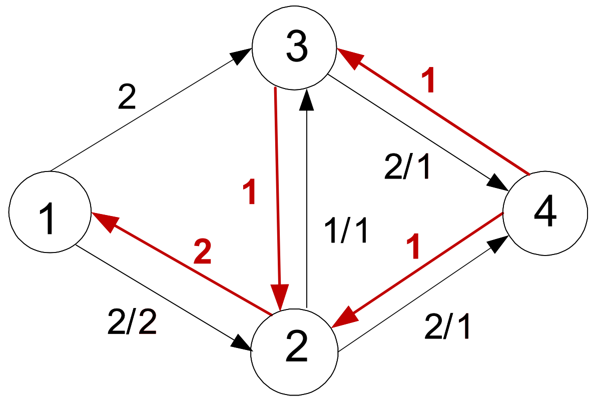
Path: $1 \rightarrow 2 \rightarrow 4$

$Cap_{min} = 1 \quad \Rightarrow \quad$ Flow = 1



$1 \rightarrow 2, 3$
$2 \rightarrow 3, 4, 1$
$3 \rightarrow 4, 2$
$4 \rightarrow 3, 2$

Maximum Flow

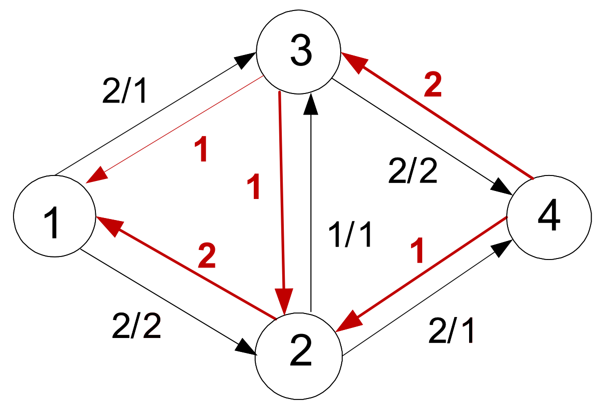


$1 \rightarrow$ 2, 3
$2 \rightarrow$ 3, 4, 1
$3 \rightarrow$ 4, 2
$4 \rightarrow$ 3, 2

3rd Iteration

Path: $1 \rightarrow 3 \rightarrow 4$

$Cap_{min} = 1 \Rightarrow$ Flow = 1

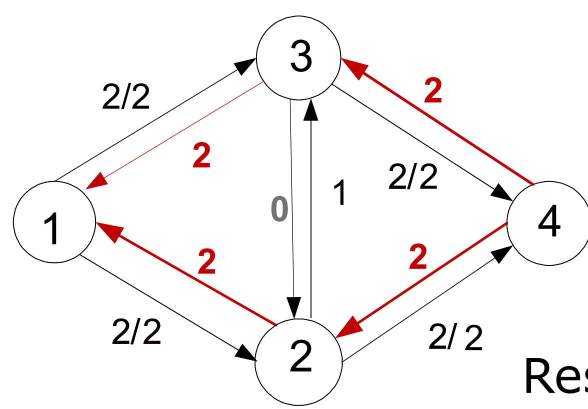


$1 \rightarrow$ 2, 3
$2 \rightarrow$ 3, 4, 1
$3 \rightarrow$ 4, 2, 1
$4 \rightarrow$ 3, 2

