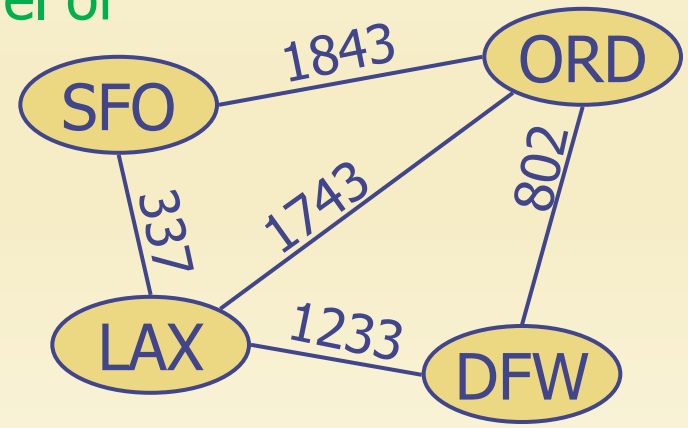
Lesson 15
Intro to Graphs,DFS and BFS

Combinatorics of Pure Intelligence

Wholeness of the Lesson

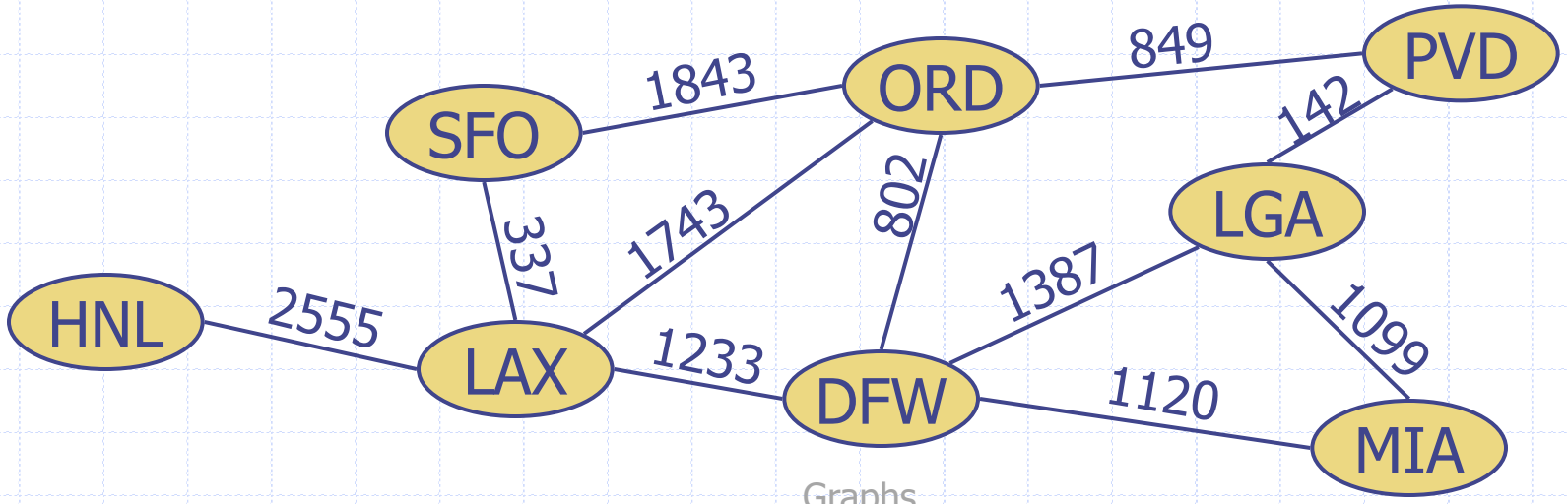
Graphs are data structures that do more than simply store and organize data; they are used to model interactions in the world. This makes it possible to make use of the extensive mathematical knowledge from the theory of graphs to solve problems abstractly, at the level of the model, resulting in a solution to real-world problems.

Science of Consciousness: Our own deeper levels of intelligence exhibit more of the characteristics of Nature's intelligence than our own surface level of thinking. Bringing awareness to these deeper levels, as the mind dives inward, engages Nature's intelligence, Nature's know-how, and this value is brought into daily activity. The benefit is greater ability to solve real-world problems, meet challenges, and find the right path for success.

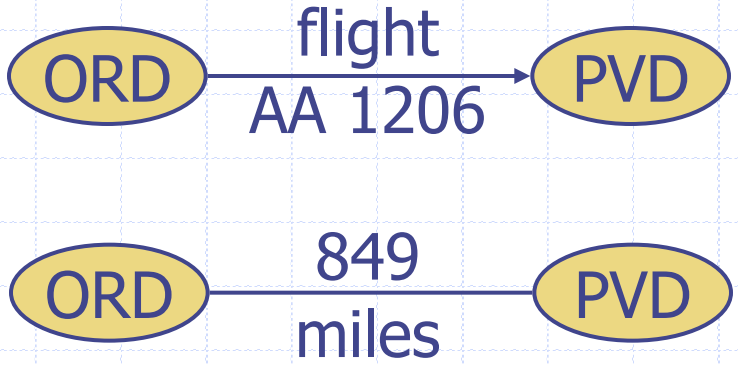


Graph

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges can be implemented so that they store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code(SFO, ORD, ..)
  - An edge represents a flight route between two airports and stores the mileage of the route.



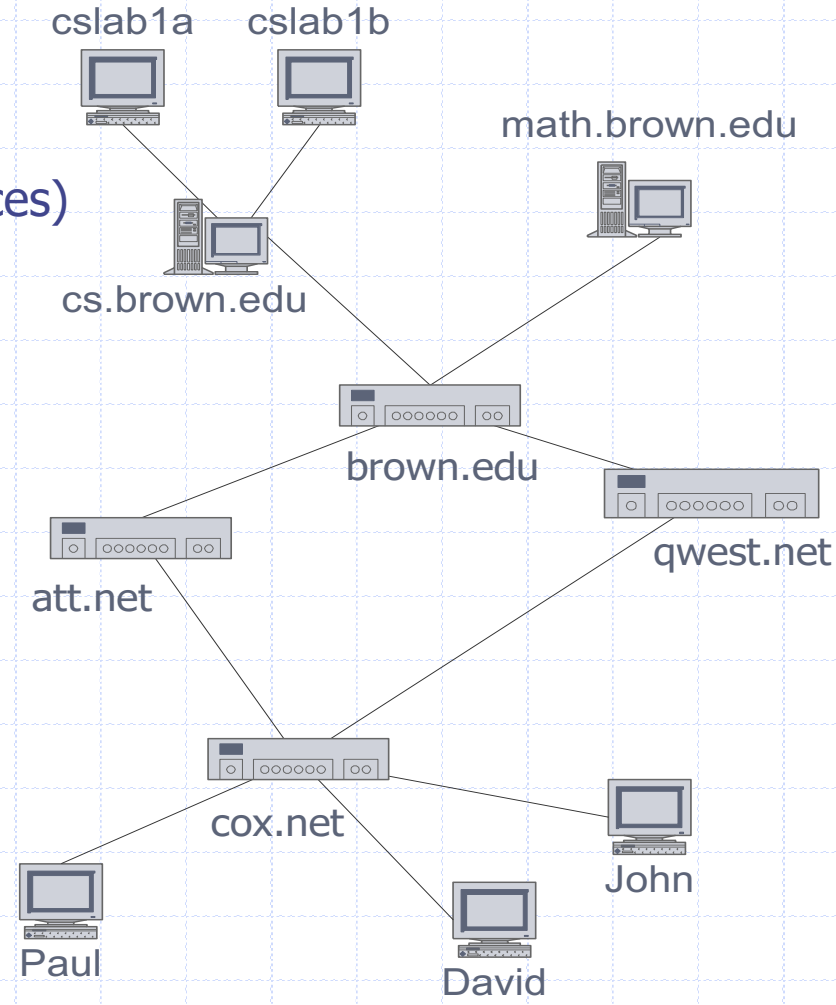
Edge Types



- Directed edge
  - ordered pair of vertices (u,v)→(ORD,PVD)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
  - form directed graphs
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
  - form undirected graphs
- Weighted edge(next lesson)
  - given a weight
  - weight could represent cost, distance, etc.
  - form weighted graphs
- Unweighted edge
  - no weight on it
  - form unweighted graphs

