Data Structures & Algorithms 2

Topic 8 – Weighted Graphs

Lecturer: Dr. Hadi Tabatabaee
Materials: Dr. Phil Maguire & Dr. Hadi Tabatabaee
Maynooth University
Online at http://moodle.maynoothuniversity.ie

Overview

Aims

- Introduce the minimum spanning tree problem.
- Introduce the shortest path problem in a graph.

Learning outcomes: You should be able to…

- Learn how to solve the minimum spanning tree problem in a weighted graph.
- Learn how to solve the shortest path problem in a weighted graph.

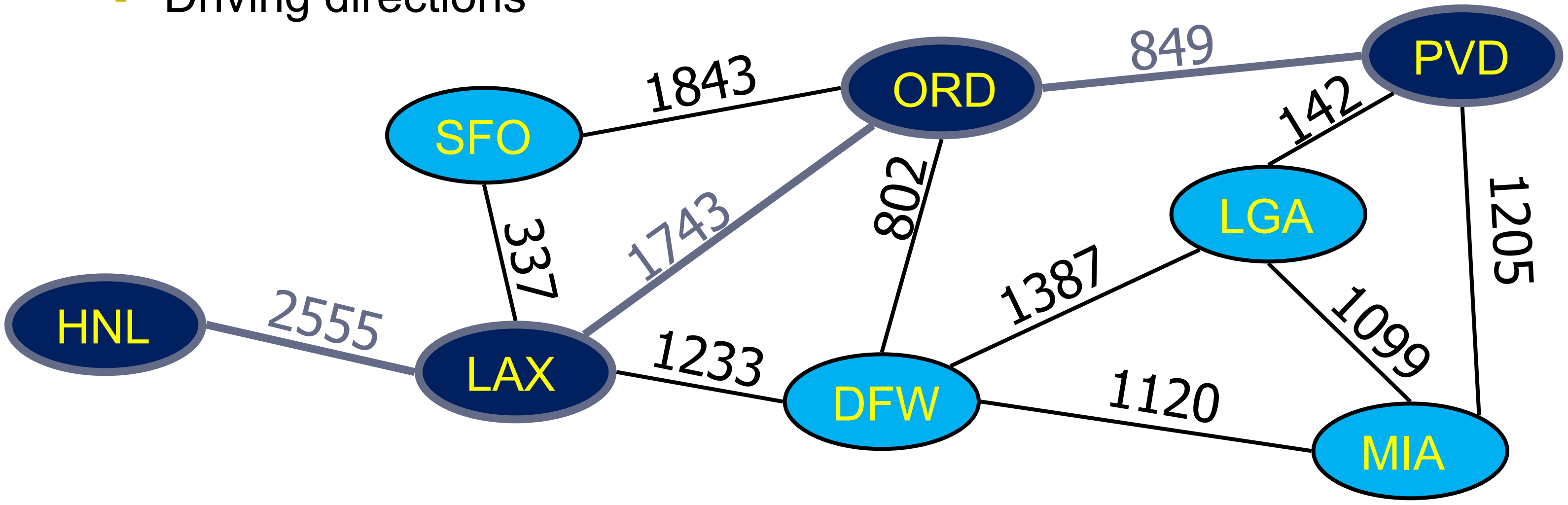
Weighted Graphs

- Now to introduce an additional edge feature that makes things more complicated: **weight**
- For example, if vertices in a weighted graph represent cities, then the weight of the edges might represent the distances between the cities or the cost of flying between them.
- The exciting question arise, such as
  - What is the shortest (or cheapest) distance between two vertices?
  - What is the minimum spanning tree (MST) for a weighted graph?
- These questions have very important applications in the real world.

Shortest path

Shortest Paths

- Given a weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v.
  - The length of a path is the sum of the weights of its edges.
- Example:
  - The shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



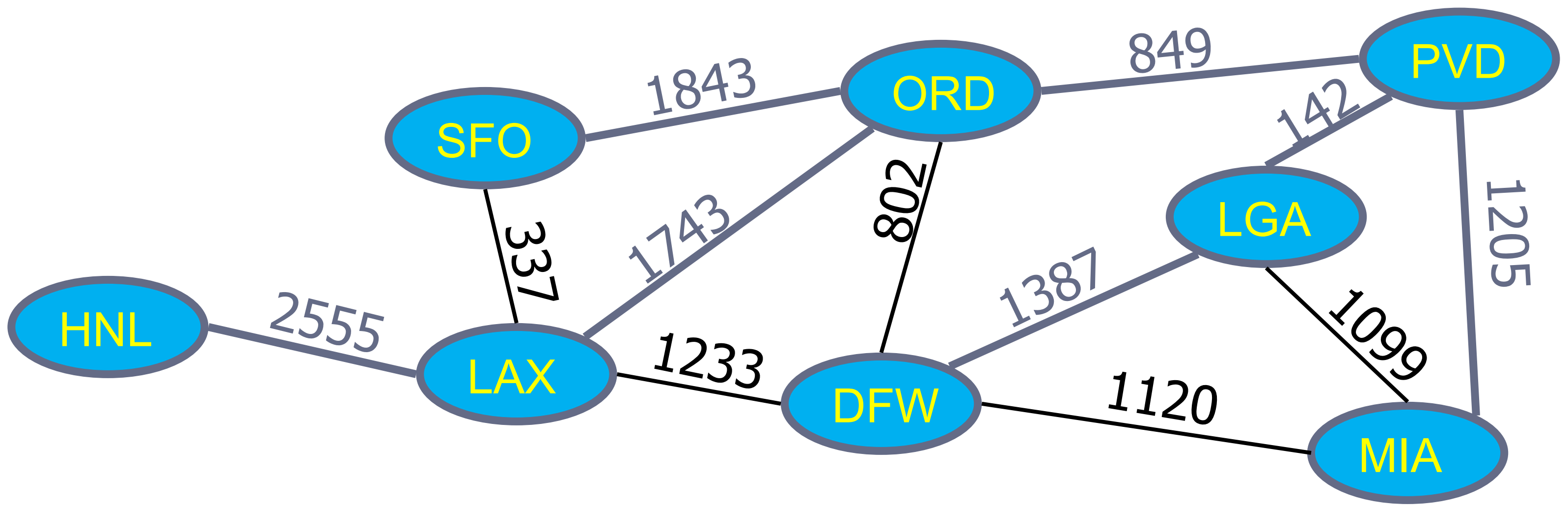
Shortest Path Properties

Property 1:

