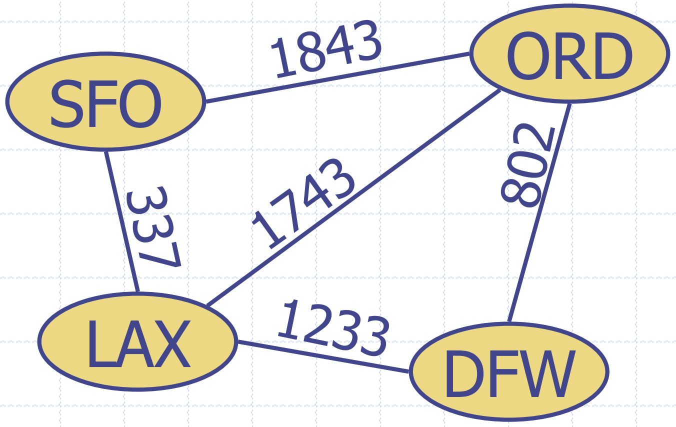
Lesson 10
Graphs
Combinatorics of Pure Intelligence

Wholeness of the Lesson

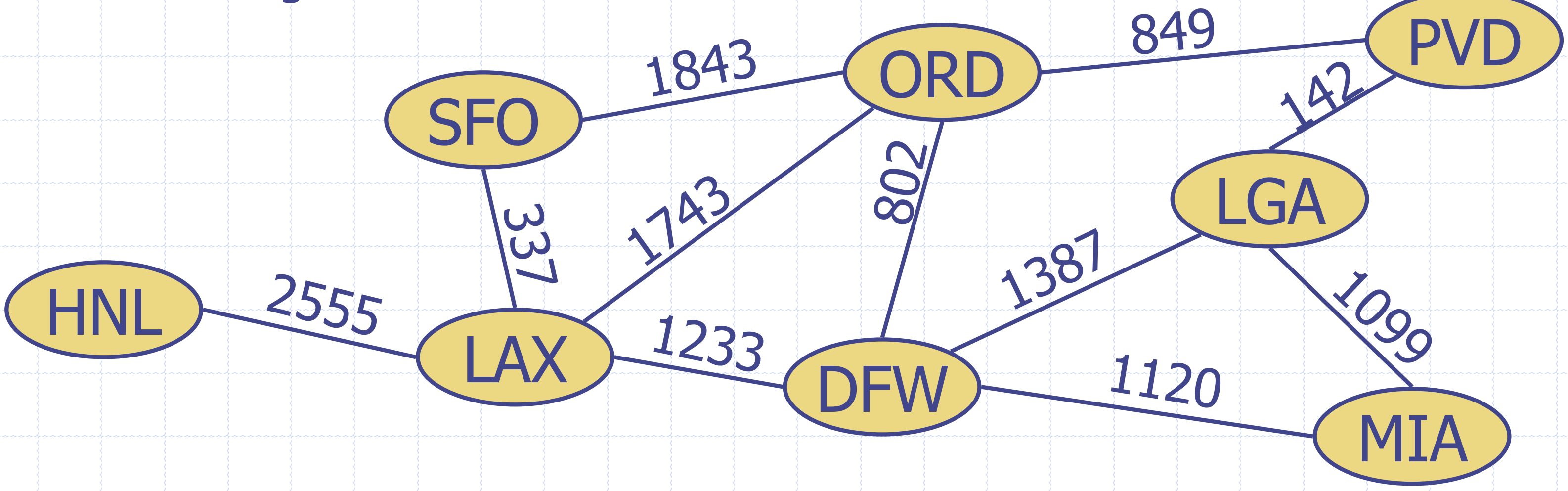
Graphs are data structures that do more than simply store and organize data; they are used to model interactions in the world. This makes it possible to make use of the extensive mathematical knowledge from the theory of graphs to solve problems abstractly, at the level of the model, resulting in a solution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



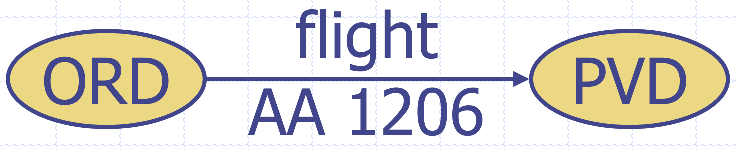
Graph

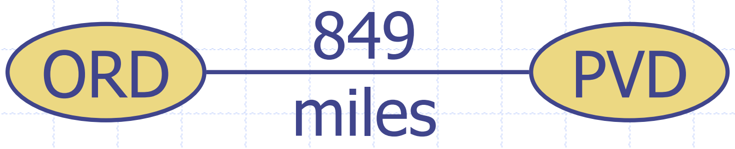
- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges can be implemented so that they store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge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



Edge Types

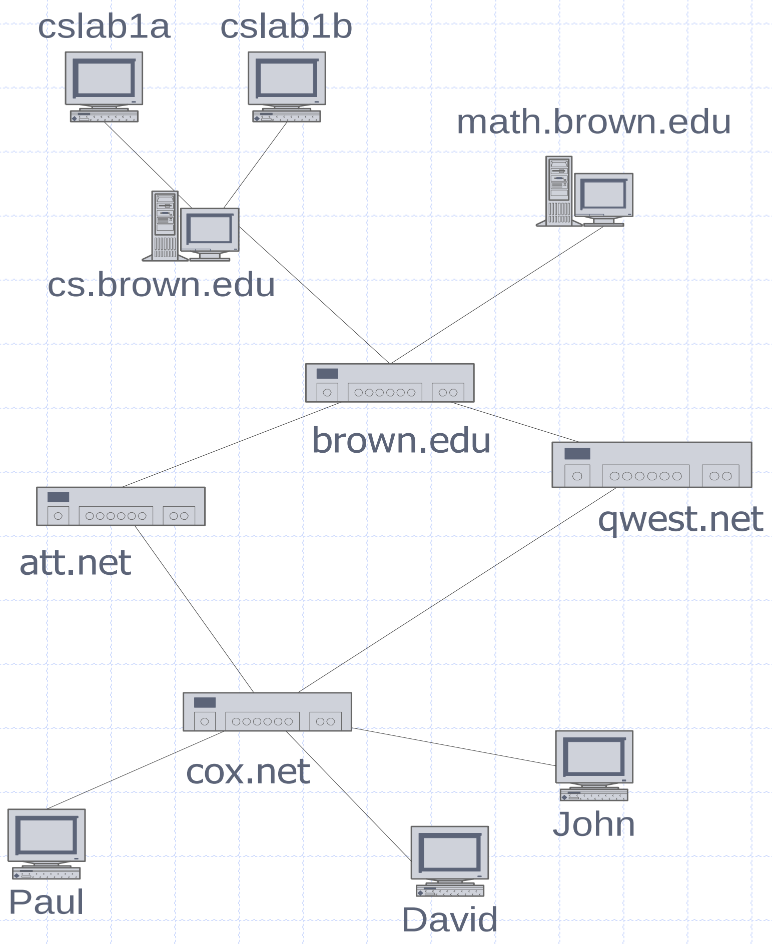
- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., flight network
- Undirected graph
  - all the edges are undirected
  - e.g., route network





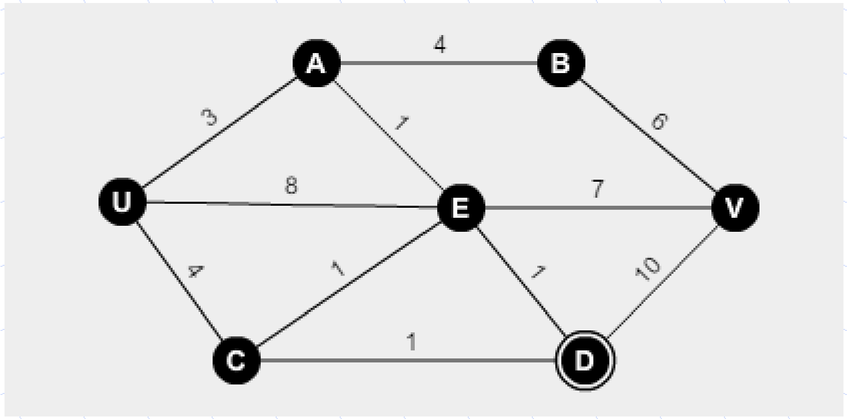
Applications

- 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



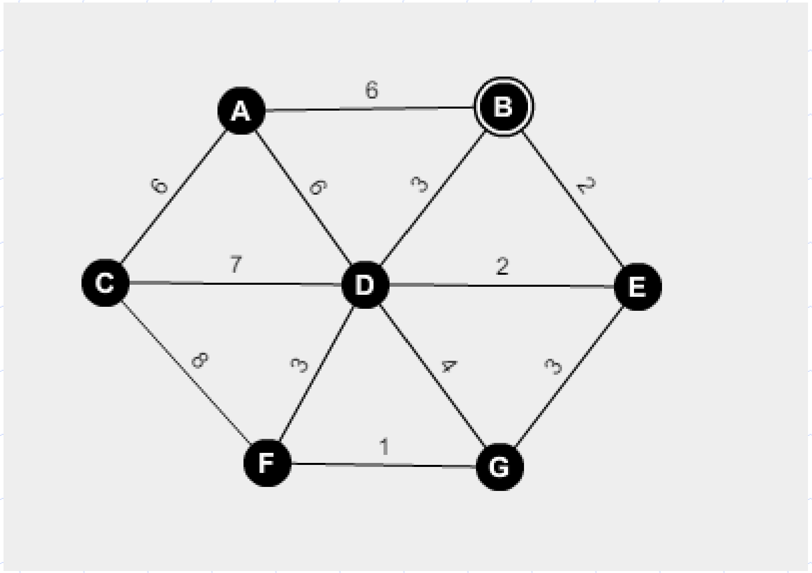
Examples

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.



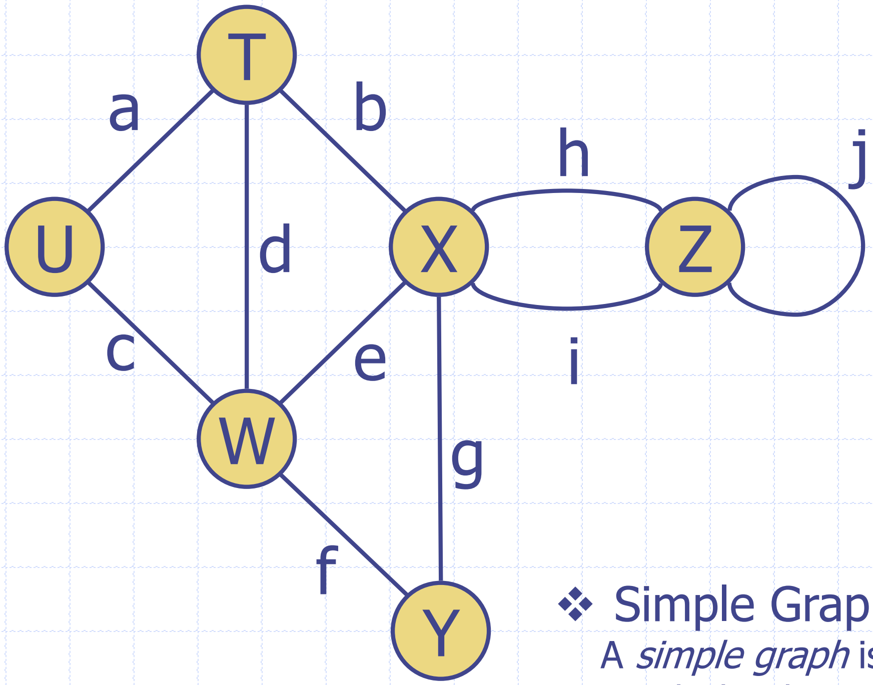
Examples (continued)

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| (also ν or *n*) is the number of vertices of G; |E| (also ϵ or *m*) is the number of edges. (ν = "nu", ϵ = "epsilon")
- 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 (= an edge between them)
  - U and T are adjacent
- Degree of a vertex (= # edges incident to the vertex)
  - X has degree 5
- Parallel edges (= two edges with same endpoints)
  - h and i are parallel edges
- Self-loop (= edge with just one endpoint)
  - j is a self-loop

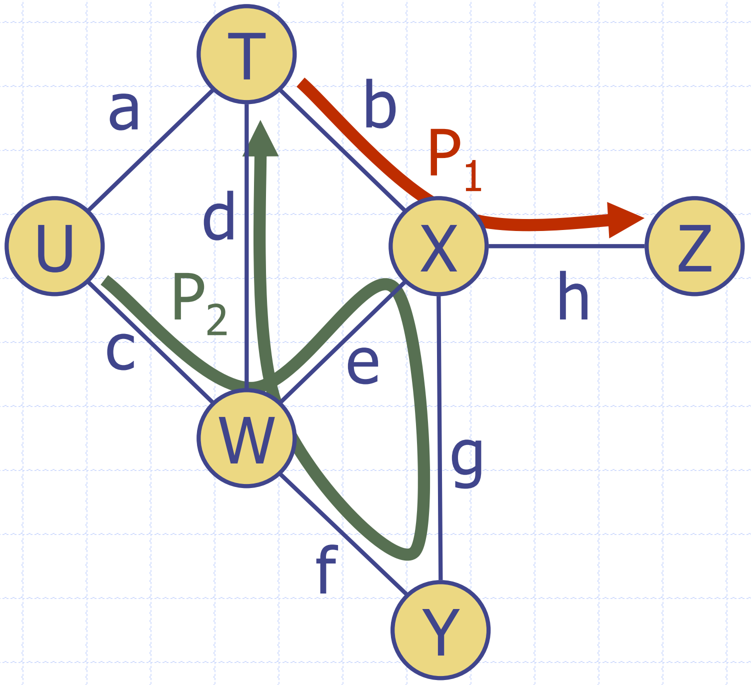


- Simple Graph

  A *simple graph* is a graph that has no self-loops or parallel edges

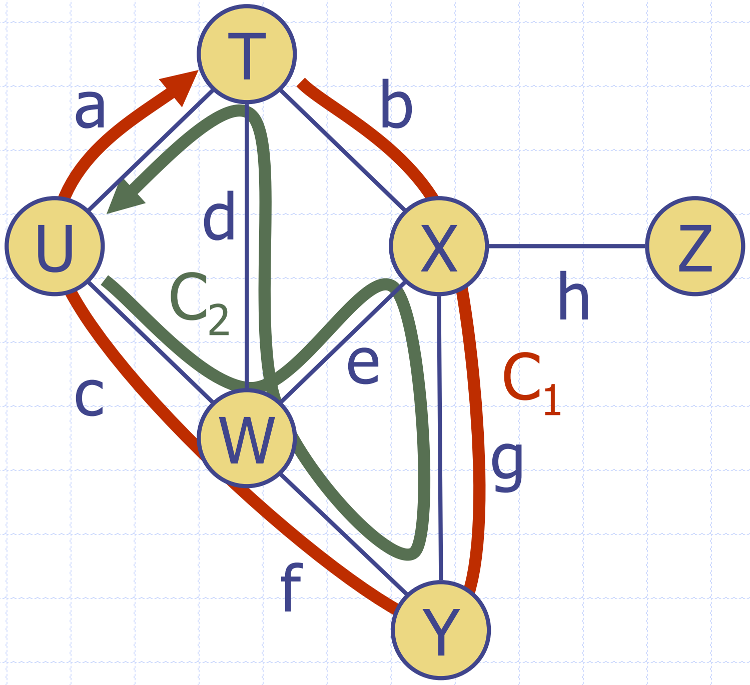
Terminology (cont.)