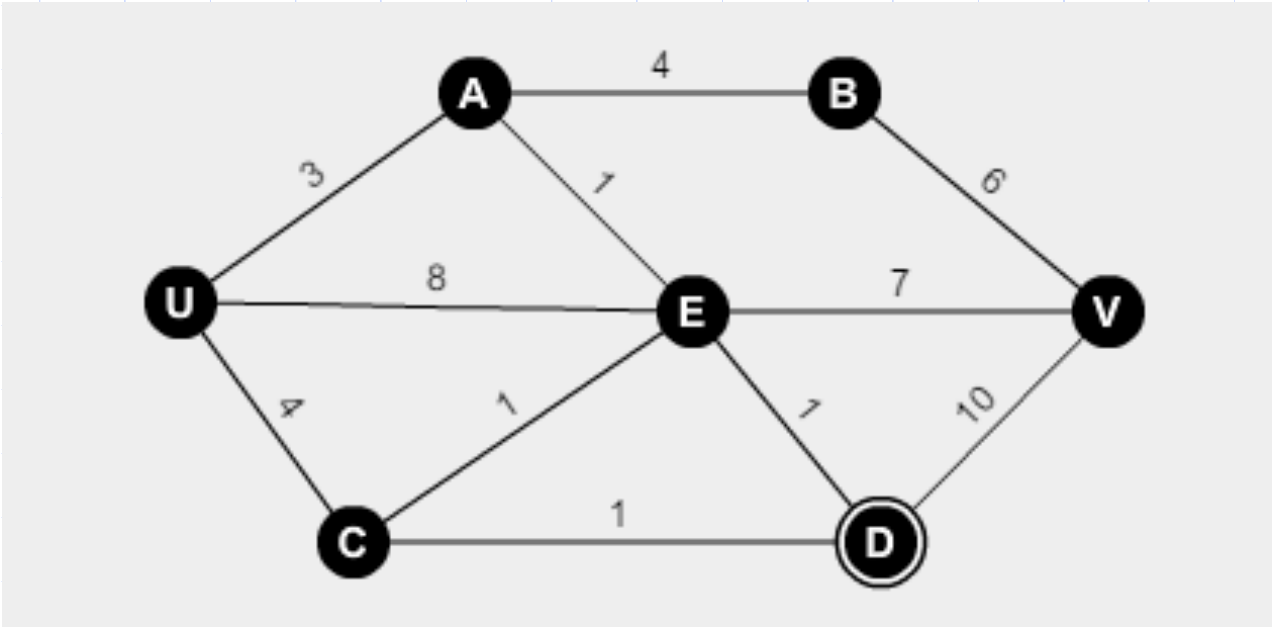
Applications

- **Electronic circuits**
  - Printed circuit board (nodes = junctions, edges are the traces)
- **Transportation networks**
  - Highway network
  - Flight network
- **Computer networks**
  - Local area network
  - Internet
  - Web
- **Databases**
  - Entity-relationship diagram
- **Physics / Chemistry**
  - Atomic structure simulations (e.g. shortest path algs)
  - Model of molecule -- atoms/bonds



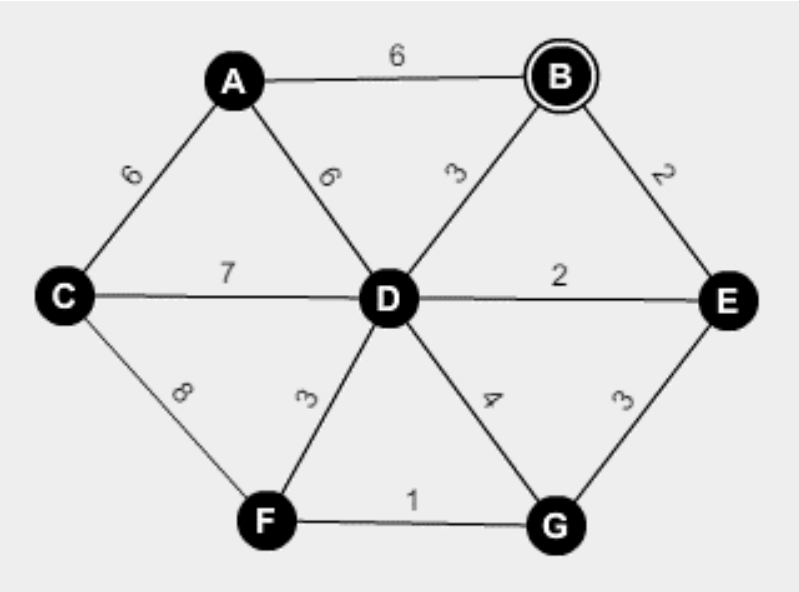
Examples

SHORTEST PATH. The diagram below schematically represents a railway network between cities; each numeric label represents the distance between respective cities. What is the shortest path from city U to city V? Devise an algorithm for solving such a problem in general. ((next lesson)

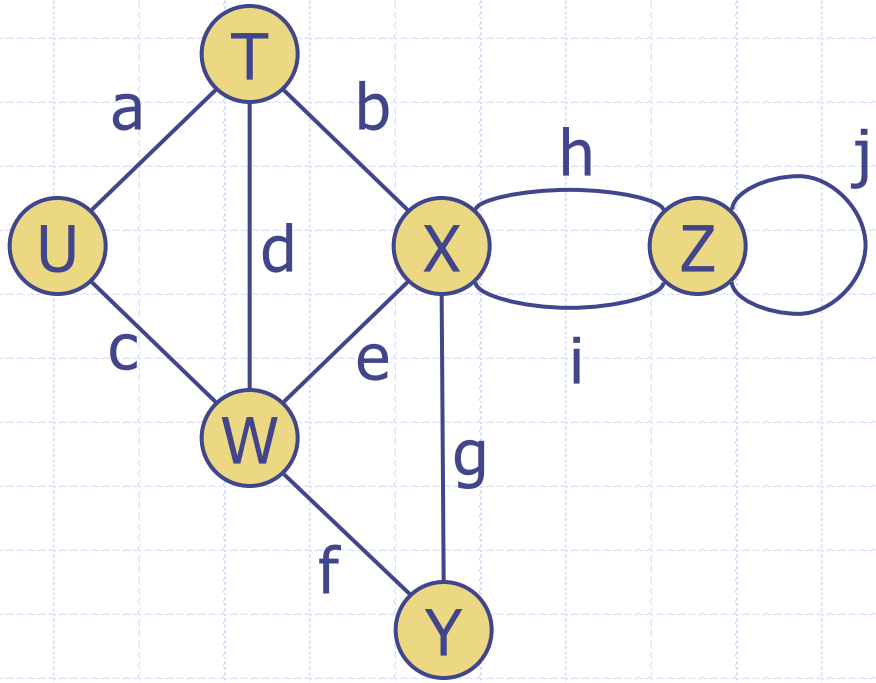


Examples (continued)

CONNECTOR. The diagram below schematically represents potential railway paths between cities; a numeric label represents the cost to lay the track between the respective cities. What is the least costly way to build the railway network in this case, given that it must be possible to reach any city from any other city by rail? Devise an algorithm for solving such a problem in general.



Terminology

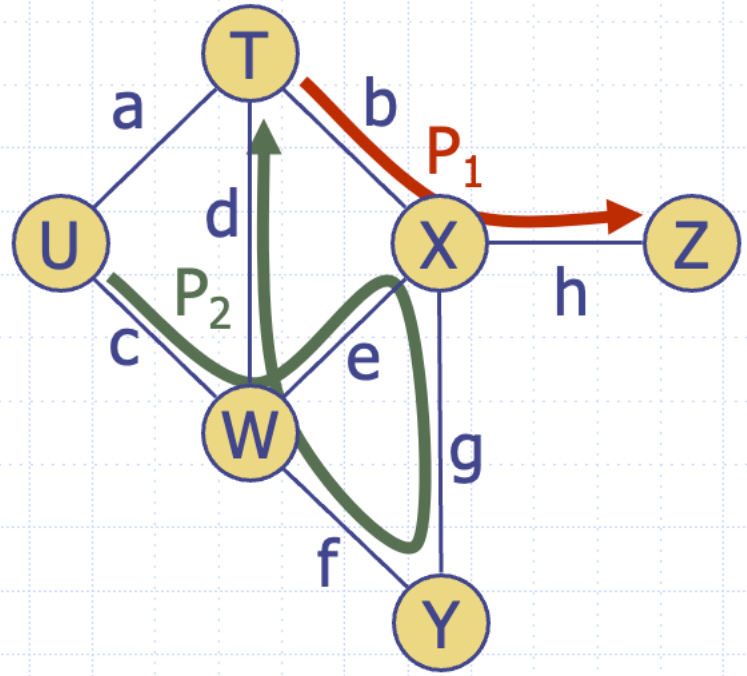


- **|V| (or *n*)** is the number of vertices of G;
  |E| (or *m*) is the number of edges.
- **End vertices** (or endpoints) of an edge
  - U and T are the endpoints of a
- **Edges incident to a vertex**
  - a, d, and b are incident to T
- **Adjacent vertices**
  - U and T are adjacent
- **Degree of a vertex**: number of edges incident to it
  - X has degree 5 denoted as: deg(X) = 5
- **Parallel edges**
  - h and i are parallel edges

- **Self-loop**
  - j is a self-loop
- **Simple Graph**
  - A simple graph is a graph that has no self-loops or parallel edges

Terminology (cont.)

- **Path**
  - sequence of alternating vertices and edges.
  - In a simple graph, can omit edges
  - begins and ends with a vertex
  - length of a path is number of edges

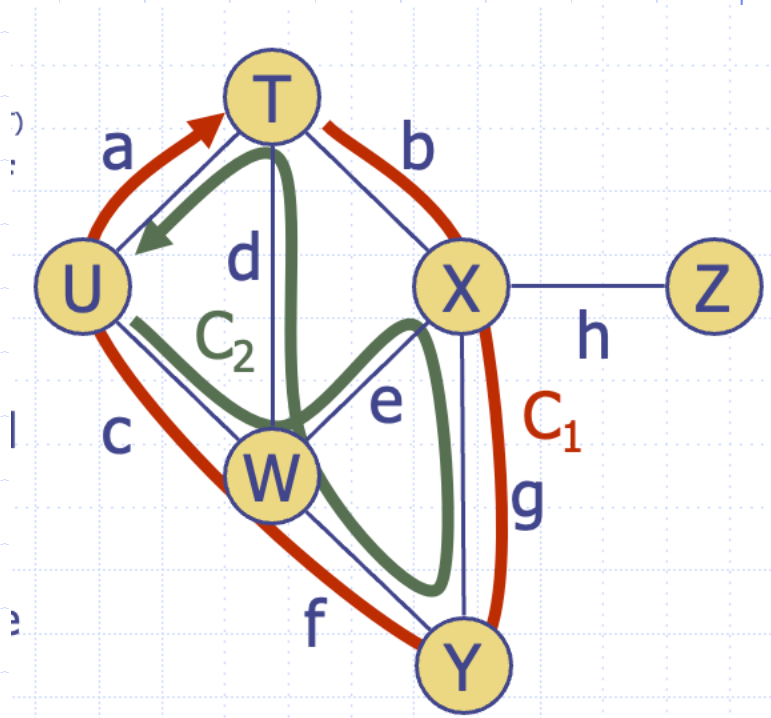


Examples

- P_1=(T, X, Z) is a simple path, length of P1 = 2
- P_2=(U, W, X, Y, W, T) is a path that is not simple. [W repeated]

Terminology (cont.)