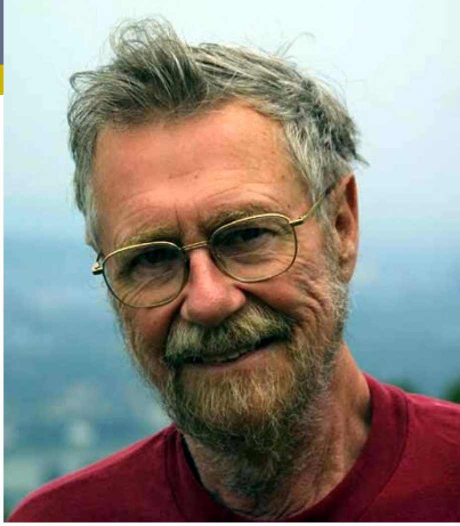
A subpath of a shortest path is itself a shortest path.

Property 2:

There is a tree of shortest paths from a start vertex to all the other vertices.



Dijkstra’s algorithm



- Dijkstra’s algorithm is also known as the shortest path problem
- The problem was conceived by Dutch computer scientist Edsger Dijkstra in 1959.
- This algorithm is often used for routing network traffic in the most efficient manner.
- The problem is also applicable to a wide variety of real-world situations, from the layout of printed circuit boards to project scheduling.

Dijkstra’s algorithm

The essentials of Dijkstra’s algorithm are as follows:

1. Start at any vertex and consider the weights on the edges leading from it.
2. Always visit the unvisited vertex that has the total cheapest path from the starting point.
3. Each time you visit a new vertex, use the new edge information to revise your list of cheapest paths to each unvisited vertex from the starting point.
4. Keep going until all vertices have been visited.

Implementation

- We need an array that keeps track of the minimum distances from the starting vertex to the other vertices.
- During execution, these distances are changed until at the end they hold the actual shortest distances for all vertices from the start vertex.
- We create an array that holds the shortest known distances to all of the vertices involved.
- This Shortest-Path Array simply has one slot for each of the vertices in the graph.

A	B	C	D	E
-	50	100	80	140

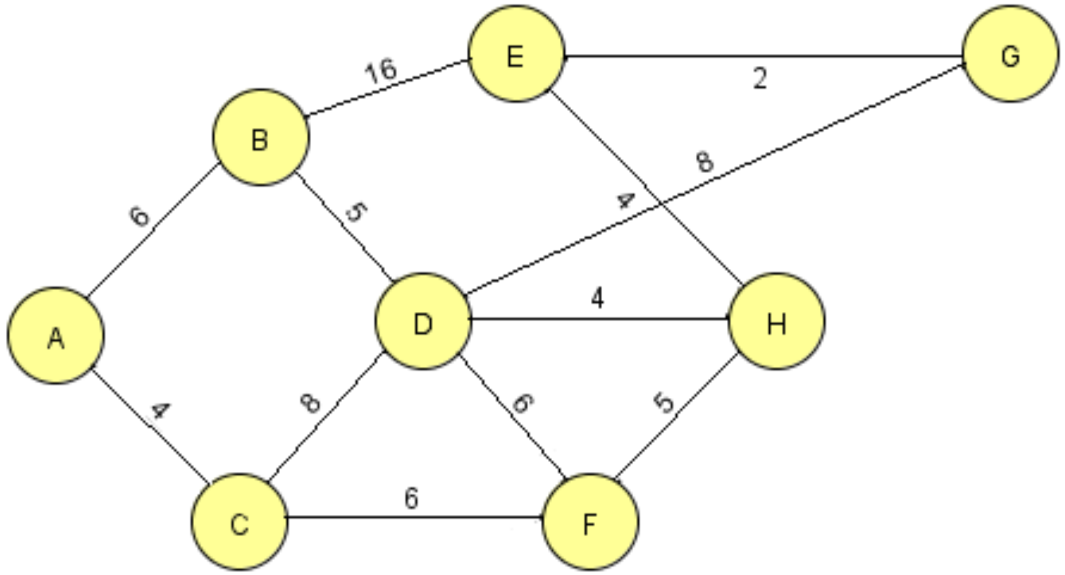
Implementation

- As well as tracking the minimum distance to each vertex, we also need to know the path taken to get there.
- Luckily, we don't have to store the whole path – just the previous step.
- For example, if we know the shortest path to E came through C then we just have to look where the shortest path to C came through etc.
- Use the Shortest-Path Array to store the shortest distance to each vertex and the previous step on the shortest path to that vertex.

