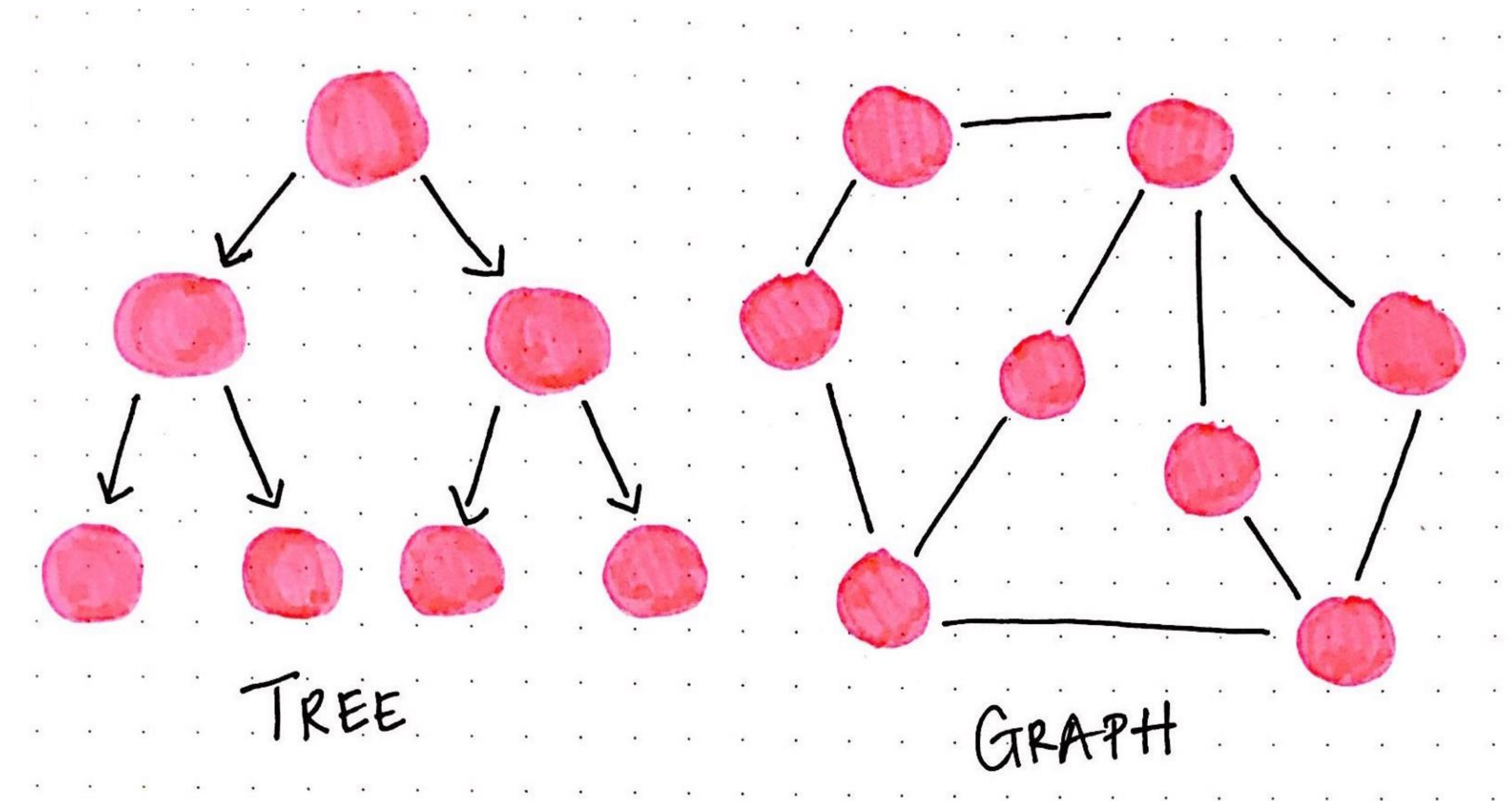
COMP1110/1140/6710
Week 11(B)

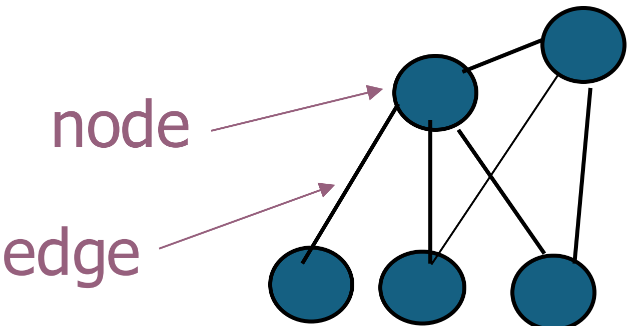
Graph ADT, Traversal and Graph Algorithms

Tree versus Graph

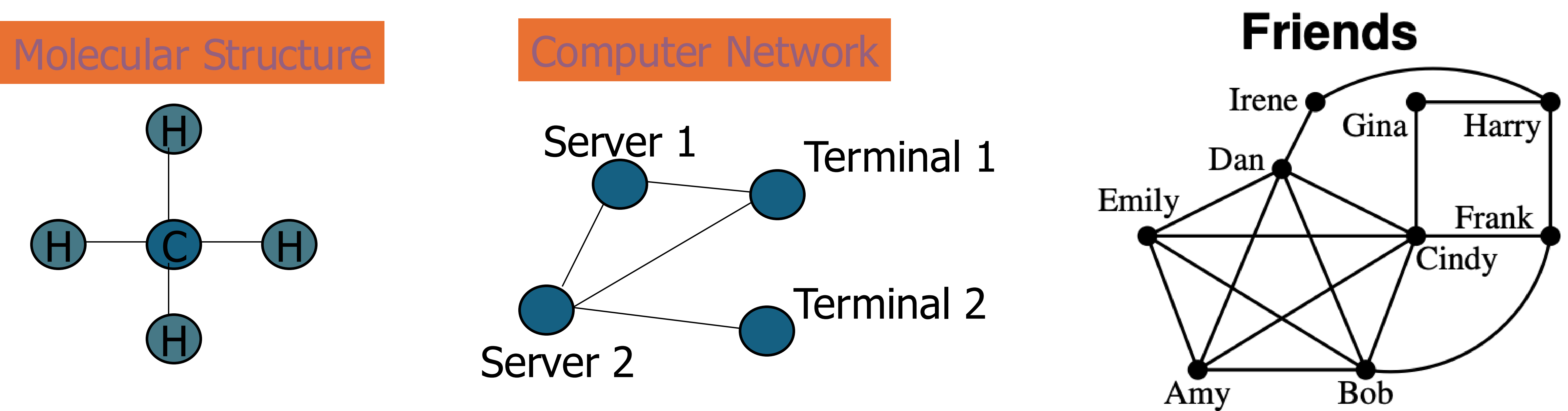


What is a graph?

- Graphs represent the relationships among data items
- A graph G consists of
  - a set V of nodes (vertices)
  - a set E of edges: each edge connects two nodes
- Each node represents an item
- Each edge represents the relationship between two items

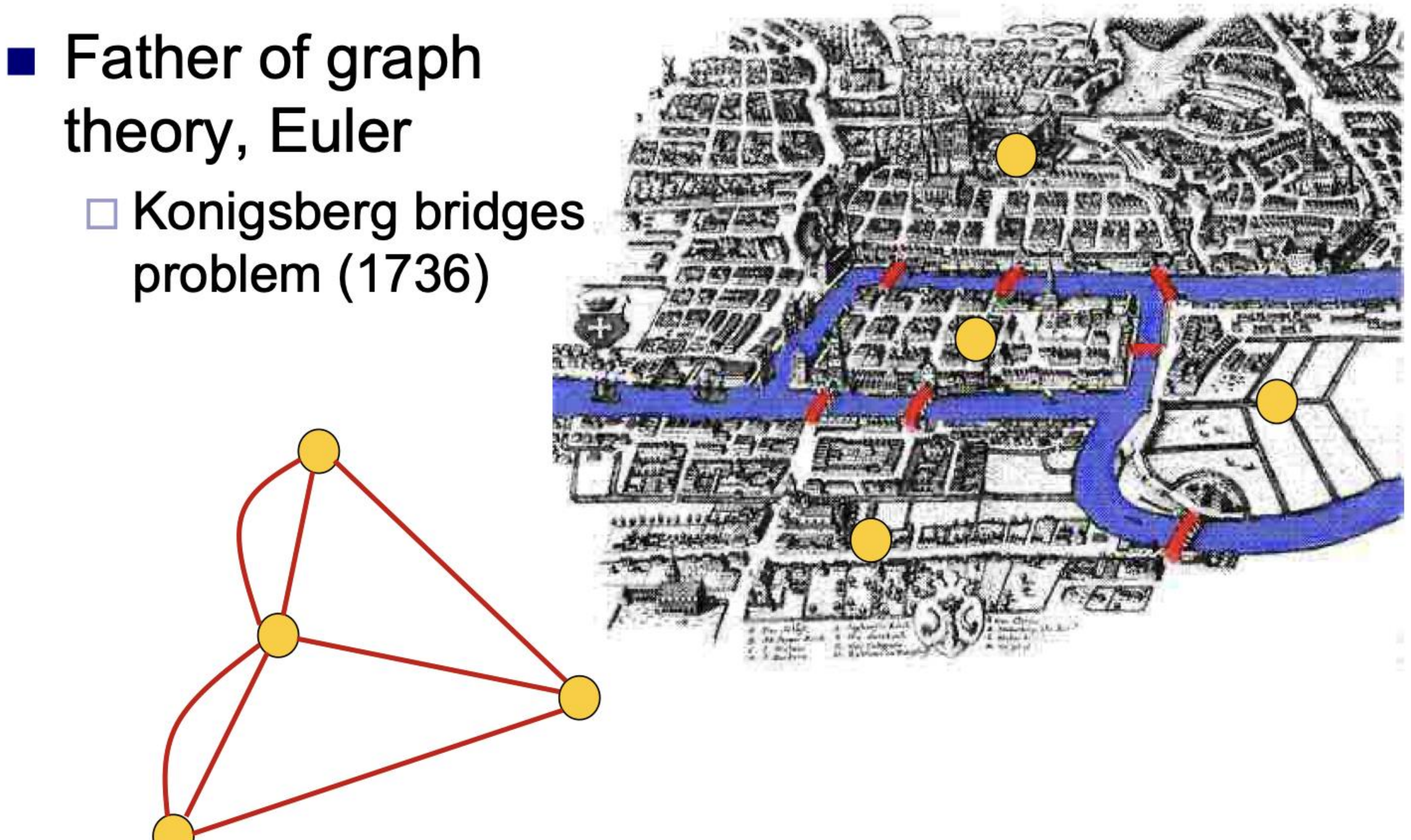


Examples of graphs



Other examples: electrical and communication networks, airline routes, flow chart, graphs for planning projects, social networks, …

A bit of history of graph theory

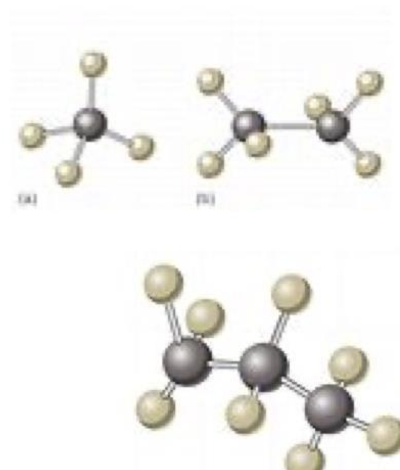


A bit of history of graph theory

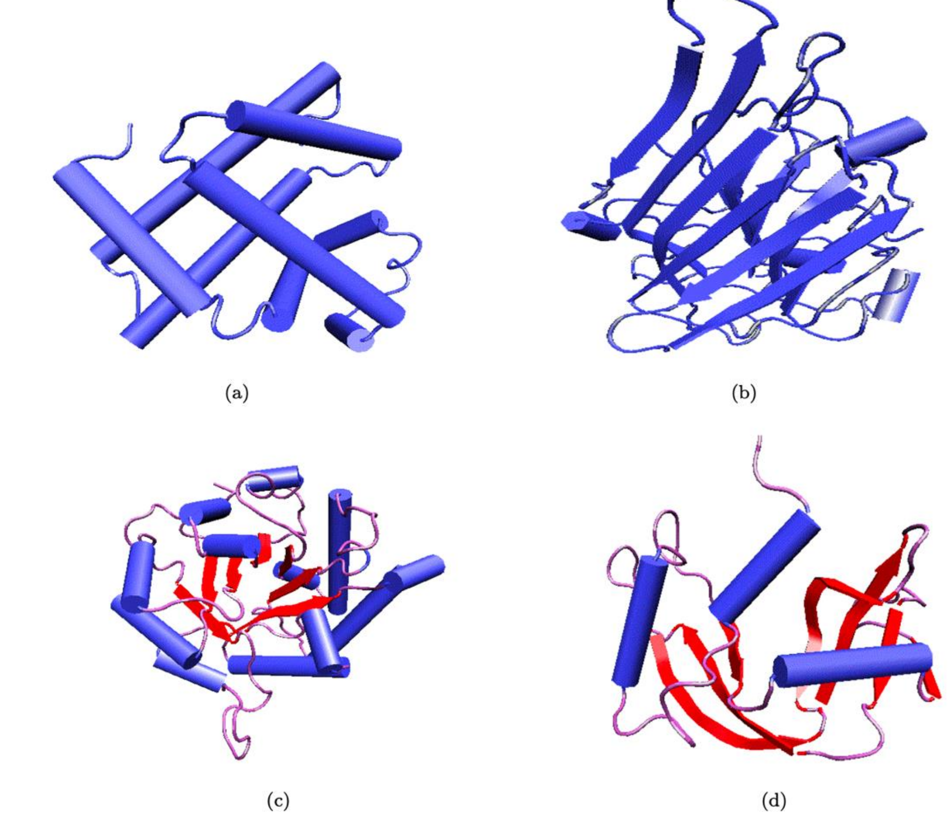
Kirchhoff and Cayley

- Kirchhoff developped the theory of trees in 1847 to solve the linear equations in branches and circuits of an electric network.

- In 1857, Cayley discovered the trees. Later he engaged in enumerating the isomers of saturated hyrocarbons with a given number of carbon atoms.



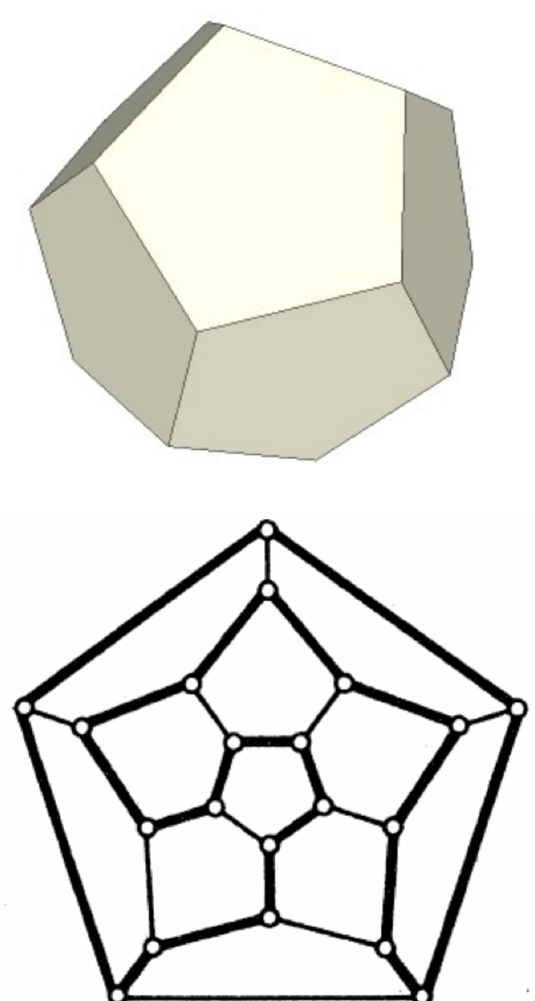
Nobel Prize for Chemistry 2024



- **Protein folding is essential a <u>graph matching</u> problem.**

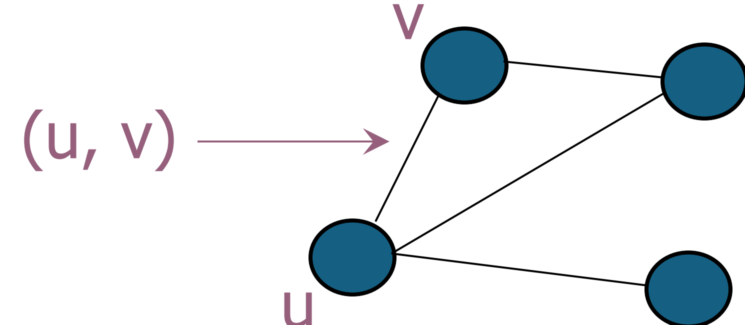
Hamilton cycle

- In 1859, Hamilton used a regular solid dodecahedron whose 20 corners are labeled with famous cities.
- The player is challenged to travel *"around the world"* by finding a closed circuit along the edges, passing through each city exactly once.