- **Cycle**
  - path in which all edges are distinct and whose first and last vertex are the same
  - *length* of a cycle is the number of edges in the cycle
- Examples
  - C_1=(T, X, Y, W, U, T) is a simple cycle
  - C_2=(U, W, X, Y, W, T, U) is a cycle that is not simple



Properties

Convention: Focus on simple_undirected graphs at first

Property 1

$$\Sigma_v \deg(v) = 2m$$

Proof: Let E_v = {e | e incident to v}.

Then $\Sigma_v \deg(v) = \Sigma_v |E_v|$

Notice every edge (v,w) belongs to just two of these sets: E_v and E_w.
So every edge is counted exactly twice.

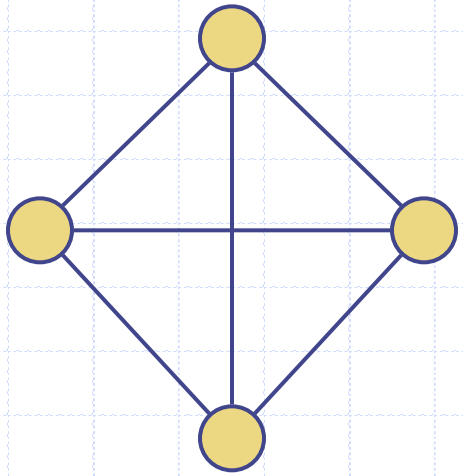
Property 2

$$m \leq n(n-1)/2$$

Proof: max number of edges is
C(n , 2) = $C_{n,2} = n(n-1)/2$

Notation:

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v

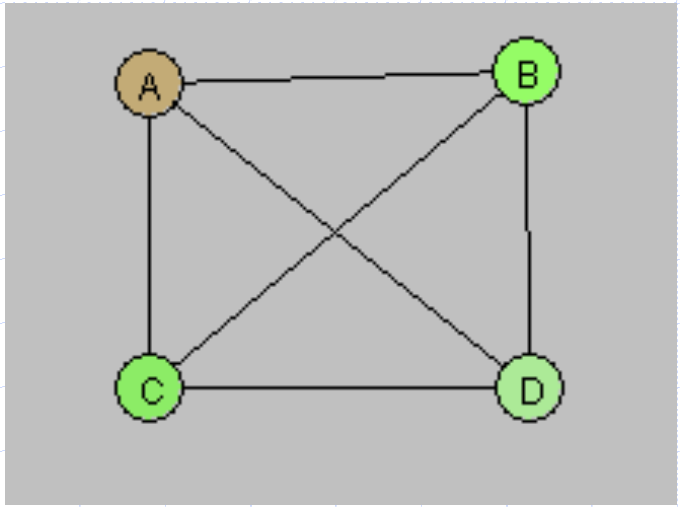


Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

Complete Graphs

- A graph G is complete if for every pair of vertices (u,v), there is an edge (u,v) in G. i.e the graph has the maximum number of edges.
- This is the complete graph on 4 vertices, denoted K_4.



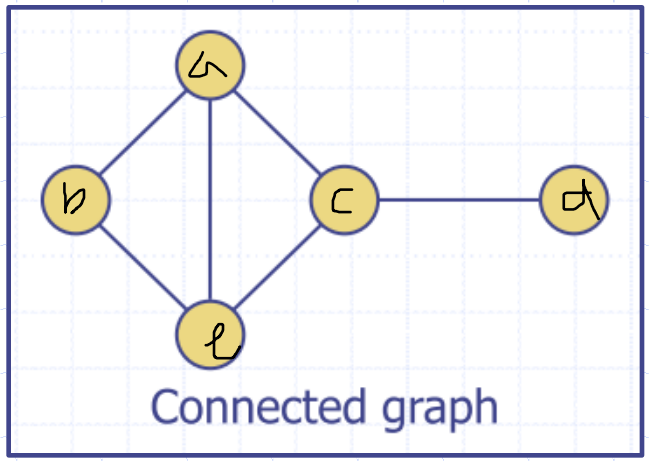
- In general, the complete graph on n vertices is denoted as K_n.
- For a complete graph G,

  $m = n(n-1)/2$

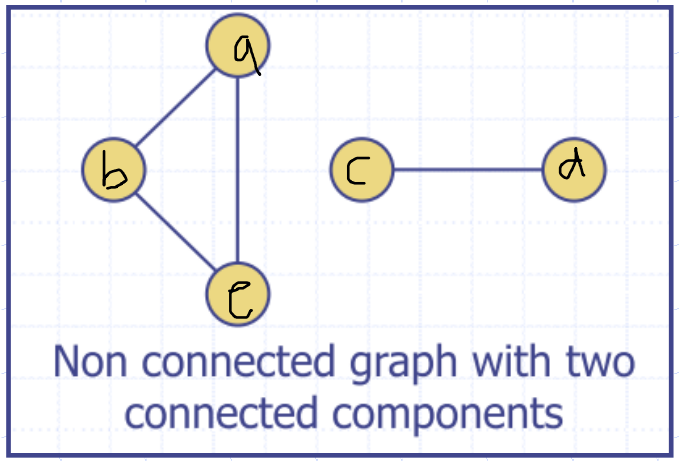
  that is to say, K_n has exactly $n(n-1)/2$ edges.

Connected Graphs And Connected Components

- A graph is connected if for any two vertices u,v in G, there is a path from u to v. Example path from a to d is a-c-d.
- **Observation** If a graph G=(V,E) is not connected, then G can be partitioned into different components, each component is connected and is connected to no additional vertices in the super graph. They are called the connected components of G.



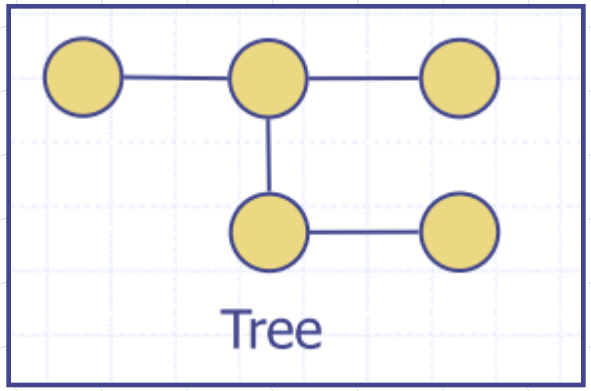
Connected graph

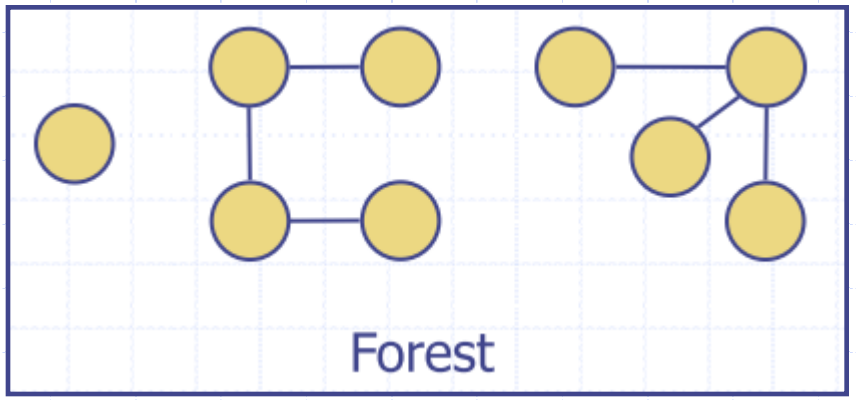


Non connected graph with two connected components

Trees and Forests

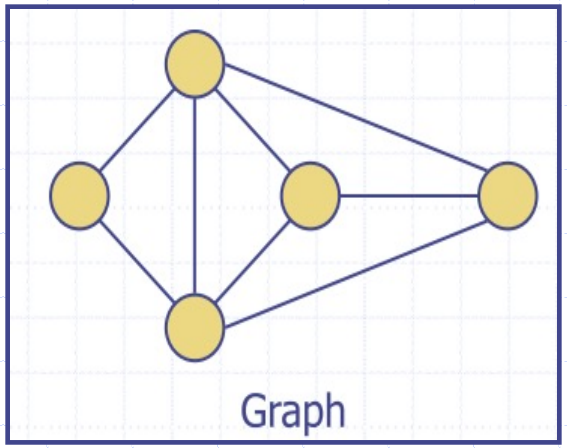
- A graph is **acyclic** if it contains no cycle.
- An acyclic connected undirected graph is called a **tree**.
- **Theorem**. If G is a tree, m = n - 1.
- A **forest** is an undirected graph without cycles. It's a disjoint union of trees.