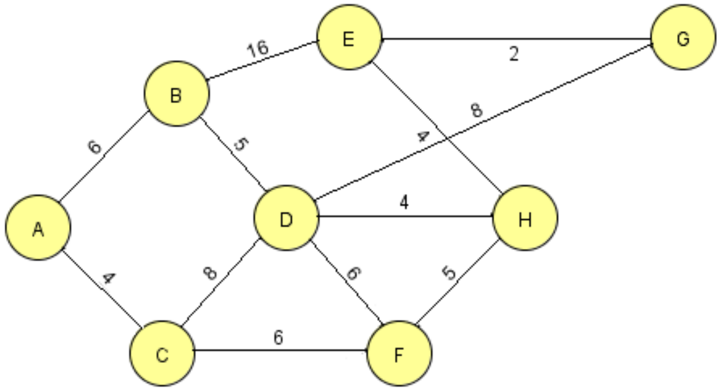
A	B	C	D	E
-	50(A)	100(D)	80(A)	140(C)

Example

Use Dijkstra’s shortest path algorithm to find the cheapest path between the vertices A and E.



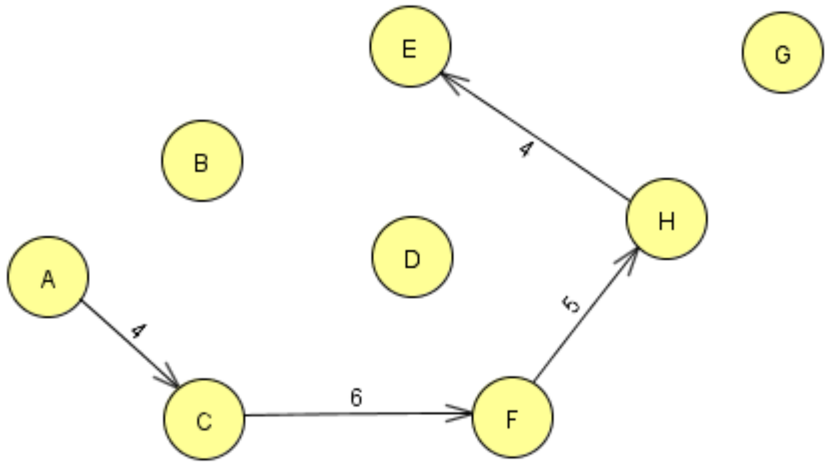
Worked Solution



Vertices Visited	A	B	C	D	E	F	G	H
A	0	6(A)	4(A)	-	-	-	-	-
A C	0	6(A)	4(A)	12(C)	-	10(C)	-	-
A C B	0	6(A)	4(A)	11(B)	22(B)	10(C)	-	-
A C B F	0	6(A)	4(A)	11(B)	22(B)	10(C)	-	15(F)
A C B F D	0	6(A)	4(A)	11(B)	22(B)	10(C)	19(D)	15(F)
A C B F D H	0	6(A)	4(A)	11(B)	19(H)	10(C)	19(D)	15(F)
A C B F D H E	0	6(A)	4(A)	11(B)	19(H)	10(C)	19(D)	15(F)

- Keep picking the shortest path to an unvisited vertex (red shows visited)
- When you move to a new vertex update the shortest paths

Solution



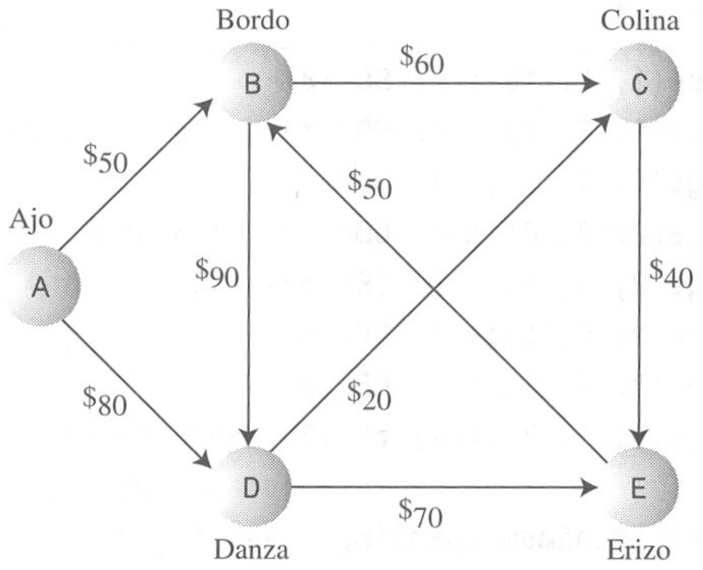
- The final row in the Shortest-Path Array represents the shortest paths to all of the vertices when starting at A

Vertices Visited	B	C	D	E	F	G	H
A C B F D H E G	6 (A)	4 (A)	11 (B)	19 (H)	10 (C)	19 (D)	15 (F)

All-pairs Shortest Path

- Dijkstra’s algorithm gives us the shortest distance from one vertex to all the other vertices in the graph.
- We can easily enhance this algorithm. Instead of just a single dimension array, it produces a matrix that gives the shortest distance from any vertex to any other vertex.
- We run Dijkstra’s algorithm several times, starting from a different vertex each time.
- This gives us the all-pairs shortest-path matrix.