Formal Definition of graph

- The set of nodes is denoted as V
- For any nodes u and v, if u and v are connected by an edge, such edge is denoted as (u, v)



- The set of edges is denoted as E
- A graph G is defined as a pair: G= (V, E)

Terminology

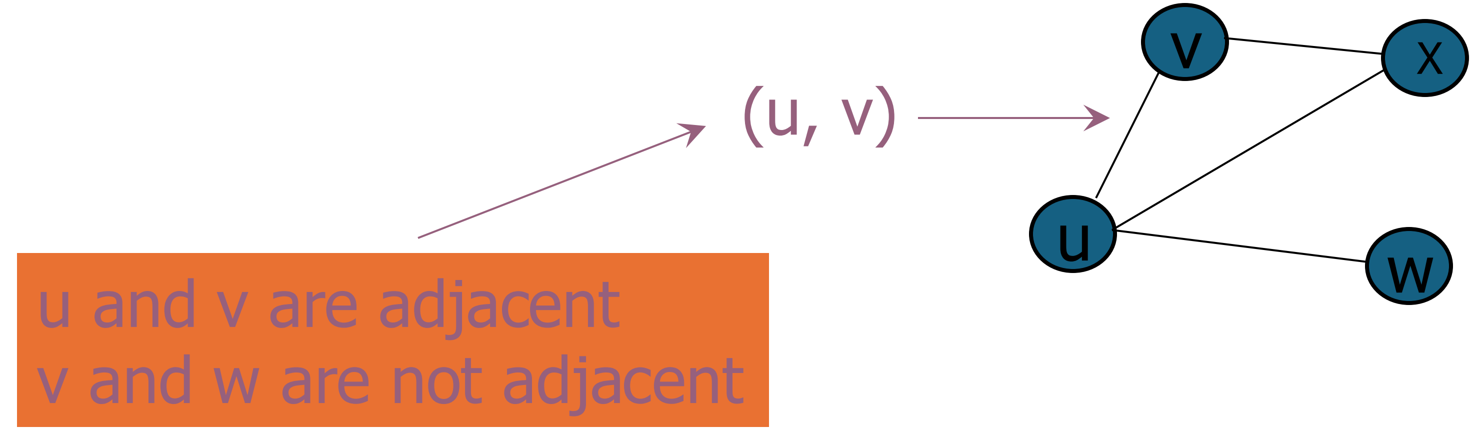
- G = {V, E}
- A graph G consists of two sets
  - A set V of vertices, or nodes
  - A set E of edges
- A subgraph
  - Consists of a subset of a graph’s vertices and a subset of its edges
- Adjacent vertices
  - Two vertices that are joined by an edge

Terminology

- A path between two vertices
  - A sequence of edges that begins at one vertex and ends at another vertex
  - May pass through the same vertex more than once
- A simple path
  - A path that passes through a vertex only once
- A cycle
  - A path that begins and ends at the same vertex
- A simple cycle
  - A cycle that does not pass through a vertex more than once

Adjacent

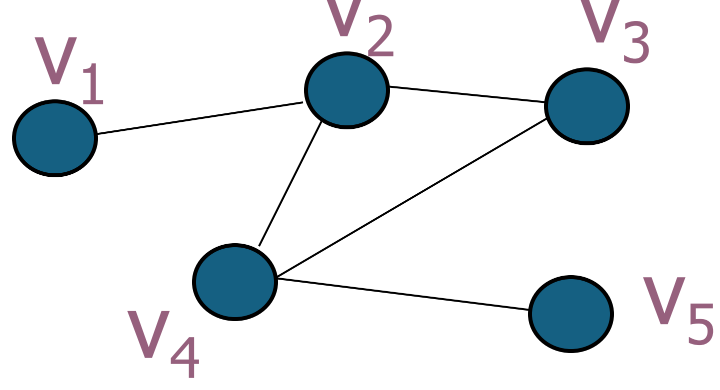
- Two nodes u and v are said to be adjacent if $(u, v) \in E$



Path and simple path

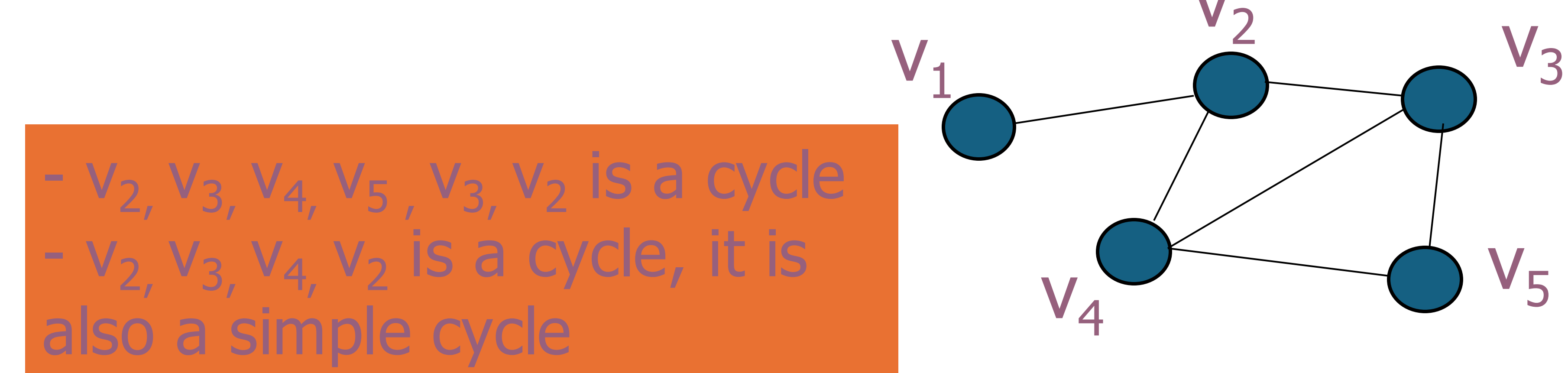
- A path from v_1 to v_k is a sequence of nodes $v_1, v_2, \ldots, v_k$ that are connected by edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$
- A path is called a simple path if every node appears at most once.

- v_2, v_3, v_4, v_2, v_1 is a path
- v_2, v_3, v_4, v_5 is a path, also it is a simple path



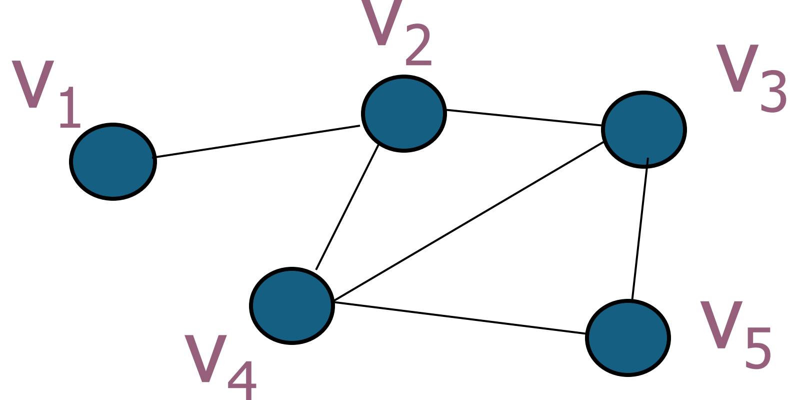
Cycle and simple cycle

- A cycle is a path that begins and ends at the same node
- A simple cycle is a cycle if every node appears at most once, except for the first and the last nodes



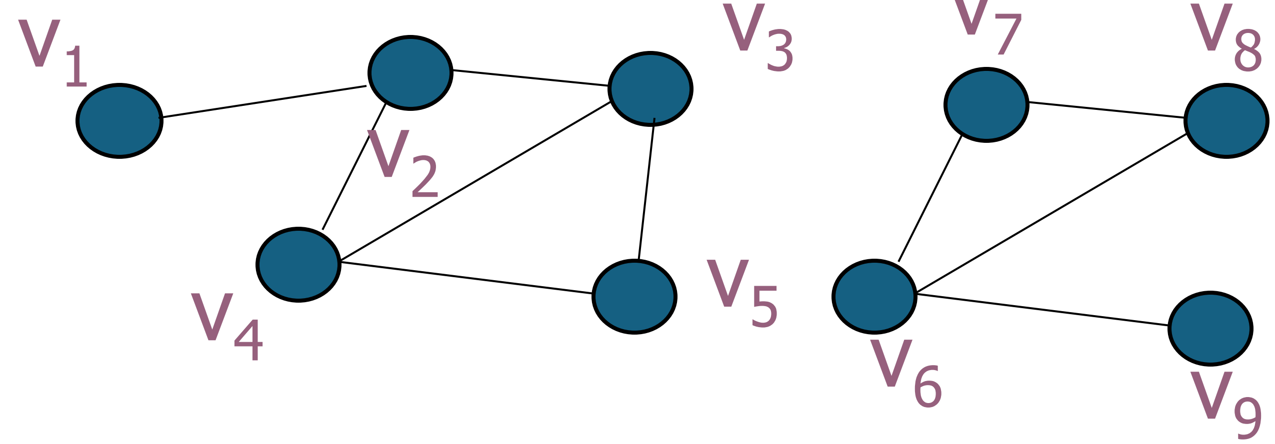
Connected graph

- A graph G is connected if there exists path between every pair of distinct nodes; otherwise, it is disconnected



This is a connected graph because there exists path between every pair of nodes

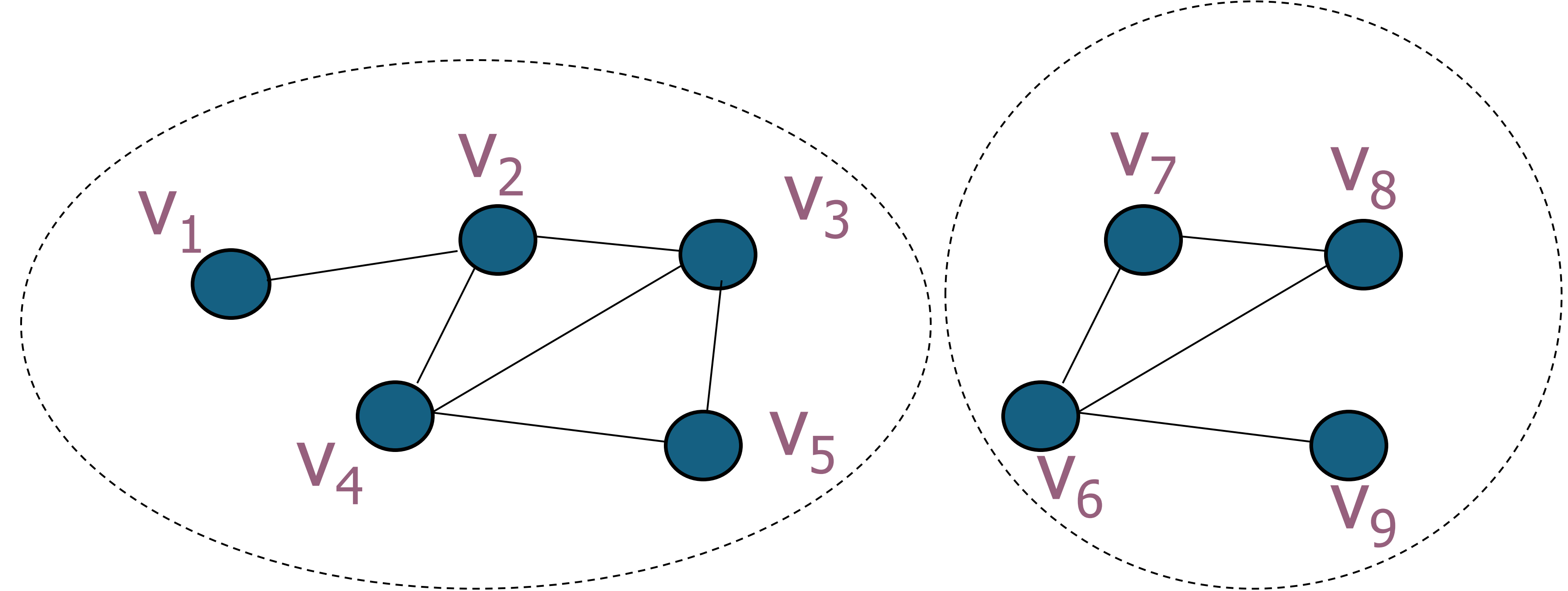
Example of disconnected graph



This is a disconnected graph because there does not exist path between some pair of nodes, says, v_1 and v_7

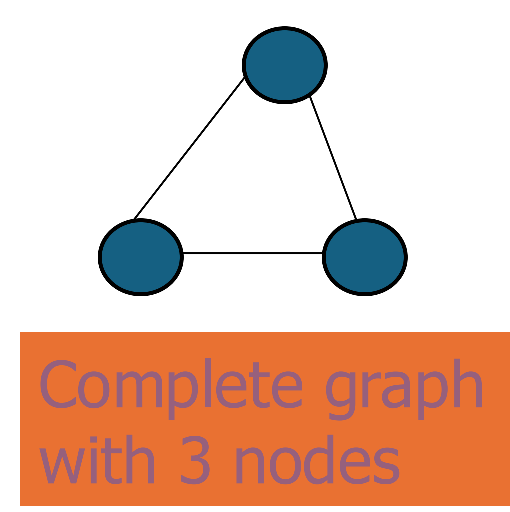
Connected component

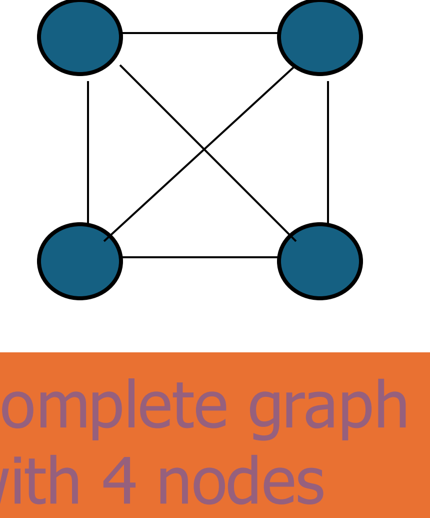
- If a graph is disconnect, it can be partitioned into a number of graphs such that each of them is connected. Each such graph is called a connected component.



Complete graph

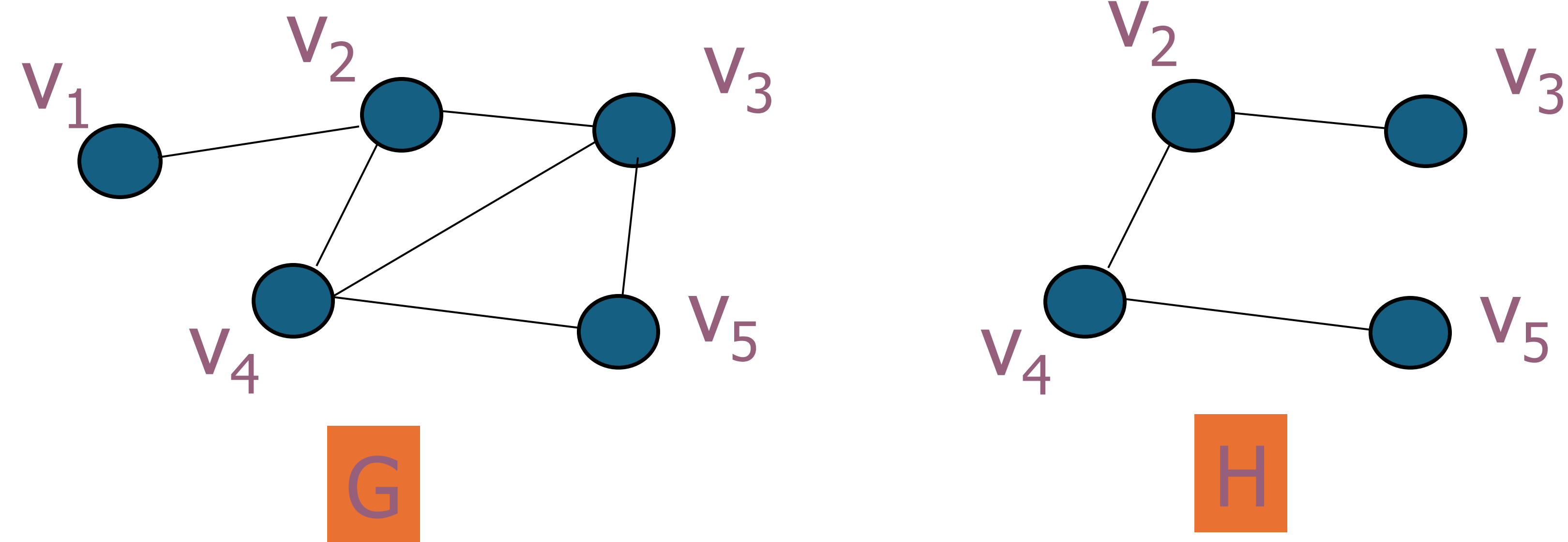
- A graph is complete if each pair of distinct nodes has an edge





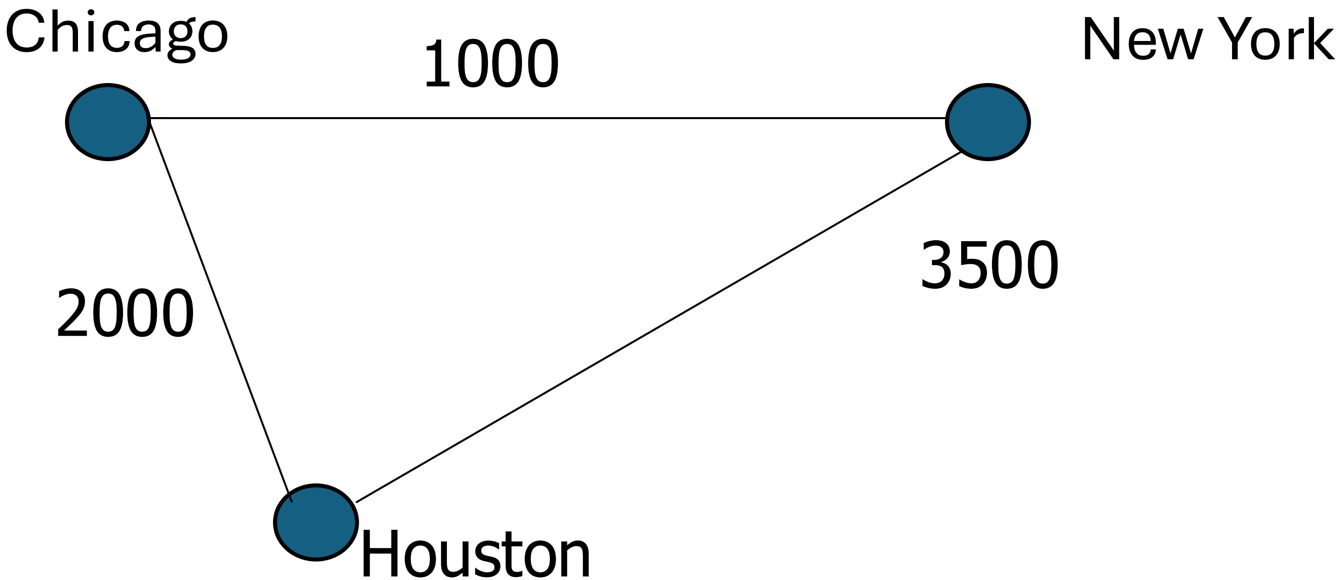
Subgraph

- A subgraph of a graph G =(V, E) is a graph H = (U, F) such that $U \subseteq V$ and $F \subseteq E$.