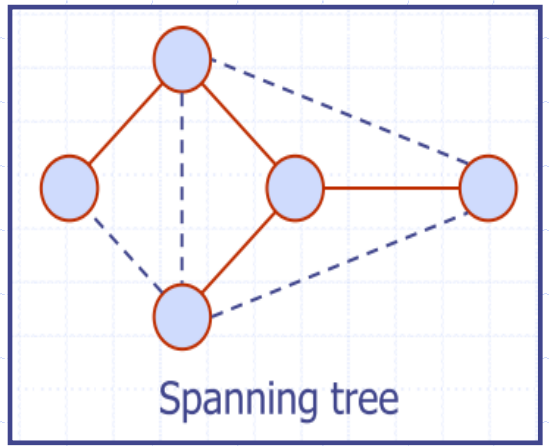


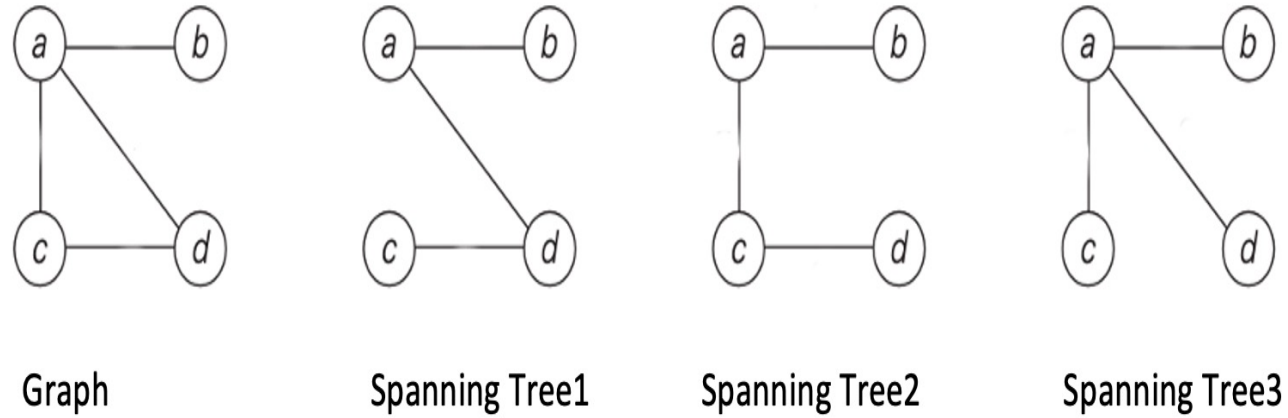


Spanning Trees and Forests

- A ***spanning tree*** of a graph is a sub graph of the original graph which is a tree that contains all vertices of the original graph.
- A spanning tree is not unique unless the graph is already a tree. You may get more than one spanning tree for a graph.
- Will discuss Spanning Tree Algorithms to realize the graph applications.

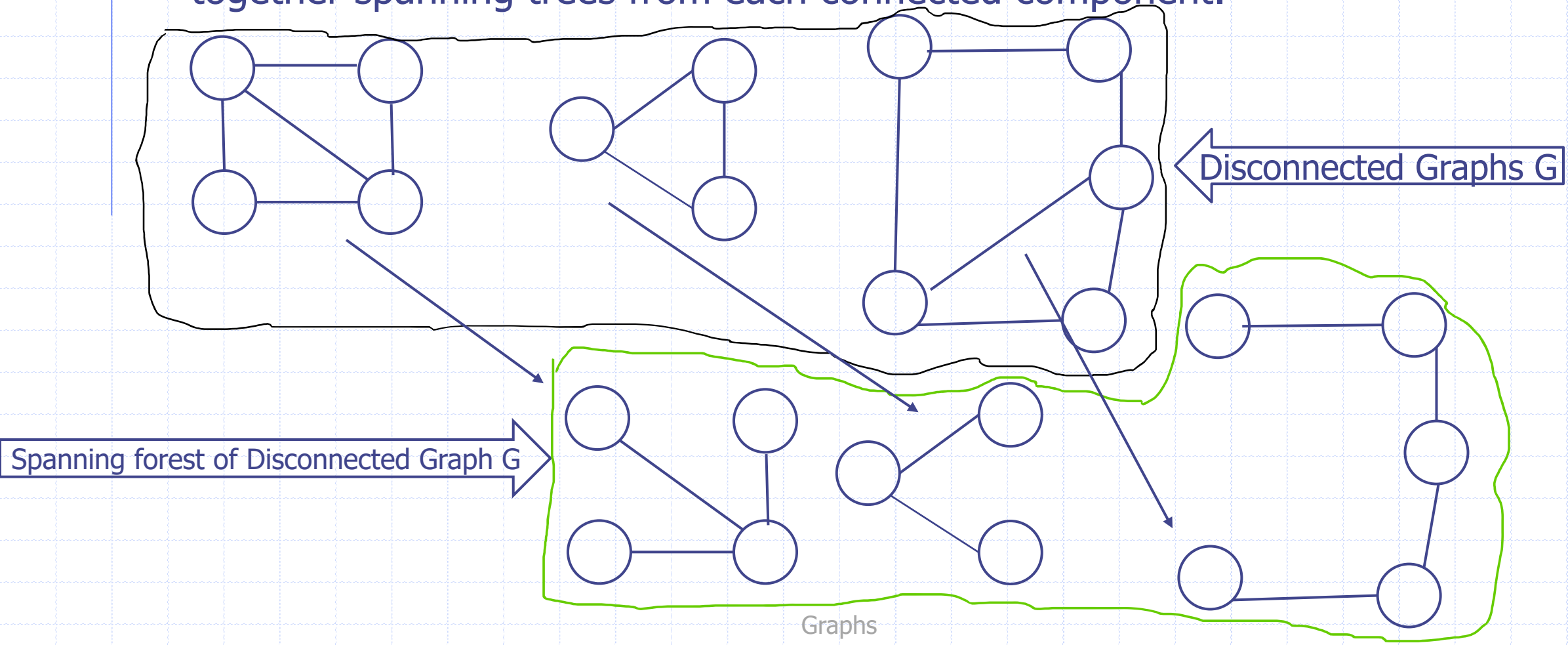






Spanning Trees and Forests

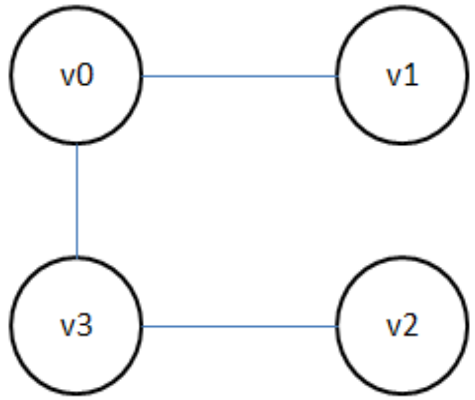
- If the graph is not connected, a spanning forest will be obtained by putting together spanning trees from each connected component.



Determining Adjacency

- Most graph algorithms rely heavily on determining whether two vertices are adjacent and on finding all vertices adjacent to a given vertex.
- These operations should be as efficient as possible.
- *Two ways to represent adjacency.*
  - Adjacency Matrix
  - Adjacency List

Determining Adjacency



- An ***Adjacency Matrix A*** is a two-dimensional *n x n* array consisting of 1's and 0's. A 1 at position *A[i][j]* means that vertices v_i and v_j are adjacent; 0 indicates that the vertices are not adjacent.

	v0	v1	v2	v3
v0	0	1	0	1
v1	1	0	0	0
v2	0	0	0	1
v3	1	0	1	0

- An ***Adjacency List*** is a table that associates to each vertex *u* the set of all vertices in the graph that are adjacent to *u.*

V_0	V_1, V_3
V_1	V_0
V_2	V_3
V_3	V_0, V_2

Determining Adjacency

- If there are relatively few edges, an adjacency matrix uses too much space.
  - **Best to use adjacency list when number of edges is relatively small.**
- If there are many edges, determining whether two vertices are adjacent becomes costly for an adjacency list.
  - **Best to use adjacency matrix when there are many edges.**

Sparse Graphs Vs Dense Graphs

- Recall the maximum number of edges in a graph is n(n - 1)/2, where n is the number of vertices.
- A graph is said to be **dense** if it has $O(n^2)$ edges. It is said to be **sparse** if it has O(n) edges.
- A dense graph is one where there are many edges, but not necessarily as many as in a complete graph.
- Sparse graphs are sparsely connected (eg. Trees)
- **Strategy. Use adjacency lists for sparse graphs and adjacency matrices for dense graphs.**

Graph Traversal Algorithms

- In computer science, graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph.

- **Motivation.** Graph traversal algorithms provide efficient procedures for visiting every vertex in a graph. There are many practical reasons for doing this.
  - Telephone network – check whether there is a break in the network

Depth-First Search

- The basic DFS strategy:
  - Pick a starting vertex and visit an adjacent vertex, and then one adjacent to that one, until a vertex is reached that has no further unvisited adjacent vertices.
  - Then backtrack to one that does have unvisited adjacent vertices and follow that path.
  - Continue in this way till all vertices have been visited.

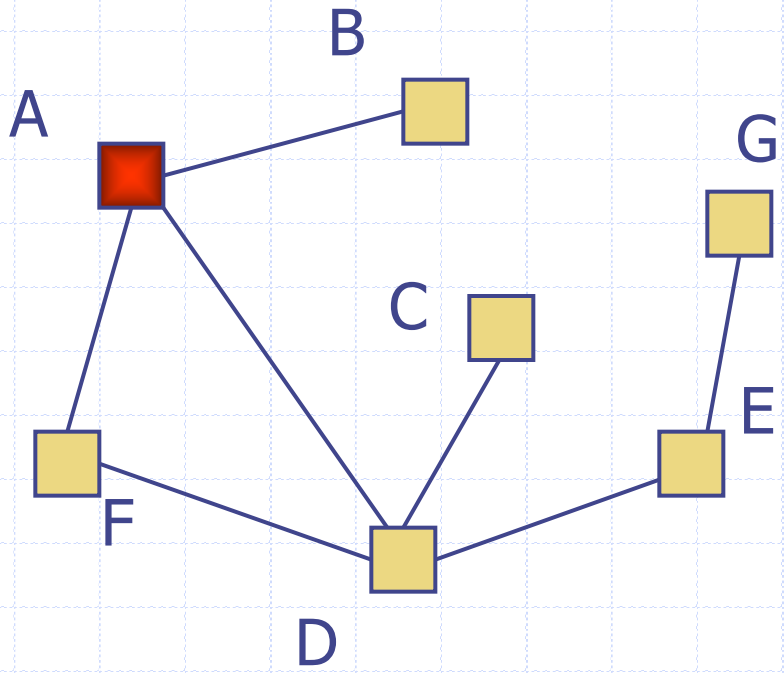
Depth-First Search

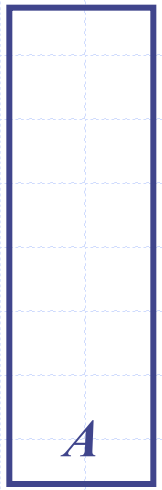
Algorithm: Depth First Search (DFS)
Input: A simple connected undirected graph G = (V,E)
Output: G, with all vertices marked as visited.

```
Initialize a stack S    //supports backtracking
Pick a starting vertex s and mark it as visited
S.push(s)
while S ≠ ∅ do
    v ← S.peek()
    if some vertex adjacent to v not yet visited then
        w ← next unvisited vertex adjacent to v
        mark w
        push w onto S
    else //if can't find such a w, backtrack
        S.pop()
```

Worked Example

Start with Vertex A, mark as visited and push into stack.



