CSC 225

Graphs

Topics

- Motivation
- Graphs
  - Undirected
  - Directed
  - DAGs
- Representation
- Traversals
- DFS and BFS
- Tree, cross, back, forward edges
- Reachability
- Connectivity
- Cycles
- Shortest paths
- Spanning trees
- Minimum Spanning Trees
- Topological sorting
- Strongly connected components
- Bi-connected components

History

- 1736
  - Euler (Swiss)
  - Seven Bridges of Königsberg
- 1878
  - The term graph was introduced by Sylvester in a Nature article
- 1930
  - Graph theory evolved into an organized brand of Mathematics
- 1959
  - Dijkstra's Shortest Path Algorithm
- 1972
  - Tarjan and Hopcroft: depth first search; Turing Award 1987

http://en.wikipedia.org/wiki/Graph_theory

Most Famous Graph Problem

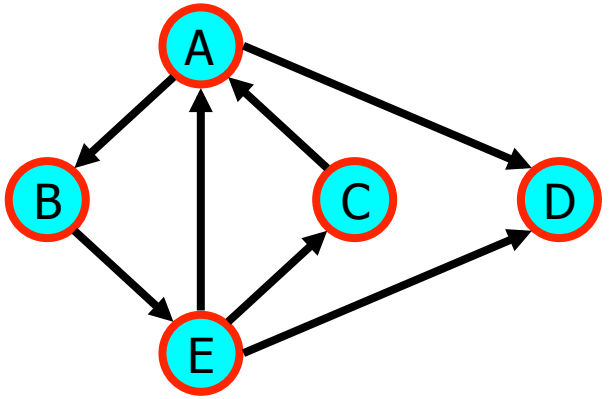
- Is it true that any map drawn in the plane may have its regions colored with ***four colors***, in such a way that any two regions having a common border have different colors?
- Problem remained unsolved for more than a century
  - 1976: First proof by Appel and Haken
  - 1996: Simpler proof by Robertson, Seymour, Sanders, Thomas

Applications and Problem Types

- Scheduling problems
  - operations research
- Network analysis
  - electrical and computer engineering, computer science
- Molecular structures
  - Chemistry and physics
- Structure of programs and software systems
  - computer science, software engineering
- Markov chains
  - queuing theory

Abstract Meaning of the Term Graph

- A *graph* $G = (V, E)$ is a set V of *vertices* (*nodes*) and a collection E of pairs from V, called *edges* (*arcs*).



- Directed Graph Example

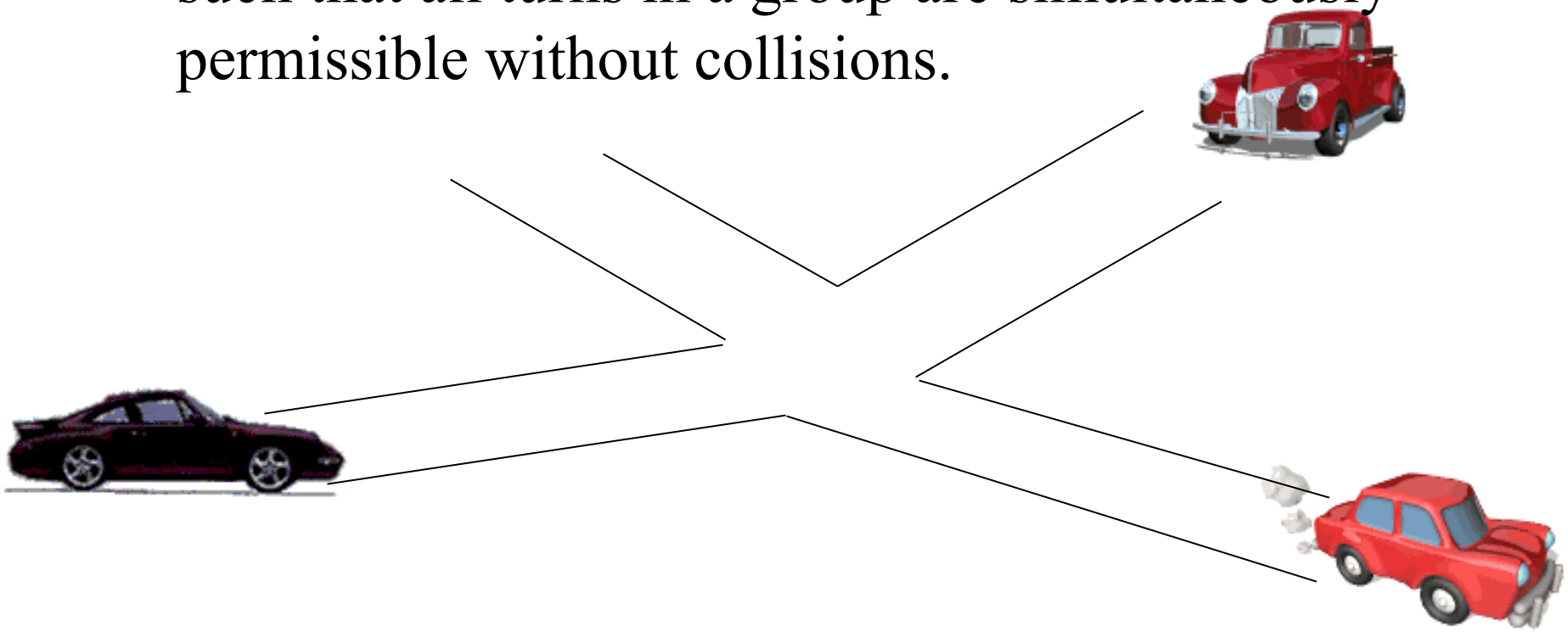
$$V = \{A, B, C, D, E\}$$

$$E = \begin{Bmatrix} (A,B);(A,D);(B,E);(C,A); \\ (E,A);(E,C);(E,D); \end{Bmatrix}$$

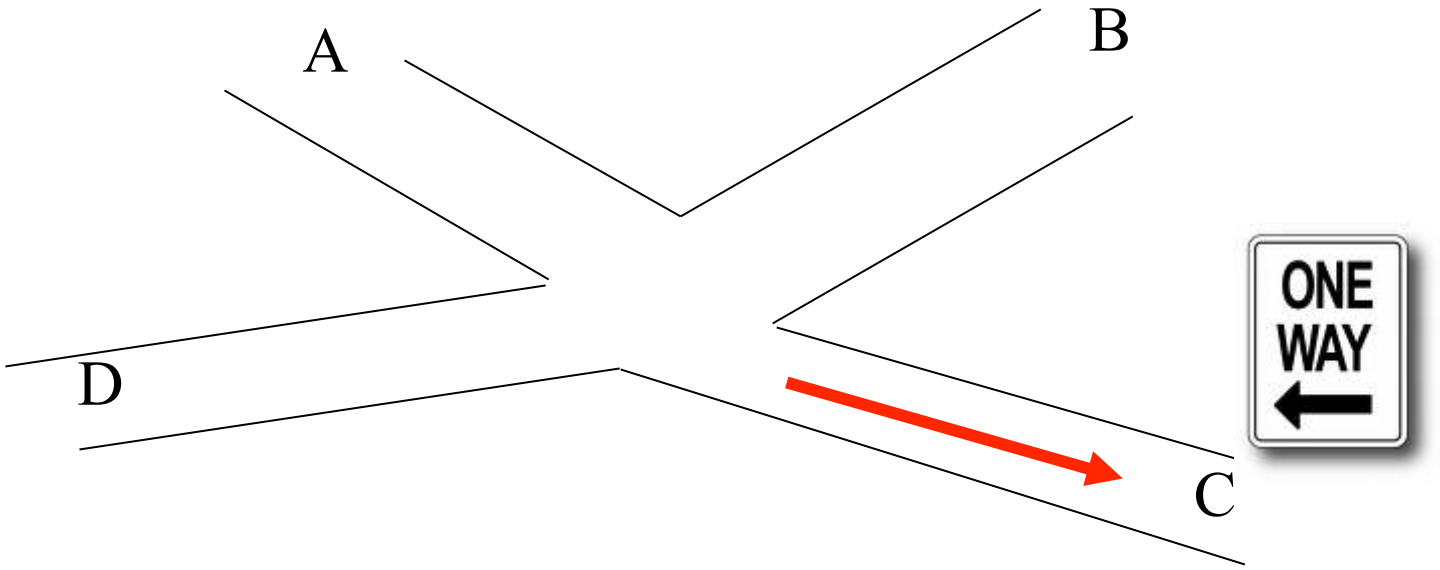
Design of traffic light phases for an intersection of roads

Given: A set of permitted turns of intersections

Goal: Partition the set into as few groups as possible such that all turns in a group are simultaneously permissible without collisions.