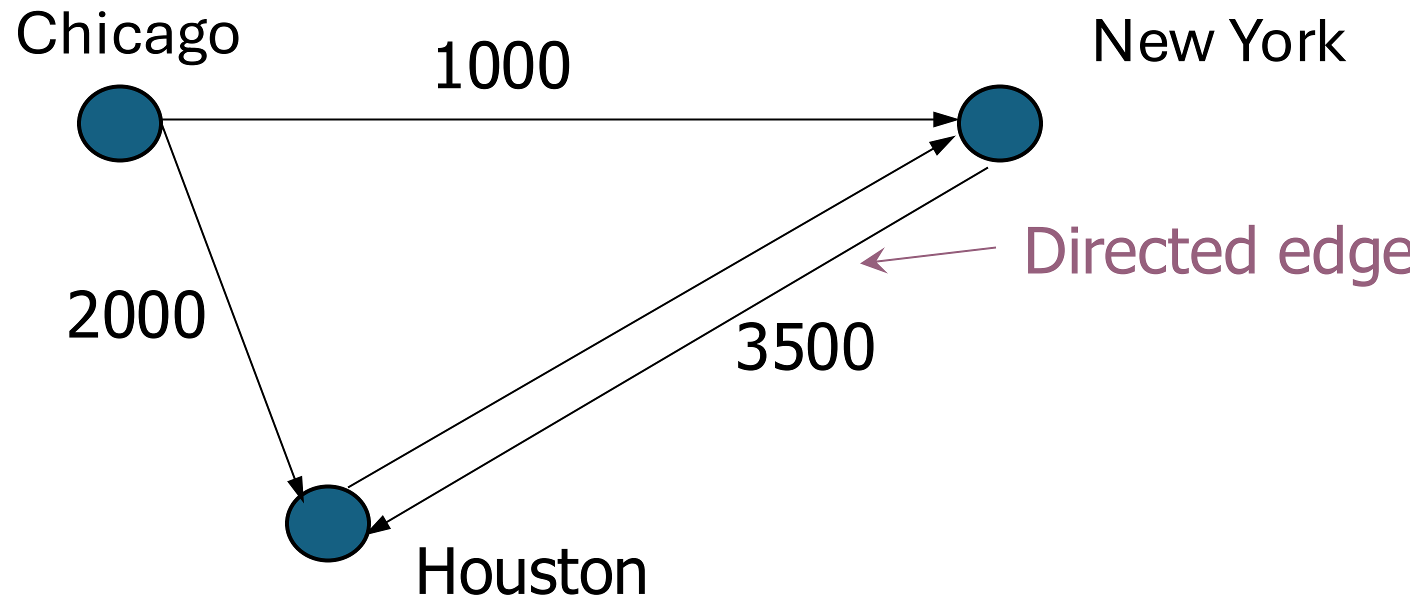
Weighted graph

- If each edge in G is assigned a weight, it is called a weighted graph, G=G(V,E,W).

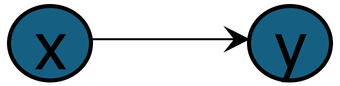


Directed graph (digraph)

- All previous graphs are undirected graph
- If each edge in E has a direction, it is called a directed edge
- A directed graph is a graph where every edges is a directed edge



More on directed graph



- If (x, y) is a directed edge, we say
  - y is adjacent to x
  - y is successor of x
  - x is predecessor of y

- In a directed graph, directed path, directed cycle can be defined similarly

Property of graph

- A undirected graph that is connected and has no cycle is a tree.
- A tree with n nodes must have exactly n-1 edges.
- A connected undirected graph with n nodes must have at least n-1 edges.

Graphs As ADTs

- Graphs form an important family of ADT.

- Two options for defining graphs
  - Vertices contain values
  - Vertices do not contain values

- Operations of the ADT graph
  - Create an empty graph
  - Determine whether a graph is empty
  - Determine the number of vertices in a graph
  - Determine the number of edges in a graph

Graphs As ADTs

- Operations of the ADT graph (Continued)
  - Determine whether an edge exists between two given vertices; for weighted graphs, return weight value
  - Insert a vertex in a graph whose vertices have distinct search keys that differ from the new vertex’s search key
  - Insert an edge between two given vertices in a graph
  - Delete a particular vertex from a graph and any edges between the vertex and other vertices
  - Delete the edge between two given vertices in a graph
  - Retrieve from a graph the vertex that contains a given search key

Implementing Graphs

- Most common implementations of a graph
  - Adjacency matrix
  - Adjacency linked list

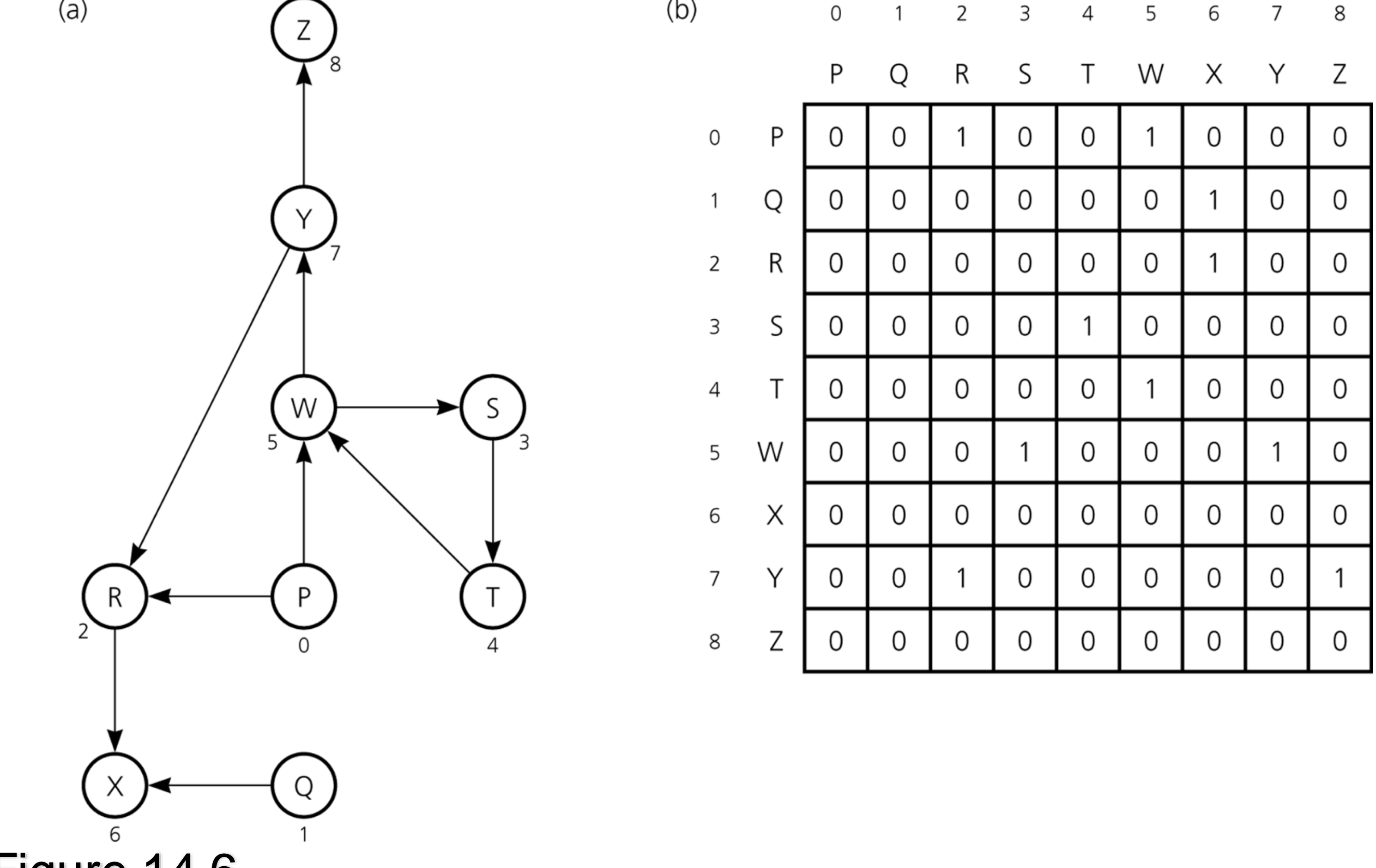
- Adjacency matrix
  - Adjacency matrix for a graph with n vertices numbered 0, 1, ..., n – 1
    - An n by n array matrix such that matrix[i][j] is
      - 1 (or true) if there is an edge from vertex i to vertex j
      - 0 (or false) if there is no edge from vertex i to vertex j

Implementing Graph

- Adjacency matrix
  - Represent a graph using a two-dimensional array

- Adjacency linked list
  - Represent a graph using n linked lists where n is the number of vertices

Adjacent Matrix



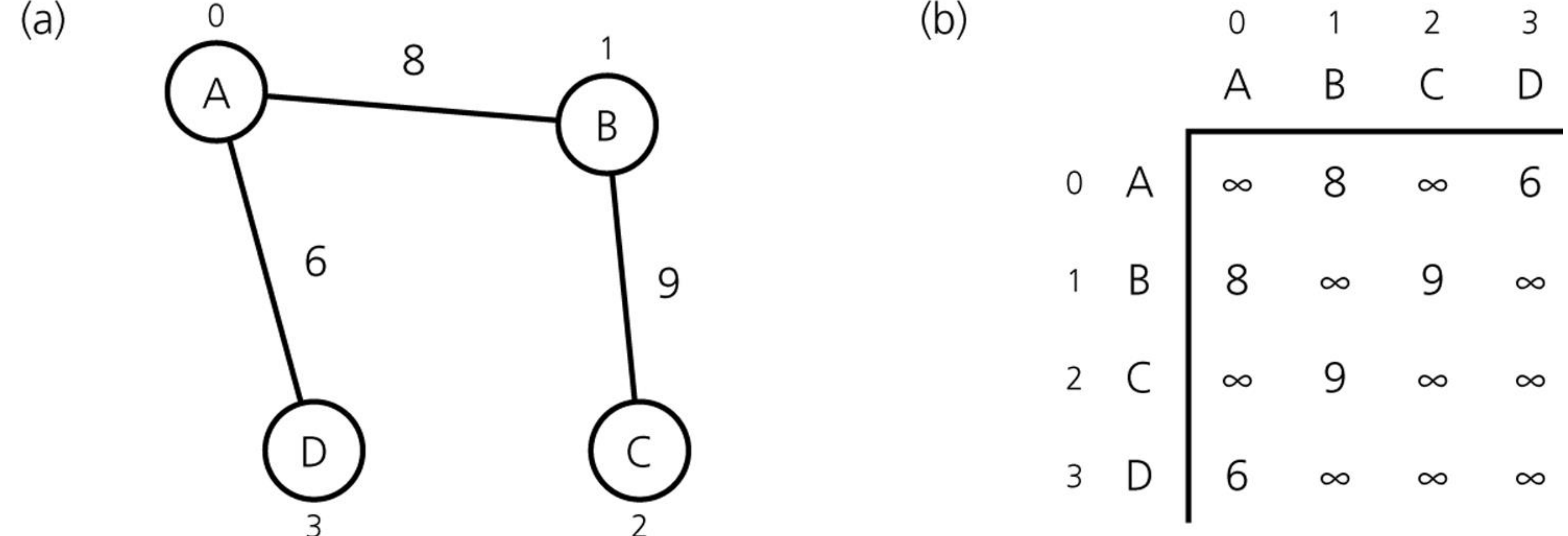
		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

Figure 14.6

a) A directed graph and b) its adjacency matrix

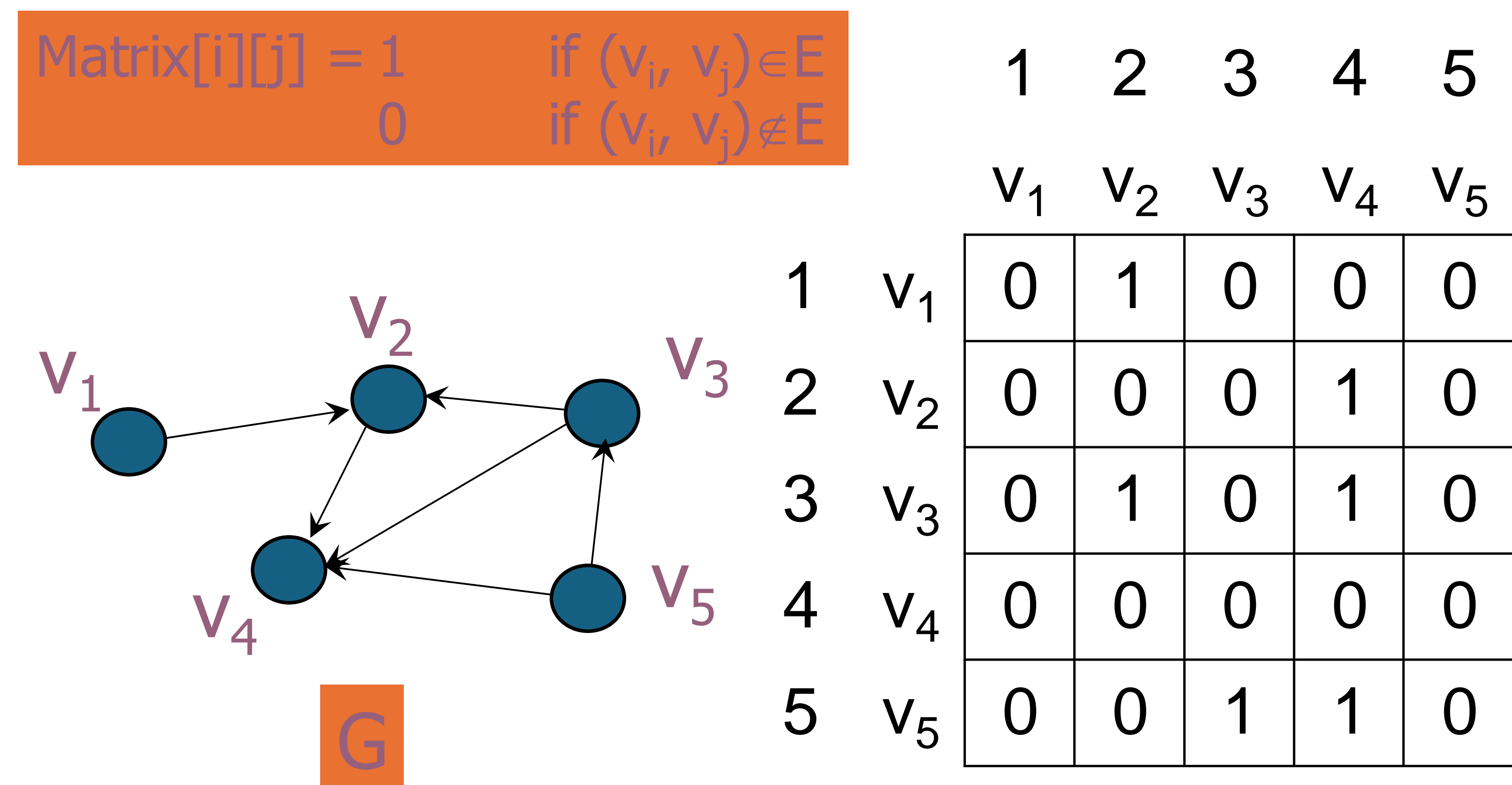
Adjacency matrix

- Adjacency matrix for a weighted graph with n vertices numbered 0, 1, …, n – 1
  - An n by n array matrix such that matrix[i][j] is
    - The weight that labels the edge from vertex i to vertex j if there is an edge from i to j
    - ∞ if there is no edge from vertex i to vertex j



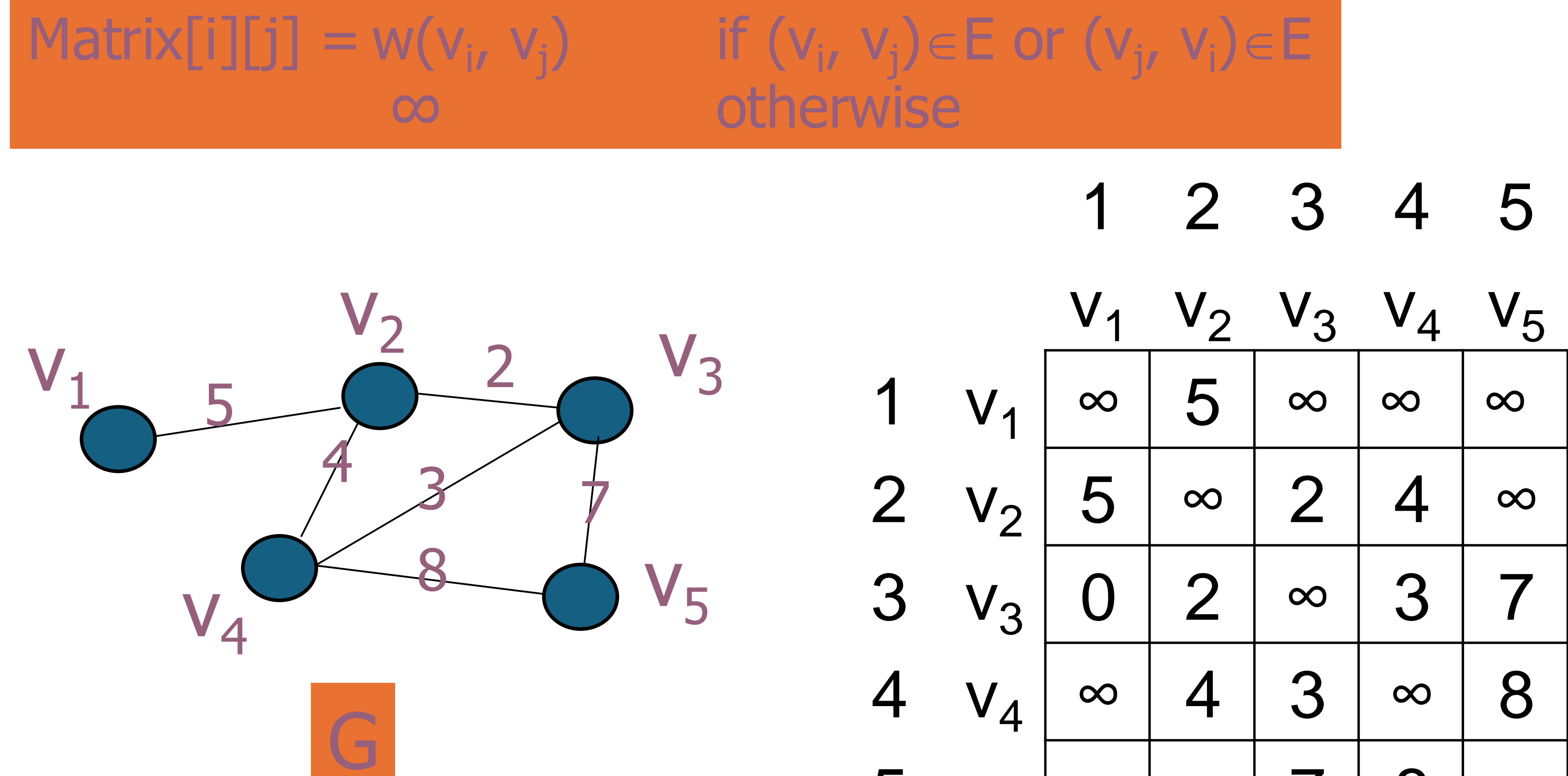
		0 A	1 B	2 C	3 D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