All-pairs Shortest Path



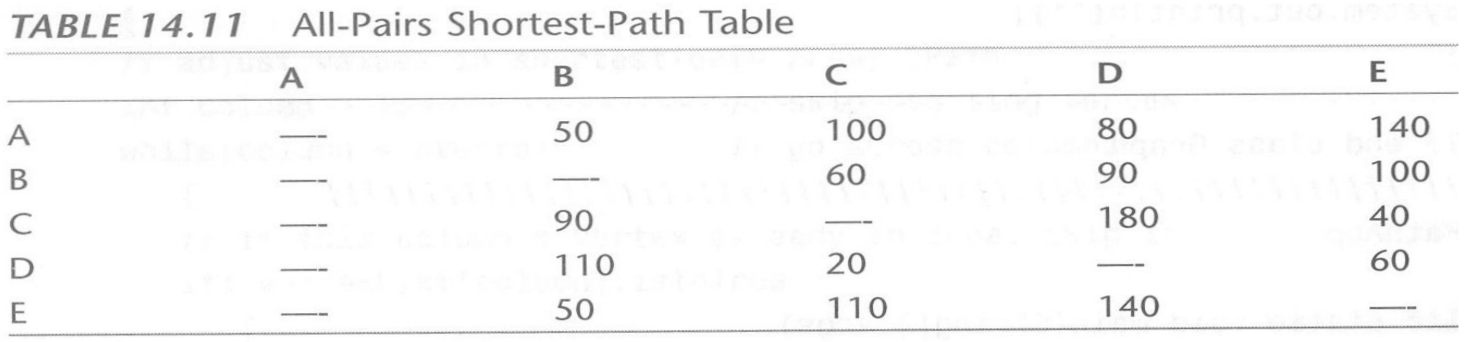


TABLE 14.11 All-Pairs Shortest-Path Table

	A	B	C	D	E
A	—-	50	100	80	140
B	—-	—-	60	90	100
C	—-	90	—-	180	40
D	—-	110	20	—-	60
E	—-	50	110	140	—-

Efficiency

- Efficiency depends on whether we're representing the graph using an adjacency matrix or an adjacency list
- Using an adjacency matrix, the algorithms mostly require O(V^2) time, where V is the number of vertices.
- This is because there is a slot in the matrix for a potential edge between a vertex and every other vertex.
- We have to check through all of these potential edges to determine whether there is a valid edge or not.
- If there are many vertices, this is bad news – however, if the graph is dense (meaning the graph has many edges), then most of the slots in the adjacency matrix are filled anyway, and there's not much we can do to improve performance.

Efficiency

- Many graphs are *sparse,* meaning that there are few edges, and most of the slots in an adjacency matrix would be empty.
- In this case, the running time can be improved by using an adjacency list.
- Adjacency lists just hold a list of all the existing edges, so you don't waste any time checking matrix slots that don't hold edges.

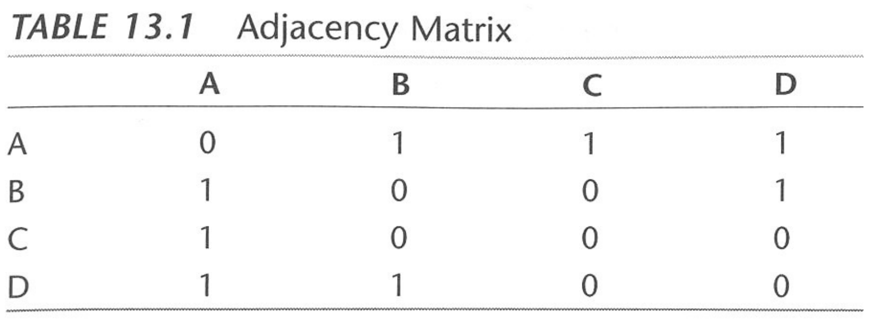


TABLE 13.1 Adjacency Matrix

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

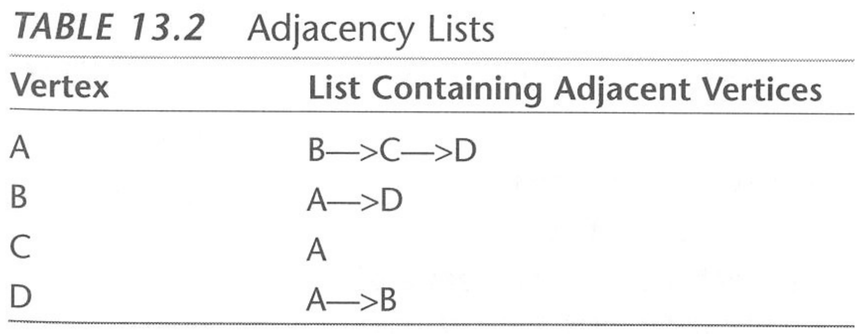


TABLE 13.2 Adjacency Lists

Vertex	List Containing Adjacent Vertices
A	B—>C—>D
B	A—>D
C	A
D	A—>B

Efficiency

- For un-weighted graphs the depth-first search with adjacency lists requires O(V+E) time where V is the number of vertices and E is the number of edges

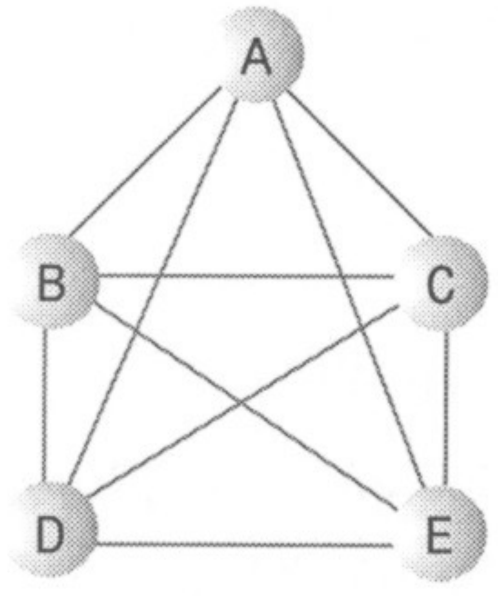
- For weighted graphs both the minimum spanning tree and shortest-path algorithms require O((E+V)logV) time using an adjacency list