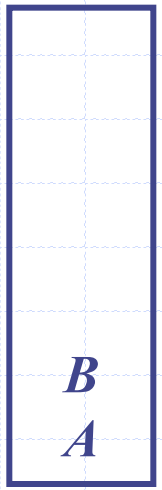


Stack

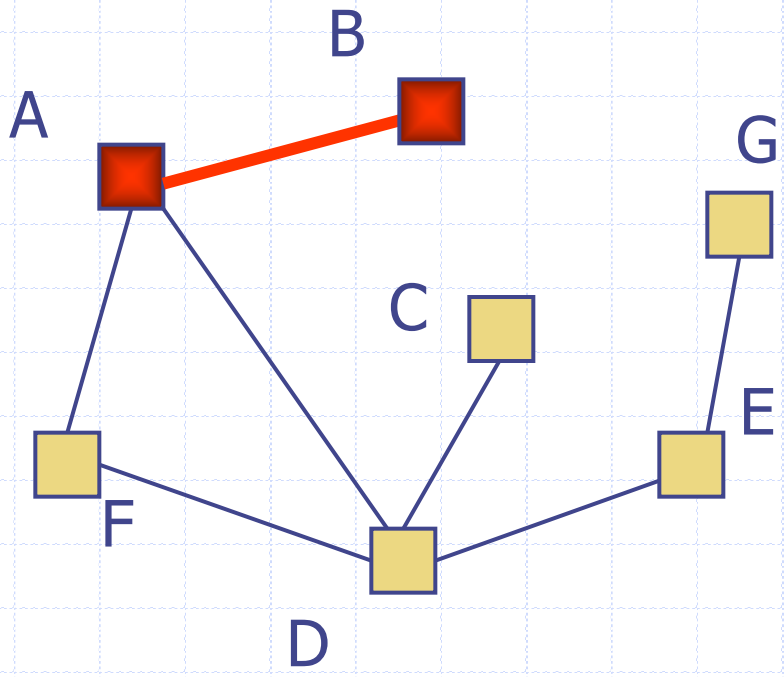
Spanning Tree:
T = {…}

Worked Example

Take a peek at the stack and find A; B is unvisited and adjacent to A; mark B as visited and push into stack.



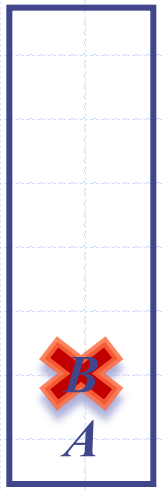
Stack



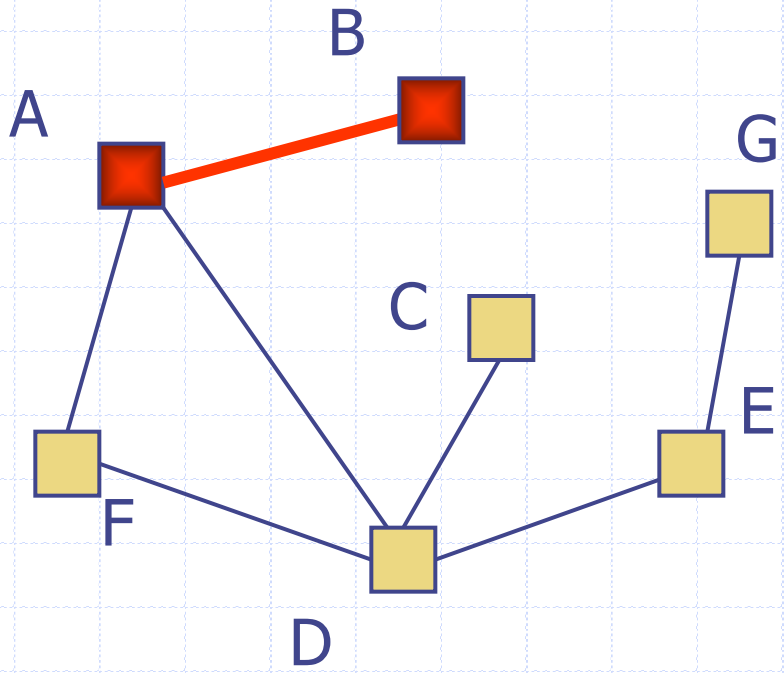
Spanning Tree:
T = {AB, …}

Worked Example

Take a peek at the stack and find B; B has no more unvisited vertices; pop the stack to remove B



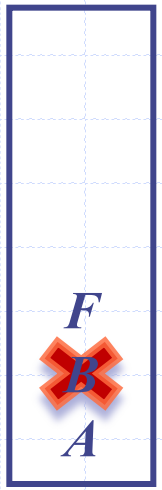
Stack



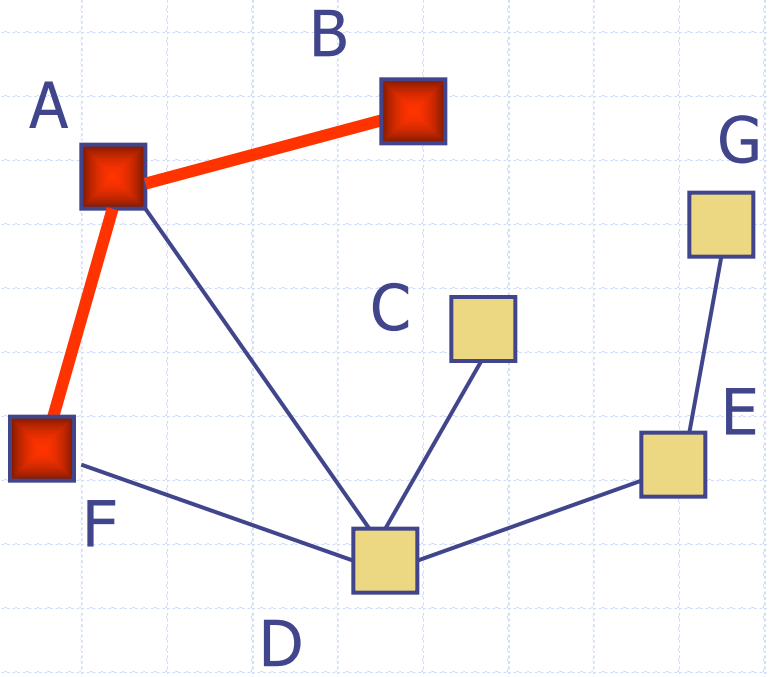
Spanning Tree:
T = {AB, …}

Worked Example

Take a peek at the stack and find A; F is adjacent to A and not yet visited; mark F as visited and push it into the stack



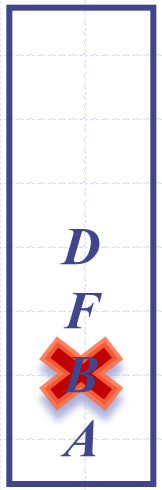
Stack



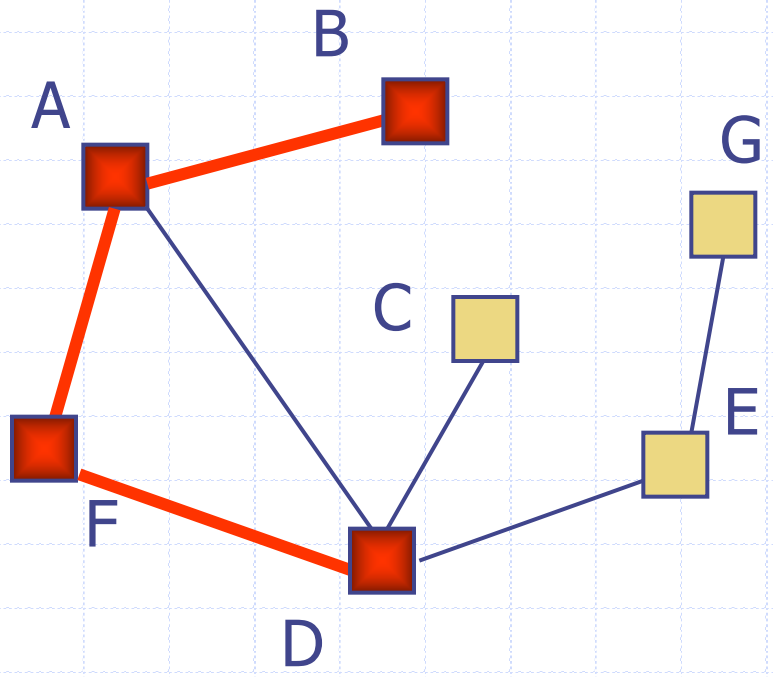
Spanning Tree: T = {AB, AF, …}

Worked Example

Take a peek at the stack and find F; D is adjacent to F and not yet visited; mark D as visited and push it into the stack