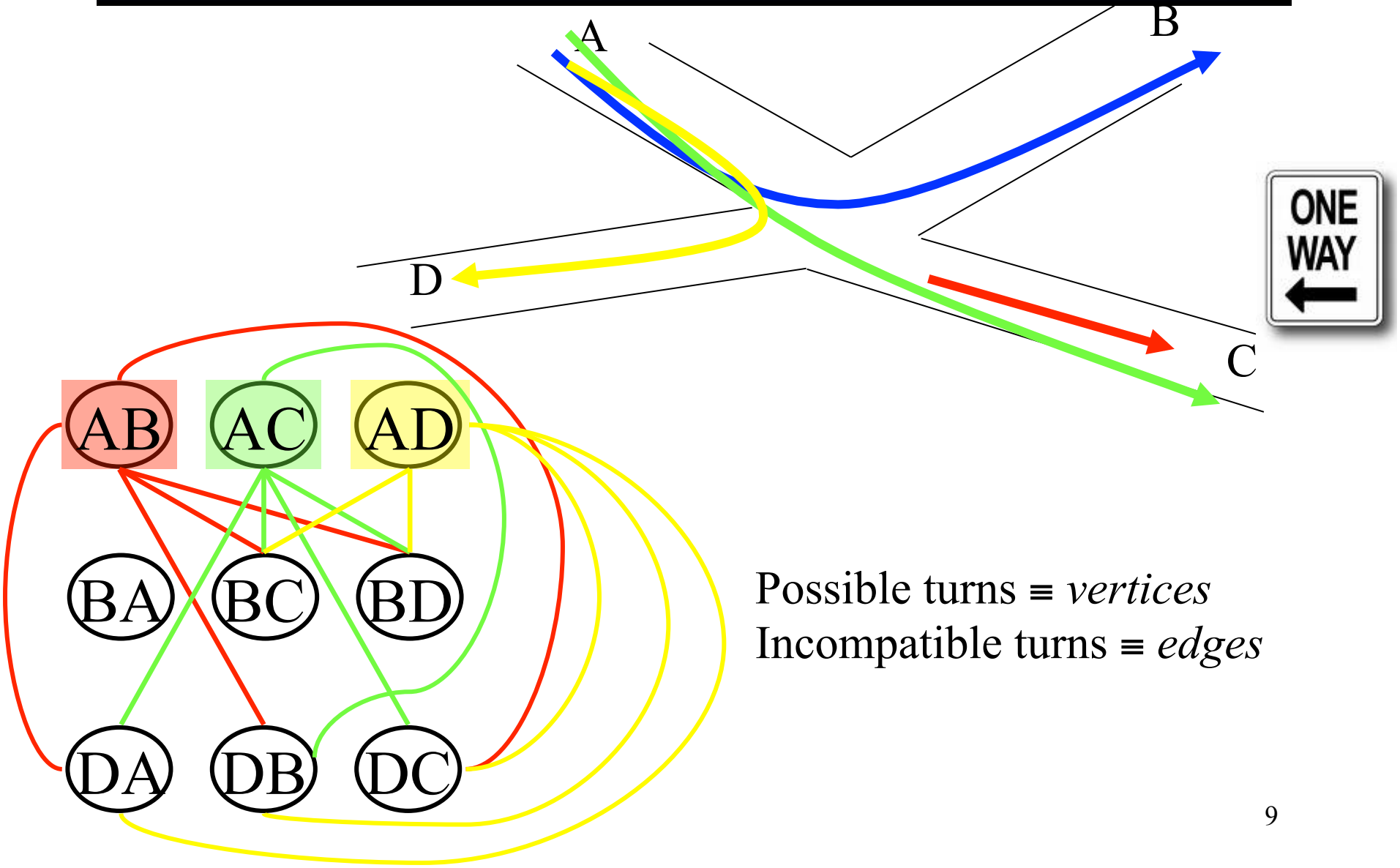


Possible Turns



- AB, AC, AD
- BA, BC, BD
- DA, DB, DC

The Model: A Conflict Graph



Possible turns ≡ *vertices*
Incompatible turns ≡ *edges*

The Model: A Conflict Graph

- Complete the conflict graph for the other two rows of the graph (done in class)

- Find a traffic phase assignment such that no two turns that have a possible collision have a green light at the same time
- Minimize the number of phases
- Implement domain-specific optimization

Problem Solving

- **Model:** Conflict Graph
- **Data Structure:** graph
- **Algorithm:** graph colouring

Graph Colouring Problem

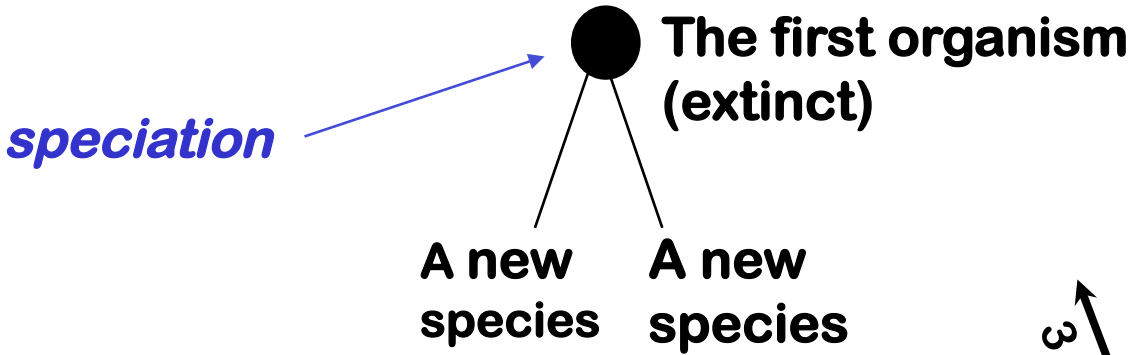
Input: A *graph* $G = (V, E)$ with *vertices* and *edges* E.

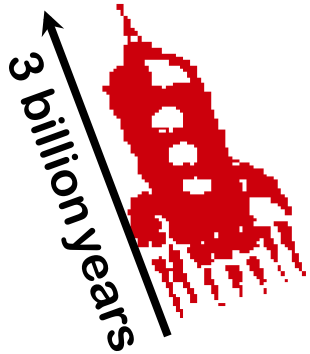
- Possible turns ≡ *vertices*
- Incompatible turns ≡ *edges*

Output: A minimum set of colours and an assignment from the colours to the vertices such that no two vertices that have an edge in common have the same colour.