- Path
  - sequence of alternating vertices and edges – in a simple graph, can omit edges (called a "walk" in graph theory)
  - begins and ends with a vertex
  - *length* of a path is number of edges
- Simple path
  - path such that all its vertices and edges are distinct (called a "path" in GT)
- Examples
  - P_1=(T, X, Z) is a simple path
  - P_2=(U, W, X, Y, W, T) is a path that is not simple



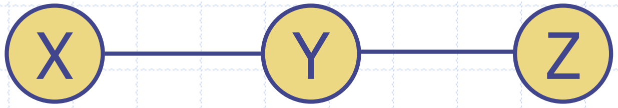
Terminology (cont.)

- Cycle
  - path in which all edges are distinct and whose first and last vertex are the same (called "closed trail" in GT)
  - *length* of a cycle is the number of edges in the cycle
- Simple cycle
  - path such that all vertices and edges are distinct except first and last vertex (called a "cycle" in GT)
- 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



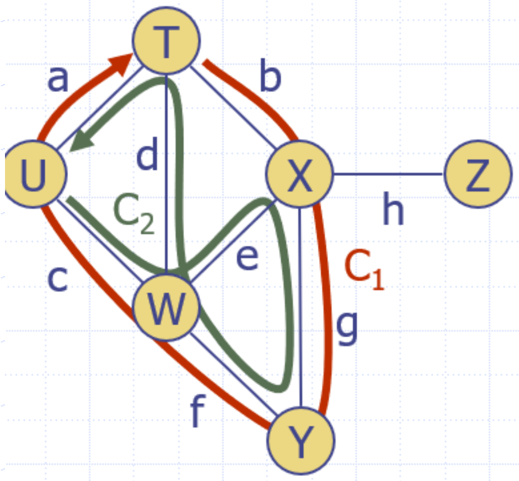
Details About Cycles

- Example: Not every path with identical first and last vertex is a cycle (edges must be distinct)



consider X - Y - Z - Y - X

- Facts
  - In a simple graph, the shortest possible cycle has length 3
  - If an edge belongs to a cycle C in G, it belongs to a simple cycle S that is a subpath of C (Examples: edges d and e both belong to C_2; edge d belongs to U-T-W-U and edge e belongs to W-X-Y-W.)
  - In a simple graph G, if there is a path joining vertices X and Y, there is a *simple* path in G joining X and Y. (Example: U-W-X-Y-W-T joins U to T; but U-W-T does also.)



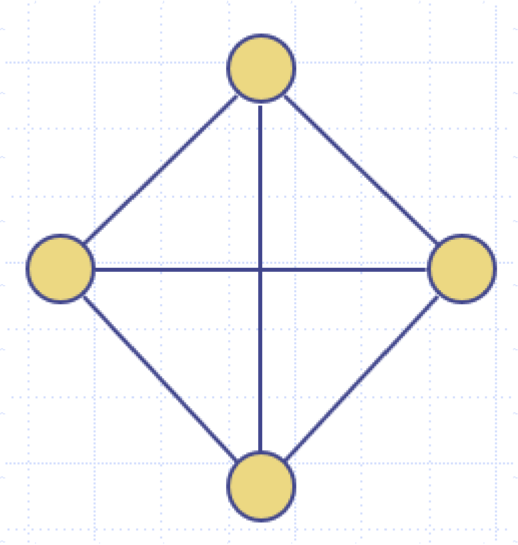
Properties

Notation

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

Example

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



Properties

Conventions:

1. All graphs considered in this class will be *simple.*
2. Mostly we will work with undirected graphs but the next lesson introduces directed graphs.

Property 1

$$\Sigma_{\mathbf{v}} \deg(\mathbf{v}) = 2\epsilon$$

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

$$\Sigma_{\mathbf{v}} \deg(\mathbf{v}) = \Sigma_{\mathbf{v}} |E_{\mathbf{v}}|$$

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

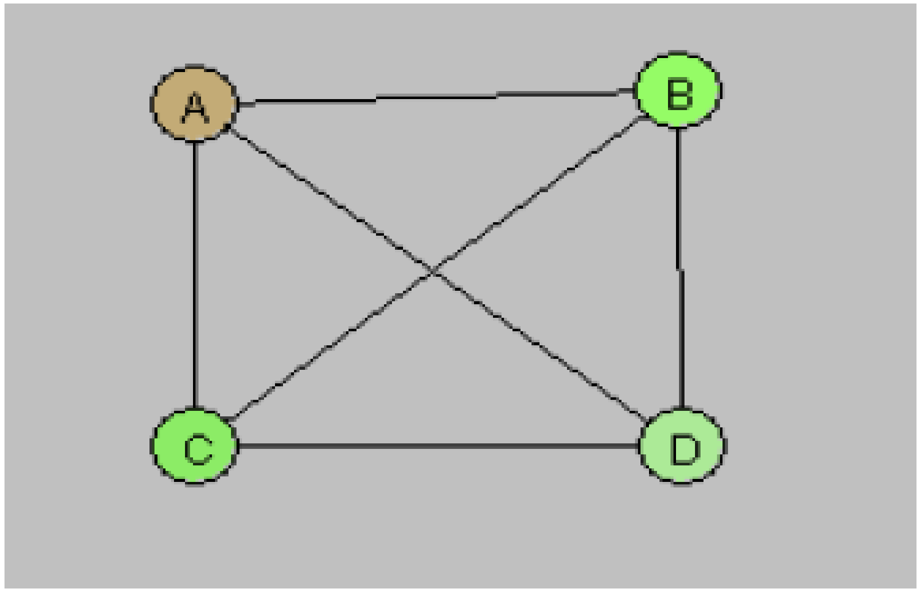
$$\epsilon \leq \nu(\nu - 1)/2$$

Proof: max number of edges is

$$C(\nu, 2) = C_{\nu,2} = \binom{\nu}{2}$$

Complete Graphs

- A graph G is **complete** if for every pair of vertices u, v, there is an edge $e \in E$ that is incident to u, v. In other words, for every u, v, $(u, v) \in E$.
- This is the complete graph on 4 vertices, denoted K_4.



Complete Graphs (continued)

- In general, the complete graph on n vertices is denoted K_n. The one-point graph is K_1.
- **Exercise.** For a complete graph G,

$$\epsilon = \frac{\nu(\nu - 1)}{2}.$$

Therefore, K_n has exactly $n(n-1)/2$ edges.

Subgraphs of a Graph

- A graph $H = (V_H, E_H)$ is a **subgraph** of a graph $G = (V_G, E_G)$ if both of the following are true:
  a. $V_H \subseteq V_G$ and $E_H \subseteq E_G$, and
  b. for every edge (u, v) belonging to E_H, both u and v belong to V_H.

  H is called a **spanning subgraph** if $V_H = V_G$.

Subgraphs of a Graph

- A graph $H = (V_H, E_H)$ is a **subgraph** of a graph $G = (V_G, E_G)$ if both of the following are true:

  a. $V_H \subseteq V_G$ and $E_H \subseteq E_G$, and

  b. for every edge (u, v) belonging to E_H, both u and v belong to V_H.

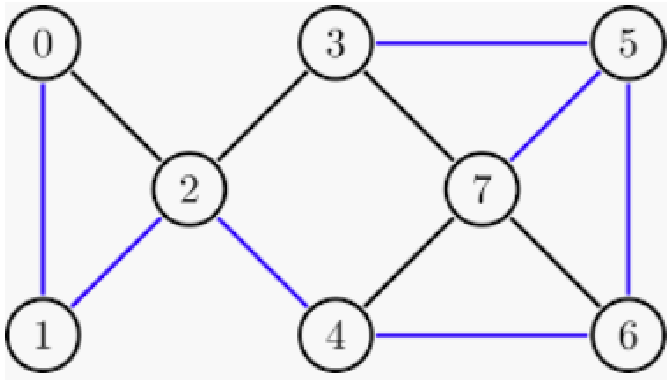
  H is called a **spanning subgraph** if $V_H = V_G$.

- Suppose $G = (V, E)$ is a graph and $W \subseteq V$, where W is nonempty. Then the the subgraph **induced by** W is the subgraph of G, denoted $G[W]$ whose vertex set is W and which has the maximum possible number of edges.

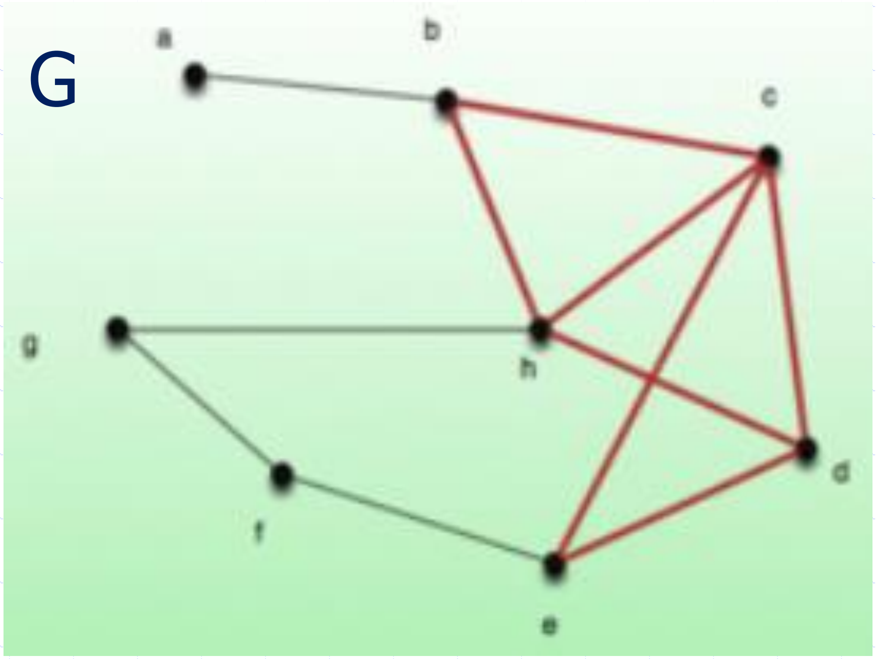
  Equivalently, it is the subgraph of G whose vertices are W and whose edges are just those edges in E both of whose ends lie in W.

Examples

G



Subgraph H (in blue) of G is a *spanning* subgraph.



Subgraph H (in red) of G is the subgraph *induced by* the set of vertices {b, c, d, e, h}

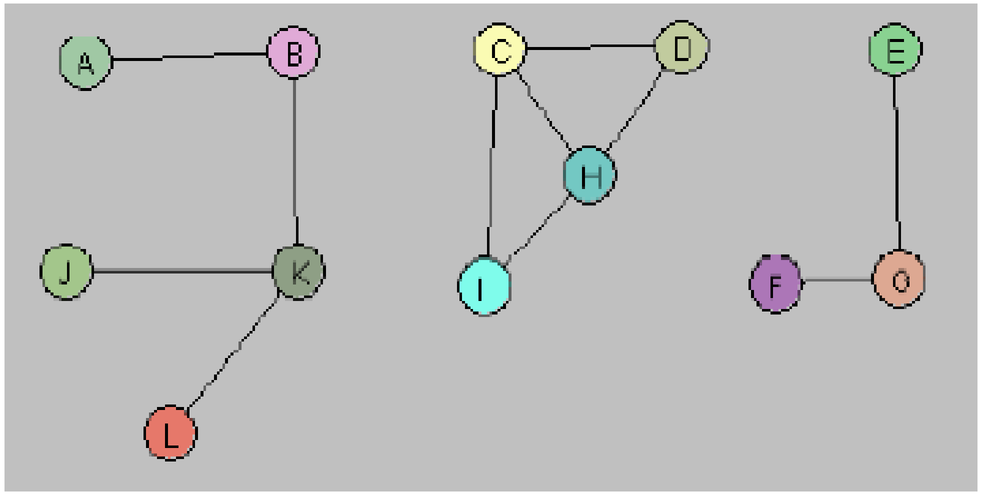
Connected Graphs and Connected Components

- A graph is **connected** if for any two vertices u, v in G, there is a path from u to v.
- **Observation** If a graph $G = (V, E)$ is not connected, then V can be partitioned into sets $V_1, V_2, \ldots, V_k$ (all disjoint) so that each of the subgraphs $G[V_1], G[V_2], \ldots, G[V_k]$ is connected; they are called the **connected components** of G.
- **In-Class Exercise.** Show that if $G = (V, E)$ is a graph, then G is connected whenever

$$\epsilon > \binom{\nu - 1}{2}.$$

(continued)

- The following graph has 3 connected components.