Adjacency matrix for directed graph



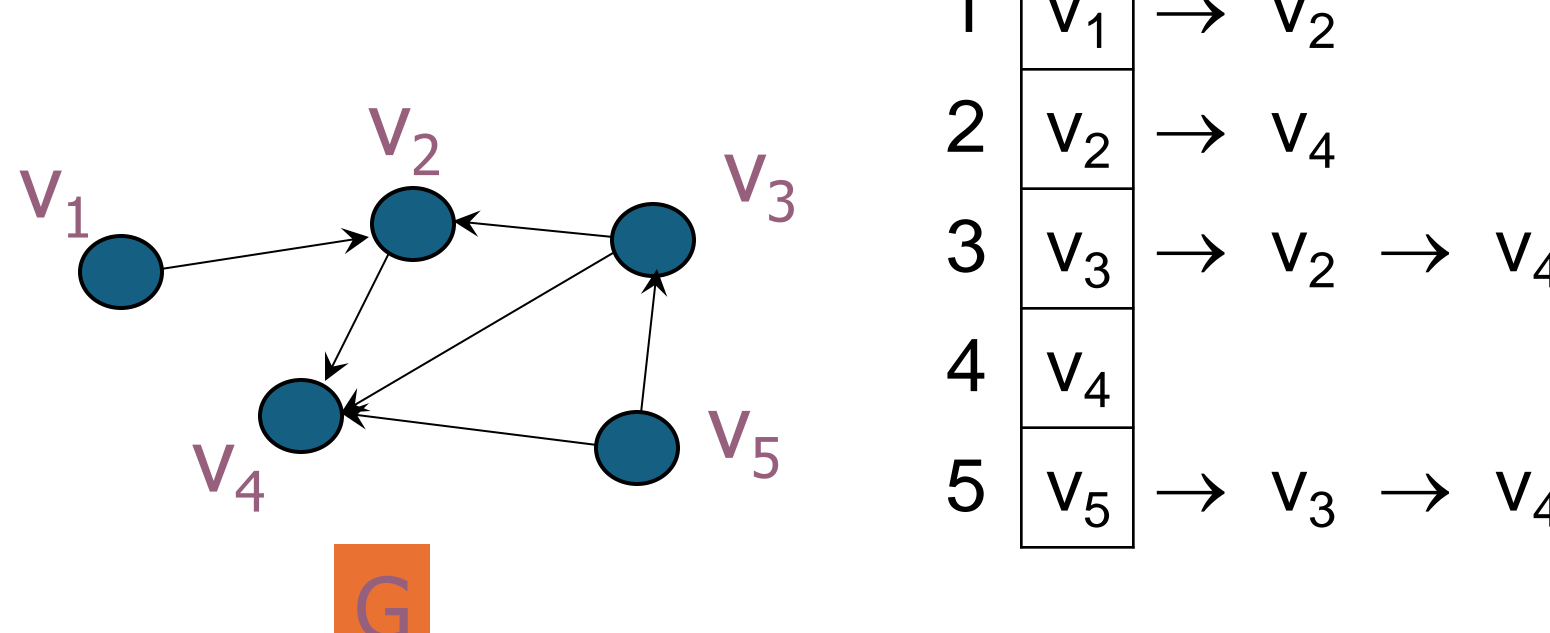
		1 v_1	2 v_2	3 v_3	4 v_4	5 v_5
1	v_1	0	1	0	0	0
2	v_2	0	0	0	1	0
3	v_3	0	1	0	1	0
4	v_4	0	0	0	0	0
5	v_5	0	0	1	1	0

Adjacency matrix for weighted undirected graph

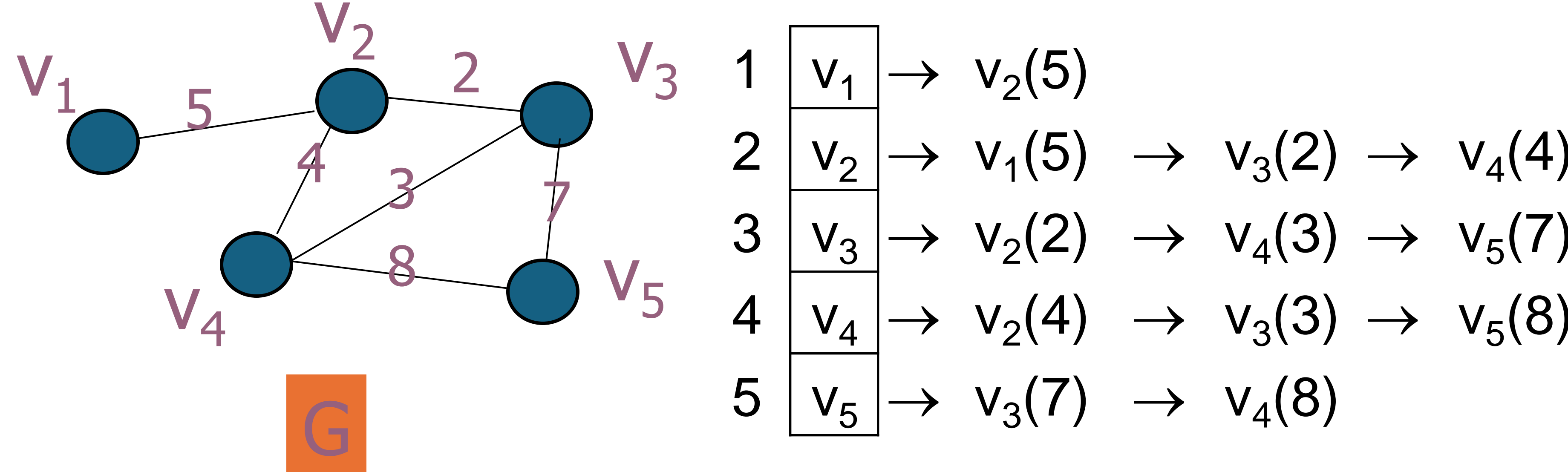


		1	2	3	4	5
		v_1	v_2	v_3	v_4	v_5
1	v_1	∞	5	∞	∞	∞
2	v_2	5	∞	2	4	∞
3	v_3	0	2	∞	3	7
4	v_4	∞	4	3	∞	8
5	v_5	∞	∞	7	8	∞

Adjacency linked list for directed graph



Adjacency list for <u>weighted</u> undirected graph



Adjacency list

An adjacency list for a graph with n vertices numbered 0, 1, …, n – 1

- Consists of n linked lists
- The ith linked list has a node for vertex j if and only if the graph contains an edge from vertex i to vertex j
    - This node can contain either
        - Vertex j's value, if any
        - An indication of vertex j's identity

Adjacency list

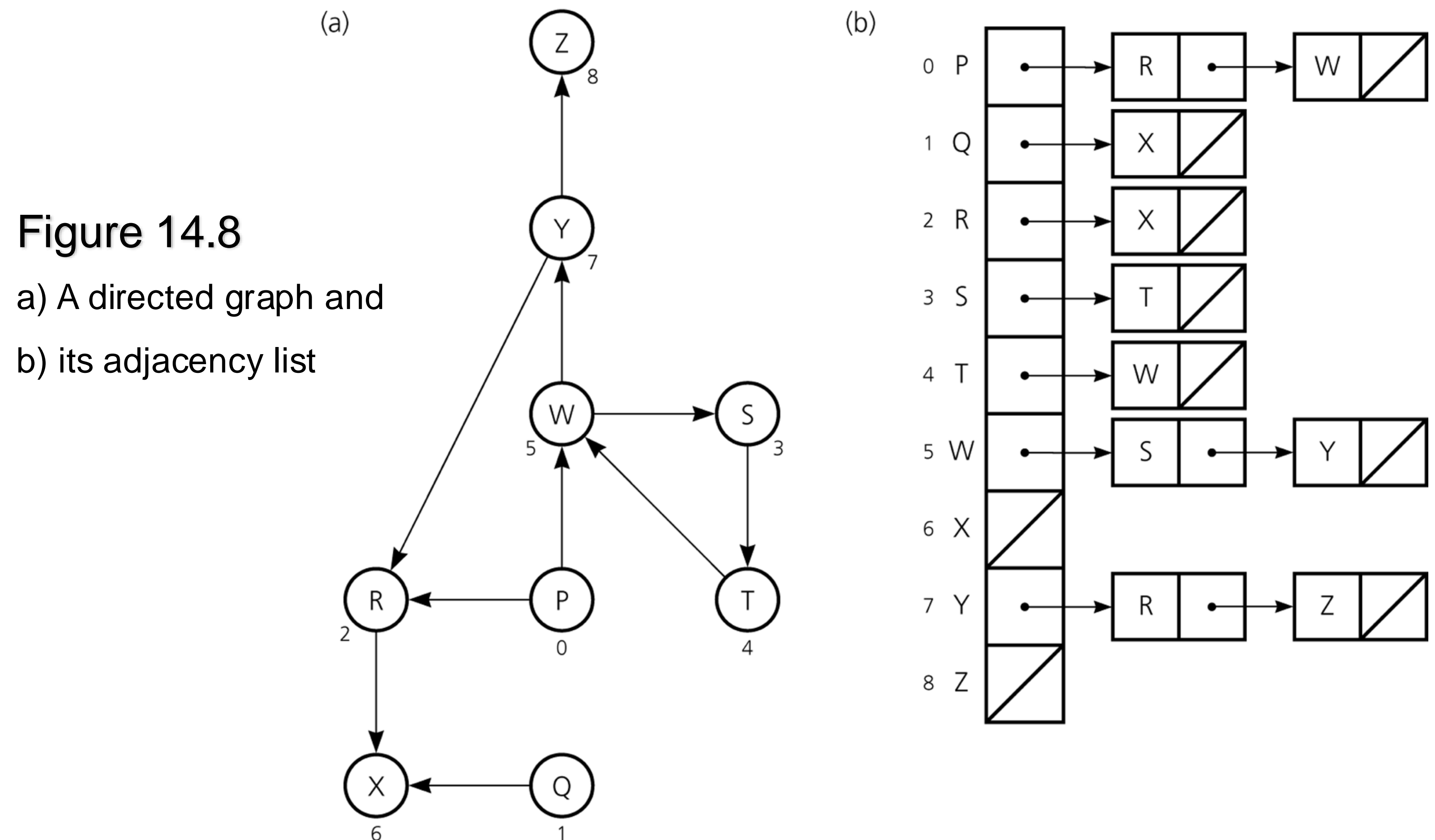


Figure 14.8

a) A directed graph and

b) its adjacency list

Adjacency list

- Adjacency list for an undirected graph
  - Treats each edge as if it were two directed edges in opposite directions

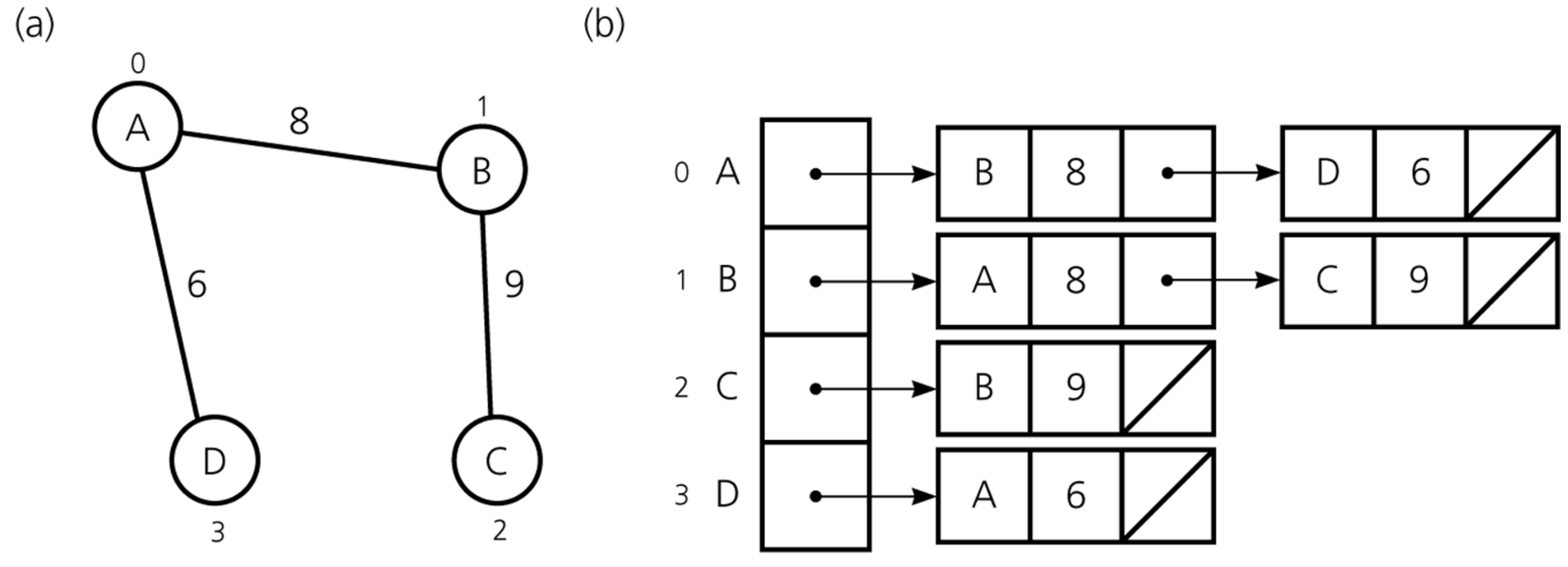
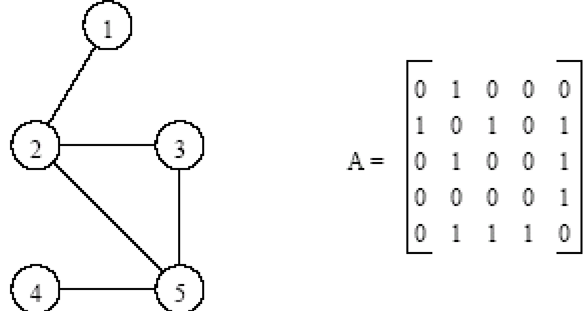


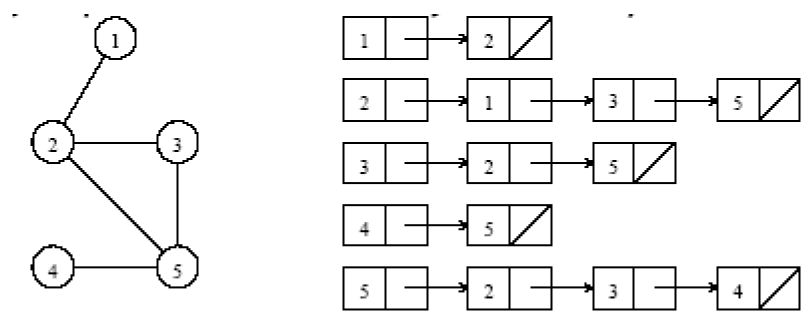
Figure 14.9

a) A weighted undirected graph and b) its adjacency list

Recap:
Two implementations (representations)



An undirected graph and its adjacency matrix representation.



An undirected graph and its adjacency list representation.

Comparison: Adjacency matrix Versus Adjacency list

- Two common operations on graphs
  1. Determine whether there is an edge from vertex i to vertex j
  2. Find all vertices adjacent to a given vertex i

- Adjacency matrix
  - Supports operation 1 more efficiently
- Adjacency list
  - Supports operation 2 more efficiently
  - Often requires less space than an adjacency matrix

Pros and Cons of the two implementations

- Adjacency matrix
  - Allows us to determine whether there is an edge from node i to node j in O(1) time;

- Adjacency list
  - Allows us to find all nodes adjacent to a given node j efficiently
  - If the graph is sparse, adjacency list requires less space

Graph problems / Graph algorithms

Problems related to Graph

- Graph Traversal
- Topological Sort
- Minimum Spanning Tree
- Shortest Path
- All-pair shortest path
- Euler path
- Hamilton circuit
- Travelling Salesman Problem
- The four colour problem.
- Graph Cut problem
- Max flow and Min Cut
- …

Graph Traversal

Graph Traversal Algorithm

- To traverse a tree, we use tree traversal algorithms like pre-order, in-order, and post-order to visit all the nodes in a tree
- Similarly, graph traversal algorithm tries to visit all the nodes it can reach.
- If a graph is disconnected, a graph traversal that begins at a node v will visit only a subset of nodes, that is, the connected component containing v.

Two basic traversal algorithms

- Depth-first-search (DFS)
  - After visit node v, DFS strategy proceeds along a path from v as deeply into the graph as possible before backing up
- Breadth-first-search (BFS)
  - After visit node v, BFS strategy visits every node adjacent to v before visiting any other nodes

Graph Traversals

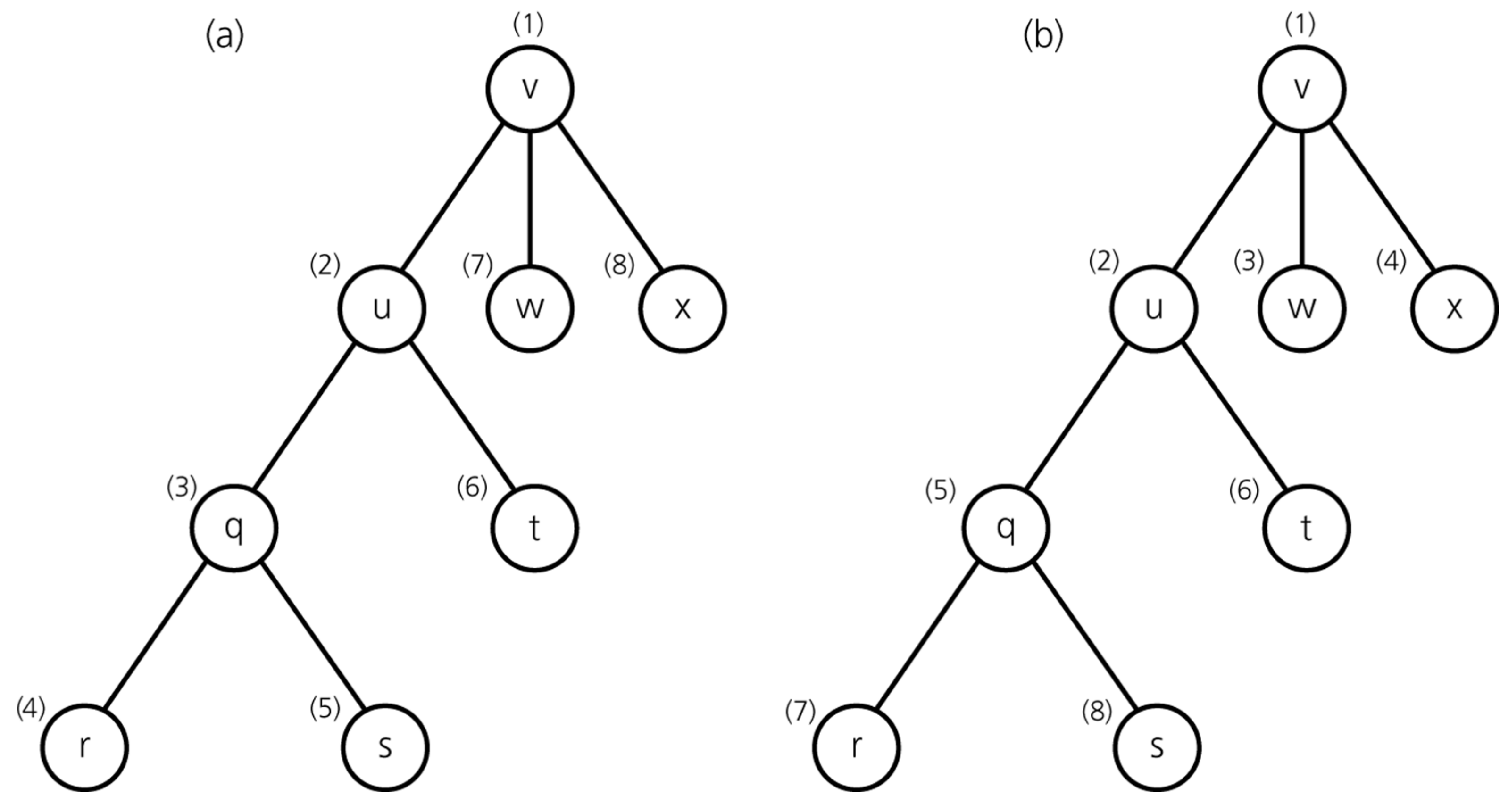


Figure 14.10

Visitation order for a) a depth-first search; b) a breadth-first search

Depth-First Search

- Depth-first search (DFS) traversal
  - Goes as deeply into the graph as possible from a vertex before *backtracking*
  - DFS strategy looks similar to pre-order. From a given node v, it first visits itself. Then, recursively visit its unvisited neighbors one by one.
  - A recursive implementation is simple
  - An iterative implementation uses a stack