- Colour ≡ *traffic light phase*

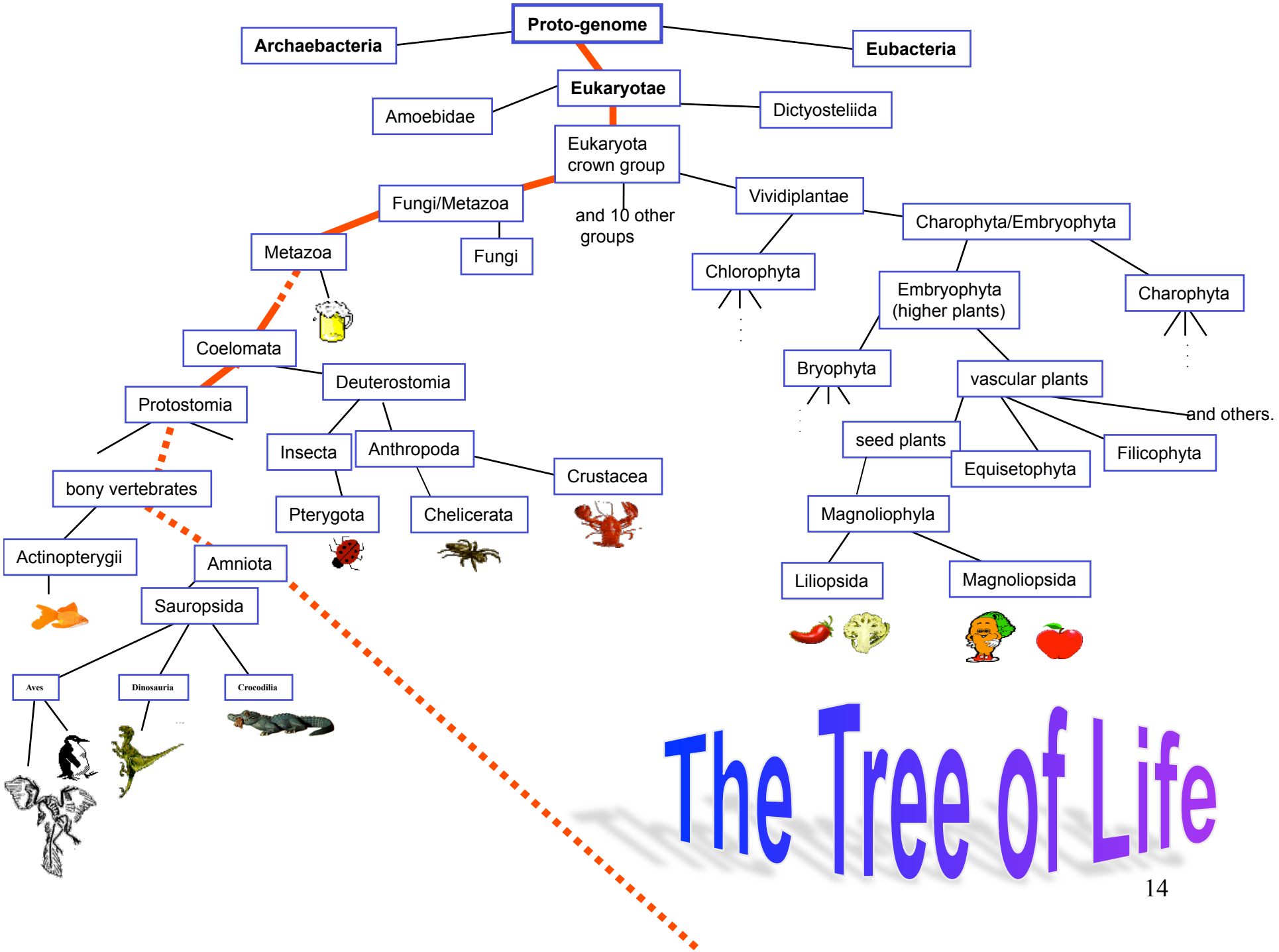
Computational Biology

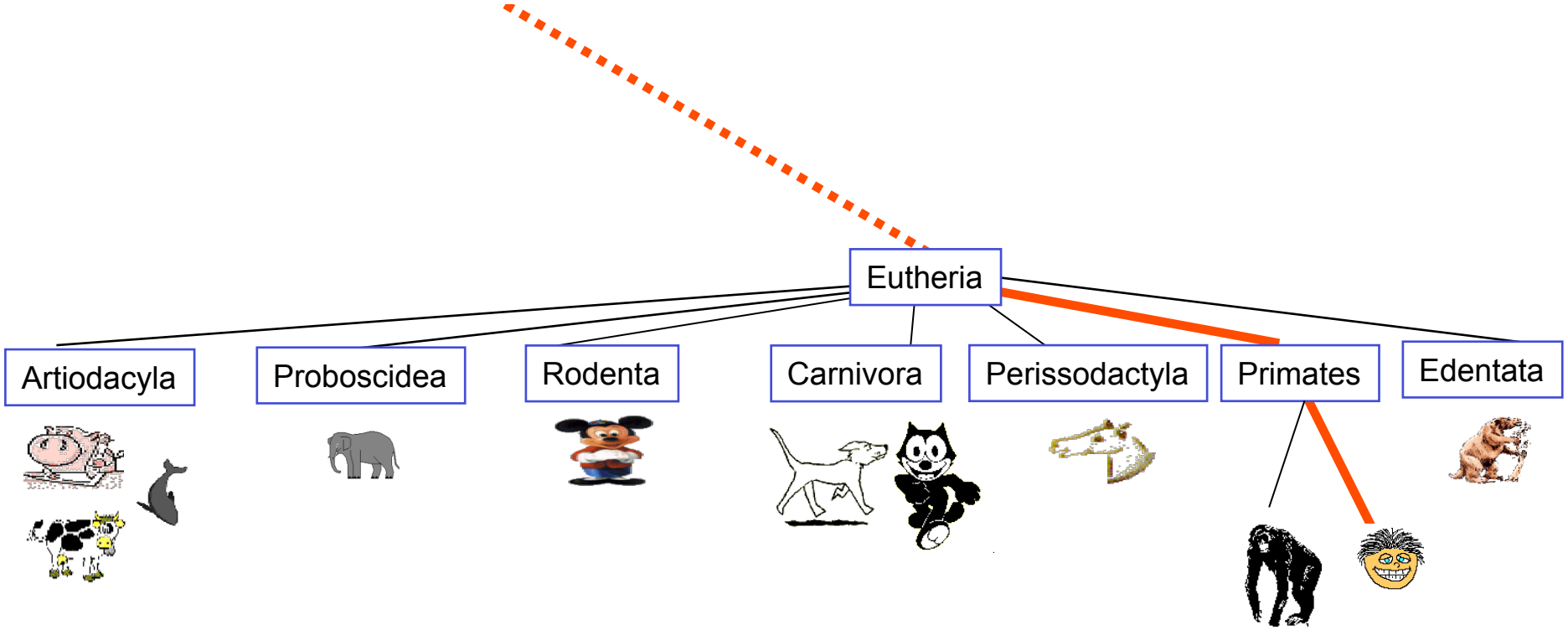




2002

How did life evolve?


Proto-genome
Archaebacteria
Eubacteria
Eukaryotae
Amoebidae
Dictyosteliida
Eukaryota crown group
and 10 other groups
Fungi/Metazoa
Fungi
Metazoa
Vividiplantae
Chlorophyta
Charophyta/Embryophyta
Embryophyta (higher plants)
Charophyta
Bryophyta
vascular plants
and others.
seed plants
Equisetophyta
Filicophyta
Magnoliophyla
Liliopsida
Magnoliopsida
Coelomata
Deuterostomia
Protostomia
Insecta
Anthropoda
Crustacea
Pterygota
Chelicerata
bony vertebrates
Actinopterygii
Amniota
Sauropsida
Aves
Dinosauria
Crocodilia
The Tree of Life



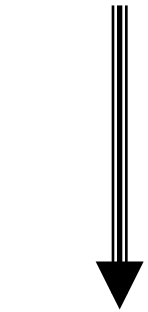
Tree of life: (1) represents the relationships among the known species

(2) binary tree

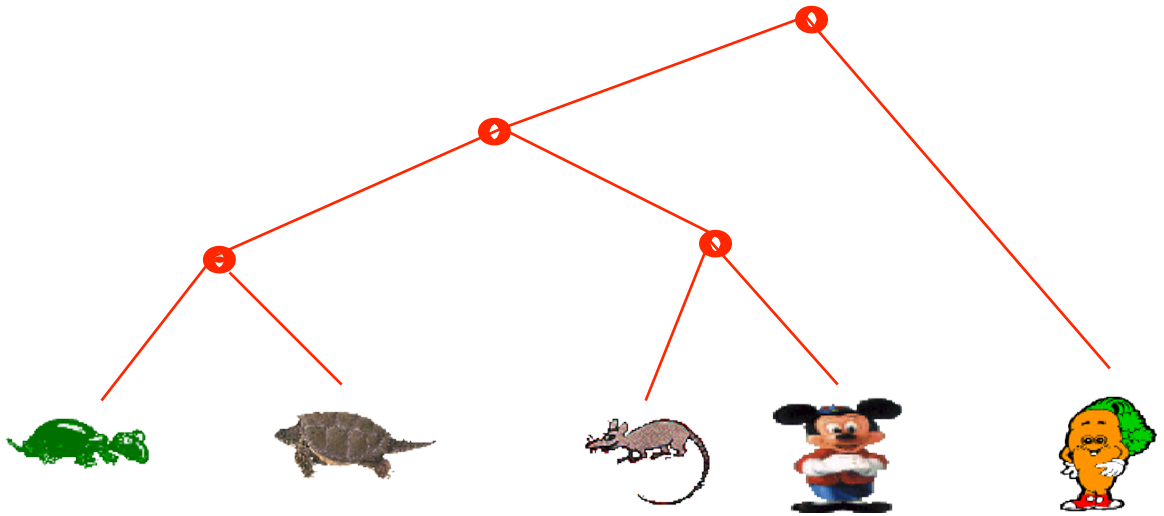
(3) leaf labeled

(4) evolutionary tree/phylogenetic tree

How can we determine the tree of life?