Trees and Rooted Trees

- A graph is **acyclic** if it contains no simple cycle.
- An acyclic connected graph is called a **tree**.
- A **rooted tree** is a tree having a distinguished vertex r.
- A subgraph T of G is a **spanning tree** if T is a spanning subgraph and also a tree.
- **Theorem.** In a tree, any two vertices are connected by a unique simple path. [In-class Exercise]

Graph Algorithms

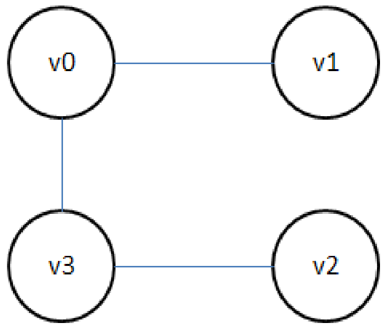
Some natural questions to ask and to solve with algorithms:

- Given two vertices, are they adjacent (is there an edge between them)?
- Given two vertices, is there a path that joins them?
- Is the graph connected? If not, how many connected components does it have?
- Does the graph contain a cycle?
- Does a graph have a spanning tree? If so, find it.
- Given two vertices, what is the length of a *shortest* path between them (if there is a path at all)?

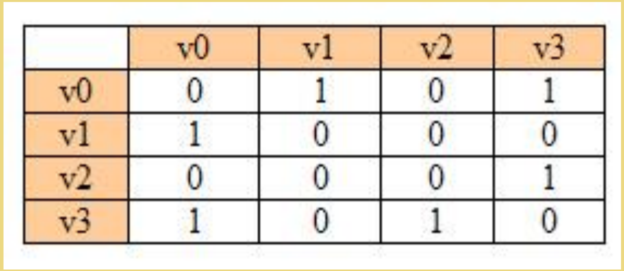
Determining Adjacency

- 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

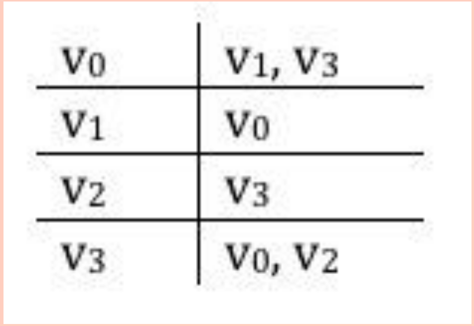


- An *Adjacency Matrix A* is an n x n matrix (representable as a two-dimensional array) consisting of 1s and 0s. A 1 in *A*[i][j] means that vertices v_i and v_j are adjacent; 0 indicates that the vertices are not adjacent.



	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.*



V0	V1, V3
V1	V0
V2	V3
V3	V0, V2

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 $\Theta(n^2)$ edges. It is said to be **sparse** if it has $O(n)$ edges.
- **Strategy**. Use adjacency lists for sparse graphs and adjacency matrices for dense graphs.
- For purposes of implementation, in this class we will use adjacency lists. But for purposes of determining optimal running time, we will assume we are using whichever implementation gives the best running time.

Implementation of Graphs in Java

- Java Classes: Vertex, Edge, Graph
- Graph constructor accepts a List of Pairs (x,y). Each of x and y becomes a vertex, and the pair (x,y) becomes an edge.
- Represent the adjacency list using a HashMap.
- See Demo in graph0 package.

In-Class Exercise

Implement the method

```
boolean areAdjacent(Vertex v, Vertex w)
```

in the Graph class.

Graph Traversal Algorithms

- To answer questions about graphs (Is it connected? Is there a path between two given vertices? Does it contain a cycle? Find a spanning tree) we need an efficient way of visiting every vertex.
- *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)
```

Correctness of BFS

Two points to check:

(1) the while loop eventually exits

(2) after the while loop exits, every vertex in G has been marked

Proof of (1) In each pass in the while loop, one marked vertex is removed, and no vertex that has been removed is ever added back to the queue.

Proof of (2) Suppose at the end of the algorithm, some vertex w has not been marked. Let s – v_1 – v_2 - . . . – v_k = w be a path to w from s. Let i be least such that v_i is unmarked; therefore v_{i-1} has been marked, and so was added to Q. Eventually, (v_{i-1}, v_i) must be examined. If v_i has not yet been marked, it will be marked at that time. Contradiction.

Running time for BFS

For each vertex v, we have the following logic:

```
get the list L of vertices adjacent to v
for each vertex w in L
    if (w is unvisited)
        mark w
        add to Q
```

There are deg(v) vertices adjacent to v; for each such vertex w, we check if w is unvisited, mark it, and add it to Q. It takes O(1) to get the list of vertices adjacent to v, then O(deg v) to loop through the adjacent vertices and process them. Therefore:

$$\mathrm{O}(\textstyle\sum_{v\in V} 1+\deg(v)) = \mathrm{O}(n+2m) = \mathrm{O}(n+m)$$

Implementation Issues

- *Indicating Visited Vertices.* Need to decide how to represent the notion that a vertex "has been visited". Often this is done by creating a special bit field for this purpose in the Vertex class. An alternative is to view visiting as being conducted by BFS, so BFS is responsible for tracking visited vertices. Can do this with a hashtable – insert (u,u) whenever u has been visited. Checking whether a vertex has been visited can then be done in O(1) time. (We use this approach.)
- *BFS As a Class.* It's useful to represent BFS as a class. Then, other algorithms that make use of the BFS strategy can be represented as subclasses. To this end, insert "process" options at various breaks in the code to allow subclasses to perform necessary processing of vertices and/or edges. (This is an application of the *template design pattern.*)

BFS for Disconnected Graphs

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

```
while there are unvisited vertices do
    s $\leftarrow$ next unvisited vertex
    initialize a queue Q
    mark s as visited
    Q.add(s)
    while Q $\neq \emptyset$ do
      v $\leftarrow$ Q.dequeue()
      for each unvisited w adjacent to v do
          mark w
          Q.add(w)
```

BFS for Disconnected Graphs Running Time

If a graph is not connected, with components G_1, G_2, . . ., G_k, so that each component G_i has n_i vertices and m_i edges, then running BFS on each component requires $O(n_i + m_i)$ time; summing over all components gives the same result as for connected graphs: $O(n + m)$ running time.

The other work that is done is to locate the next unvisited vertex after a all vertices of a component of the graph have been visited. Making sure this step is not costly is another (small) implementation issue: We handle by maintaining an iterator for the list of vertices; this ensures that each vertex is accessed only once

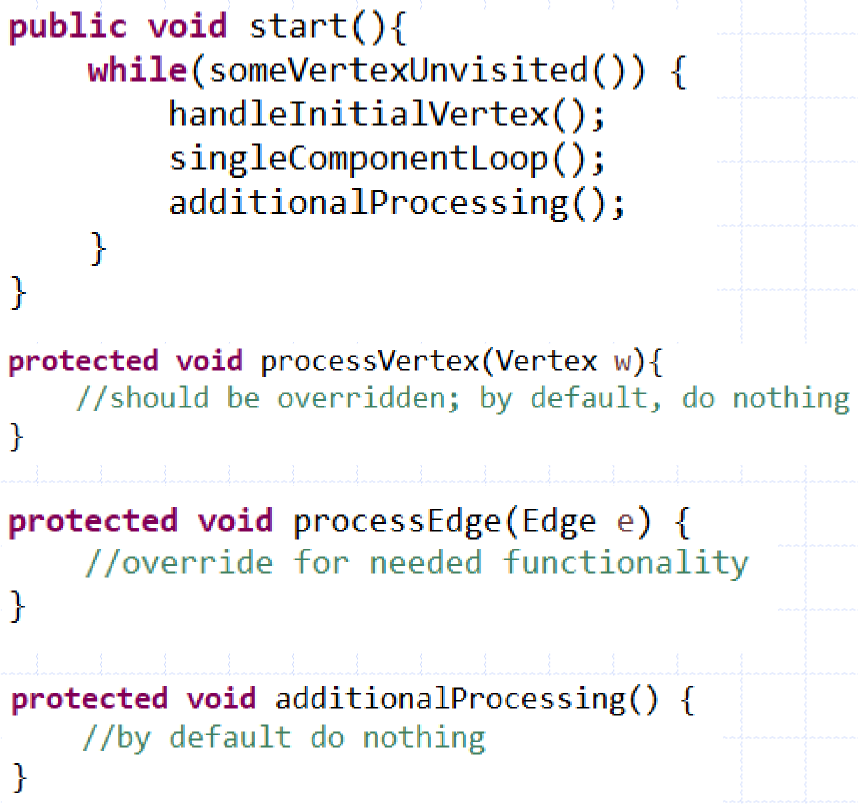
BFS for Disconnected Graphs Running Time (continued)

For each vertex s that is read, if s is unvisited, then it belongs to one of the connected components, say G_i, that has m_i edges and n_i vertices. So total processing that occurs when that vertex is read is $O(m_i + n_i)$. However if s has already been visited, the condition of the if statement fails and processing is O(1). This leads to the following computation of running time:

$$O(1) + O(1) + . . . + O(m_1 + n_1) + O(1) + . . . + O(1) + O(m_i + n_i) + . . . + O(m_k+n_k)$$
$$= O(n) + O(m + n) = O(m+n).$$

Implementation of BFS

See Demo



```
public void start(){
    while(someVertexUnvisited()) {
        handleInitialVertex();
        singleComponentLoop();
        additionalProcessing();
    }
}

protected void processVertex(Vertex w){
    //should be overridden; by default, do nothing
}