Recursive Depth-First Search

- Recursive DFS traversal can be defined as follows:

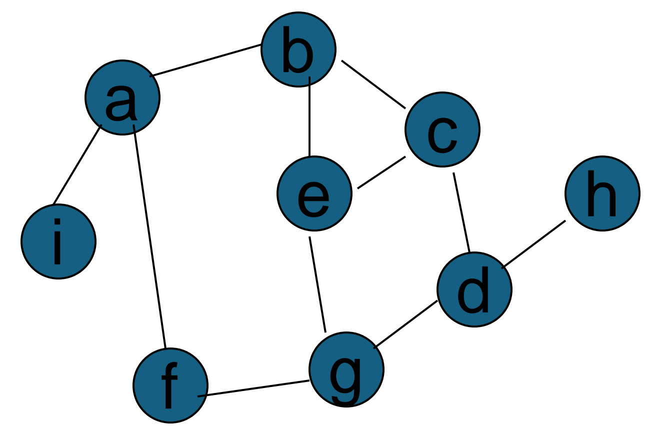
```
dfs(v)
  print(v);
  mark v as visited
  for (each unvisited vertex u adjacent to v)
    dfs(u)
```

Iterative Depth-First Search

- Iterative DFS traversal

```
dfs(v)
   s.createStack()
   s.push(v)
   mark v as visited
   while (!s.isEmpty()) {
      if (no unvisited vertices adjacent to the vertex on the stack
   top)
               s.pop() // backtrack
      } else {
               select an unvisited vertex u adjacent to the vertex on
   the
                  stack top
               s.push(u)
               mark u as visited
      }
   }
```

Iterative Depth-First Search



Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e

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

Breadth-First Search

- Breadth-first search (BFS) traversal
  - Visits every vertex adjacent to a vertex v that it can before visiting any other vertex
  - A first visited, first explored strategy

  - An iterative implementation uses a queue
  - A recursive implementation is possible, but not simple

Breadth-first search (BFS)

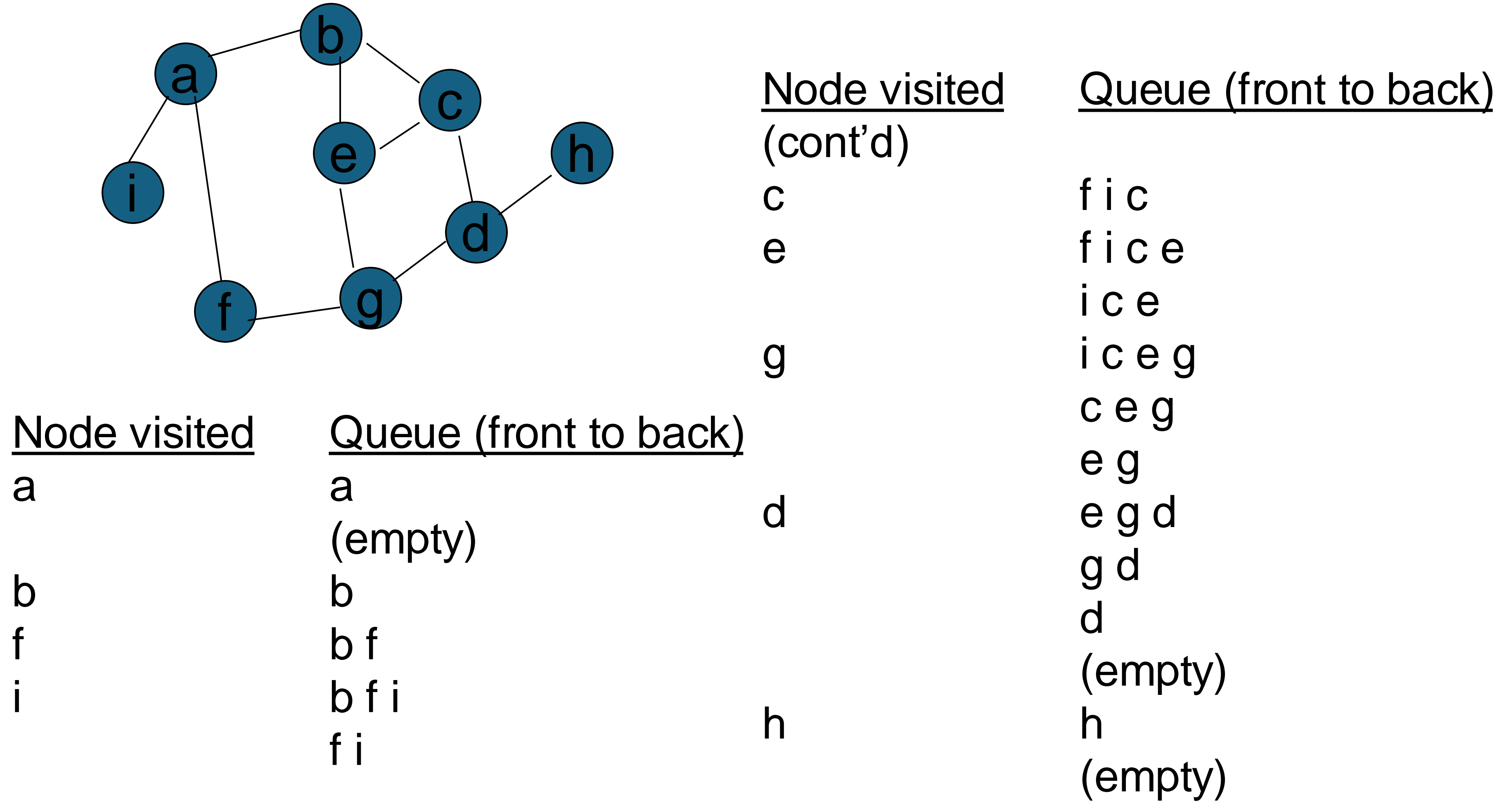
- BFS strategy looks similar to level-order. From a given node v, it first visits itself. Then, it visits every node adjacent to v before visiting any other nodes.
  - 1. Visit v
  - 2. Visit all v's neigbours
  - 3. Visit all v's neighbours' neighbours
  - …
- Similar to level-order, BFS is based on a queue.

Iterative Breadth-First Search

- Iterative BFS traversal

```
bfs(v)
  q.createQueue()
  q.enqueue(v)
  mark v as visited
  while (!q.isEmpty()) {
    w = q.dequeue()
    for (each unvisited vertex u adjacent to w) {
            mark u as visited
            q.enqueue(u)
    }
  }
```

Iterative Breadth-First Search



Node visited	Queue (front to back)
a	a
	(empty)
b	b
f	b f
i	b f i
	f i

Node visited (cont'd)	Queue (front to back)
c	f i c
e	f i c e
	i c e
g	i c e g
	c e g
	e g
d	e g d
	g d
	d
	(empty)
h	h
	(empty)

BFS example

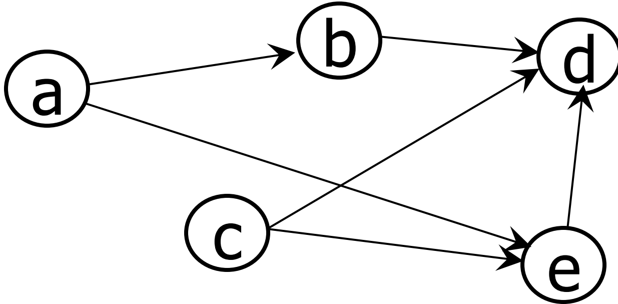
- Start from v_5

Visit	Queue (front to back)
v_5	v_5
	empty
v_3	v_3
v_4	v_3, v_4
	v_4
v_2	v_4, v_2
	v_2
	empty
v_1	v_1
	empty

Topological order and topological sorting

Topological order

- Consider the prerequisite structure for courses:



- Each node x represents a course x
- (x, y) represents that course x is a prerequisite to course y
- Note that this graph should be a directed graph without cycles (called a directed acyclic graph).
- A linear order to take all 5 courses while satisfying all prerequisites is called a topological order.
- E.g.
  - a, c, b, e, d
  - c, a, b, e, d

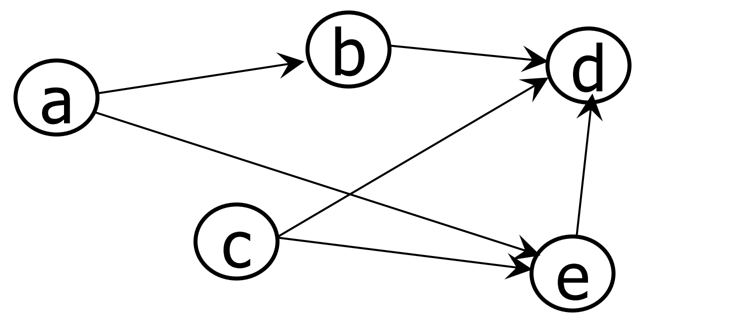
Topological sort

- Arranging all nodes in the graph in a topological order

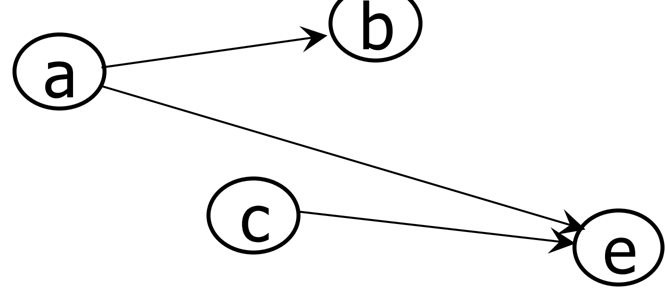
Algorithm TopoSort

```
n = |V|;
for i = 1 to n {
   select a node v that has no successor;
   aList.add(1, v);
   delete node v and its edges from the graph;
}
return aList;
```

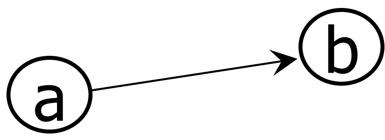
Example: topo-sort



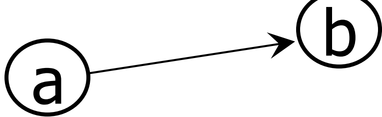
1. d has no successor! Choose d!



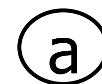
2. Both b and e have no successor! Choose e!



3. Both b and c have no successor! Choose c!



4. Only b has no successor! Choose b!



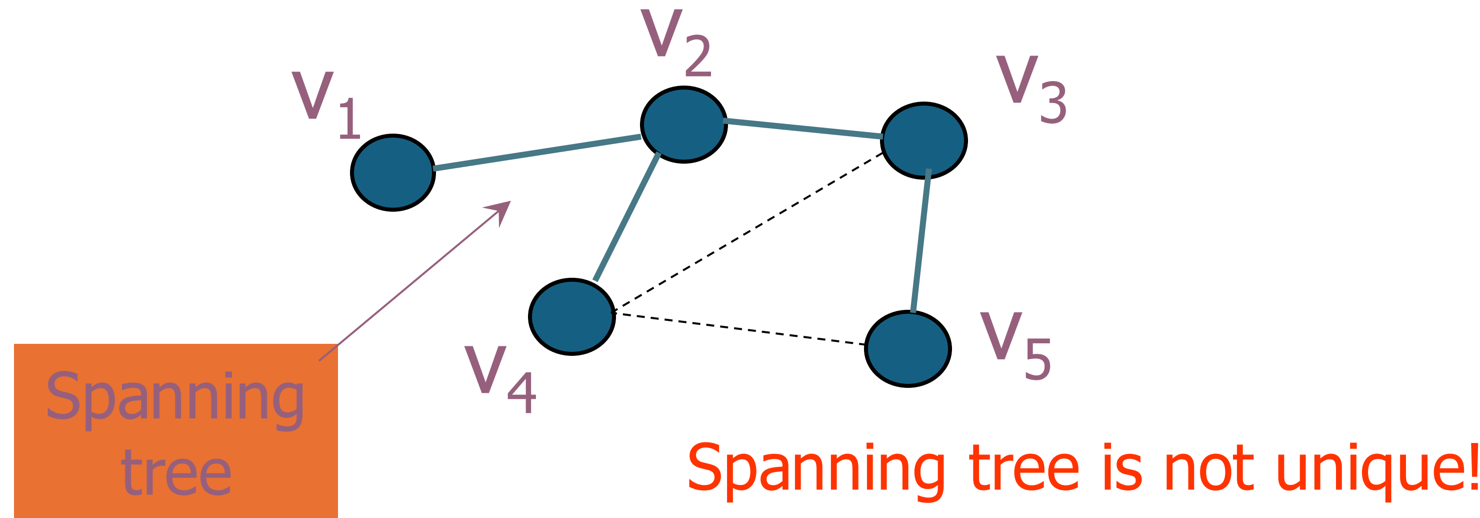
5. Choose a!

The topological order is **a,b,c,e,d**

Spanning tree and MST

Spanning Tree

- Given a connected undirected graph G, a spanning tree of G is a subgraph of G that contains all of G's nodes and enough of its edges to form a tree.



DFS spanning tree

- Generate the spanning tree edge during the DFS traversal.

```
Algorithm dfsSpanningTree(v)
mark v as visited;
for (each unvisited node u adjacent to v) {
  mark the edge from u to v;
  dfsSpanningTree(u);
}
```

- Similar to DFS, the spanning tree edges can be generated based on BFS traversal.

Minimum (cost) Spanning Tree

- Consider a connected undirected graph where
  - Each node x represents a city x
  - Each edge (x, y) has a number which measures the cost of placing telephone line between city x and city y
- Problem: connecting all cities while minimizing the total cost
- Solution: find a spanning tree with minimum total weight, that is, minimum spanning tree

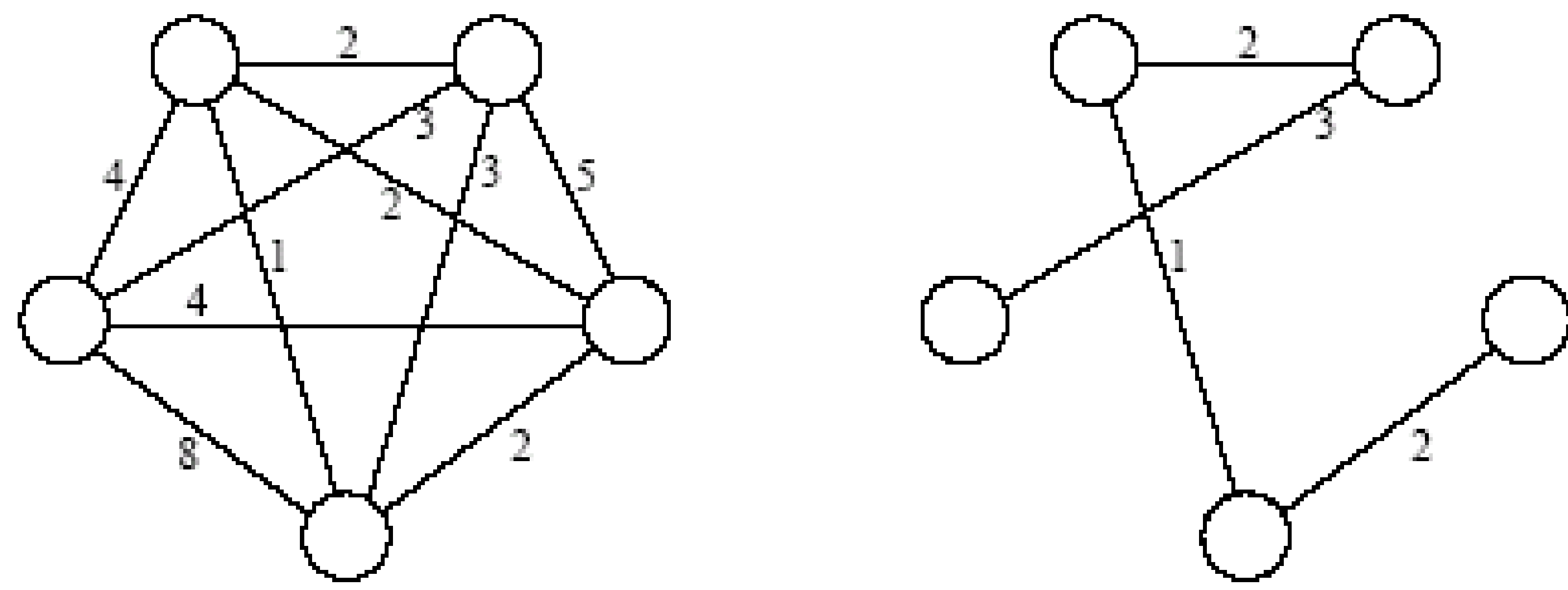
Minimum Spanning Tree

- A *spanning tree* of an undirected graph G is a subgraph of G that is a tree containing all the vertices of G.

- In a weighted graph, the weight of a subgraph is the sum of the weights of the edges in the subgraph.

- A *minimum spanning tree* (MST) for a weighted undirected graph is a spanning tree with minimum weight.

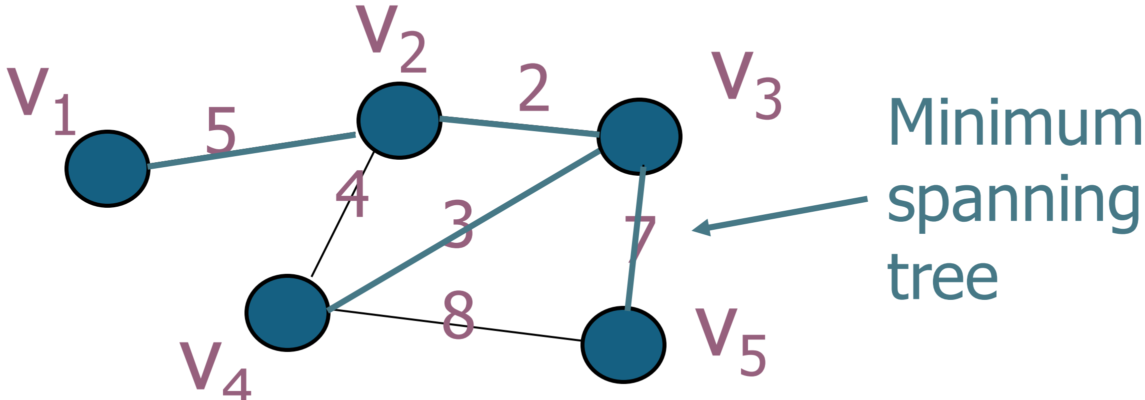
Minimum Spanning Tree



An undirected graph and its minimum spanning tree.