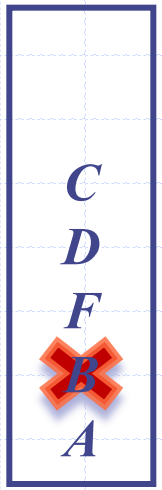
Stack



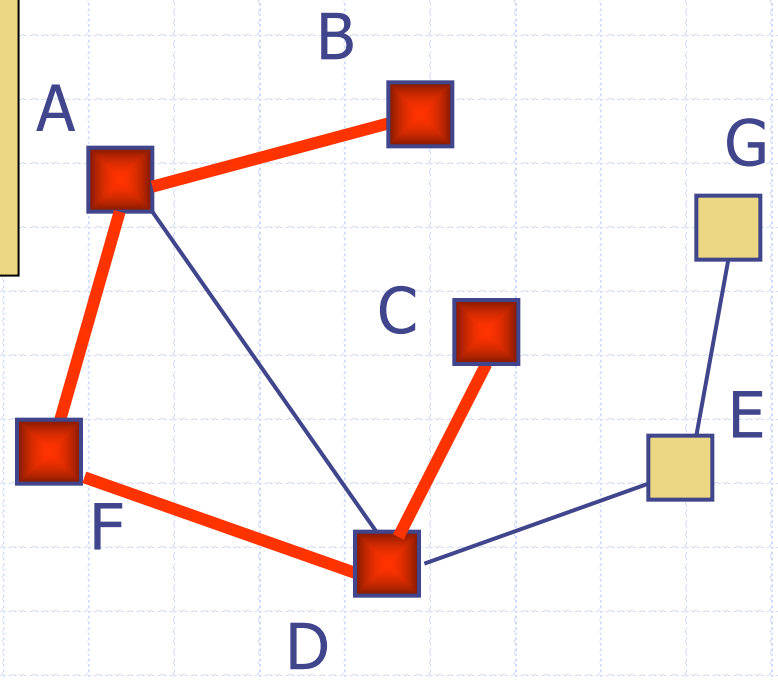
Spanning Tree: T = {AB, AF, FD, …}

Worked Example

Take a peek at the stack and find D; we do not go back to A since A was visited (avoids cycle creation); C is adjacent to D and not yet visited, so mark C and push it.



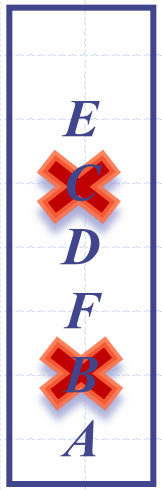
Stack



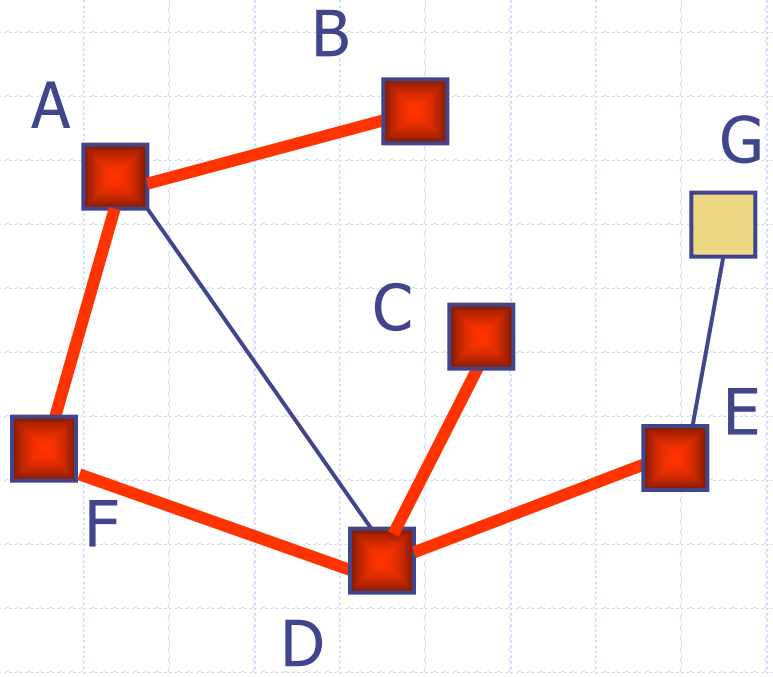
Spanning Tree: T = {AB, AF, FD, DC…}

Worked Example

Peek at the stack and find C; C has no more unvisited vertices, so pop C. Peek at the stack and find D; E is adjacent to D and not yet visited; mark E and push onto the stack



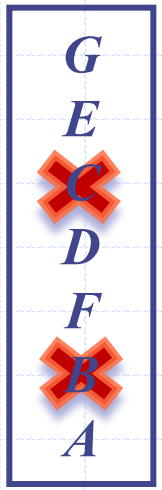
Stack



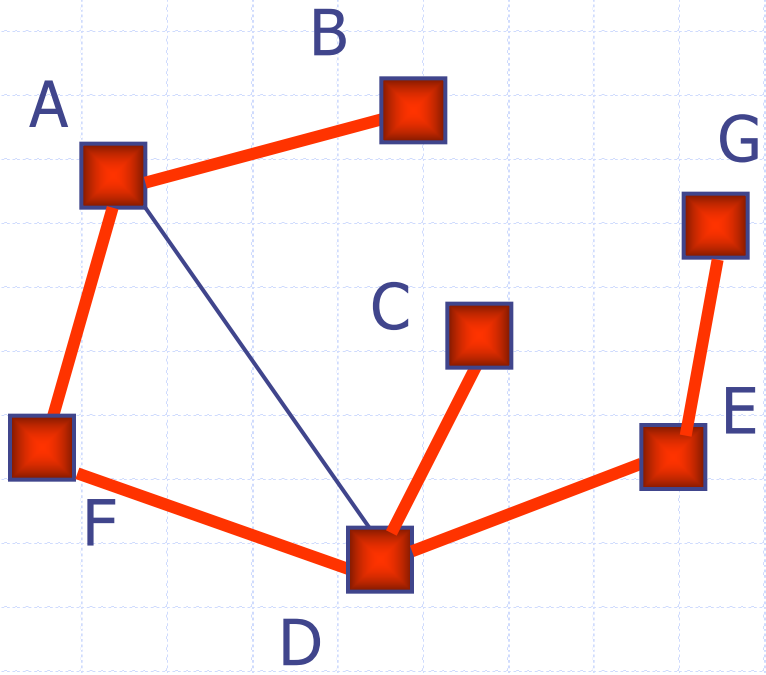
Spanning Tree: T = {AB, AF, FD, DC, DE…}

Worked Example

Peek at the stack and find E; G is adjacent to E and not yet visited; mark G and push onto the stack



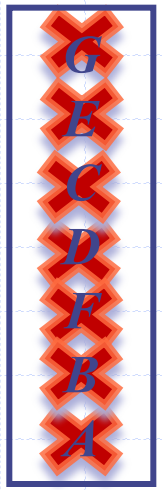
Stack



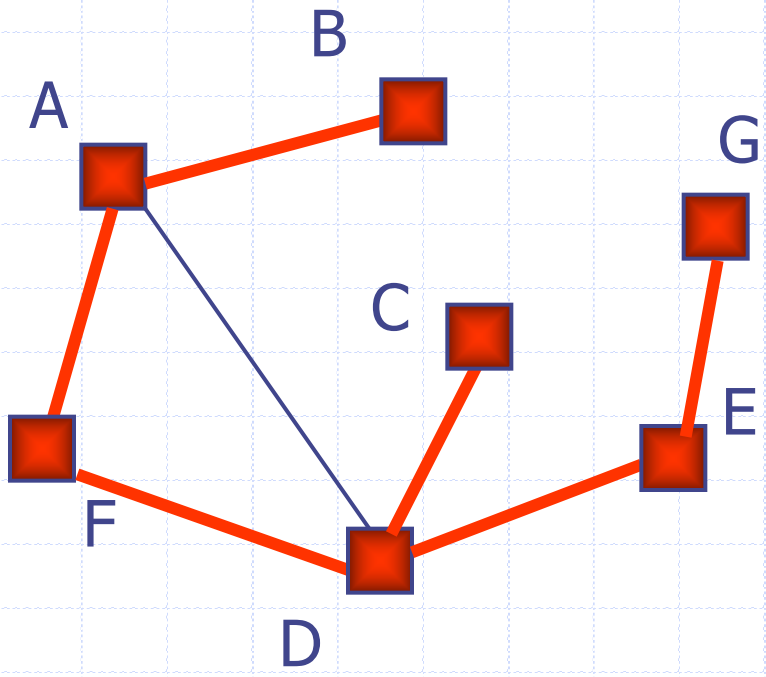
Spanning Tree: T =
{AB, AF, FD, DC, DE, EG}

Worked Example

Now we have no more unvisited vertices, so pop all vertices that are still in the stack.