Minimum spanning tree

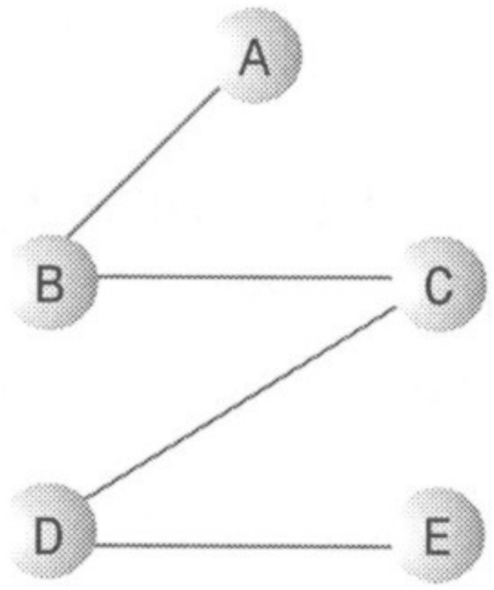
Minimum Spanning Tree

- Suppose you want to connect all the vertices in your graph using the least number of edges.
- This is called a *minimum spanning tree* (MST)
- The fewest number of edges needed will always be one less than the number of vertices.
  - E = V – 1
- For unweighted graphs, this is simple –use a searching algorithm to visit every vertex.
- For weighted graphs, it is a bit more complex.

Minimum Spanning Tree



a) Extra Edges



b) Minimum Number of Edges

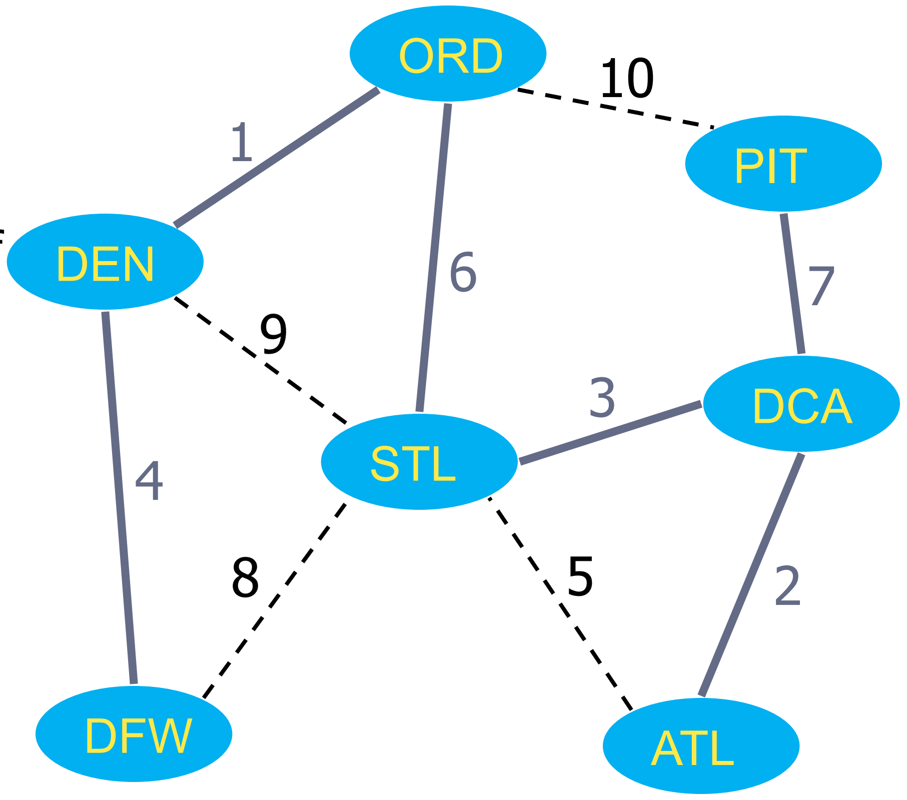
Minimum Spanning Trees

Spanning subgraph
- Subgraph of a graph ***G*** containing all the vertices of ***G***

Spanning tree
- Spanning subgraph that is itself a (free) tree

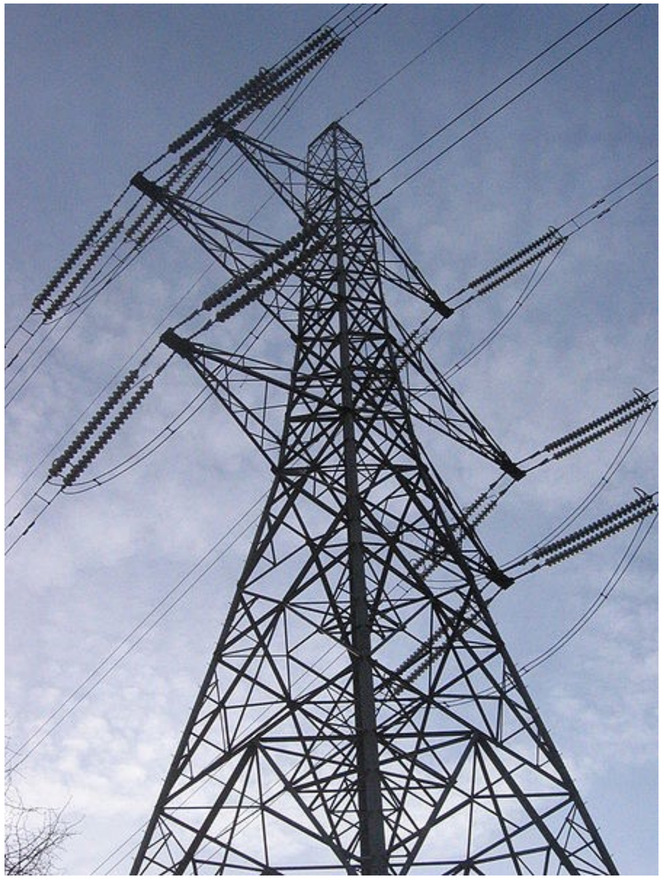
Minimum spanning tree (MST)
- Spanning tree of a weighted graph with minimum total edge weight