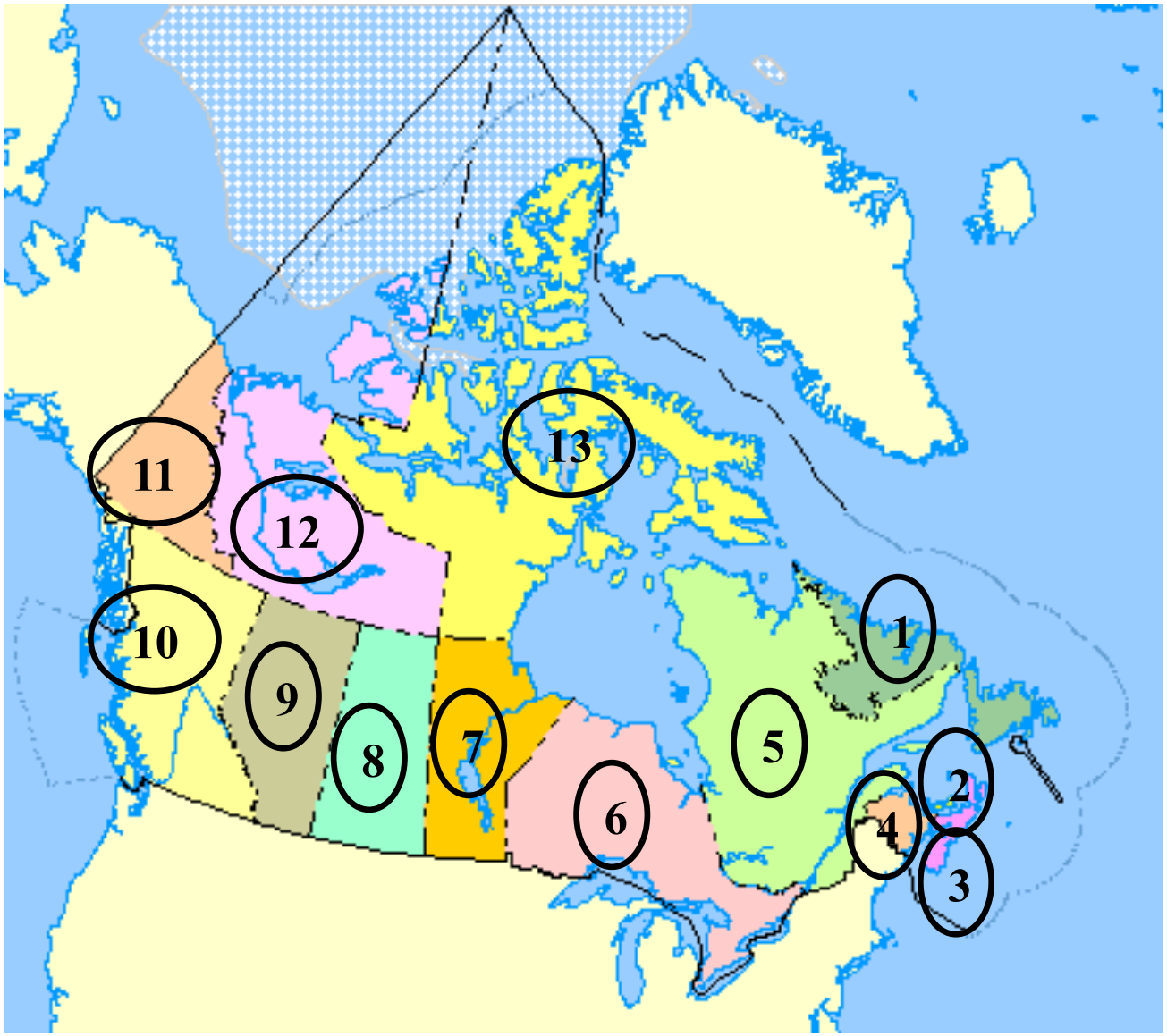
Determining the relationships: Which species are most similar?



Different types and special cases of graphs

- *Directed* graphs *(Digraphs):* The edges of the graph are *directed* (i.e., the pairs are ordered, $(a,b) \neq (b,a)$)
- *Undirected* graphs: The edges of the graph are *undirected* (i.e., the pairs are not ordered, $(a,b) = (b,a)$)
- Trees
  - Rooted trees
  - Unrooted trees
  - Binary trees
  - AVL trees
  - Heaps