Formal definition of minimum spanning tree

- Given a connected undirected graph G.
- Let T be a spanning tree of G.
- cost(T) = $\sum_{e \in T}$weight(e)
- The minimum spanning tree is a spanning tree T which minimizes cost(T)



Prim's Algorithm for MST

- Prim's algorithm for finding an MST is a greedy algorithm.
- Start by selecting an arbitrary vertex, include it into the current MST.
- Grow the current MST by inserting into it the vertex closest to one of the vertices already in current MST.

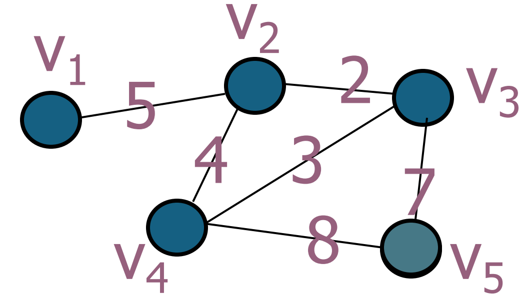
Prim’s algorithm (II)

Algorithm PrimAlgorithm(v)

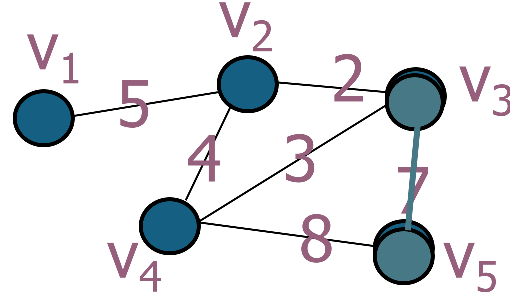
- Mark node v as visited and include it in the minimum spanning tree;
- while (there are unvisited nodes) {
  - find the minimum edge (v, u) between a visited node v and an unvisited node u;
  - mark u as visited;
  - add both v and (v, u) to the minimum spanning tree;

  }

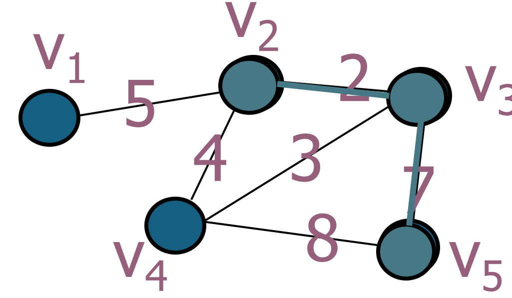
Prim's algorithm (I)



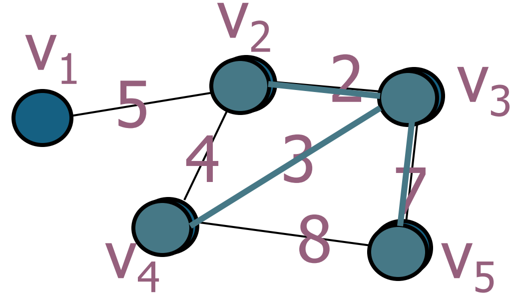
Start from v_5, find the minimum edge attach to v_5



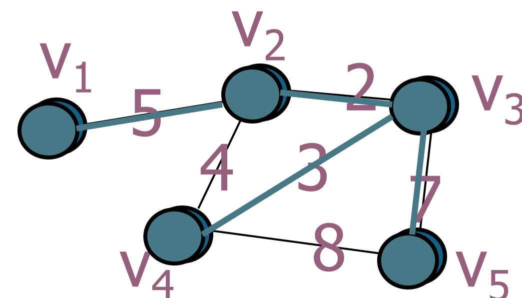
Find the minimum edge attach to v_3 and v_5



Find the minimum edge attach to v_2, v_3 and v_5



Find the minimum edge attach to v_2, v_3 , v_4 and v_5



Shortest path problem

Shortest path

- Consider a weighted directed graph
  - Each node x represents a city x
  - Each edge (x, y) has a number which represent the cost of traveling from city x to city y

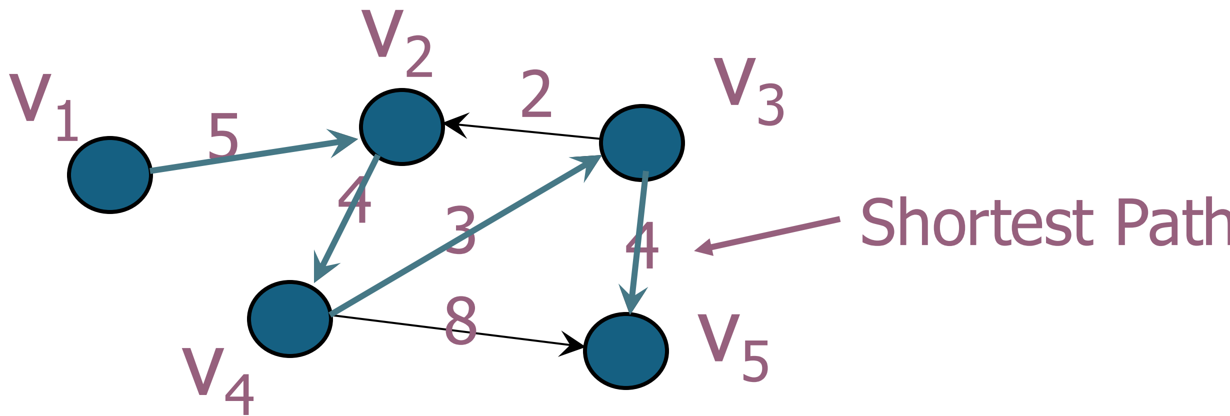
- Problem: find the minimum cost to travel from city x to city y
- Solution: find the shortest path from x to y

Single-Source Shortest Paths

- For a weighted graph $G = (V,E,w)$, the *single-source shortest paths* problem is to find the shortest paths from a vertex $v \in V$ to all other vertices in V.
- Dijkstra's algorithm is similar to Prim's algorithm.
  - maintains a set of nodes for which the shortest paths are known.
  - grow this set based on the node closest to source using one of the nodes in the current shortest path set.

Formal definition of shortest path

- Given a weighted directed graph G.
- Let P be a path of G from x to y.
- cost(P) = $\sum_{e \in P}$weight(e)
- The shortest path is a path P which minimizes cost(P)



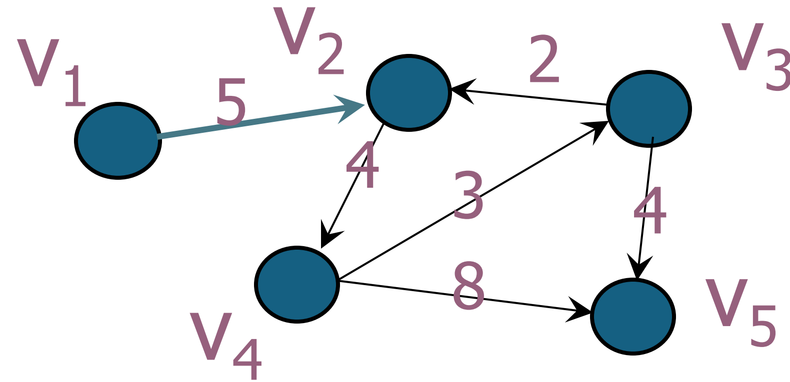
Dijkstra's algorithm

- Consider a graph G, each edge (u, v) has a weight w(u, v) > 0.
- Suppose we want to find the shortest path starting from v_1 to any node v_i
- Let VS be a subset of nodes in G
- Let cost[v_i] be the weight of the shortest path from v_1 to v_i that passes through nodes in VS only.

Dijkstra’s algorithm: code

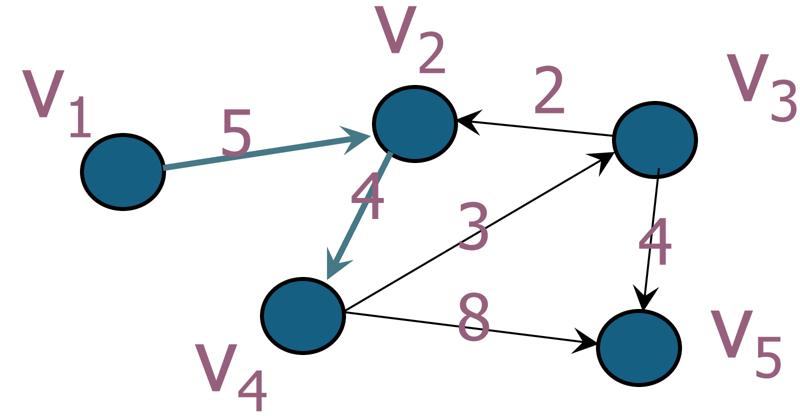
```
Algorithm shortestPath()
n = number of nodes in the graph;
for i = 1 to n
    cost[v_i] = w(v_1, v_i);
VS = { v_1 };
for step = 2 to n {
    find the smallest cost[v_i] s.t. v_i is not in VS;
    include v_i to VS;
    for (all nodes v_j not in VS) {
            if (cost[v_j] > cost[v_i] + w(v_i, v_j))
                    cost[v_j] = cost[v_i] + w(v_i, v_j);
      }
}
```

Example for Dijkstra's algorithm



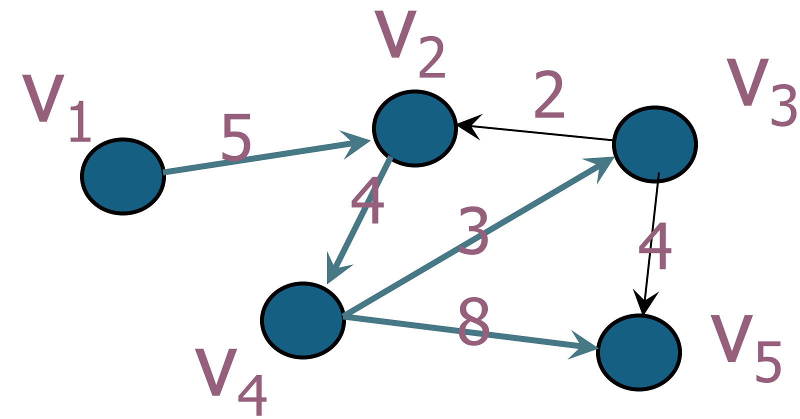
	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞

Example for Dijkstra's algorithm



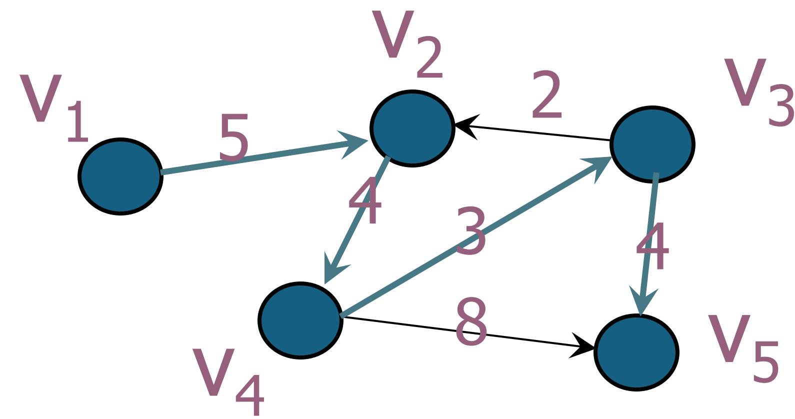
	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞

Example for Dijkstra's algorithm



	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞
3	v_4	[v_1, v_2, v_4]	0	5	12	9	17

Example for Dijkstra's algorithm



	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞
3	v_4	[v_1, v_2, v_4]	0	5	12	9	17
4	v_3	[v_1, v_2, v_4, v_3]	0	5	12	9	16
5	v_5	[v_1, v_2, v_4, v_3, v_5]	0	5	12	9	16

Complexity of Dijkstra's algorithm

The **time complexity** of Dijkstra's Algorithm is typically **O(V^2)** when using a simple array implementation or **O((V + E) log V)** with a priority queue, where **V** represents the number of vertices and **E** represents the number of edges in the graph. The **space complexity** of the algorithm is **O(V)** for storing the distances and predecessors for each node, along with additional space for data structures like priority queues or arrays.

Aspect	Complexity
Time Complexity	O((V + E) log V)
Space Complexity	O(V)

All-Pairs Shortest Paths

- Given a weighted graph *G(V,E,w)*, the *all-pairs shortest paths* problem is to find the shortest paths between all pairs of vertices $v_i, v_j \in V$.
- A number of algorithms are known for solving this problem.
  - Matrix multiplication
  - Dijkstra algorithm
  - …

All-Pairs Shortest Paths:

Matrix-Multiplication Based Algorithm

- Consider the multiplication of the weighted adjacency matrix with itself - except,
  - replace the multiplication operation in matrix multiplication by addition, and the summation operation by minimization.
- Notice that the product of weighted adjacency matrix with itself returns a matrix that contains shortest paths of length 2 between any pair of nodes.
- It follows from this argument that A^n contains all shortest paths.

Analogy of matrix multiplication

- Compute the matrix product $C = A \cdot B$ of two $n \times n$ matrices A and B. Then, for $i, j = 1, 2, \ldots, n$, we compute

$$C_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

- If we make the substitutions in $l_{ij}^{(m)} = \min_{1 \le k \le n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\}$

$$l^{(m-1)} \to a$$

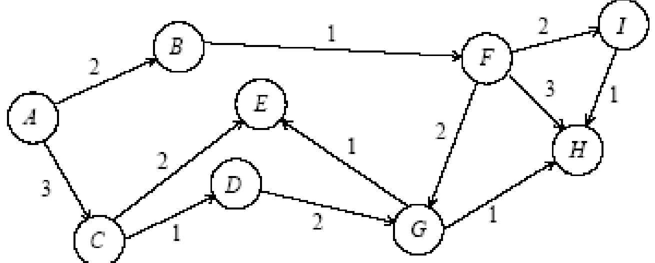
$$w \to b$$

$$l^{(m)} \to c$$

$$\min \to \sum \quad (+)$$

$$+ \to \cdot$$

Matrix-Multiplication Based Algorithm



$$
A^1 = \begin{pmatrix}
0 & 2 & 3 & \infty & \infty & \infty & \infty & \infty & \infty \\
\infty & 0 & \infty & \infty & \infty & 1 & \infty & \infty & \infty \\
\infty & \infty & 0 & 1 & 2 & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & 0 & \infty & \infty & 2 & \infty & \infty \\
\infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & \infty & 0 & 2 & 3 & 2 \\
\infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0
\end{pmatrix}
\qquad
A^2 = \begin{pmatrix}
0 & 2 & 3 & 4 & 5 & 3 & \infty & \infty & \infty \\
\infty & 0 & \infty & \infty & \infty & 1 & 3 & 4 & 3 \\
\infty & \infty & 0 & 1 & 2 & \infty & 3 & \infty & \infty \\
\infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\
\infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\
\infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0
\end{pmatrix}
$$

$$
A^4 = \begin{pmatrix}
0 & 2 & 3 & 4 & 5 & 3 & 5 & 6 & 5 \\
\infty & 0 & \infty & \infty & 4 & 1 & 3 & 4 & 3 \\
\infty & \infty & 0 & 1 & 2 & \infty & 3 & 4 & \infty \\
\infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\
\infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\
\infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0
\end{pmatrix}
\qquad
A^8 = \begin{pmatrix}
0 & 2 & 3 & 4 & 5 & 3 & 5 & 6 & 5 \\
\infty & 0 & \infty & \infty & 4 & 1 & 3 & 4 & 3 \\
\infty & \infty & 0 & 1 & 2 & \infty & 3 & 4 & \infty \\
\infty & \infty & \infty & 0 & 3 & \infty & 2 & 3 & \infty \\
\infty & \infty & \infty & \infty & 0 & \infty & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty & 3 & 0 & 2 & 3 & 2 \\
\infty & \infty & \infty & \infty & 1 & \infty & 0 & 1 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 0 & \infty \\
\infty & \infty & \infty & \infty & \infty & \infty & \infty & 1 & 0
\end{pmatrix}
$$

Time complexity analysis: Matrix-Multiplication Based Algorithm

- A^n is computed by doubling powers - i.e., as A, A^2, A^4, A^8, and so on.
- We need $\log n$ matrix multiplications, each taking time $O(n^3)$.
- The serial complexity of this procedure is $O(n^3 \log n)$.
- This algorithm is not optimal, since the best known algorithms have complexity $O(n^3)$.

APSP using Dijkstra's SSSP algorithm

- Execute n instances of the single-source shortest path problem, one for each of the n source vertices.

- Complexity is $O(n^3)$.

Circuit Problems

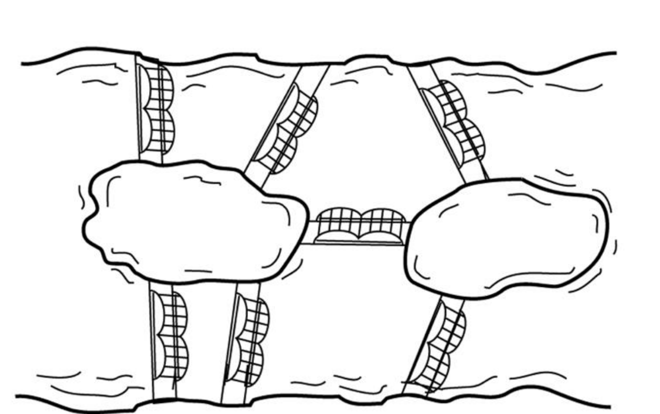
Circuits

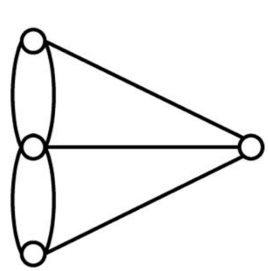
- A circuit
  - A special cycle that passes through every vertex (or edge) in a graph exactly once

- Euler Circuit is a circuit that begins at a vertex v, passes through every **edge** exactly once, and terminates at v.

- Euler Circuit exists if and only if each vertex touches an even number of edges (i.e., has an even *degree*)

a) Euler's bridge problem and b) its multigraph representation





Euler and Hamilton

- An **Euler path** is a path that passes through every edge exactly once. If it ends at the initial vertex then it is an *Euler circuit*.

- A **Hamiltonian path** is a path that passes through every vertex exactly once (NOT every edge). If it ends at the initial vertex then it is a *Hamiltonian circuit*.