protected void processEdge(Edge e) {
    //override for needed functionality
}

protected void additionalProcessing() {
    //by default do nothing
}
```

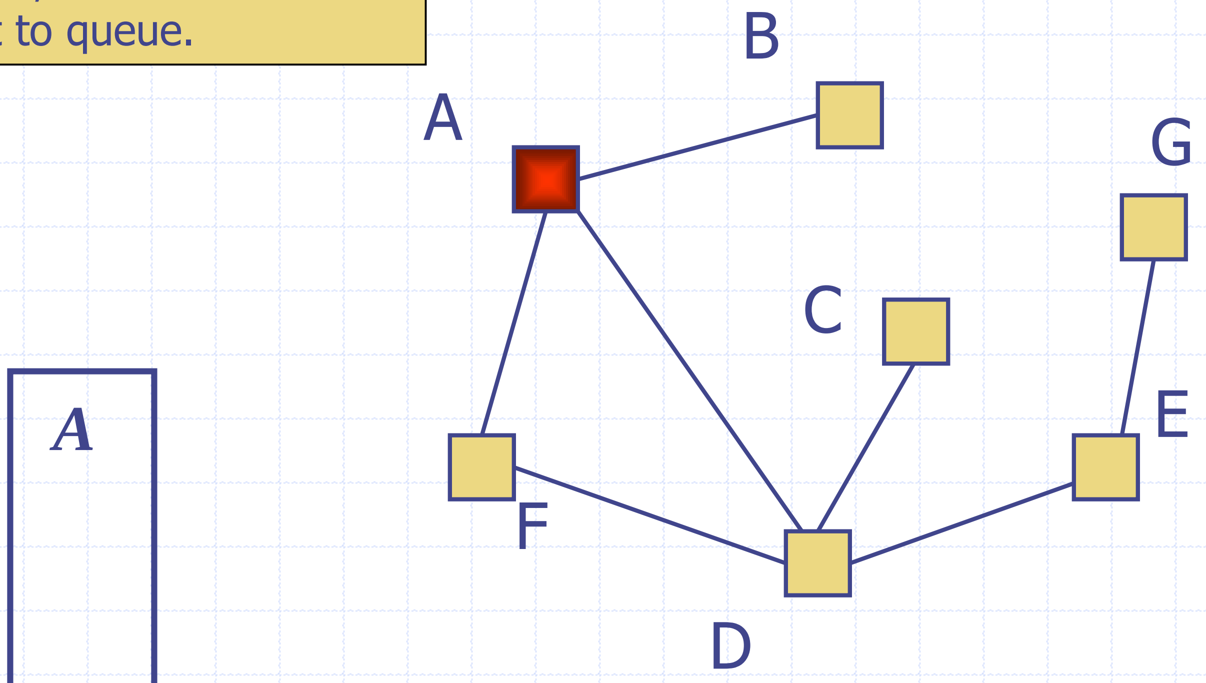
- BFS begins with the `start()` method.
- `handleInitialVertex()` locates start vertex and marks as visited, calls `processVertex`
- `singleComponentLoop()` performs BFS starting from initial vertex *v*, marking all vertices reachable from *v*; performs `processVertex` and `processEdge` after each *v* is marked
- `additionalProcessing` called at end of loop to permit between-component processing

Using BFS to Find a Spanning Tree

- A spanning tree for a connected graph (or a spanning *forest* for a disconnected graph) can be found by using BFS and recording each new edge as it is discovered. Since every vertex is visited, a subgraph is created, in this way, that includes every vertex. No cycle is created because we do not record edges when encountering already visited vertices.
- The spanning tree obtained using BFS in this way, with start vertex *s*, is called the *BFS rooted tree with root s.*

Worked Example

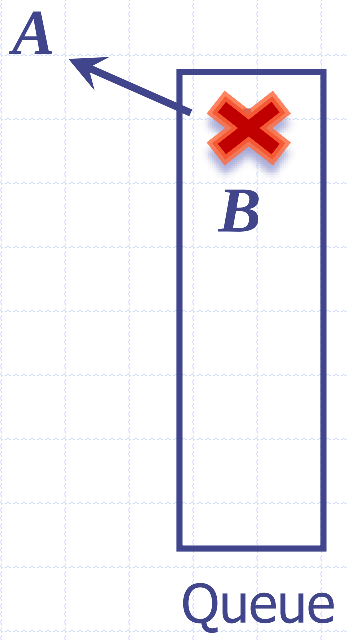
Start with A, mark it as visited and add it to queue.

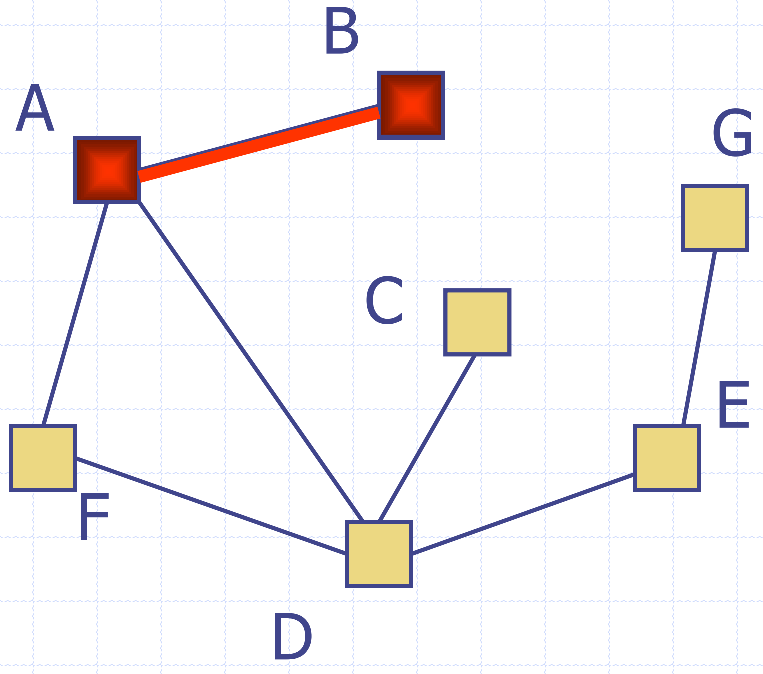


Spanning Tree: T = {…}

Worked Example

Remove A from Queue. Since B is adjacent to A, insert B into Queue. Add AB to tree T.

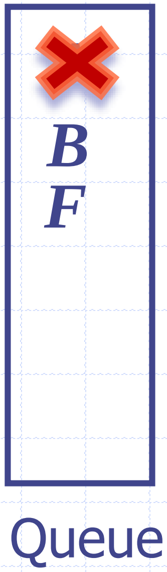


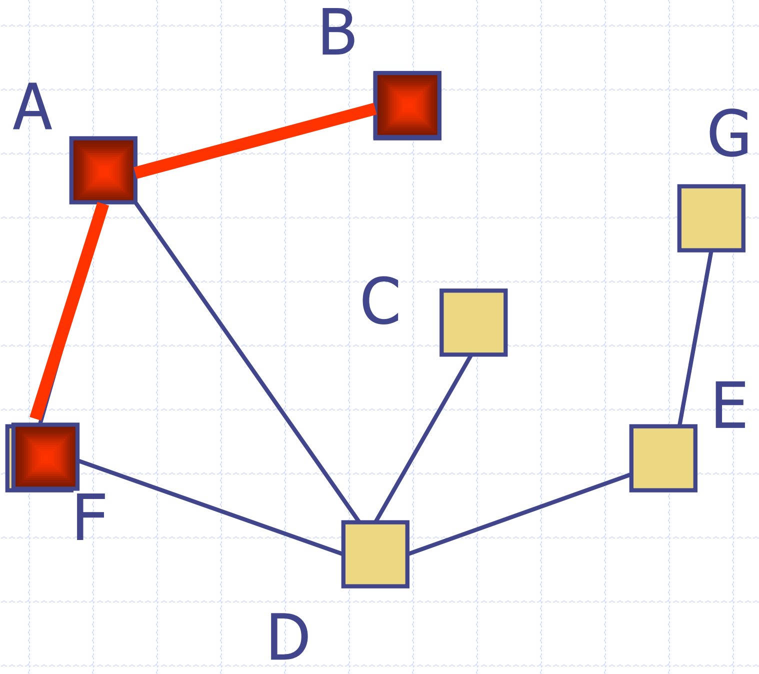


Spanning Tree: T = {AB . . .}

Worked Example

Since F is adjacent to A, insert F into Queue. Add AF to tree T.

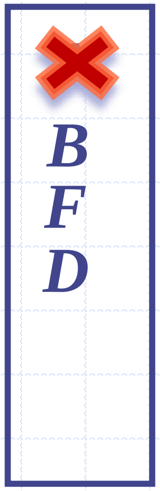




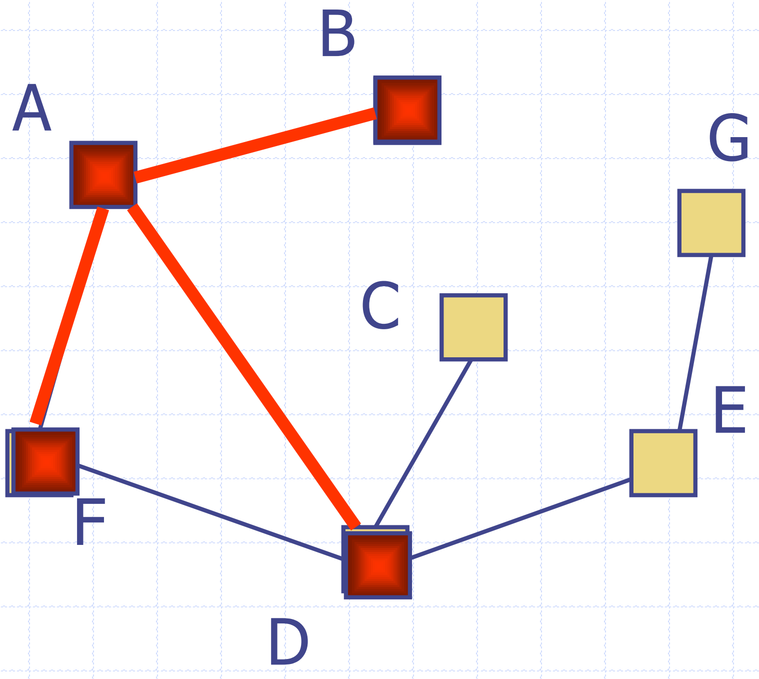
Spanning Tree: T = {AB, AF . . .}

Worked Example

Since D is adjacent to A, insert D into Queue. Add AD to tree T.



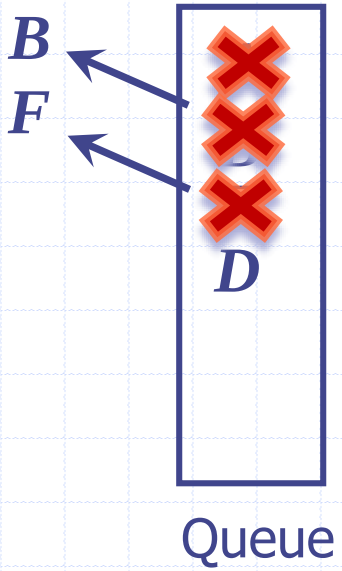
Queue

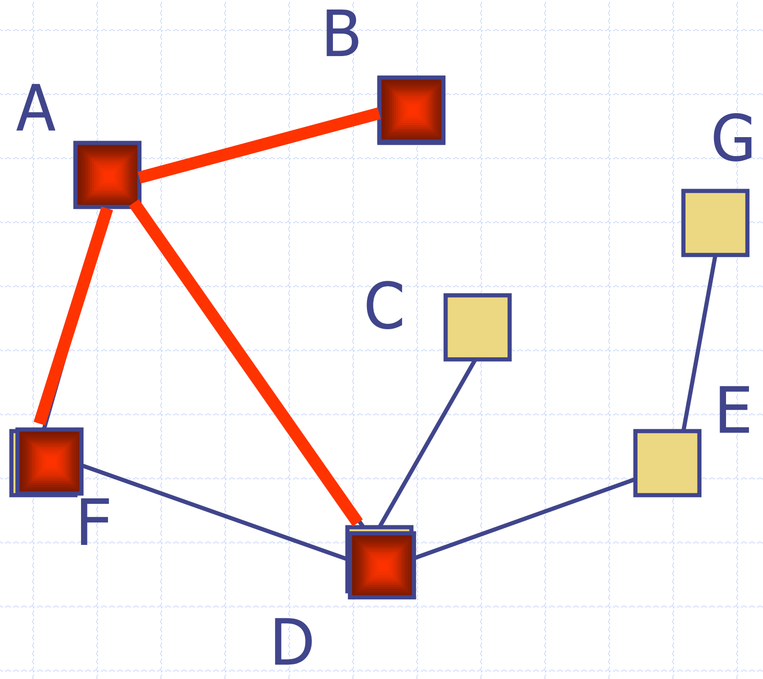


Spanning Tree: T = {AB, AF, AD . . .}

Worked Example