4th Iteration

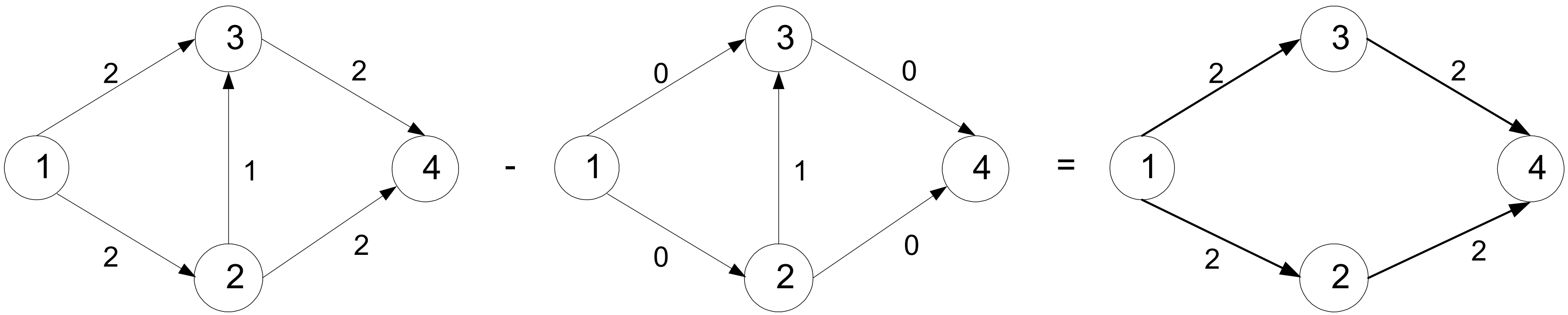
Path: $1 \rightarrow 3 \leftarrow 2 \rightarrow 4$