- Applications
  - Communications networks
  - Transportation networks



Minimum Spanning Tree

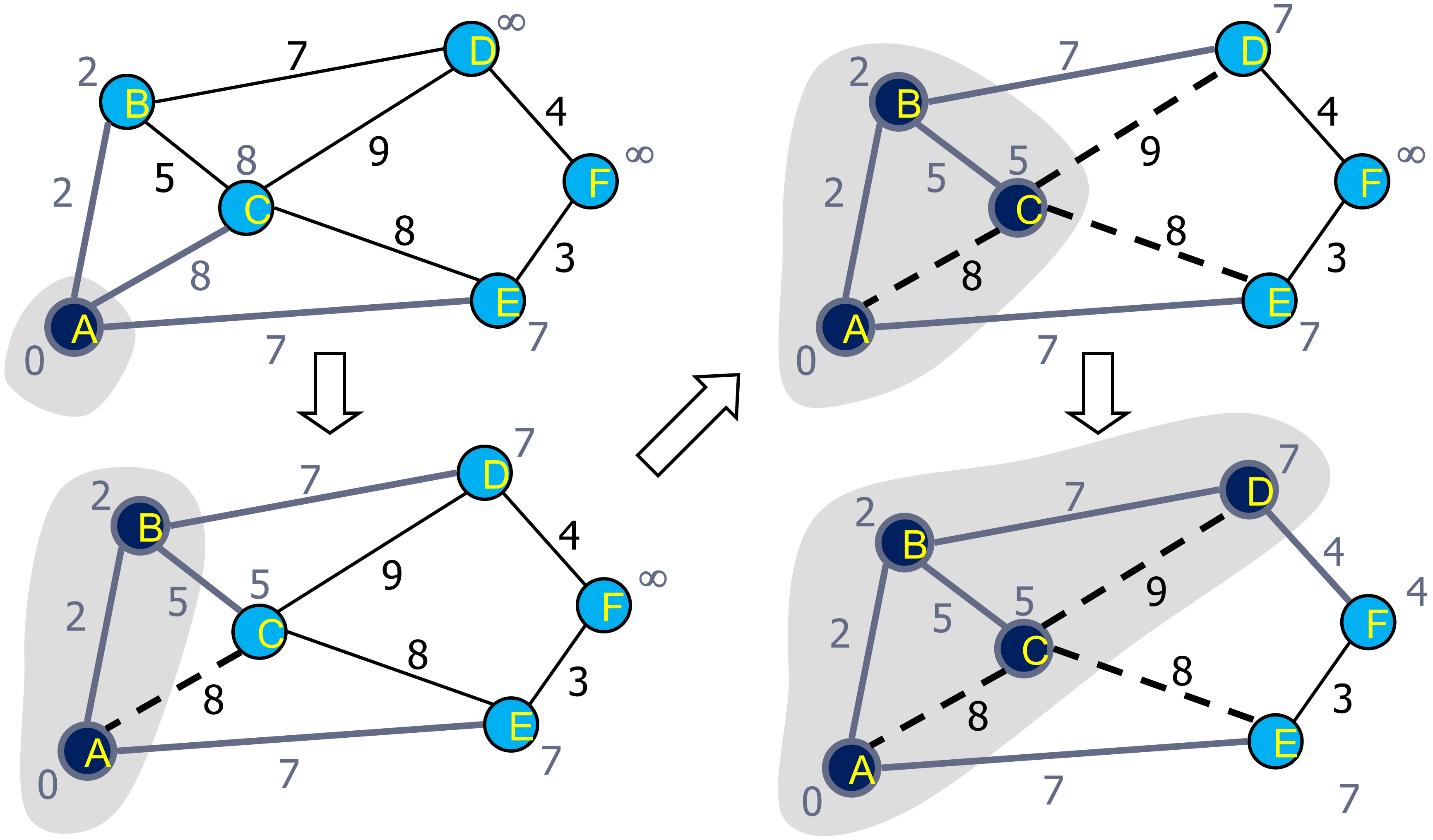
- Because the edges have different weights, it is essential to choose the proper selection to link all vertices.
- A real-world application of the MST algorithm would be to link a series of towns up to the electricity grid.
- What is the cheapest way to link them all together?