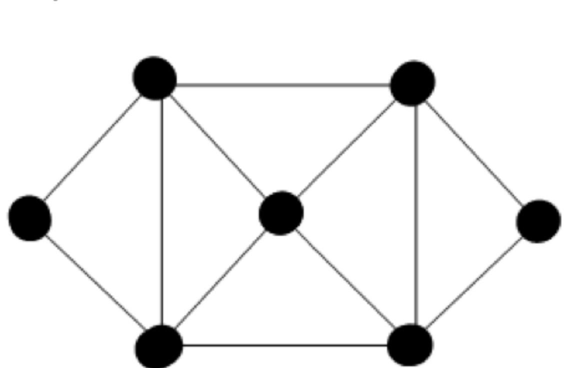
- Note:
  - In an Euler path you might pass through a vertex more than once.
  - In a Hamiltonian path you may not pass through all edges.

Exercise

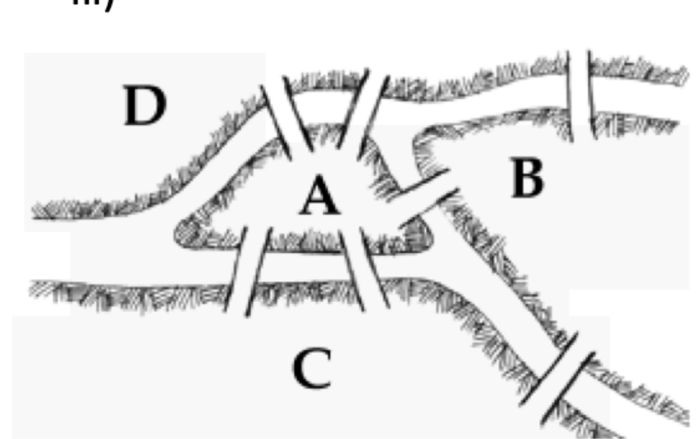
Which of the following graphs have a Eulerian circuit?

i)

ii)



iii)



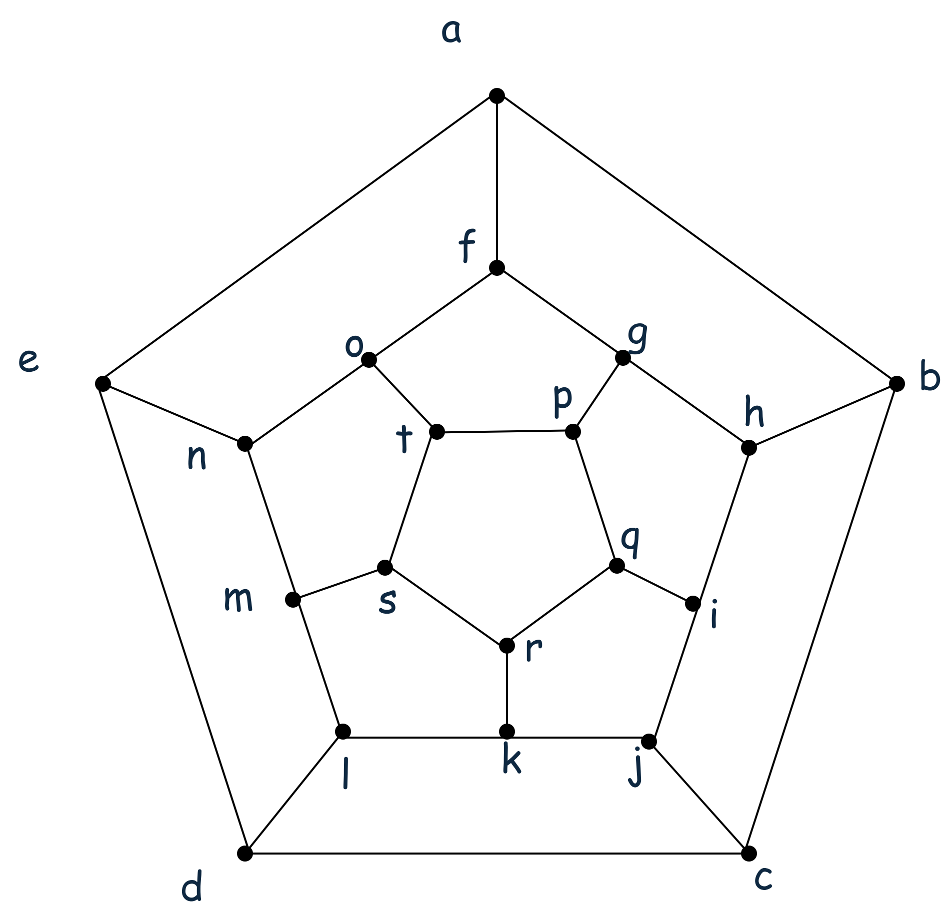
Therefore, a Eulerian circuit always exist if all of its vertices are of even degree.

In other words, a Eulerian circuit always exist if there are not odd degree vertices.

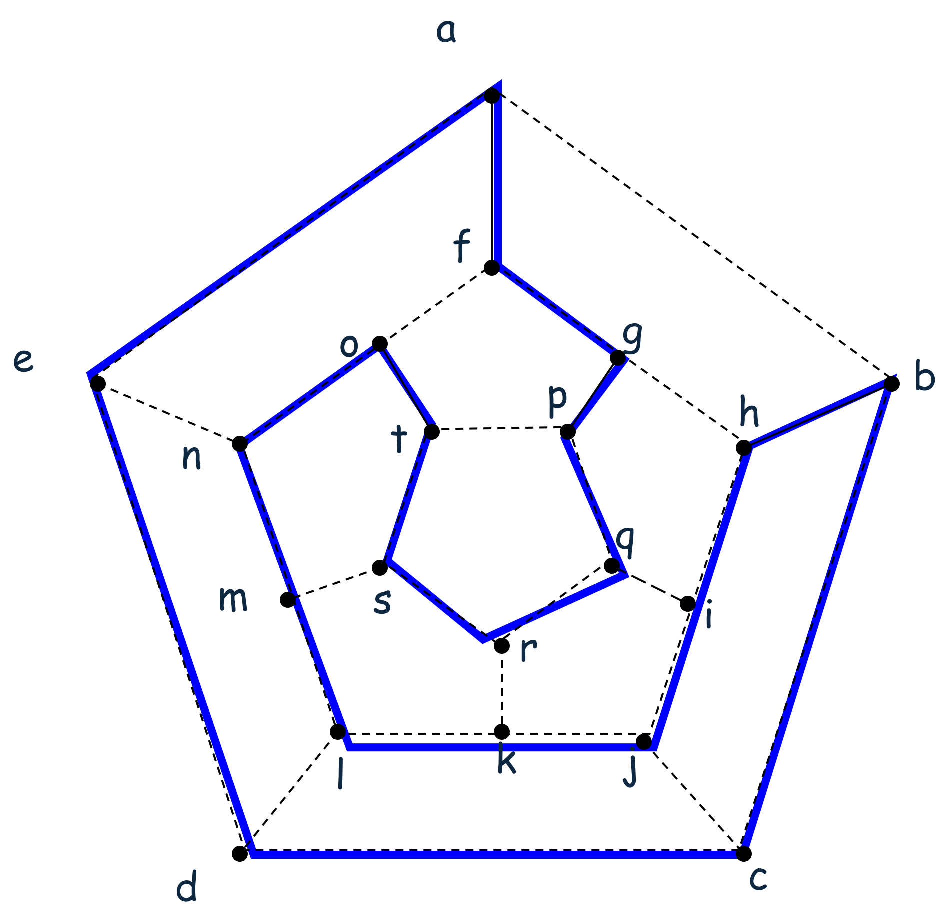
Hamilton circuit

- A Hamilton circuit is a cycle that begins at a vertex v, passes through every **vertex** in the graph exactly once, and terminates at v.
- Hamiltonian circuit = visit each vertex once and only once and come back to where you started from
- Note: In a Hamiltonian path (or circuit) you may not pass through all edges.
- Given a graph, does it have a Hamiltonian circuit ?

Hamilton Circuit: example



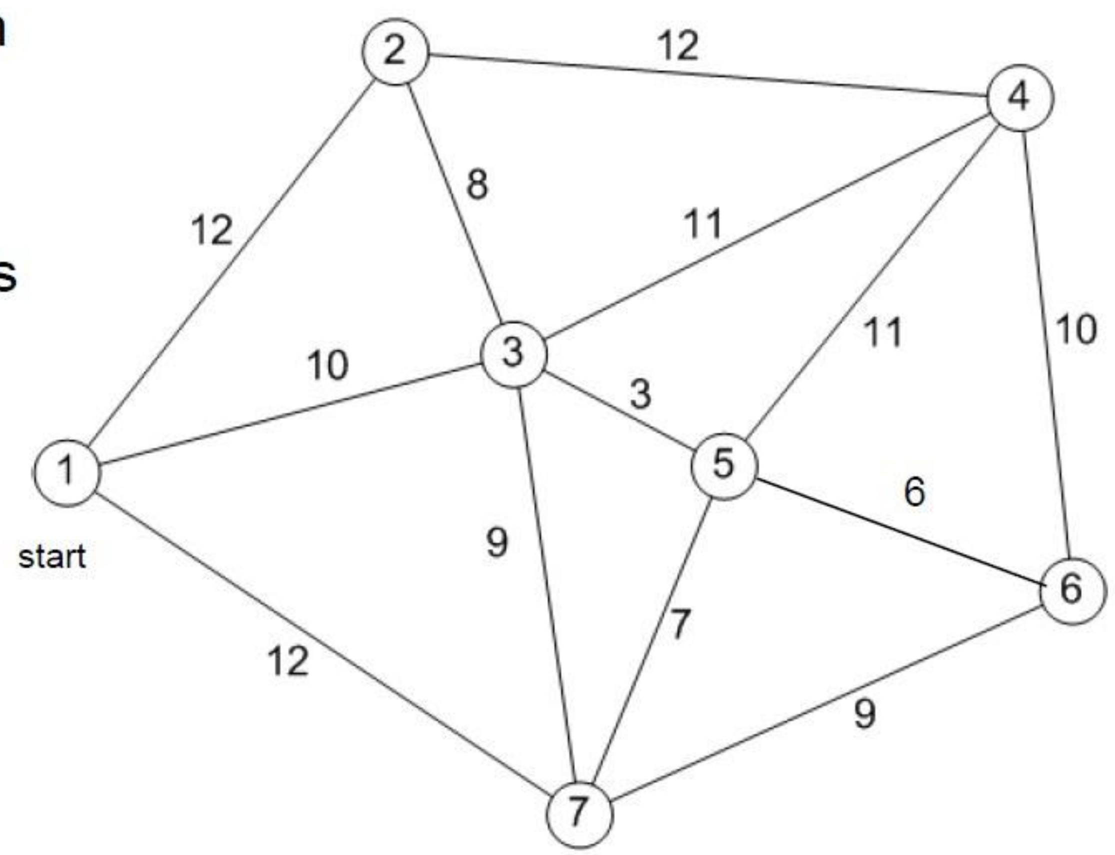
Hamilton Circuit: example



TSP problem

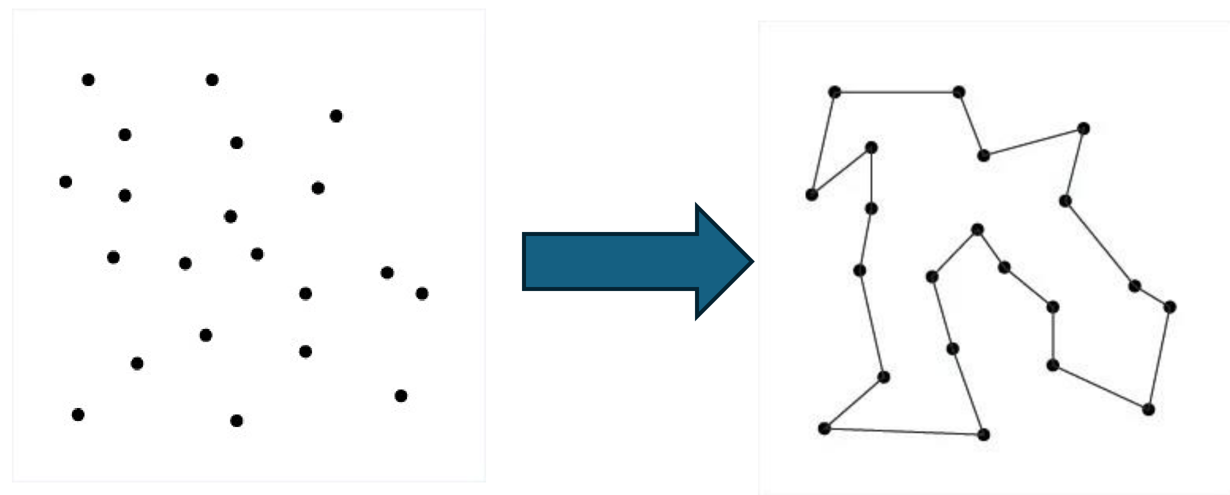
The Traveling Salesman Problem

- Starting from city 1, the salesman must travel to all cities once before returning home
- The distance between each city is given, and is assumed to be the same in both directions
- Only the links shown are to be used
- Objective - Minimize the total distance to be travelled



TSP (The travelling salesman problem)

- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city ?
- This is an optimization problem



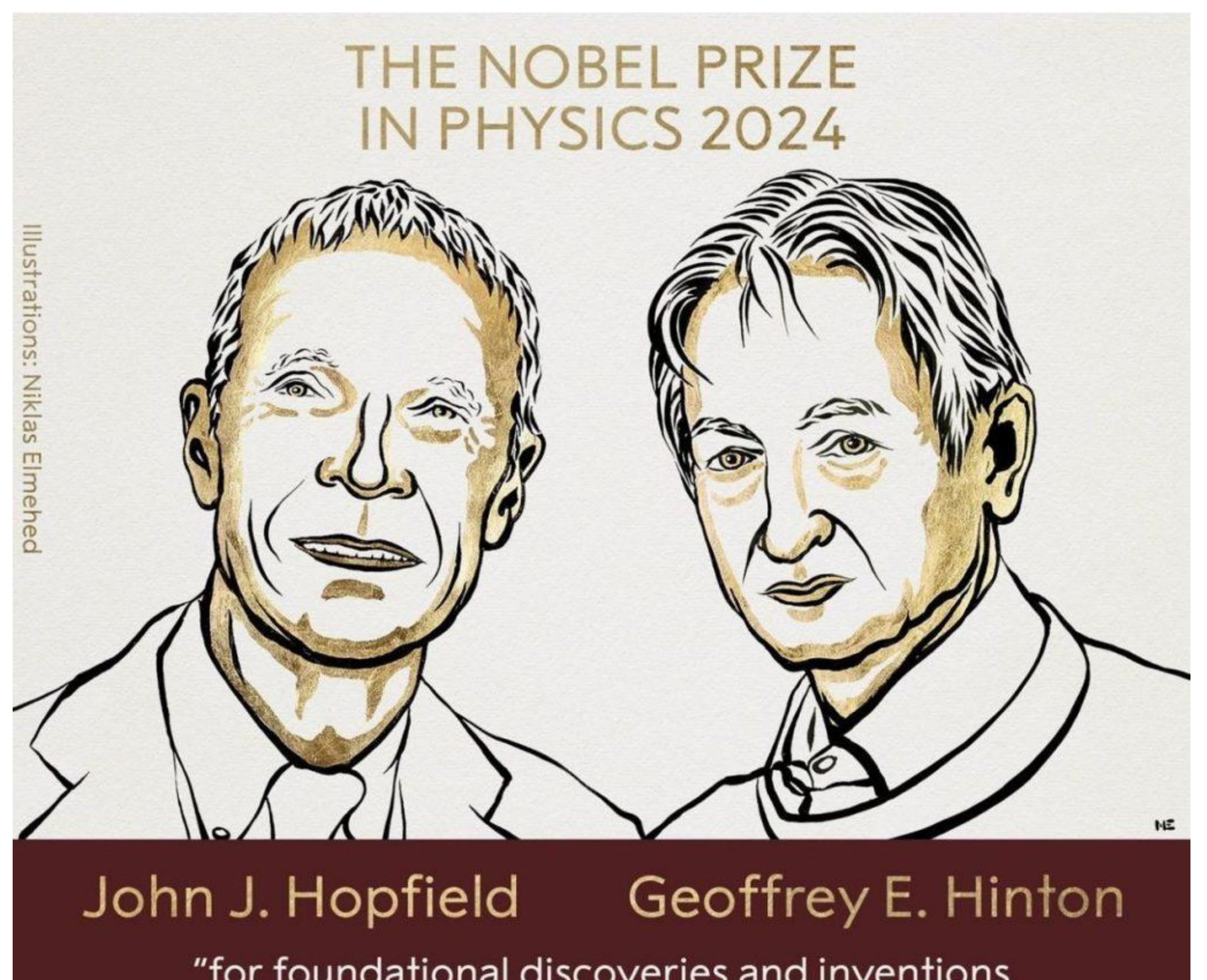
The Traveling Salesman Problem (TSP) and the Hamiltonian Circuit Problem are closely related but not identical.

- **Traveling Salesman Problem (TSP)**: Given a set of cities and the distances between each pair of cities, the TSP asks for the shortest possible route that visits each city exactly once and returns to the origin city. It focuses on minimizing the total travel cost (distance, time, etc.).

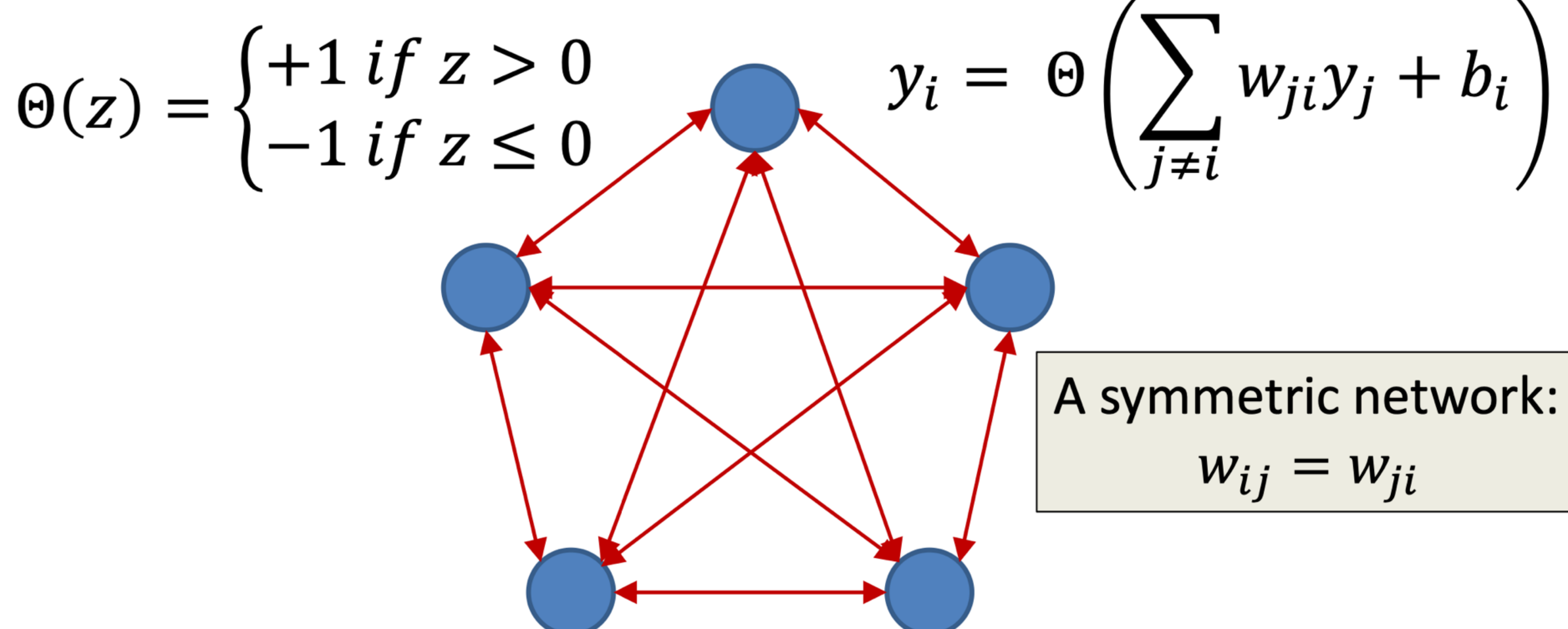
- **Hamiltonian Circuit Problem**: This problem involves finding a Hamiltonian circuit in a graph, which is a cycle that visits each vertex exactly once and returns to the starting vertex. It does not consider the weights or costs associated with the edges; it merely asks if such a circuit exists.