Example of an undirected graph



1 Newfoundland and Labrador

2 Prince Edward Island

3 Nova Scotia

4 New Brunswick

5 Quebec

6 Ontario

7 Manitoba

8 Saskatchewan

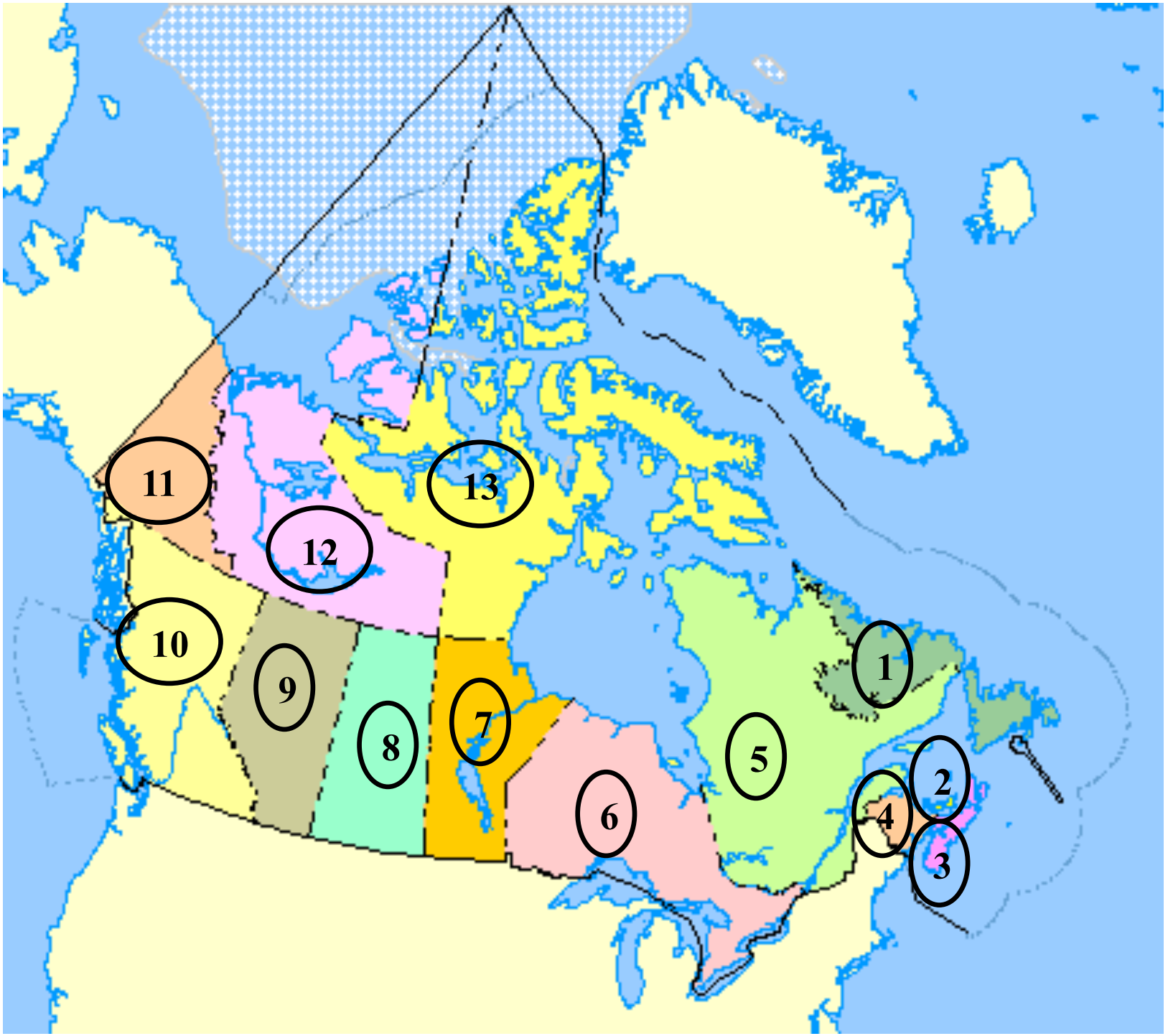
9 Alberta

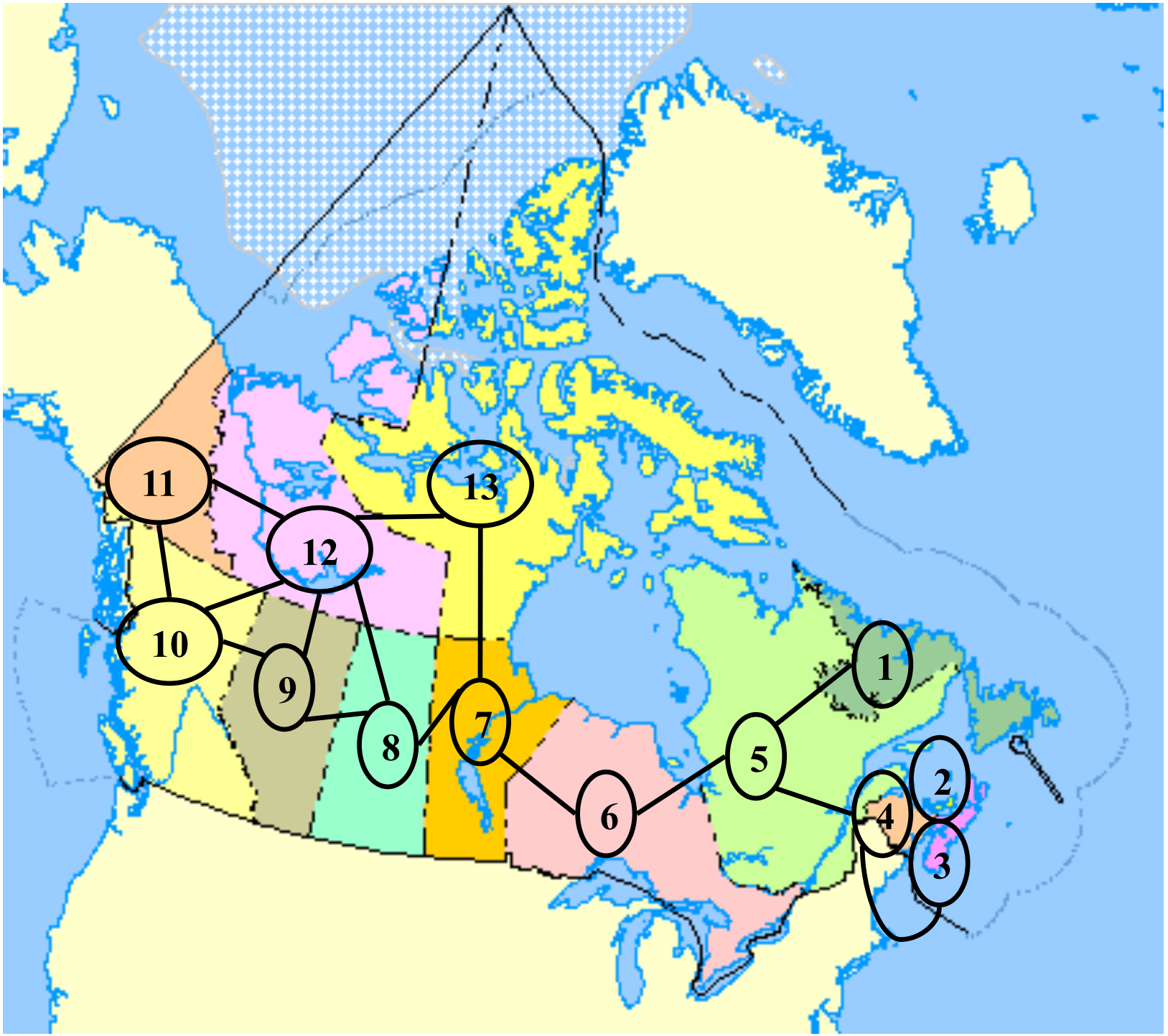
10 British Columbia

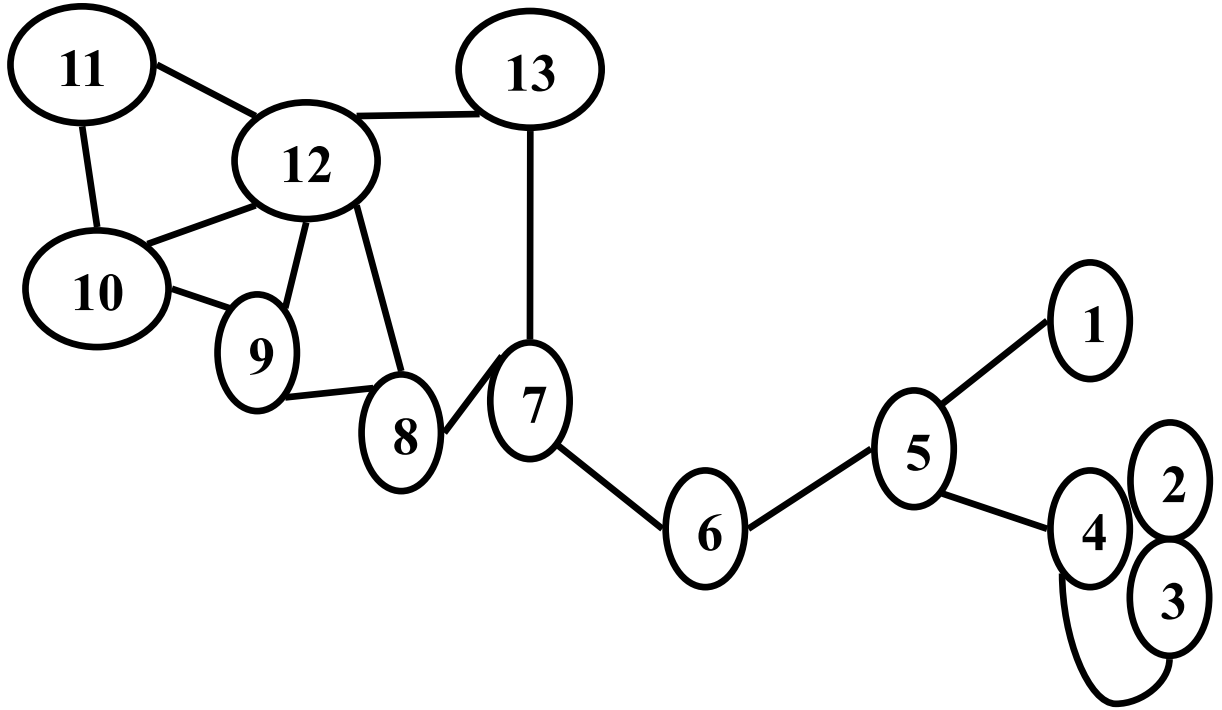
11 Yukon Territory

12 Northwest Territories

13 Nunavut


11
13
12
10
9
8
7
6
5
1
2
4
3


11
13
12
10
9
8
7
6
5
1
2
4
3

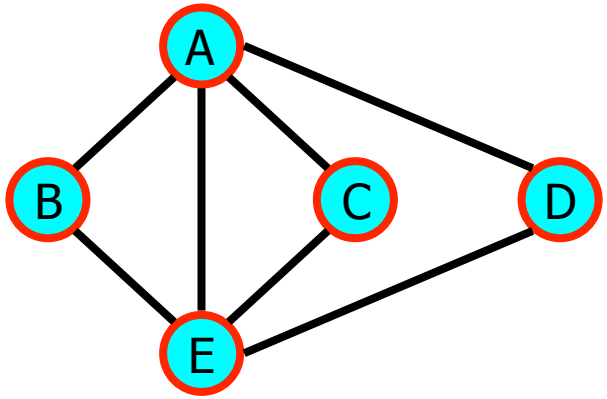


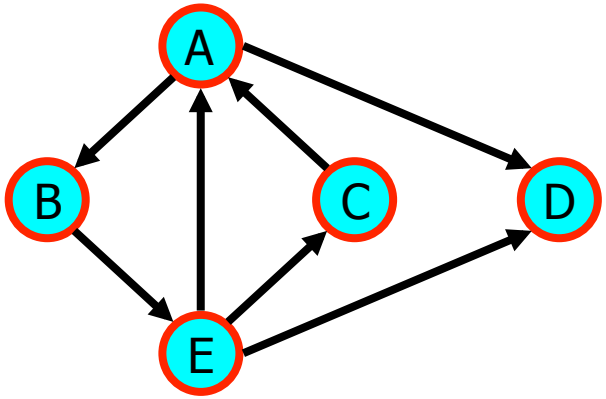
V={1,2,3,4,5,6,7,8,9,10,11,12,13}

E= {(1,5),(3,4),(5,4),(5,6),(6,7),(7,8),(8,12),
(7,13),(8,9),(13,12),(11,12),
(11,10),(12,10),(9,12),(10,9)}

Graph Terminology

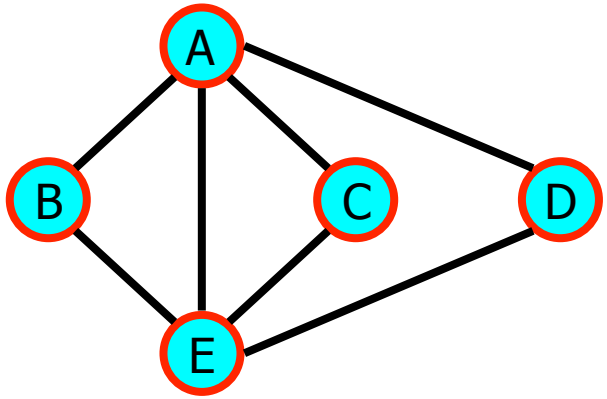
- Much of the terminology for graphs is applicable to directed graphs and undirected graphs





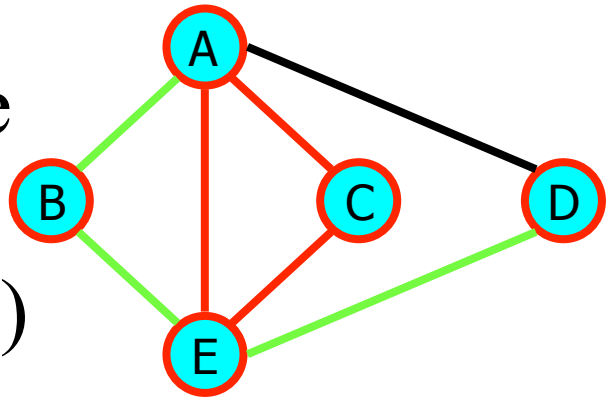
Undirected Edges

- An *undirected edge e* represents a *symmetric* relation between two vertices *v* and *w* represented by the vertices. We usually write $e = (v, w)$, here (w, w) is represented by an unordered pair.