No other vertices adjacent to A. Remove B from Queue. Since no unvisited vertices are adjacent to B, remove F from Queue

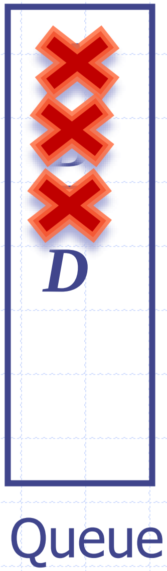


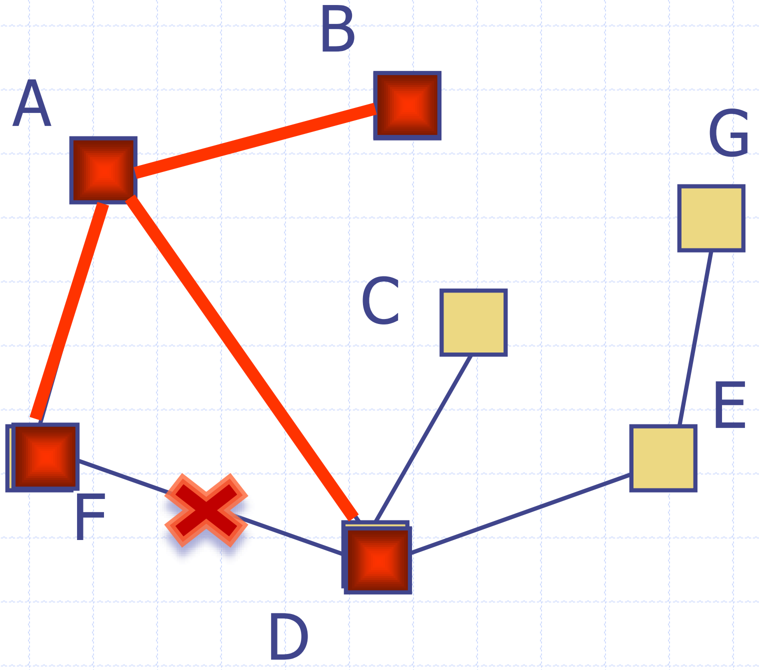


Spanning Tree: T = {AB, AF, AD . . .}

Worked Example

D is adjacent to F but D has been visited so do not mark D again or add it to the Queue again. (If we were to add the edge FD to T, it would create a cycle.)

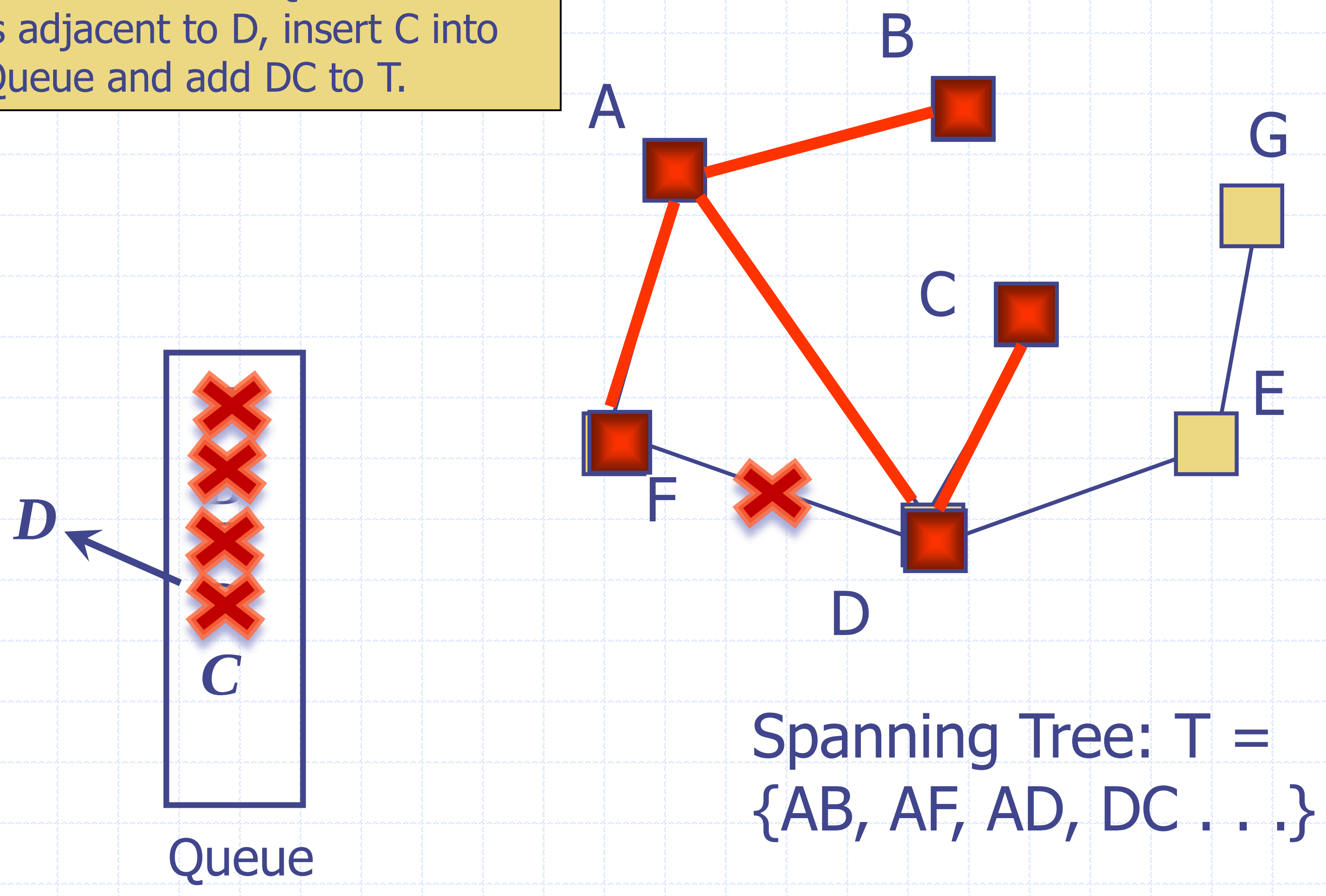




Spanning Tree: T = {AB, AF, AD . . .}

Worked Example

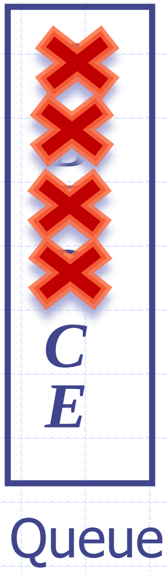
Remove D from Queue. Since C is adjacent to D, insert C into Queue and add DC to T.

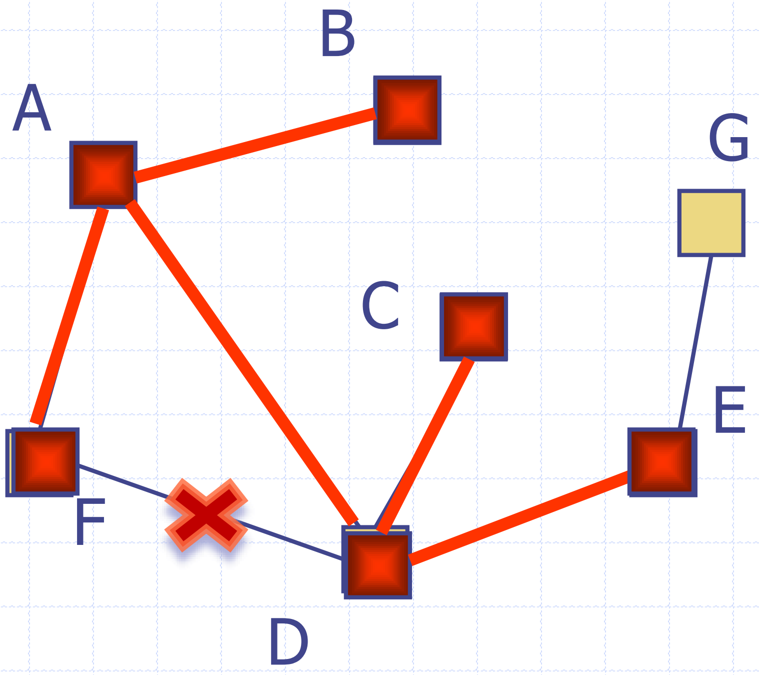


Spanning Tree: T = {AB, AF, AD, DC . . .}

Worked Example

Since E is adjacent to D, insert E into Queue and add DE to T.

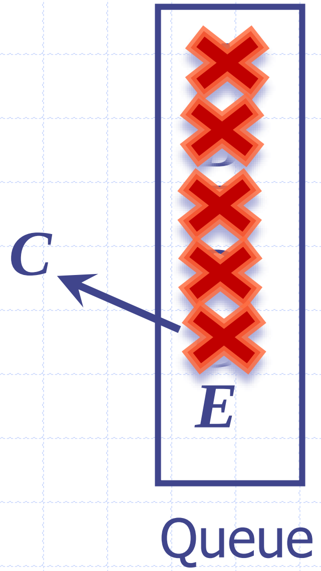


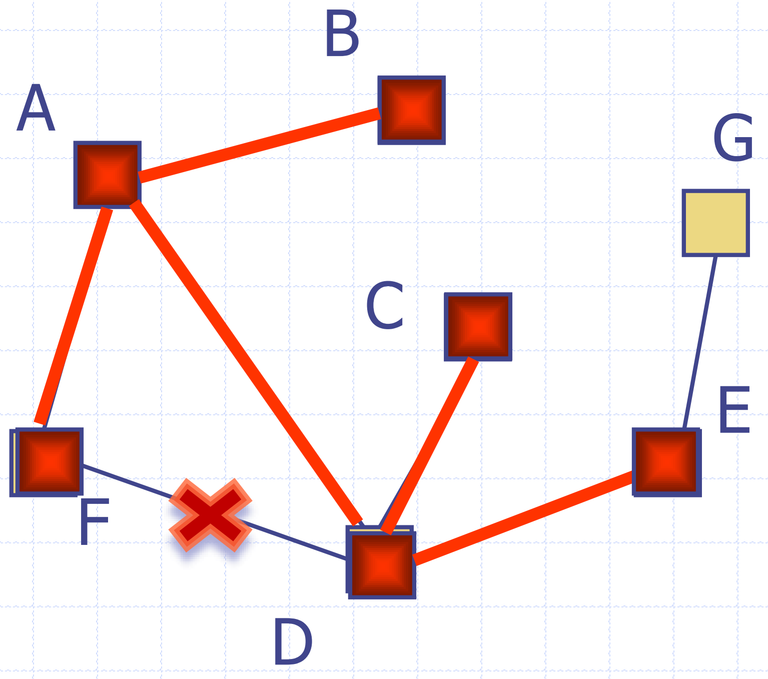


Spanning Tree: T = {AB, AF, AD, DC, DE . . .}

Worked Example

Remove C from Queue. D is adjacent to C but has already been visited so we do not mark it again or add it to Queue again

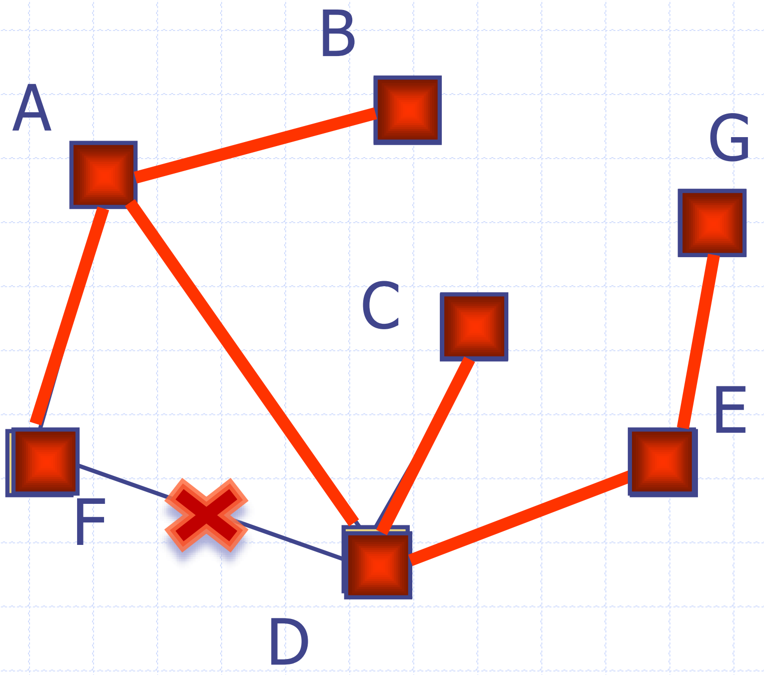


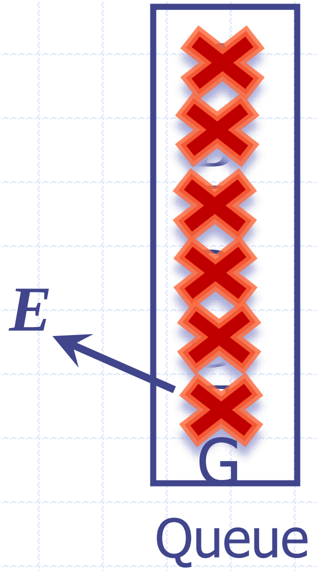


Spanning Tree: T = {AB, AF, AD, DC, DE . . .}

Worked Example

Remove E from Queue. Since G is adjacent to E, insert G into Queue and add EG to T

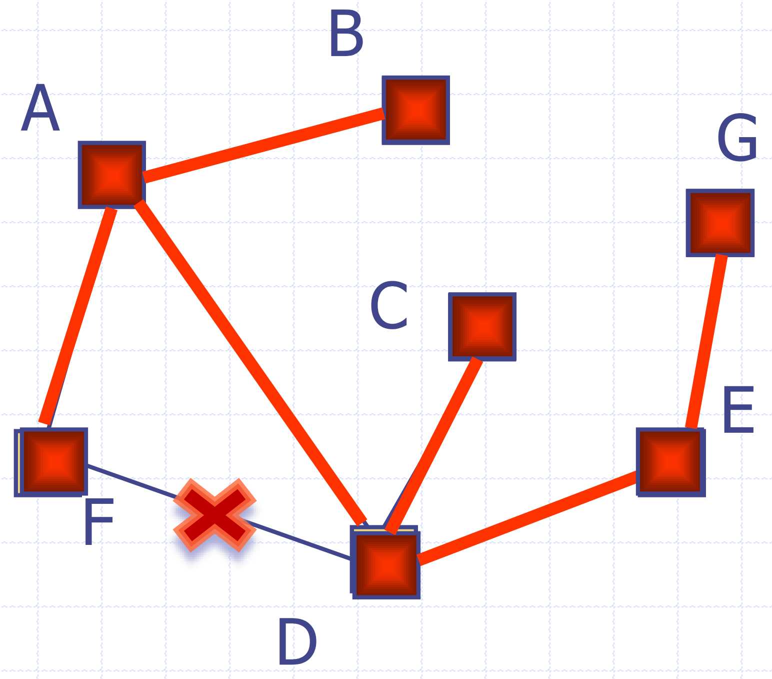


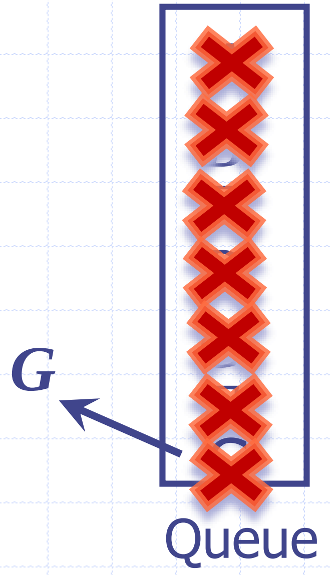


Spanning Tree: T = {AB, AF, AD, DC, DE, EG }

Worked Example

Remove G from Queue. There are no unvisited vertices adjacent to G and now Queue is empty. BFS algorithm terminates.





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

Implementation of Spanning Tree Algorithm

Can implement the spanning tree algorithm in Java by creating a subclass FindSpanningTree of BFS and, in each processEdge step, add the discovered edge to a list. At the end, use the list of edges to create a new Graph object, and return