$Cap_{min} = 1 \Rightarrow$ Flow = 1



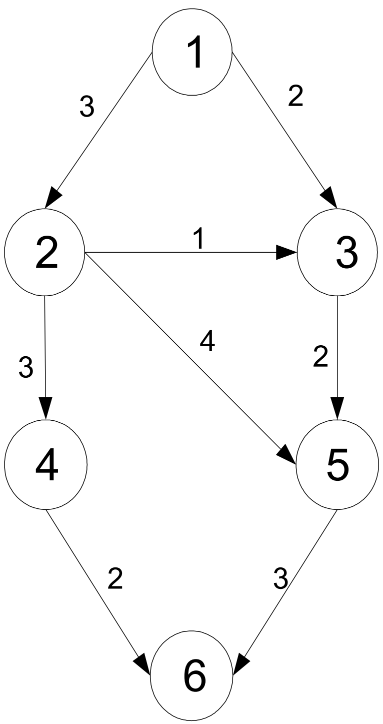
Residual Graph

Maximum Flow Graph



Initial Graph **minus** Residual Graph **=** Maximum Flow Graph

Ford-Fulkerson Algorithm

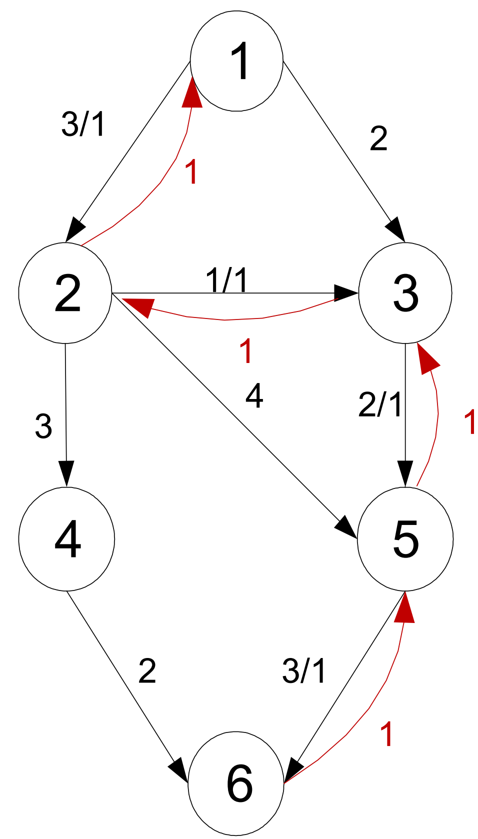


1 → 2, 3
2 → 3, 4, 5
3 → 5
4 → 6
5 → 6

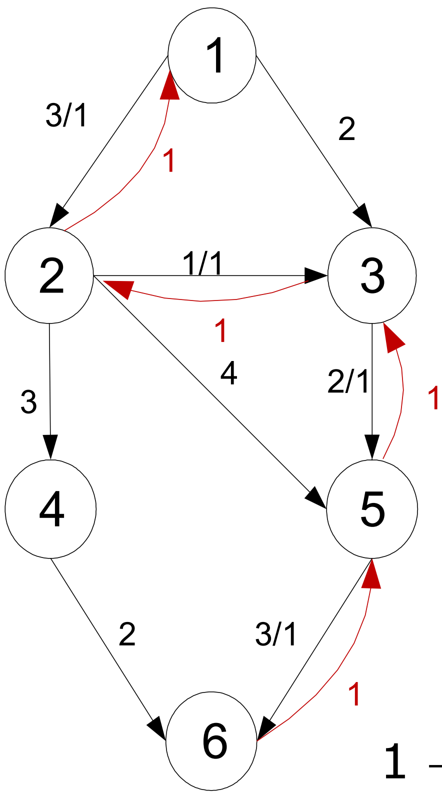
1st Iteration

Path: 1→2→3→5→6

Cap (3, 1, 2, 3) cap_{min} = 1



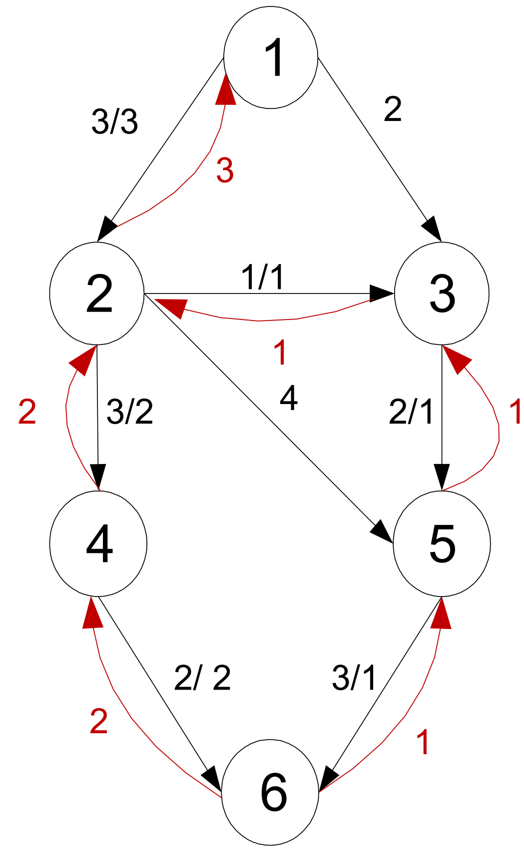
Ford-Fulkerson Algorithm



2nd Iteration

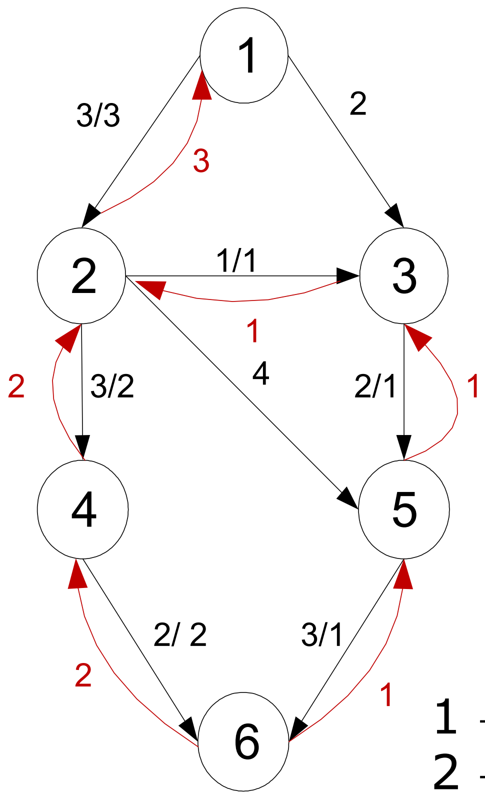
Path: 1→2→4→6

Cap (2, 3, 2) Cap_{min}= 2



1 → 2, 3
2 → 3, 4, 5, 1
3 → 5, 2
4 → 6
5 → 6, 3
6 → 5

Ford-Fulkerson Algorithm



3rd Iteration

Path: 1→2→5→6

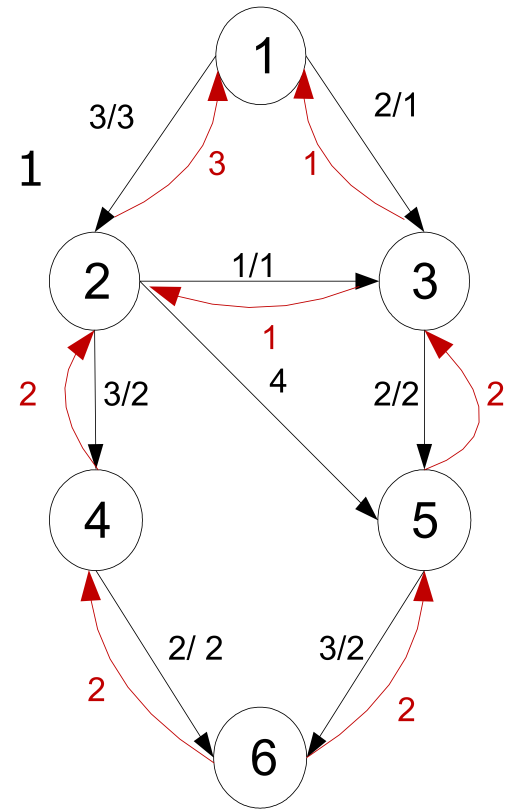
Cap (0, 4, 2) Cap_{min} = 0

4th Iteration

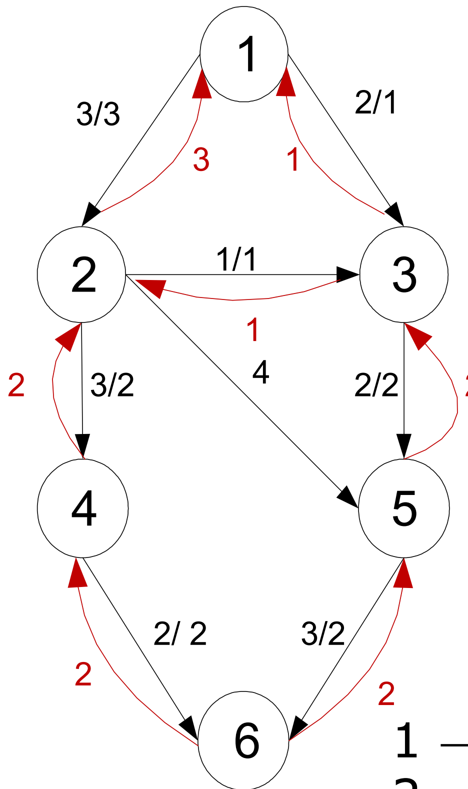
Path: 1→3→5→6

Cap(2, 1, 2) Cap_{min} = 1

1 → 2, 3
2 → 3, 4, 5, 1
3 → 5, 2