  - *v*, *w* are the endpoints of the edge
  - *v* is adjacent to *w*
  - *e* is incident upon *v* and *w*
- The degree of a vertex is the number of incident edges



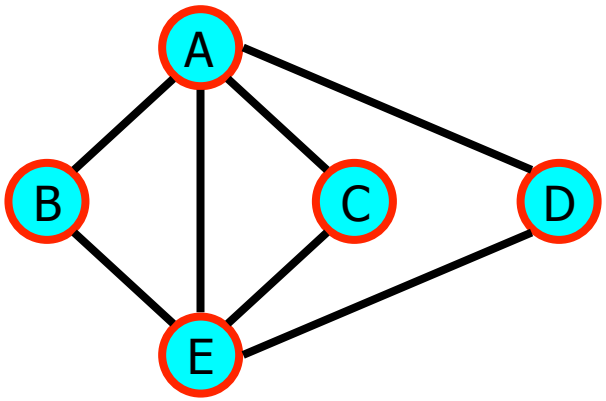
Undirected Paths

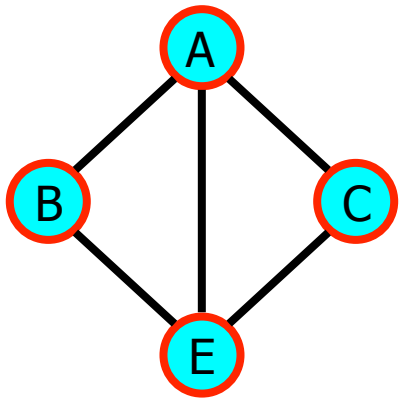
- A ***path*** in a graph is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that there are edges from v_1 to v_2, from v_2 to v_3, …, and from v_{n-1} to v_n
- The ***length*** of a path is the number of edges in the path
- A path is ***simple*** if all vertices on the path, except possibly the first and the last, are distinct (i.e., no *parallel* or *multi-edges*)



Connected Graph

- A graph is ***connected*** if every pair of vertices is connected.
- Example
  - Two connected components of a graph
  - Unconnected graph



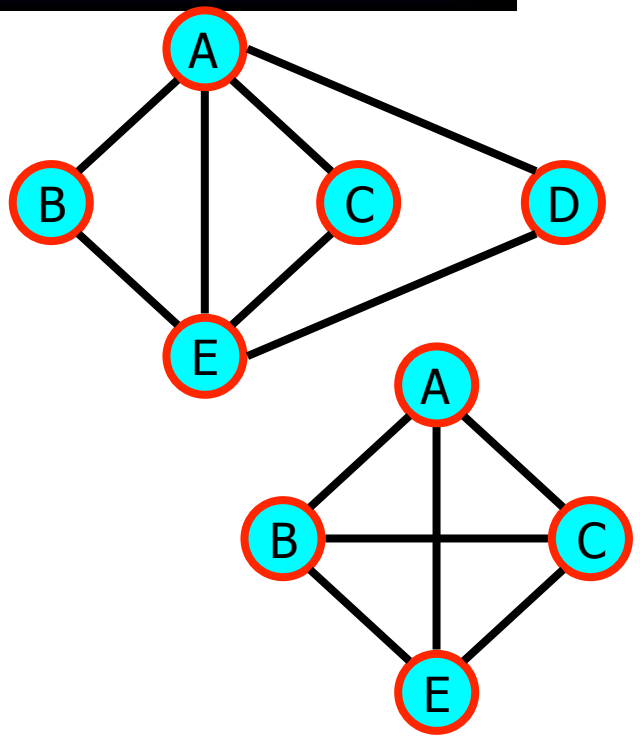


Simple and Complete Graphs

- A ***simple graph*** is a graph with no self-loops and no parallel or multi-edges

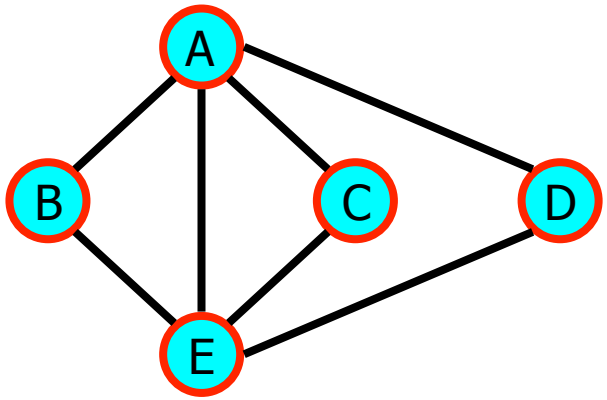
- A ***complete graph*** is a graph is a simple graph where an edge connects every pair of vertices