```
public class FindSpanningTree extends BreadthFirstSearch {
    private ArrayList<Edge> tree = new ArrayList<Edge>();
    public FindSpanningTree(Graph graph) {
        super(graph);
    }
    protected void processEdge(Edge e) {
            tree.add(e);
    }

    public Graph computeSpanningTree() {
        start();
        //tree is loaded
        Edge[] edges = tree.toArray(new Edge[0]);
        Graph newG  = new Graph(edges);
        return newG;
    }

}
```

Using BFS to Solve Other Problems

- √ Given two vertices, are they adjacent (is there an edge between them)?
- Given two vertices, is there a path that joins them?
- Is the graph connected? If not, how many connected components does it have?
- Does the graph contain a cycle?
- √ Does the graph have a spanning tree? If so, find it.
- Given two vertices, what is the length of the *shortest* path between them (if there is a path at all)?

Exercise: Answer these questions by using BFS as was done to find a spanning tree

More on Trees: Specifying Levels of a Rooted Tree

- Recall that if G is a simple graph, G is a *tree* if G is connected and acyclic, and G is a *rooted tree* if G is a tree with some distinguished vertex (which we designate as the *root* of the tree).

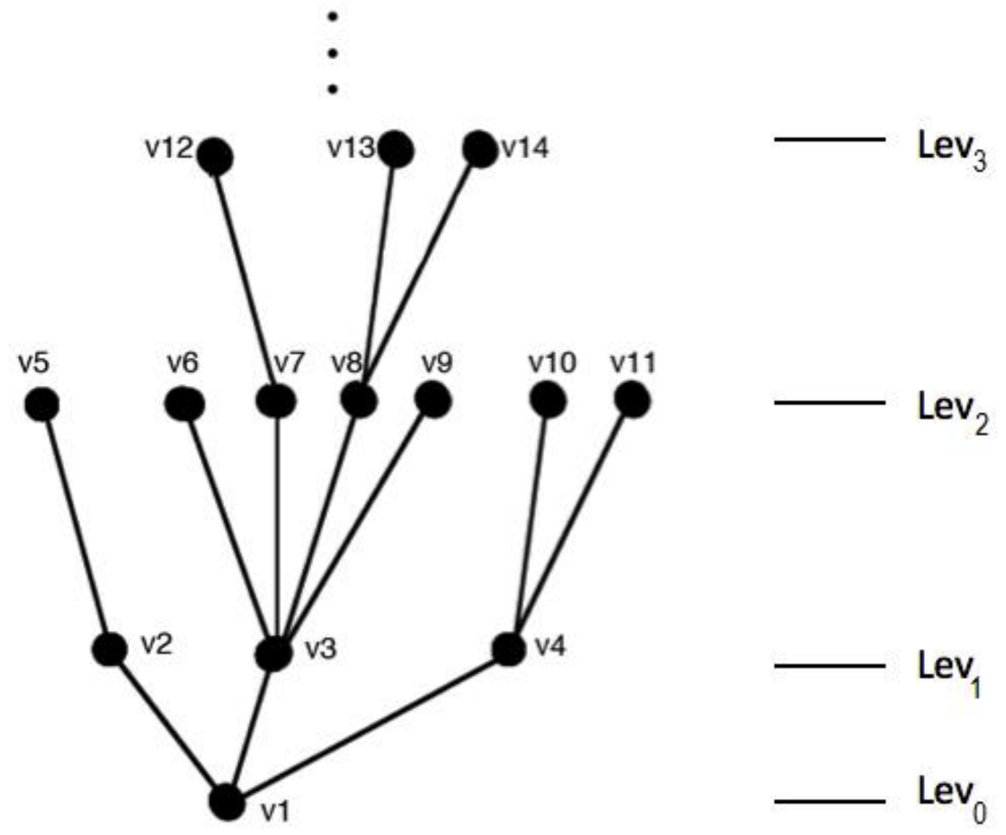
- Rooted trees can be organized into levels, just like trees used as data structures:
  Level 0 = {root of G}
  Level 1 = {all vertices adjacent to root}
  Level 2 = {all vertices adjacent to a Level 1 vertex other than root}
  . . .

- Various algorithms about trees make use of this level structure. The levels can be tracked using a hashtable as BFS executes.

Vertex	Level
A	0
B	1
. . .	. . .

Vertex – Level Map

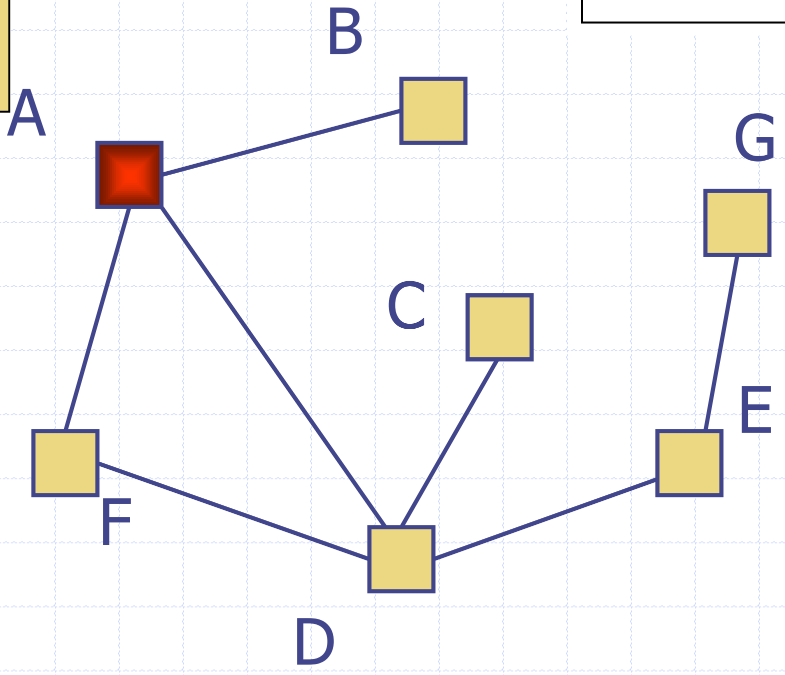
A Rooted Tree with Levels Shown

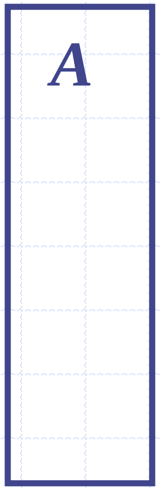


Vertex	Level
A	0

Worked Example

Start with A, mark it as visited and add it to queue. Set Lev(A)=0





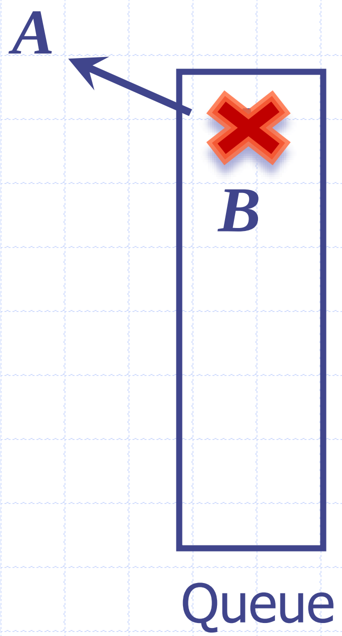
Queue

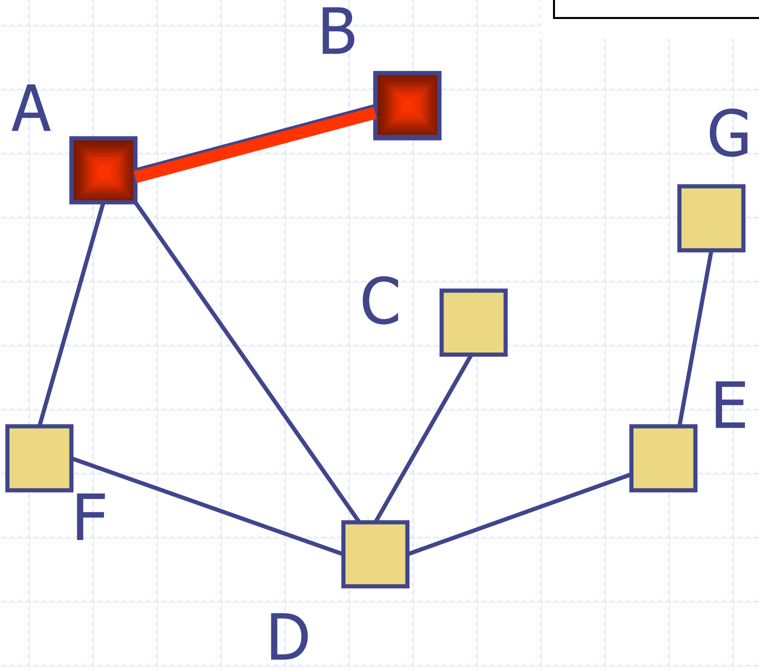
Spanning Tree: T = {…}

Vertex	Level
A	0
B	1

Worked Example

Remove A from Queue. Since B is adjacent to A, insert B into Queue. Add AB to tree T. Set Lev(B)=1.



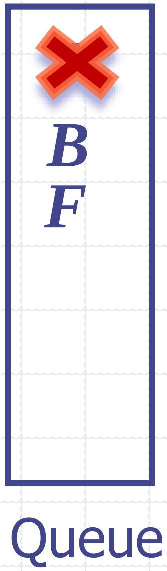


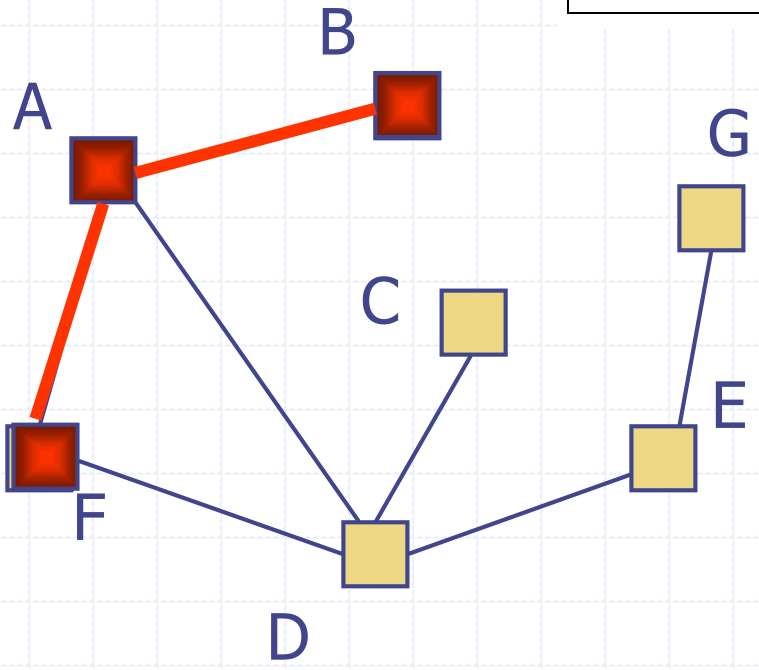
Spanning Tree: T = {AB . . .}

Vertex	Level
A	0
B	1
F	1

Worked Example

Since F is adjacent to A, insert F into Queue. Add AF to tree T. Set Lev(F) = 1.



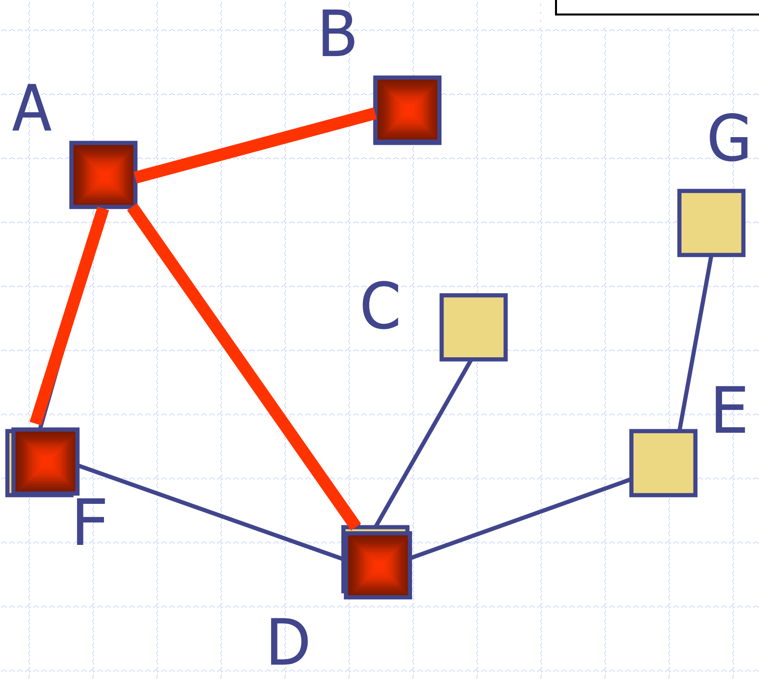


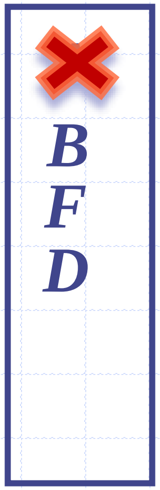
Spanning Tree: T = {AB, AF . . .}

Worked Example

Vertex	Level
A	0
B	1
F	1
D	1

Since D is adjacent to A, insert D into Queue. Add AD to tree T. Set Lev(D) = 1.





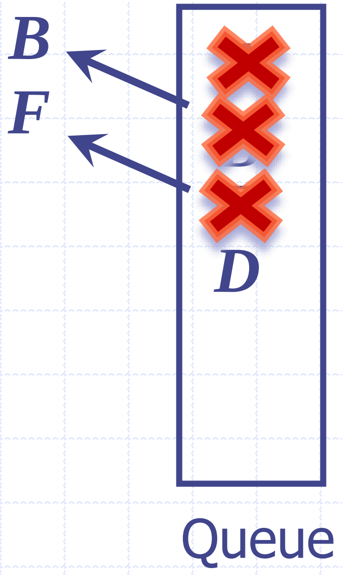
Queue

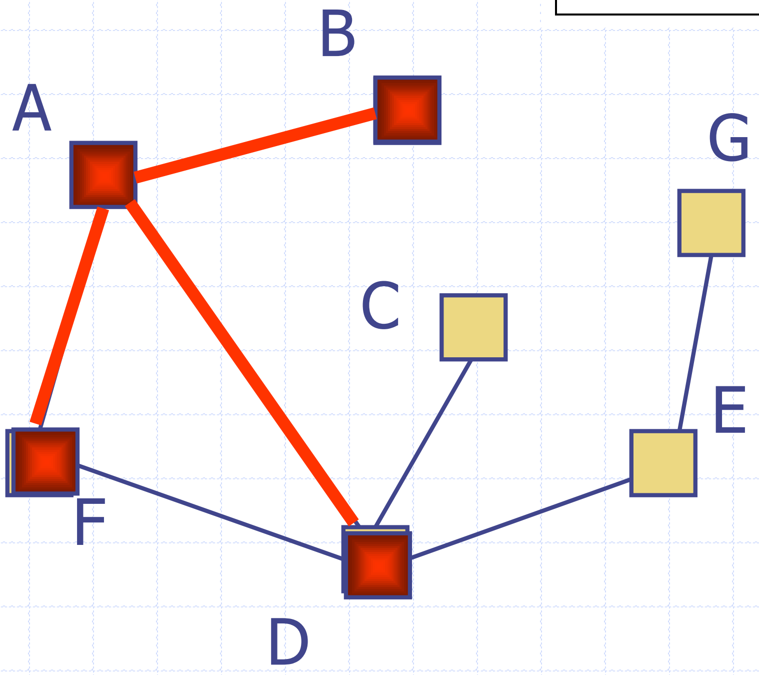
Spanning Tree: T = {AB, AF, AD . . .}

Worked Example

Vertex	Level
A	0
B	1
F	1
D	1

No other vertices adjacent to A. Remove B from Queue. Since no unvisited vertices are adjacent to B, remove F from Queue.



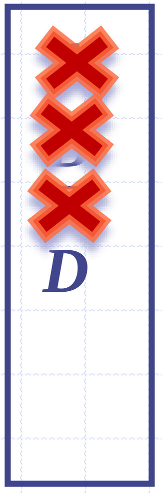


Spanning Tree: T = {AB, AF, AD . . .}

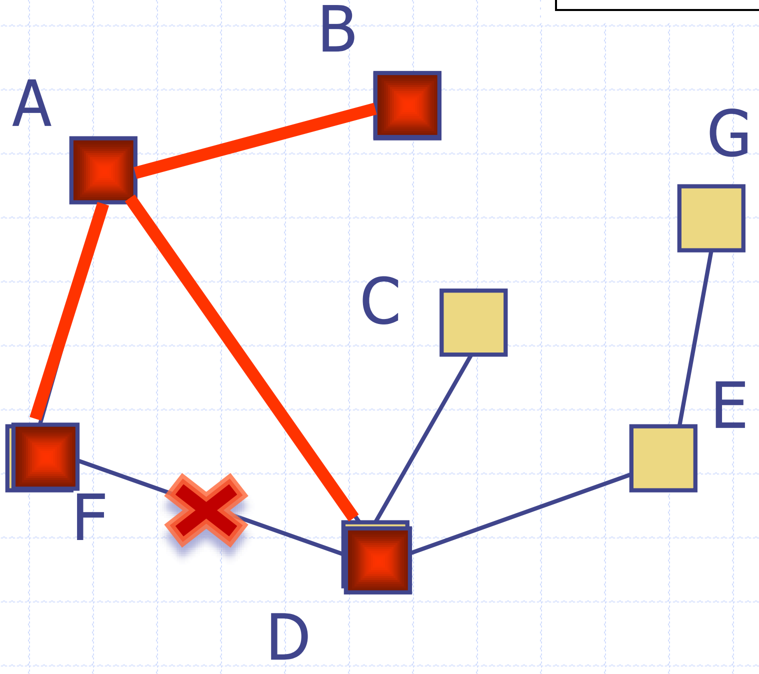
Worked Example

Vertex	Level
A	0
B	1
F	1
D	1

D is adjacent to F but D has been visited so do not mark D again or add it to the Queue again. (If we were to add the edge FD to T, it would create a cycle.)



Queue

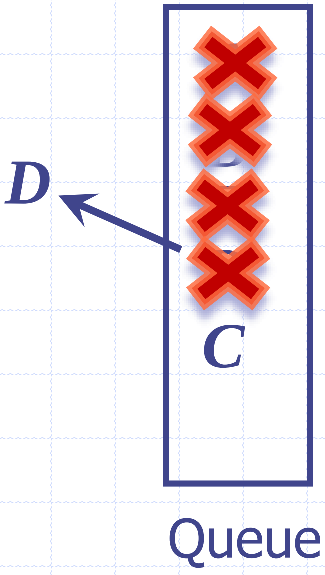


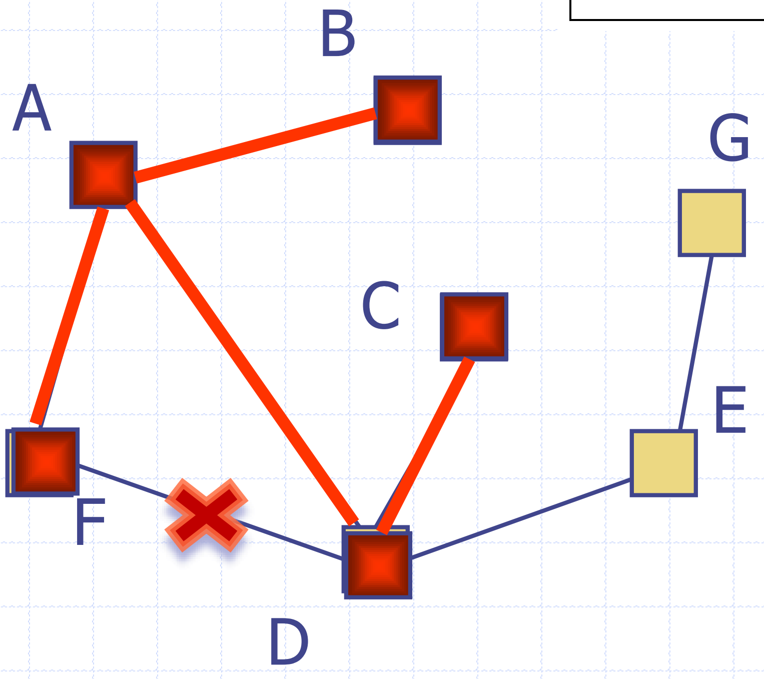
Spanning Tree: T = {AB, AF, AD . . .}

Worked Example

Vertex	Level
A	0
B	1
F	1
D	1
C	2

Remove D from Queue. Since C is adjacent to D, insert C into Queue and add DC to T. Set Lev(C)=2.



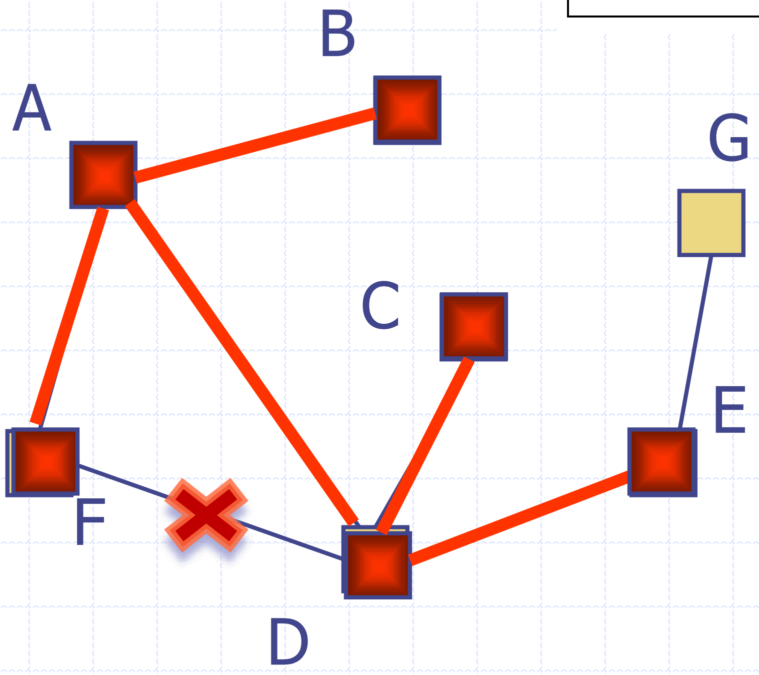


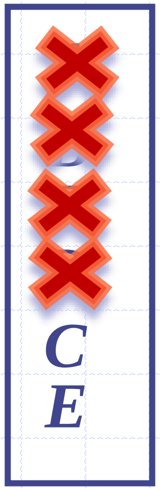
Spanning Tree: T = {AB, AF, AD, DC . . .}

Worked Example

Vertex	Level
A	0
B	1
F	1
D	1
C	2
E	2

Since E is adjacent to D, insert E into Queue and add DE to T. Set Lev(E) = 2.





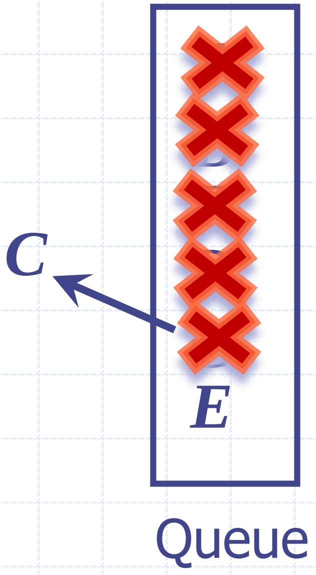
Queue

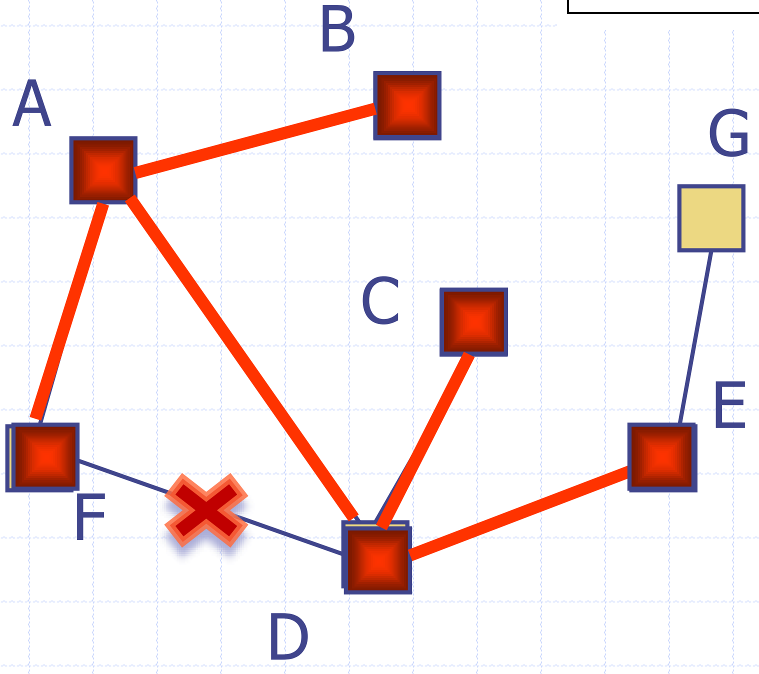
Spanning Tree: T = {AB, AF, AD, DC, DE . . .}

Worked Example

Vertex	Level
A	0
B	1
F	1
D	1
C	2
E	2

Remove C from Queue. D is adjacent to C but has already been visited so we do not mark it again or add it to Queue again



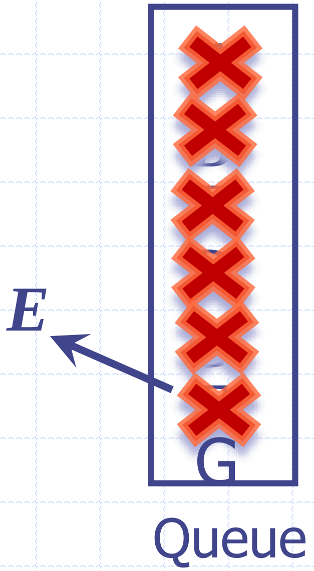


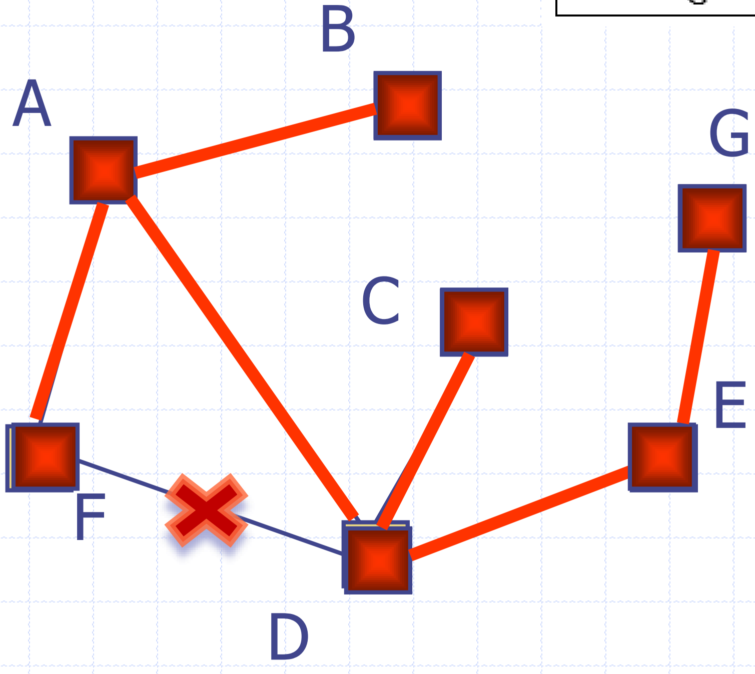
Spanning Tree: T = {AB, AF, AD, DC, DE . . .}

Vertex	Level
A	0
B	1
F	1
D	1
C	2
E	2
G	3

Worked Example

Remove E from Queue. Since G is adjacent to E, insert G into Queue and add EG to T



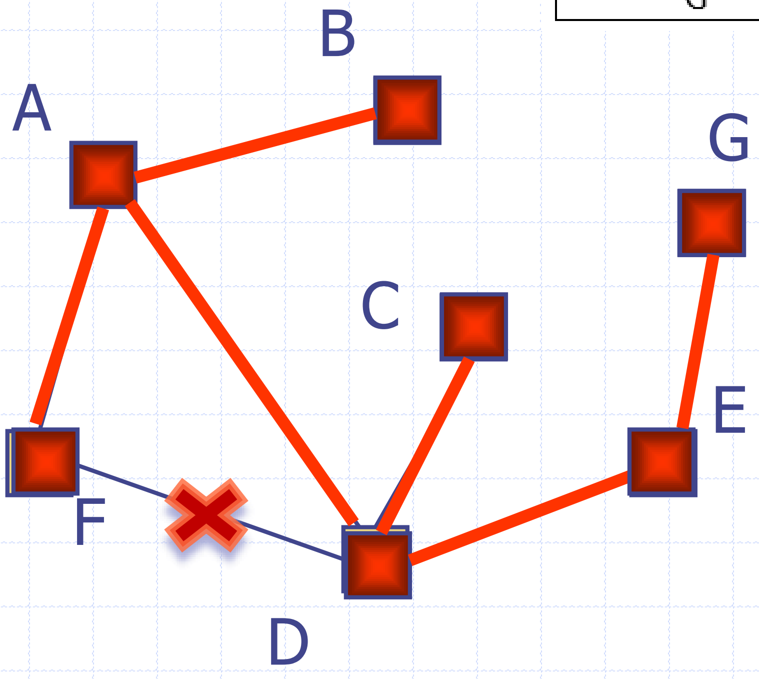


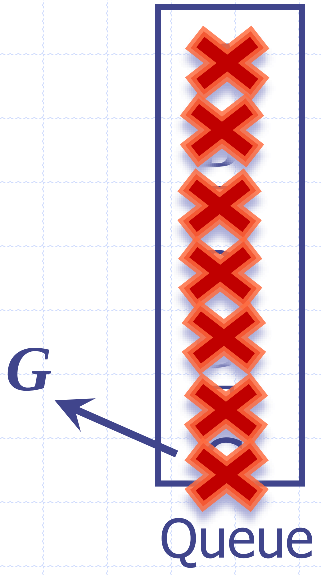
Spanning Tree: T = {AB, AF, AD, DC, DE, EG }

Vertex	Level
A	0
B	1
F	1
D	1
C	2
E	2
G	3

Worked Example

Remove G from Queue. There are no unvisited vertices adjacent to G and now Queue is empty. BFS algorithm terminates. Set Lev(G) = 3.





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

Facts About Trees

- Suppose T is a rooted tree with root r and levels Lev0, Lev1, Lev2, … have been specified. Then if vertices u and v belong to the same level, they are not adjacent.
  [Proof: Otherwise, T would contain a cycle]

- Suppose T is a rooted tree with root r and levels Lev0, Lev1, Lev2, … have been specified. Then every vertex other than the root has a parent.
  [Proof: By construction of the levels]