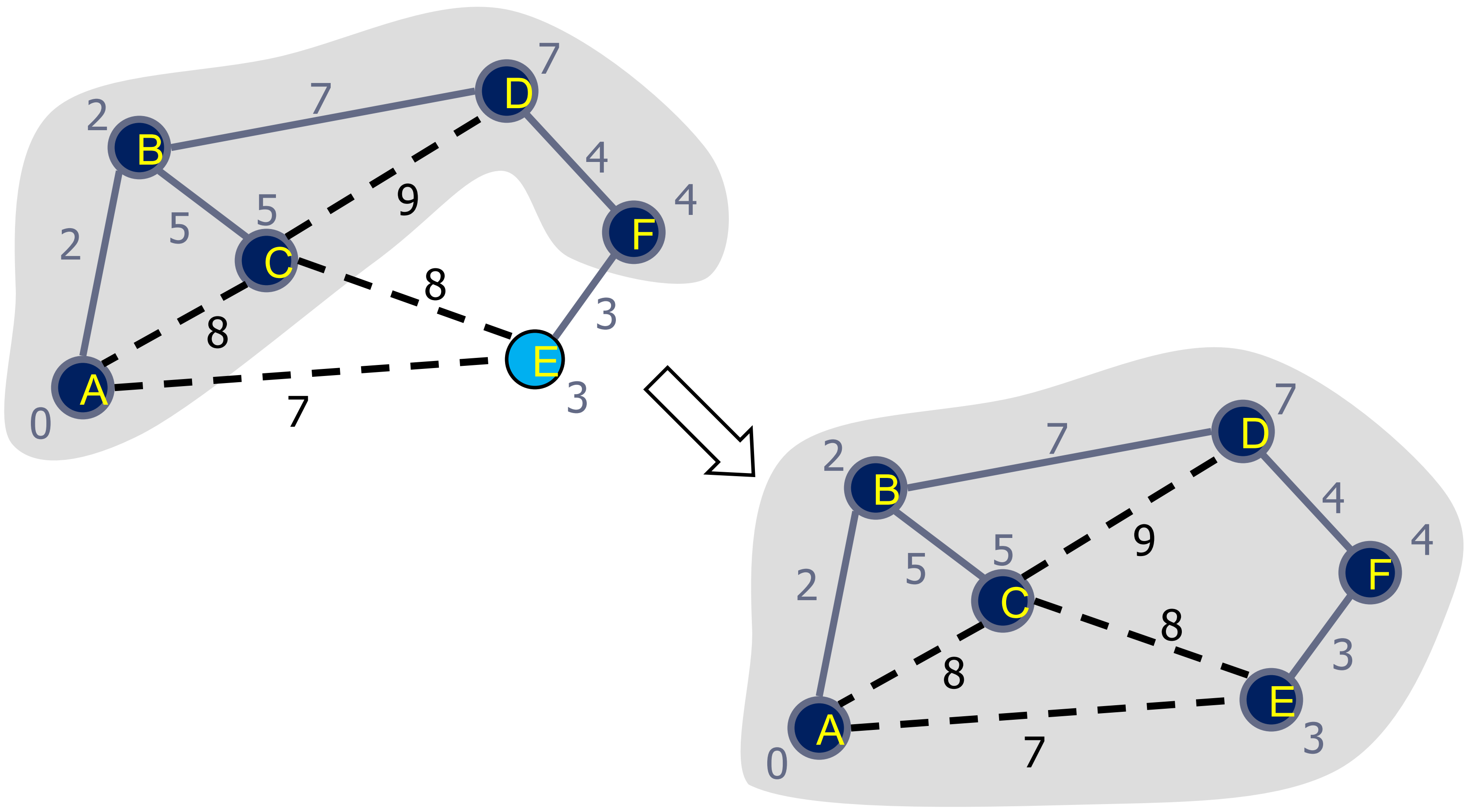
The MST Algorithm (Prim’s algorithm)

1. Start with any vertex and add it to the minimum spanning tree.
2. Add newly available edges to a priority queue.
3. Pick the edge with the lowest weight, remove it from the priority queue, and print it out as part of the solution. The vertex it leads to is added to the minimum spanning tree.
4. You don’t want to visit a vertex more than once. Therefore, when you visit a vertex, make sure to remove all of the remaining edges that connect already-visited vertices.
5. Keep going until every vertex has been visited. The graph is now completely connected.

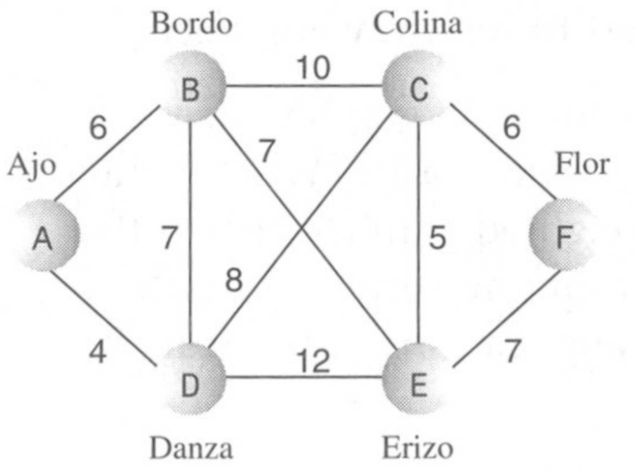
Example

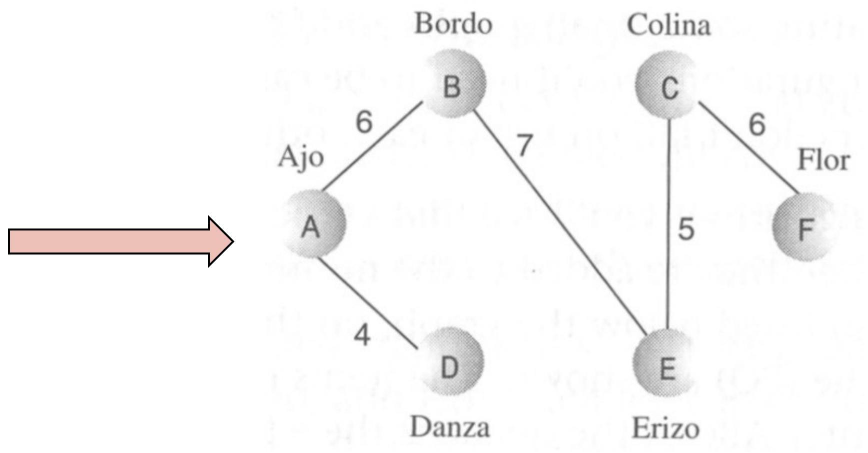

A
B
C
D
E
F
0
2
8
∞
∞
7
2
7
5
8
9
4
3
8
7
0
2
5
7
∞
7
2
7
5
8
9
4
3
8
7
0
2
5
7
∞
7
2
7
5
8
9
4
3
8
7
0
2
5
7
4
7
2
7
5
8
9
4
3
8
7

Example (contd.)