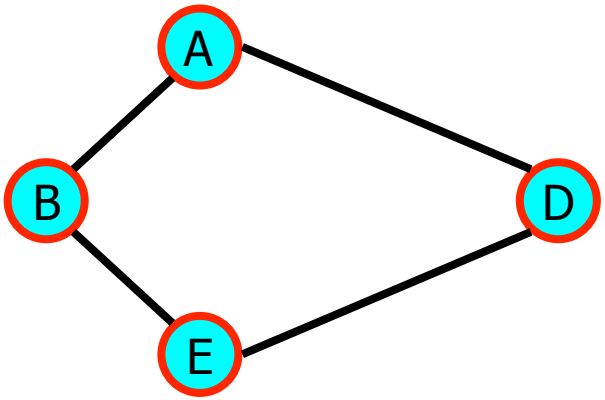
- The complete graph on n vertices has n vertices and $n(n-1)/2$ edges
- A complete graph with at most one self loop per vertex on n vertices has n vertices and $n(n+1)/2$ edges



Subgraph

- Let $G = (V, E)$ be a graph with vertex set V and edge set E.
- A ***subgraph*** of G *is a graph* where $G' = (V', E')$
  - V' is a subset of V
  - E' consists of edges (v, w) in E such that both v and w are in V'

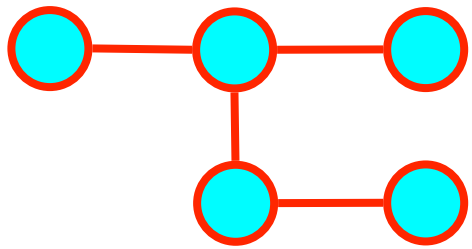




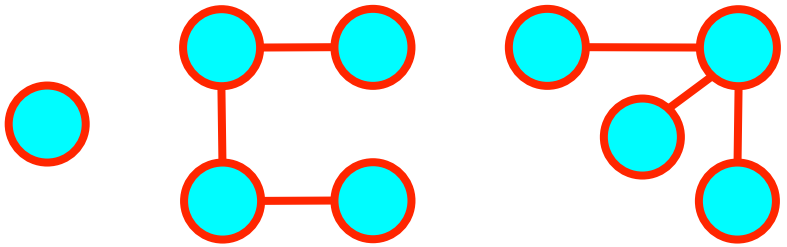
Trees and Forests

- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles

  This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



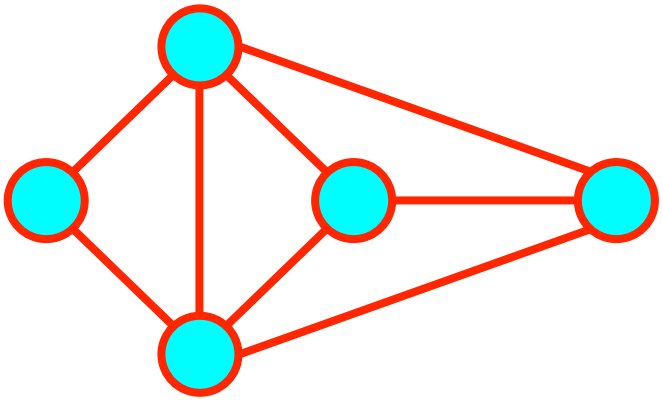
Tree



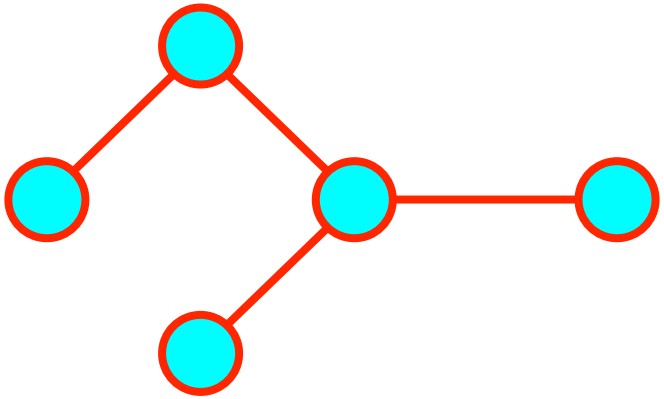
Forest

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest



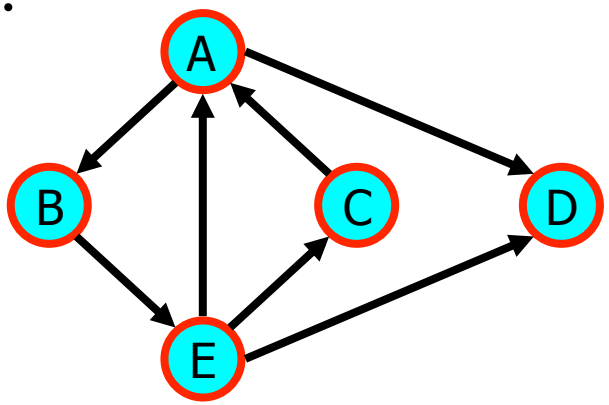
Graph



Spanning tree

Directed Edges or Arcs

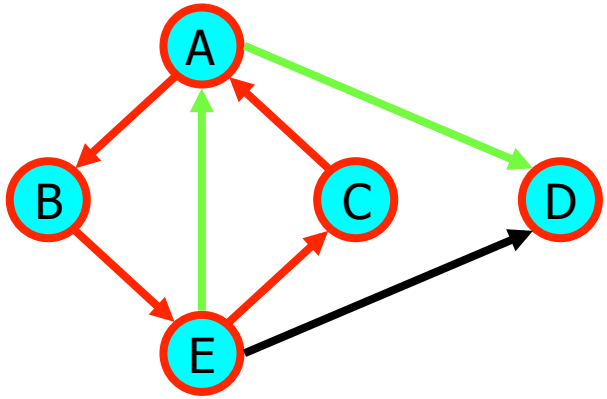
- A *directed edge* (*or arc*) *e* represents an *asymmetric* relation between two vertices *v* and *w*.
  e = (*v*, w) denotes an ordered pair.
  - *v*, *w* are the endpoints of the edge
  - *v* is adjacent to *w*
  - *e* is incident upon *v* and *w*
  - The arc goes from the source vertex *v* to the destination vertex *w*



- The indegree of a vertex is the number of incoming arcs
- The outdegree of a vertex is the number of outgoing arcs

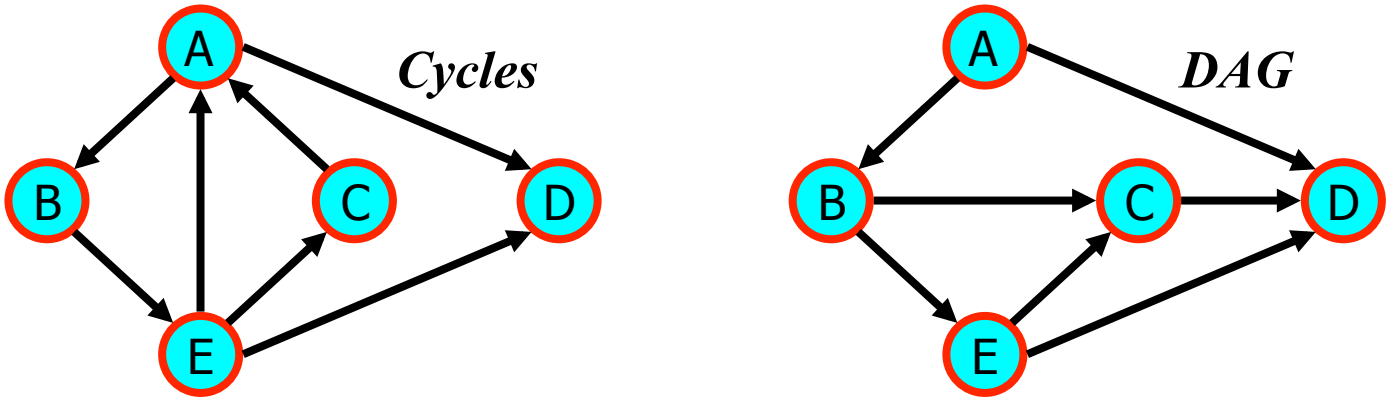
Directed Paths

- A ***path*** in a digraph is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that there are arcs from v_1 to v_2, from v_2 to v_3, …, and from v_{n-1} to v_n
- The ***length*** of a directed path is the number of arcs in the path
- A path is ***simple*** if all vertices on the directed path, except possibly the first and the last, are distinct (i.e., no *parallel* or *multi-edges*)



Directed Acyclic Graphs (DAGs)

- A directed acyclic graph (DAG) is a graph with no cycles.
- DAGs are more general than trees, but less general than arbitrary directed graphs.