- Every rooted tree with one or more vertices has at least one vertex of degree 1.
  [Proof: Consider a vertex at the bottom level.]

Number of Edges in a Tree

Theorem. Every tree with $n \geq 1$ vertices has exactly $n - 1$ edges.

Proof. Proceed by induction on n. The base case $n = 1$ is obvious. Assume any tree with n vertices has $n - 1$ edges and let T be a tree with $n + 1$ vertices. We show T has n edges. T has a vertex v of degree 1; let uv be the only edge that is incident to v in T. Let $T' = T - \{uv\}$ (T' is the graph obtained by removing uv from T.)

Note: T' is also a tree. (Verify!) Now T' has n vertices so by the induction hypothesis, T' has $n - 1$ edges. We can reconstruct T by adding uv back to T': $T = T' \cup \{uv\}$. It is now clear that T has n edges. This completes the induction.

Minimum Number of Edges in a Connected Graph

Theorem. Suppose G is a connected graph with n vertices and m edges. Then m ≥ n – 1.

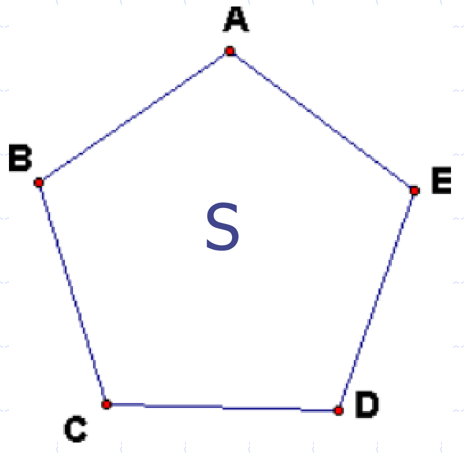
Proof. Since G is connected, we can use the Spanning Tree algorithm to obtain a subgraph T of G that is spanning and that is a tree. Since T spans G, T has n vertices. Since T is a tree, T has n -1 edges. Since T is a subgraph of G, G must have at least as many edges as T; in other words, m ≥ n – 1

When Must a Graph Contain a Cycle?

Theorem. Suppose G is a connected graph with n vertices and m edges. Then G contains a cycle if and only if $m \geq n$.

Proof. Suppose $m \geq n$ but G contains no cycle, then G must be a tree. In that case, $m = n - 1$ (so $m < n$), contradicting the fact that $m \geq n$. Therefore $m \geq n$ implies G contains a cycle.

Conversely, suppose G contains a cycle. It follows that G has a simple cycle S – something like the image on the right. Starting from S, we can rebuild G one edge at a time. Suppose S has k vertices and k edges. If we restore one edge, we will increase number of edges and vertices by 1. The next time we add an edge, we will increase the number of edges by 1 and number of vertices by 0 or 1. Continuing, it will always be true that there are at least as many edges as vertices. Therefore, G has the property that $m \geq n$.



When Must a Graph Contain a Cycle?

Corollary. If two vertices in a tree are joined by adding a new edge, the resulting graph contains a cycle.

Proof. Suppose the starting tree has n vertices and m = n-1 edges. Joining two vertices with an edge produces a graph with n vertices and m = n edges. Since joining two vertices with an edge does not destroy connectedness, the Theorem tells us that the new graph contains a cycle.

Theorem. Suppose G is any simple graph (not necessarily connected) having n vertices and m edges. If $m \geq n$, then G contains a cycle.

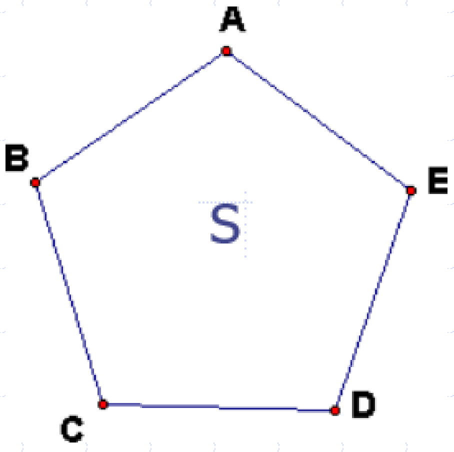
Proof. Exercise (Lab 10)

Question. Is it possible for a graph G to contain a cycle even if $m < n$?

Relationship Between Connected and Acyclic Graphs

Theorem. Suppose G is a graph with n vertices and m edges and suppose m = n – 1. Then G is connected if and only if G is acyclic.

Proof that G connected implies G acyclic. Argue by contradiction: Assume G actually has a cycle. By the theorem on the previous slide, it would follow that m ≥ n, a contradiction



continued

Proof that G acyclic implies G connected. Assume that G is is acyclic but not connected; we arrive at a contradiction.

Since G is not connected, it is the union of two or more connected components: $G = G_1 \cup G_2 \cup . . . \cup G_k$. For each i, let n_i be the number of vertices in G_i and let m_i be the number of edges in G_i. Since G is acyclic, each component is acyclic, and so each G_i is a tree. It follows that for each i, $m_i = n_i - 1$. We compute total number m of edges in G:

$$m = m_1 + m_2 + . . . + m_k = (n_1 - 1) + (n_2 - 1) + . . . + (n_k - 1)$$
$$= (n_1 + n_2 + . . . + n_k) - k = n - k < n - 1 \text{ --- a contradiction!}$$

More About Why the Spanning Tree Algorithm Works

1. Notice that as the Spanning Tree algorithm executes, the number of edges selected is always one less than the number of vertices visited so far. This means that at each step of the algorithm, the list of edges named so far forms a tree.
2. Also, the algorithm does not allow an edge to be selected when an already visited vertex occurs. If an edge were selected at that point, it would join two already visited vertices – in other words, two vertices already on the tree built so far would be joined by an edge. By the Corollary, this would create a cycle.

A Criterion for Detecting a Cycle

Theorem. Suppose G is a graph with n vertices and m edges. Let F be a spanning forest for G (F is the union of othe spanning trees inside each component of G). Let m_F denote the number of edges in F. Then

G has a cycle if and only if $m > m_F$

Proof. Write $G = G_1 \cup G_2 \cup . . . \cup G_k$ as the union of its connected components. For each i, let n_i be the number of vertices in G_i and let m_i be the number of edges in G_i. Let $F = F_1 \cup F_2 \cup . . . \cup F_k$, where each F_i is a spanning tree of G_i. Let m_{Fi} denote the number of edges in F_i. Notice that for each i, $m_i \geq m_{Fi}$. If some G_i contains a cycle, then we have $m_i \geq n_i > m_{Fi}$; it follows that $m > m_F$. Conversely, if $m > m_F$, then for some i, $m_i > m_{Fi} = n_i - 1$; in other words, $m_i >= n_i$. It follows that G_i contains a cycle, and so G itself contains a cycle.