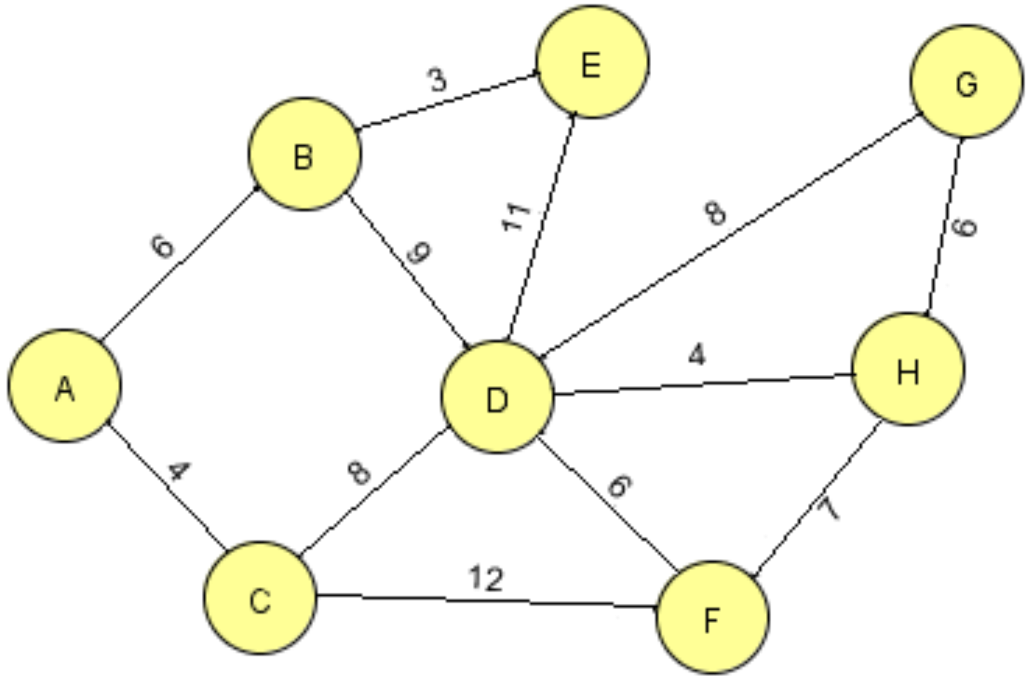
Example 2





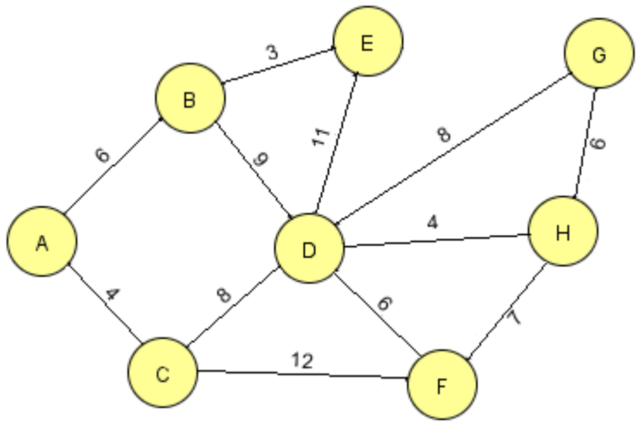
Vertices Visited	Priority Queue	Edge Selected
A	AD4, AB6	AD4
A D	AB6, DB7, DC8, DE12	AB6
A D B	BE7, DC8, BC10, DE12	BE7
A D B E	EC5, EF7, DC8, BC10	EC5
A D B E C	CF6, EF7	CF6
A D B E C F	-	-

Example 2



Find a minimum spanning tree Start at A

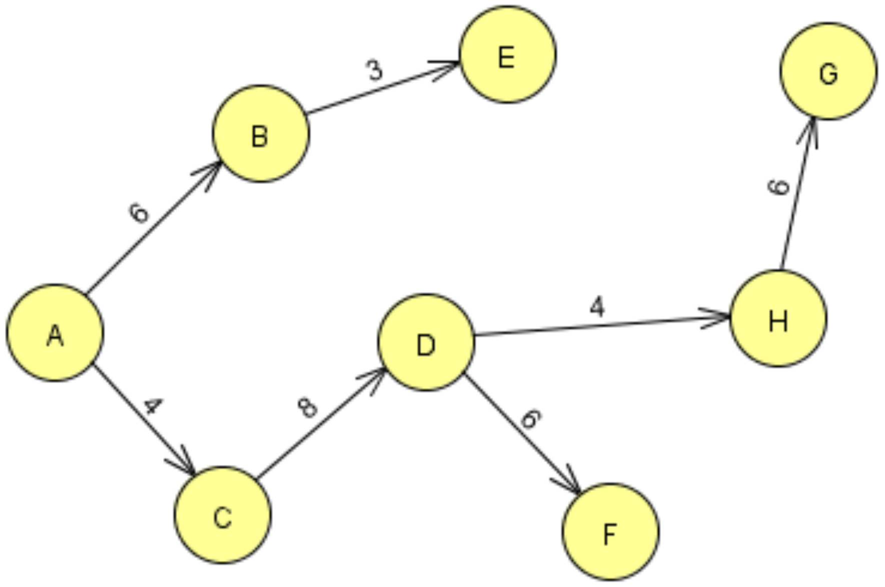
Worked Solution



Vertices Visited	Priority Queue	Edge Selected
A	AC4, AB6	AC4
A C	AB6, CD8, CF12	AB6
A C B	BE3, CD8, BD9, CF12	BE3
A C B E	CD8, ED11, CF12	CD8
A C B E D	DH4, DF6, DG8, CF12	DH4
A C B E D H	DF6, HG6, HF7, DG8, CF12	DF6
A C B E D H F	HG6, DG8	HG6

Just keep selecting the shortest edge in the priority queue
Then add in the new potential edges leading from that vertex

Solution



7 edges needed to connect 8 vertices
All vertices connected using minimum weight

Questions