4 → 6, 2
5 → 6, 3
6 → 5, 4



Ford-Fulkerson Algorithm



5th Iteration

Path: 1→3←2→4→6

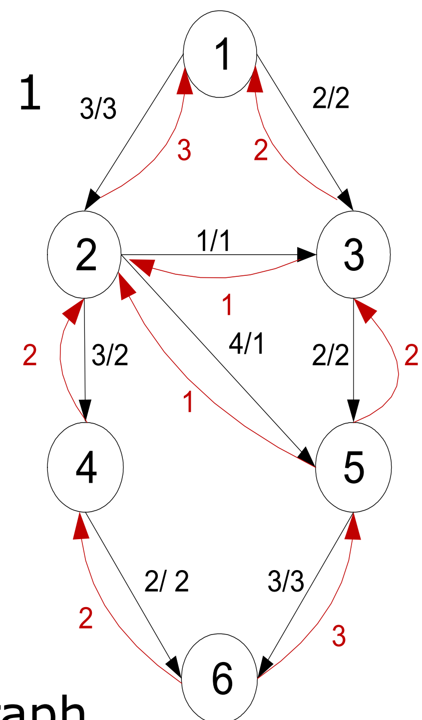
Cap(1, 1, 1, 0) $\quad$ Cap_{min}= 0

6th Iteration

Path: 1→3←2→5→6

Cap(1, 1, 4, 1) $\quad$ Cap_{min}= 1

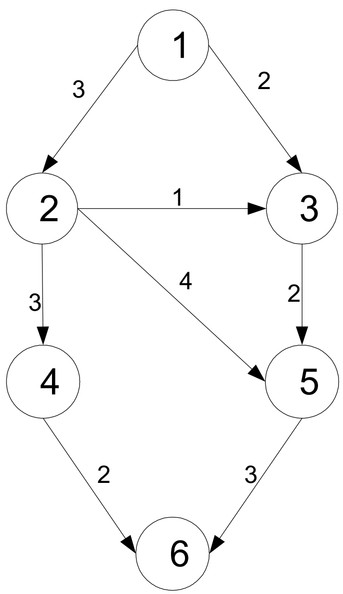
1 → 2, 3
2 → 3, 4, 5, 1
3 → 5, 2, 1
4 → 6, 2
5 → 6, 3
6 → 5, 4



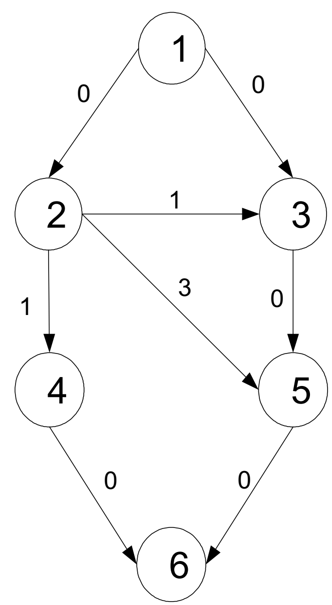
Residual Graph

Ford-Fulkerson Algorithm

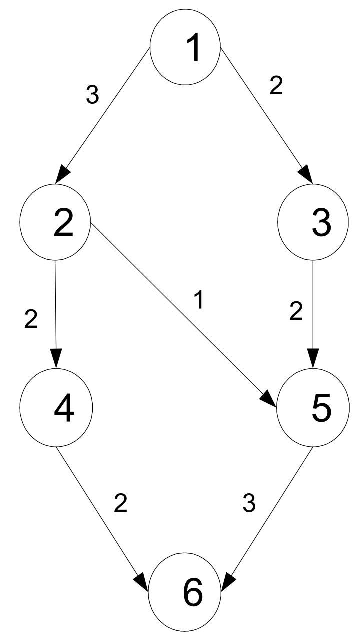
Capacity Graph



Residual Graph



Maximum Flow Graph



Maximum Flow: 5 unidades

Complexity Analysis Ford-Fulkerson Algorithm

- Assumption: capacities are integer-valued
- Finding a flow path takes O(VxE)
- We send at least 1 unit of flow through the path

  If the max-flow is f⋆, the time complexity is O((VxE)xf⋆)
  - "Bad" in that it depends on the output of the algorithm
  - Nonetheless, easy to code and works well in practice

Graphs can be used to solve too many other problems

- **Constraint Satisfaction**
  - course scheduling
- **Matching Problems**
  - match workers to their jobs
  - Stable marriage problem
- **Project Management**
- **Map Coloring**

Graphs: Algorithms Studied

- **Floyd-Warshall**
  - Transitive closure
  - Matrix with minimum distance between all vertices
- **Graph Traversals**
  - Breadth-First
  - Depth-First
- **Paths**
  - All paths between two vertices
  - Shortest Path for an unweighted/weighted graph, Dijkstra's algorithm
  - Shortest Path for a Graph with negative weights, Bellman-Ford algorithm
  - Constrained Shortest Path
- **Circuits**
  - Circuit Euler, Hierholzer algorithm
- **Topological Sort**
  - Two algorithms
- **Minimum Spanning Tree**
  - Kruskall algorithm
  - PRIM algorithm
- **Maximum Network Flow**
  - Ford-Fulkerson algorithm