Implementing Algorithms to Solve Basic Problems

1. Is G connected? How many components does G have?

   Solution Keep track of number of components by incrementing a counter in additionalProcessing step

2. Given a graph G, does G have a cycle?
   Solution ??

3. Given vertices x, y in a graph G, is there a path in G from x to y?
   Solution: ??

Finding Shortest Path Length

- A special characteristic of the BFS style of traversing a graph is that, with very little extra processing, it outputs the shortest path between any two vertices in the graph.
- If p: $v_0 - v_1 - v_2 - \ldots - v_k$ is a path in G, recall that its length is k, the number of edges in the path. BFS can be used to compute the shortest path between any two vertices of the graph.

Finding Shortest Path Length

- Given a connected graph G with vertices s and v, here is the algorithm to return the shortest length of a path in G from s to v
  - Perform BFS on G starting with s to obtain the BFS rooted tree T with root s, together with a map recording the levels of T
  - Return the level of v in T

Optional Proofs

Optional: Simple Paths

Theorem: Suppose G is a simple graph, u, v are distinct vertices, and there is a path p in G that starts at u and ends at v. Then there is a simple subpath of p in G that starts at u and ends at v.

Proof: It is enough to show that whenever there is a path p in G from u to v that is not simple, there is another shorter subpath of p from u to v in G.

Suppose p: u = $v_1, v_2, . . . , v_k$ = v is a path from u to v in G that is not simple.

Case 1: *Some vertex occurs twice in p*. Suppose $i < j$ and $v_i = v_j$ in p. Then notice that u = $v_1, v_2, . . . , v_i, v_{j+1}, . . ., v_k$ = v is another shorter path from u to v.

Case 2: *Some edge occurs twice in p*. In this case, some vertex must also occur twice, and we return to the argument in Case 1.

Optional: Simple Cycles

Theorem: Suppose G is a simple graph and e is an edge in G. Suppose G contains a cycle C, and e is an edge in C. Then some simple cycle in G also has e as an edge; in fact, such a simple cycle can be found that is a subpath of C.

Proof: Suppose G is a simple graph having edge e, and that G has a cycle C of length n that contains e. We show e belongs to a simple cycle in G that is a subpath of C, by induction on n.

Base Case: n = 3. In that case, C must already be simple.

Induction Step: Assume that whenever a cycle C′ of length < n in G contains e, e belongs to a simple cycle in G that is a subpath of C′. Let C be a cycle of length n in G, and write C as

$$v=v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_n = v$$

and let e be the edge (v_i, v_{i+1}). Assume C is not simple, so there are j < k where $v_j = v_k$

There are 3 cases:

(continued)

Case 1 v_j precedes v_i and v_{i+1} precedes v_k

$$v=v_1, v_2, \ldots, v_j, \ldots, v_i, v_{i+1}, \ldots, v_k, \ldots, v_n = v$$

Then since the section from v_j to v_k is a cycle C′ containing e, there is a simple cycle C″, which is a subpath of C′, which contains e. Now C″ must also be a subpath of C

Case 2 Both v_j and v_k precede v_i

$$v=v_1, v_2, \ldots, v_j, \ldots, v_k, \ldots, v_i, v_{i+1}, \ldots, v_n = v$$

Then if $1 < j$, we can delete the section from v_j to v_k (include v_j not v_k) and the resulting path is a shorter cycle containing e, so there is a simple cycle in G containing e, which again must be a subpath of C. If $j = 1$, then if we delete v_j up to v_k (include v_j not v_k), it follows that $v_1 = v_k = v$, so the path from v_k to v_n is a shorter cycle containing e, so again there is a simple cycle, also a subpath of C.

Case 3 Both v_j and v_k occur after v_{i+1}

$$v=v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_j, \ldots, v_k, \ldots, v_n = v$$

If $k < n$, delete section from v_j up to (not including) v_k; again we have a shorter cycle containing e. If $k = n$, if we delete from v_j up to (not including v_k), since $v_j = v$, we again have a shorter cycle containing e. Either way, there is a simple cycle in G containing e which is also a subpath of C.

Optional: Odd Cycles

Theorem: Suppose G is a simple graph that has an odd cycle C (a cycle with odd length). Then G has an odd *simple* cycle that is a subpath of C.

Proof: We show that whenever C is an odd cycle of length n in G, either C is simple or some subpath of C is an odd simple cycle; we proceed by induction on n.

Base Case: n = 3. In that case, C must already be simple.

Induction Step: Assume that whenever C′ is an odd cycle of length < n in G, C′ contains an odd simple subcycle. Let C be an odd cycle of length n. Write C as

$$v=v_1, v_2, \ldots, v_i, v_{i+1}, \ldots, v_n = v.$$

Assuming C is not simple, let i < j be such that $v_i = v_j$. Then both

$v_i, v_{i+1}, \ldots, v_j$ and $v_1, v_2, \ldots, v_i\ v_{j+1}, \ldots, v_n$ are subcycles of C with no common edges. Therefore, one of these cycles must be odd. By the induction hypothesis, the odd cycle must have an odd simple subcycle.