Stack



Spanning Tree: T =
{AB, AF, FD, DC, DE, EG}

Depth-First Search

Algorithm: Depth First Search (DFS)
Input: A simple connected undirected graph G = (V,E)
Output: G, with all vertices marked as visited.

```
Initialize a stack S   //supports backtracking
Pick a starting vertex s and mark it as visited
S.push(s)
while S ≠ ∅ do
    v ← S.peek()
    if some vertex adjacent to v not yet visited then
        w ← next unvisited vertex adjacent to v
        mark w
        push w onto S
    else //if can't find such a w, backtrack
        S.pop()
```

Running Time for DFS

- Every vertex eventually is marked and pushed onto the stack, and then is eventually popped from the stack, and each of these occurs only once. Therefore, each vertex undergoes O(1) steps of processing.
- In addition, each vertex v will experience a peek operation, at which time the algorithm will search for an unvisited vertex adjacent to v. This peek step, together with the search, will take place repeatedly until every vertex adjacent to v has been visited – in other words, deg(v) times.
- Therefore, for each v, O(1) + O(deg(v)) steps are executed. The sum over all v in V is

$$\mathbf{O(\Sigma_v\,(1 + deg(v))) = O(n + 2m) = O(n+m)}$$

Finding a Spanning Tree with DFS

- A spanning tree for a connected graph can be found by using DFS and recording each new edge as it is discovered. If the graph is not connected, the algorithm can be done in each component to create a *spanning forest.*

Breadth-First Search

- An equally efficient way to traverse the vertices of a graph is to use the *Breadth First Search* (BFS) algorithm.
- As we saw in the depth-first search, the algorithm acts as though it wants to get as far away from the starting point as quickly as possible.
- In the breadth-first search, on the other hand, the algorithm likes to stay as close as possible to the starting point. It visits all the vertices adjacent to the starting vertex, and only then goes further ahead.
- This kind of search is implemented using a queue instead of a stack.

Breadth-First Search

- *BFS Idea:*
  - Pick a starting vertex.
  - Visit every adjacent vertex.
  - Then take each of those vertices in turn and visit every one of its adjacent vertices.
  - And so forth.

Breadth-First Search

Algorithm: Breadth First Search (BFS)

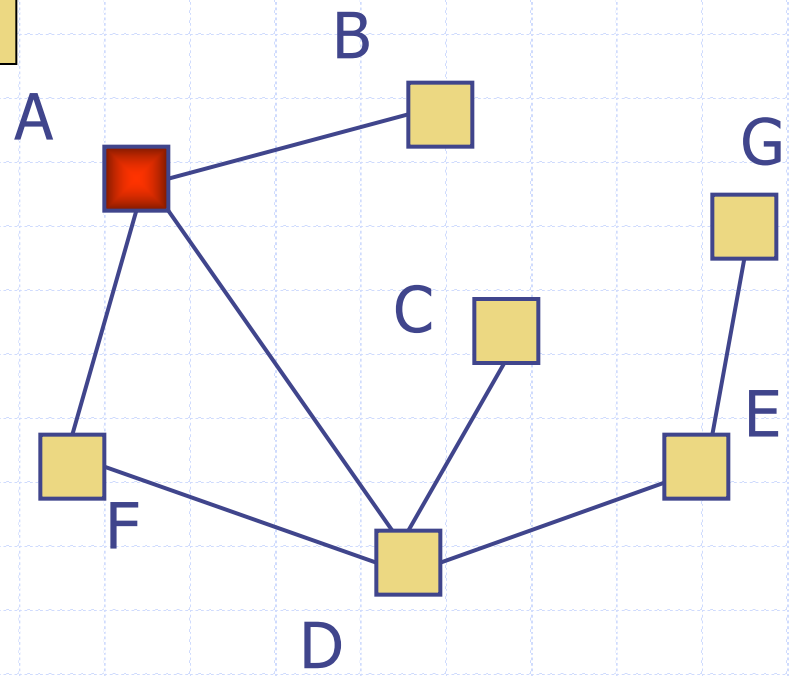
Input: A simple connected undirected graph G = (V,E)

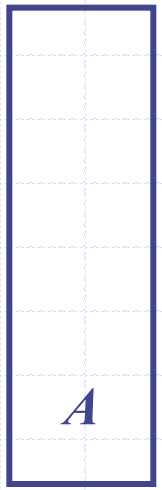
Output: G, with all vertices marked as visited.

```
Initialize a queue Q
Pick a starting vertex s and mark s as visited
Q.add(s)
while Q ≠ ∅ do
    v ← Q.dequeue()
    for each unvisited w adjacent to v do
        mark w
        Q.add(w)
```

Worked Example

Start with A, mark it as visited and add it to queue.



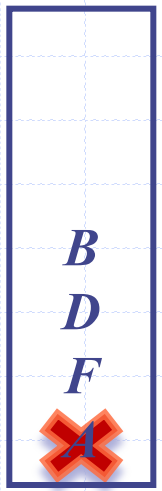


Queue

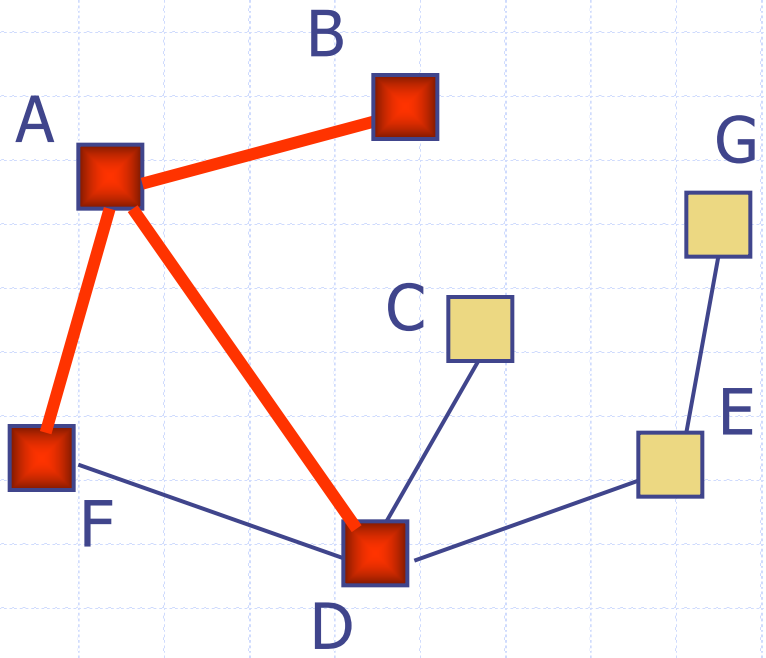
Spanning Tree: T = {…}

Worked Example

Dequeue (removes A); B, D and F are adjacent to A and not yet visited, so mark all three and add them to queue.



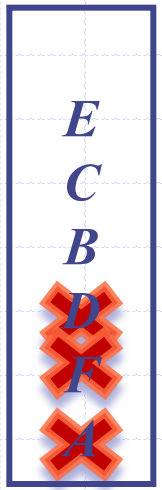
Queue



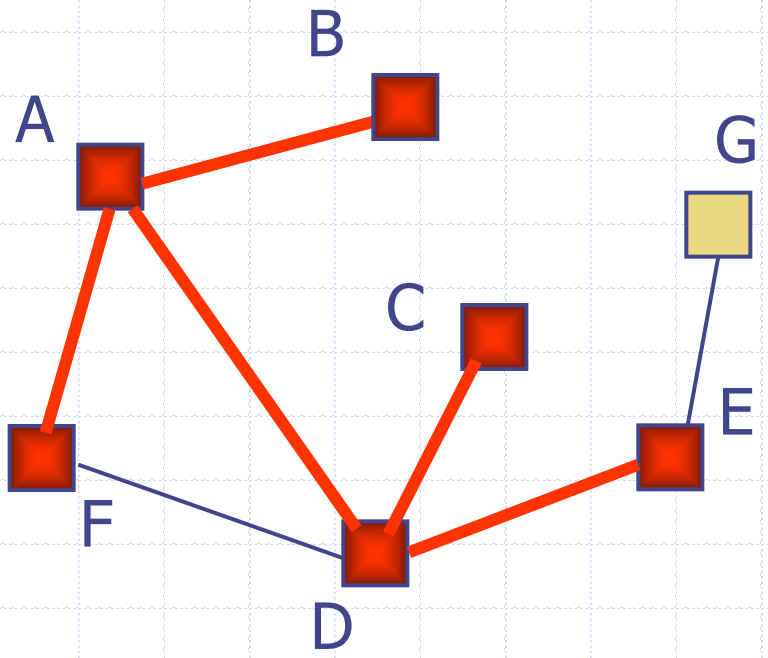
Spanning Tree: T = {AB, AD, AF, ...}

Worked Example

Dequeue (removes F); D is adjacent to F but already visited, so we don't visit D again (avoids cycle creation)
Dequeue (removes D); C and E are adjacent to D and not yet visited, so mark them and add to the queue.



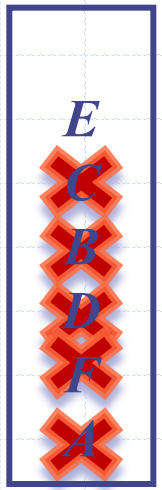
Queue



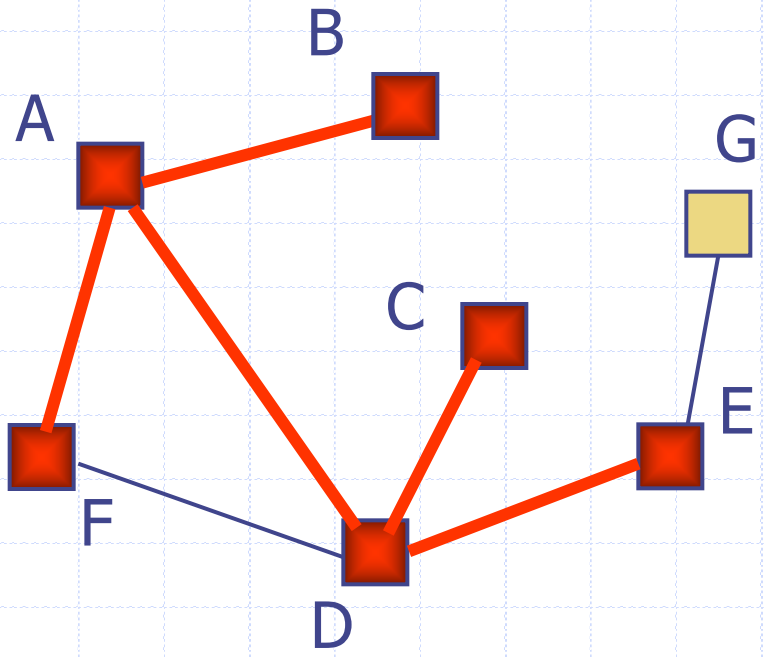
Spanning Tree: T =
{AB, AD, AF, DC, DE ...}

Worked Example

Dequeue twice (removing B and C); B and C have no more unvisited adjacent vertices.