How do we really solve TSP?

- Stochastic Gradient descent
- Simulated annealing
- Genetic algorithms
- Ant colony algorithms
- Randomized algorithms
- Hopfield neural network algorithm (Nobel Prize for Physics 2024).


THE NOBEL PRIZE
IN PHYSICS 2024
Illustrations: Niklas Elmehed
John J. Hopfield
Geoffrey E. Hinton
"for foundational discoveries and inventions
that enable machine learning
with artificial neural networks"
THE ROYAL SWEDISH ACADEMY OF SCIENCES

Hopfield net and Traveling Salesman problem



Traveling Salesman Problem

- Goal
  - Come back to the city A, visiting j = 2 to n (n is number of cities) exactly once and minimize the total distance.
- To solve by a Hopfield-Net we need to decide the *architecture*:
  - How many neurons?
  - What are the weights?

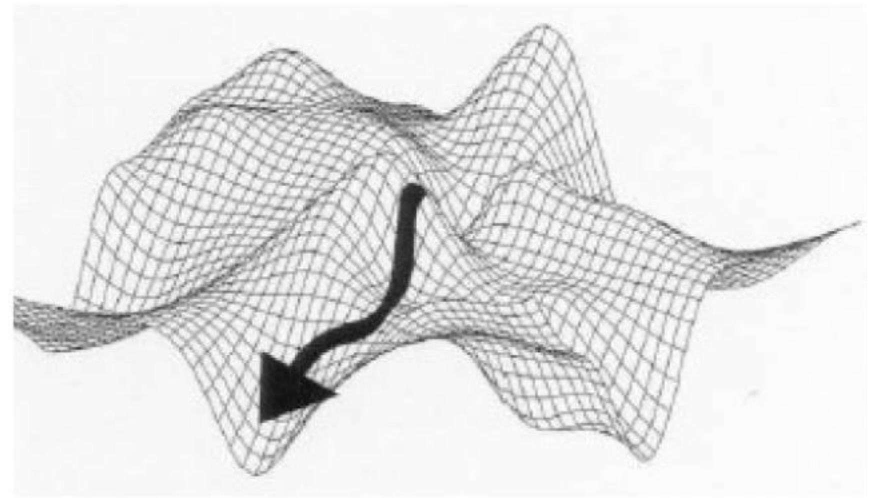
Solving TSP by continuous Hopfield model:

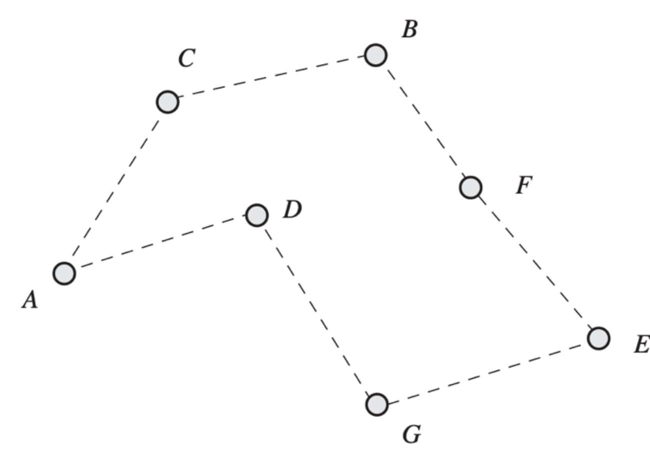
1. Design the network structure
2. Define an energy function
   - punish violation of (strong) constraint with large amount of energy
   - lower energy associates to shorter circuits (weak constraint)
3. Find weight matrix from the energy function such that energy always decreases whenever the network moves (according to H model and ***W***)

Hopfield-Net energy function (loss function) for TSP

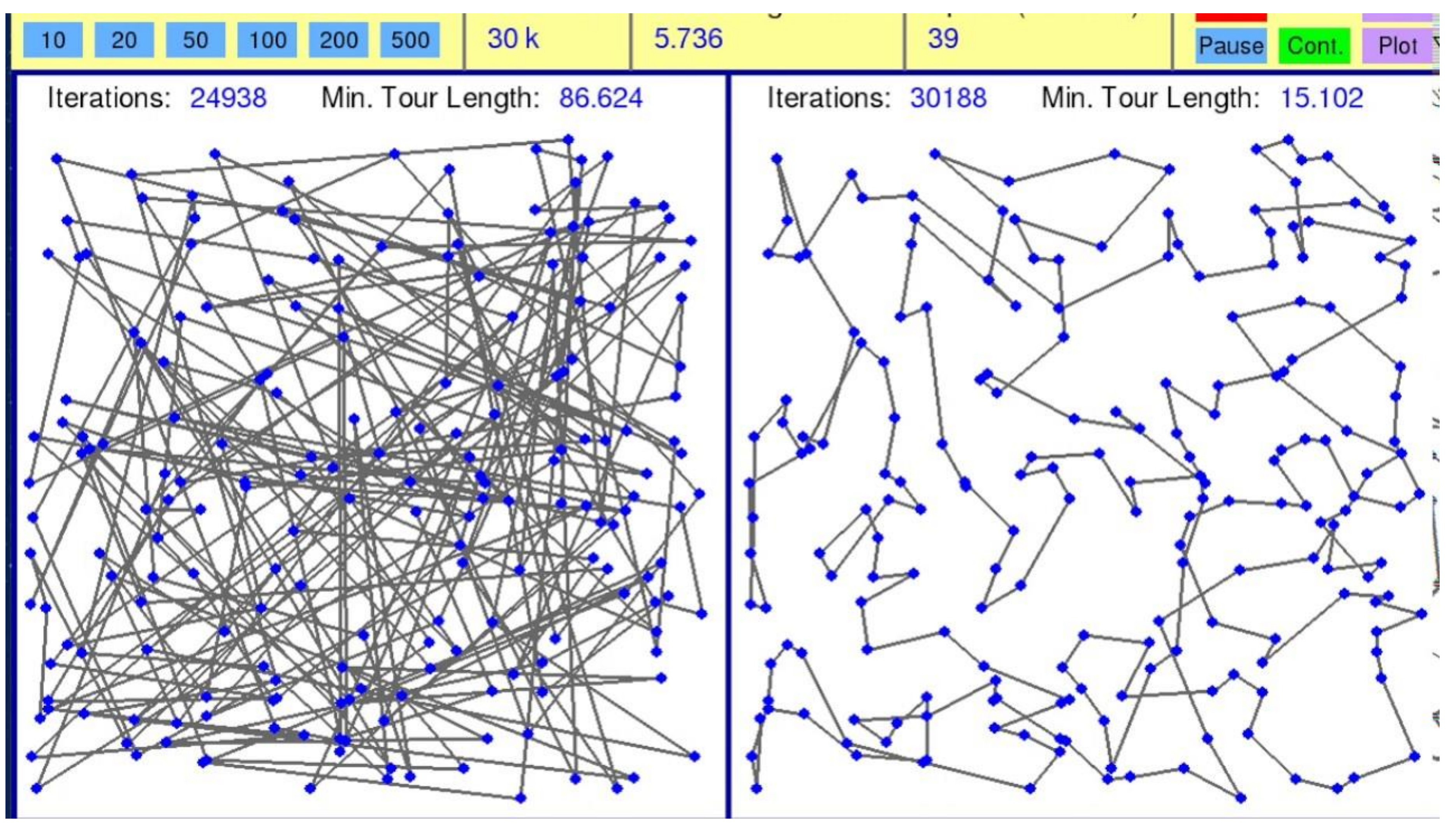
The distance between the city S_i and the city S_j is d_{ij}. To find the shortest path, the network will minimize the function :

$$E_{tsp} = \underbrace{\frac{1}{2}\sum_{i,j,k}^{n} d_{ij}x_{ik}x_{j,k+1}}_{L} + \underbrace{\frac{\gamma}{2}(\sum_{j=1}^{n}(\sum_{i=1}^{n} x_{ij} - 1))^2 + \sum_{i=1}^{n}(\sum_{j=1}^{n} x_{ij} - 1)^2))}_{Condition} \quad (7)$$

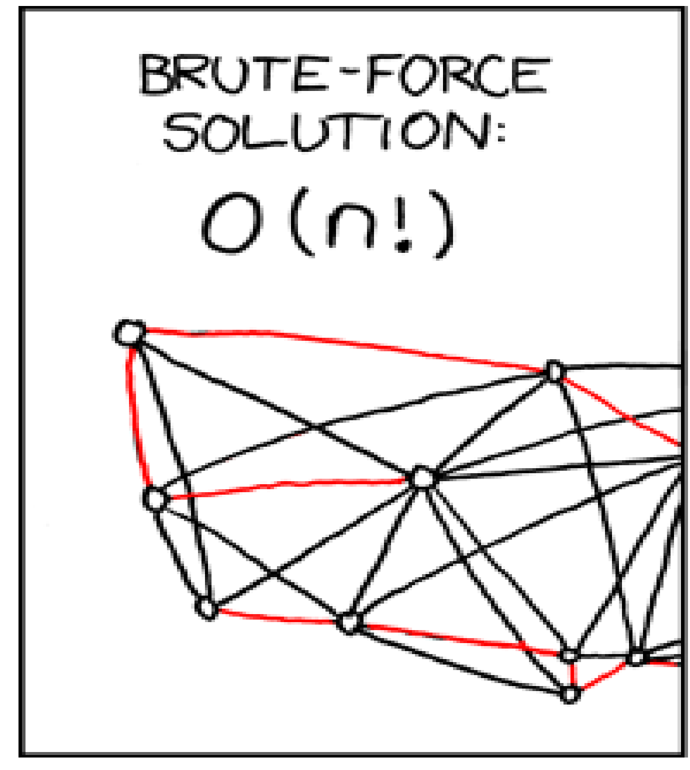


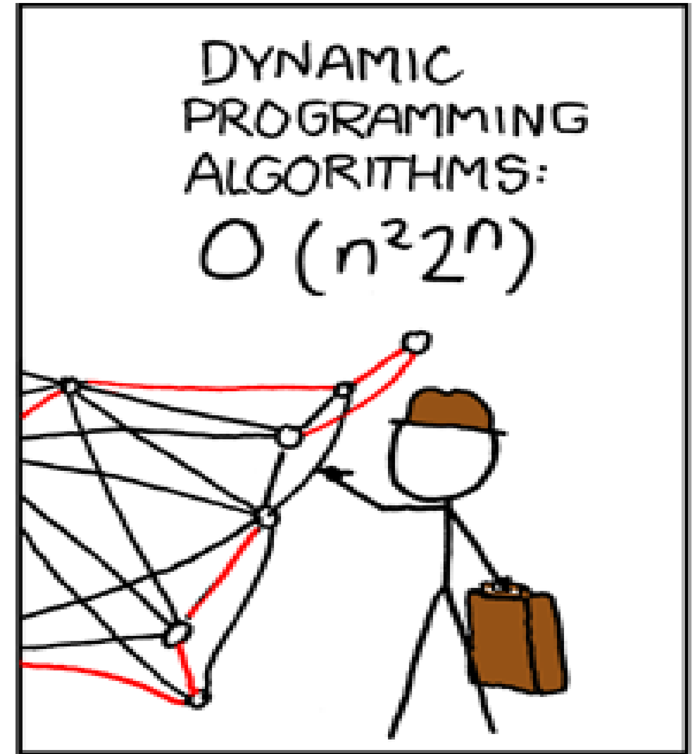


An example solution



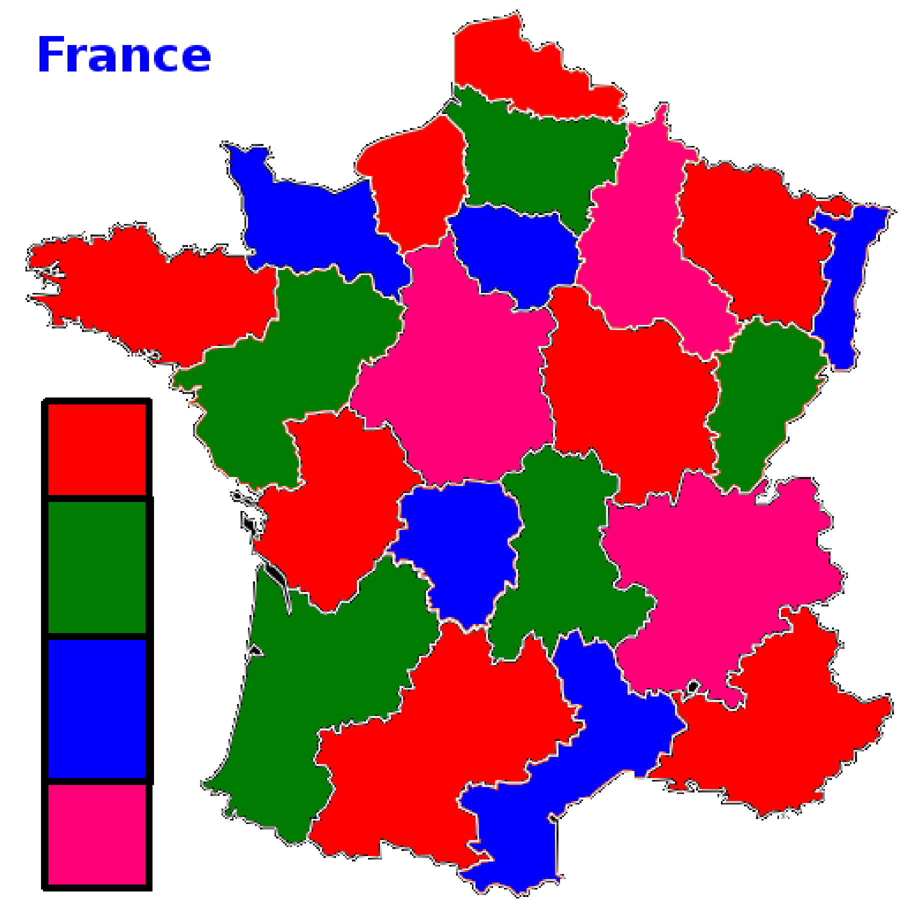
Comics





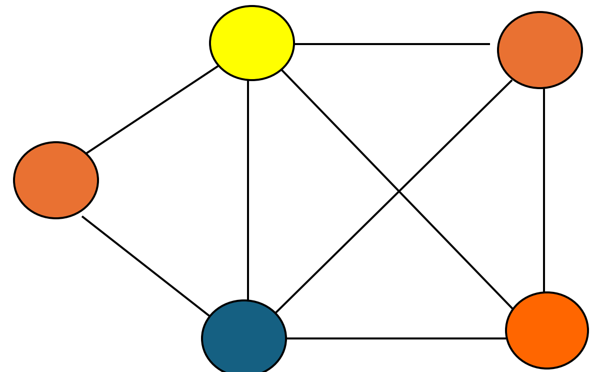
Graph Coloring problem

- Color a map such that two regions with a common border are assigned different colors.
- Each map can be represented by a graph:
  - Each region of the map is represented by a vertex;
  - Edges connect two vertices if the regions represented by these vertices have a common border.



The Graph Coloring Problem

- **Definition**: A graph has been colored if a color has been assigned to each vertex in such a way that adjacent vertices have different colors.

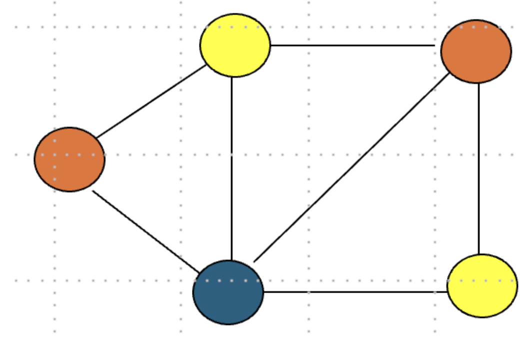


- **Definition:** The chromatic number of a graph is the smallest number of colors with which it can be colored.

In the example above, the chromatic number is 4.

Graph Coloring problem

- **Definition**: A graph is planar if it can be drawn in a plane without edge-crossings.



- **The four color theorem:** For every planar graph, the chromatic number is ≤ 4.

  Was posed as a conjecture in the 1850s. Finally proved in 1976 (Appel and Haken) by the aid of computers.

Summary

- Graphs can be used to represent many real-life problems.
- Implementation of Graph ADT
- There are numerous important graph algorithms.
- We have studied some basic concepts and algorithms.
  - Graph Traversal
  - Topological Sort
  - Spanning Tree
  - Minimum Spanning Tree
  - Shortest Path
  - Circuit problems
  - Graph coloring.
  - …

Week-12: class review.

J: Java

- J1 Introductory Java, part 1
- J2 Introductory Java, part 2
- J3 Arrays
- J5 Control Flow: Branching
- J6 Control Flow: Iteration
- J9 Higher-order programming
- J12 Generics
- J14 Collections
- J15 Exceptions

O: Object Orientation

- O1 Objects and Classes, part 1
- O2 Objects and Classes, part 2
- O4 Inheritance

S: Software Engineering

- S1 Software Development Tools
- S4 Unit testing
- S5 Software Design

C: Core Computer Science

- C1 Recursion
- C1 (part 2) Recursion revisited
- C2 Computational Complexity
- C3 Graph Traversal
- C4 Hash Functions
- C6 Files
- C7 Threads

A: Abstract Data Types

- A1 ADTs: Lists
- A2 List Implementations
- A3/A6 Sets & Maps
- A4 Sets: HashSet
- A5 Trees