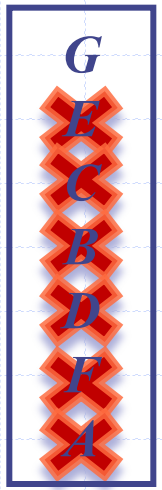
Queue



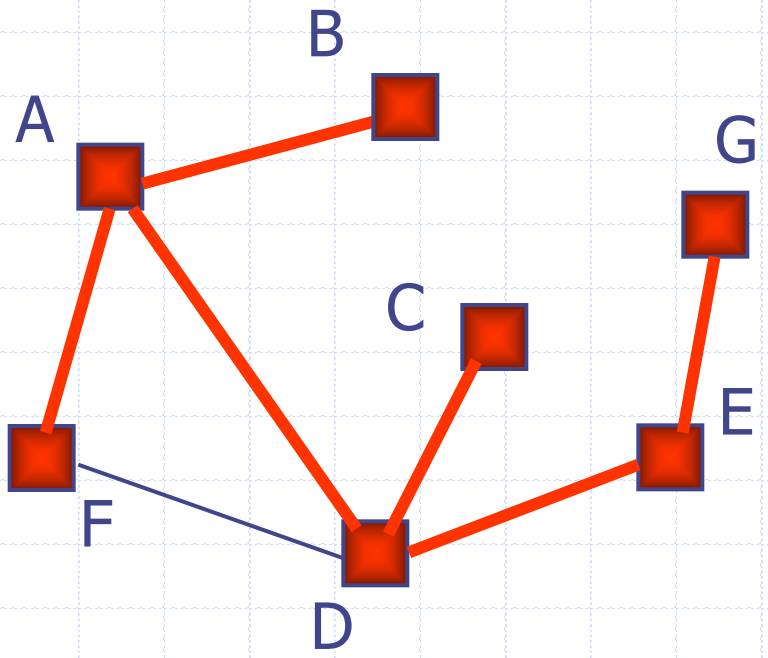
Spanning Tree: T =
{AB, AD, AF, DC, DE ...}

Worked Example

Dequeue (removes E); G is adjacent to E and not yet visited. Mark G and add it to queue.



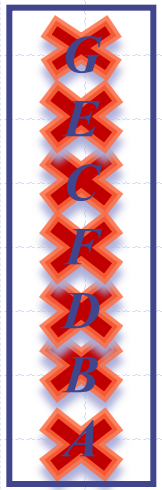
Queue



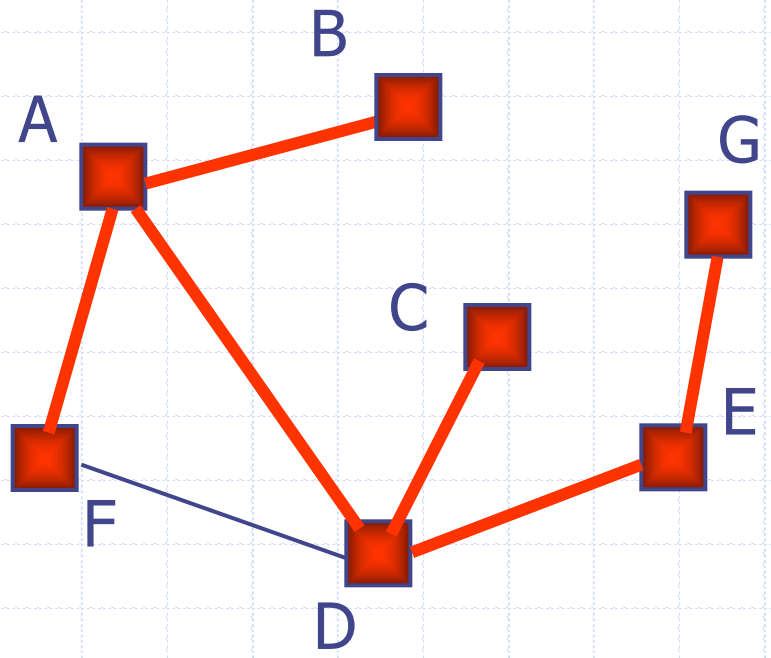
Spanning Tree: T =
{AB, AD, AF, DC, DE, EG}

Worked Example

Dequeue (removes G); The queue is now empty, the algorithm stops.



Queue



Spanning Tree: T =
{AB, AD, AF, DC, DE, EG}

Running time for BFS

- Analysis is similar to that for DFS.
- In the outer loop, 2 steps of processing occur on each vertex v, and then all edges incident to v are examined (some are discarded, others are processed).
- Therefore, for each v, O(1) + O(deg(v)) steps are executed. The sum is then

$$O(\sum v\ (1 + \deg(v))) = O(n + 2m) = O(n+m)$$

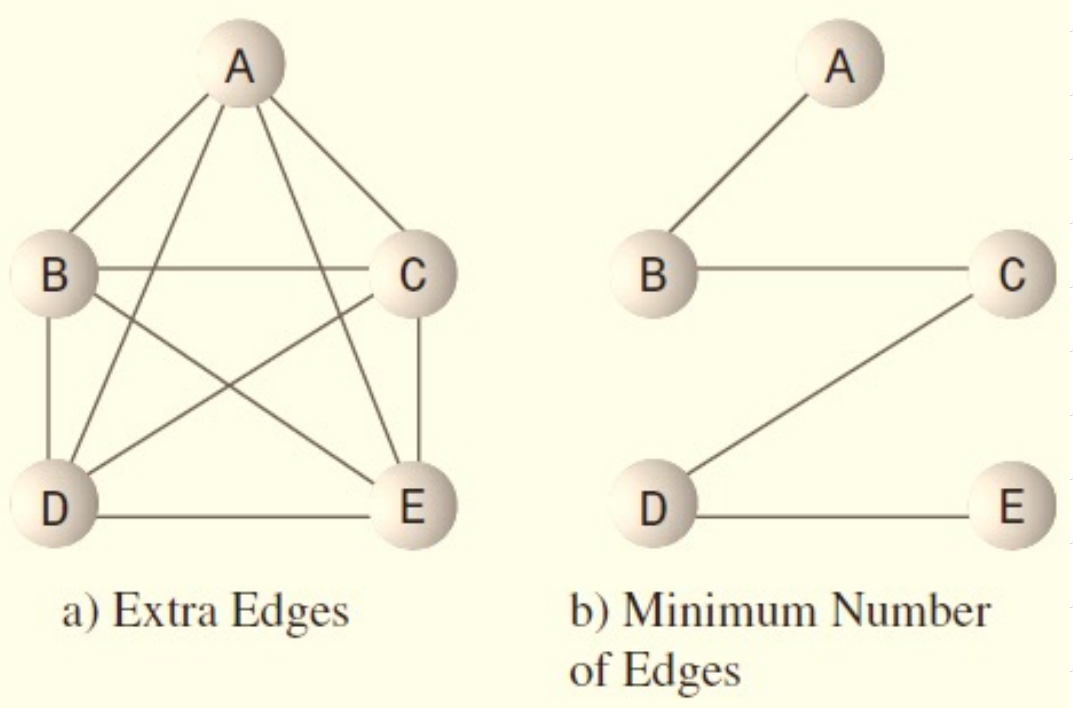
BFS

- BFS has an interesting property: It first finds all the vertices that are one edge away from the starting point, then all the vertices that are two edges away, and so on.
- This is useful if you're trying to find the shortest path from the starting vertex to a given vertex.
- You start a BFS, and when you find the specified vertex, you know the path you've traced so far is the shortest path to the node.
- If there were a shorter path, the BFS would have found it already.

Minimum Spanning Tree (MST)

- When a graph shows the minimum number of edges to connect all the vertices, it's called the MST.
- There are many possible minimum spanning trees for a given set of vertices.
- The number of edges E in a minimum spanning tree is always one less than the number of vertices V:

  E = V – 1



Main Point

The BFS and DFS algorithms are procedures for visiting every vertex in a graph. BFS proceeds "horizontally", examining every vertex adjacent to the current vertex before going deeper into the graph. DFS proceeds "vertically", following the deepest possible path from the starting point.

These approaches to graph traversal are analogous to the horizontal and vertical means of gaining knowledge, as described in SCI: The horizontal approach focuses on a breadth of connections at a more superficial level and reaches deeper levels of knowledge more slowly. The vertical approach dives deeply to the source right from the beginning; having fathomed the depths, subsequent gain of knowledge in the horizontal direction enjoys the influence of the depths of knowledge already gained.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

1. Both DFS and BFS algorithm provide efficient procedures for traversing all vertices in a given graph.
2. By tracking edges during DFS or BFS, we can obtain a spanning tree/forest for a given graph without much effort.
3. *Transcendental Consciousness* is the field of all possibilities, located at the source of thought by an effortless procedure of transcending.
4. *Impulses within the Transcendental field:* The entire structure of the universe is designed in seed form within the transcendental field, all in an effortless manner.
5. *Wholeness moving within itself:* In Unity Consciousness, each expression of the universe is seen as the effortless creation of one's own unbounded nature.