Labeled Digraphs

- A graph with edge labels is called a labeled graph
- Edge labels denote edge attributes such as distance, throughput, capacity, time, etc.
- A vertex can have both a name and a label

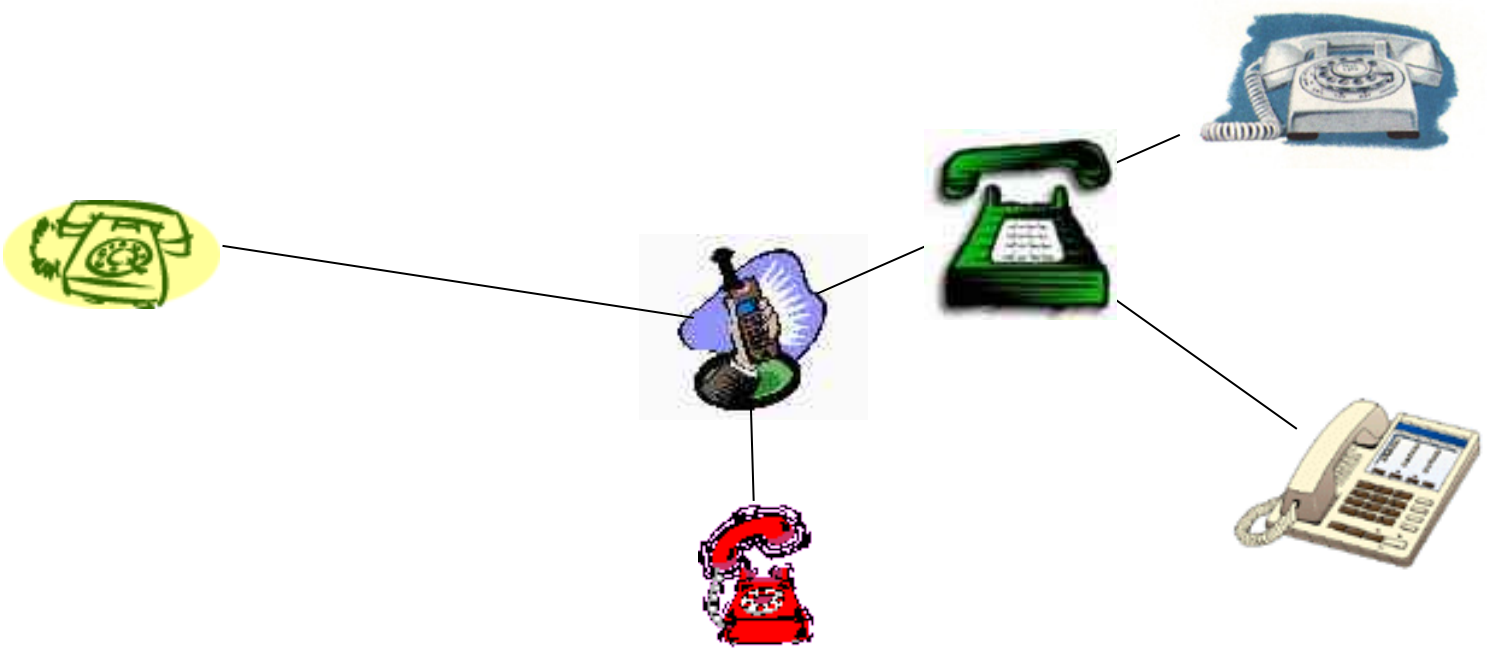
Example of a Directed Graph

- Street map of a city
  - directed edges model
  - one way and two-way streets
  - Intersections are vertices
  - Connecting road segments are edges
  - One-way streets are directed edges


Gorge Waterway
foot bridge
Gorge Rd East
Banfield Park
Victoria West Park
Esquimalt
VIC WEST
Upper Harbour
Victoria Harbour
Inner Harbour
Tourist Information Center
JAMES BAY
MacDonald Park
Beacon Hill Park
DOWNTOWN
Royal Athletic Park
Fisherman's Wharf
Helijet Terminal
Ogden Point
Cruise Ship Terminal
M.V. Coho – Vehicle Ferry to Port Angeles
Victoria Clipper – Passenger Ferry to Seattle
To Highway 1
To Highway 17
Johnson St. Bridge
Bay St. Bridge
Scale
.5 mile
.5 km
© Pacific Island Publishers 2001

Network Design

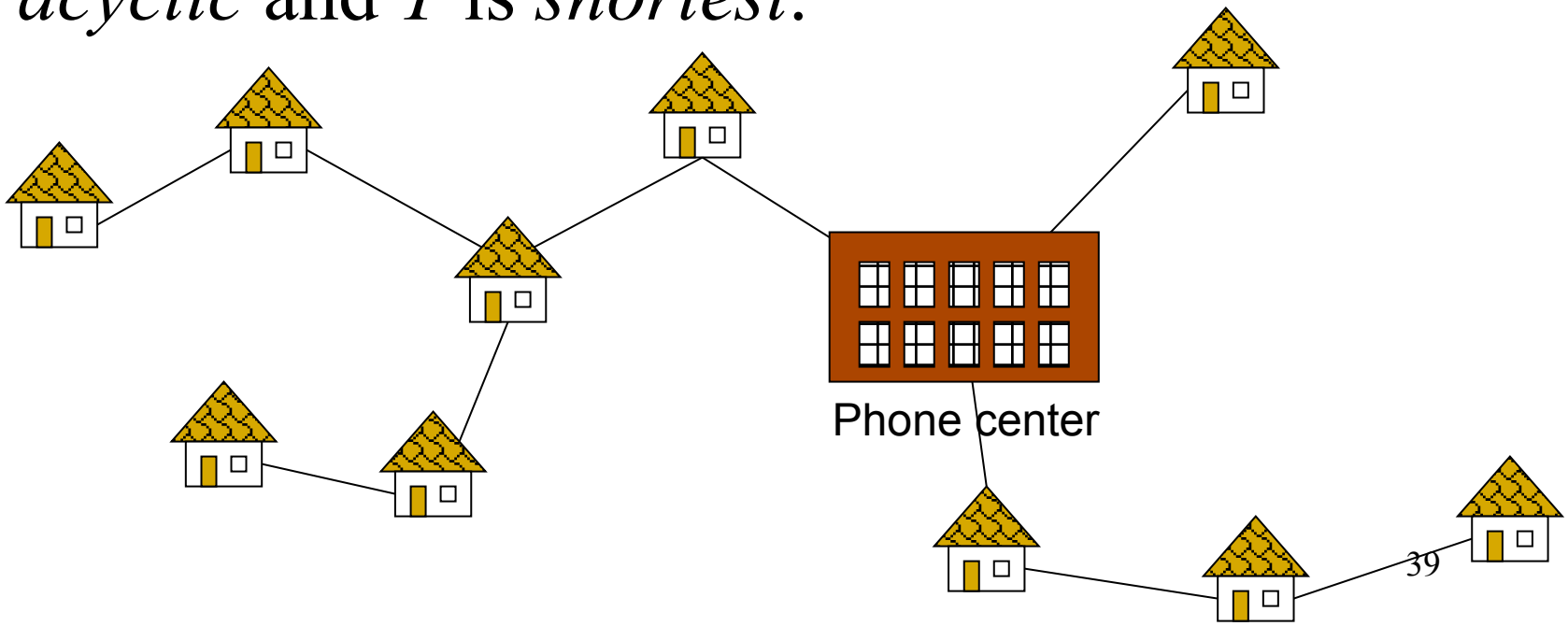
- A group of individuals, who are separated by varying distances, wish to be connected together in a telephone network.
- Determine the least costly network connecting everyone at a minimum cost.



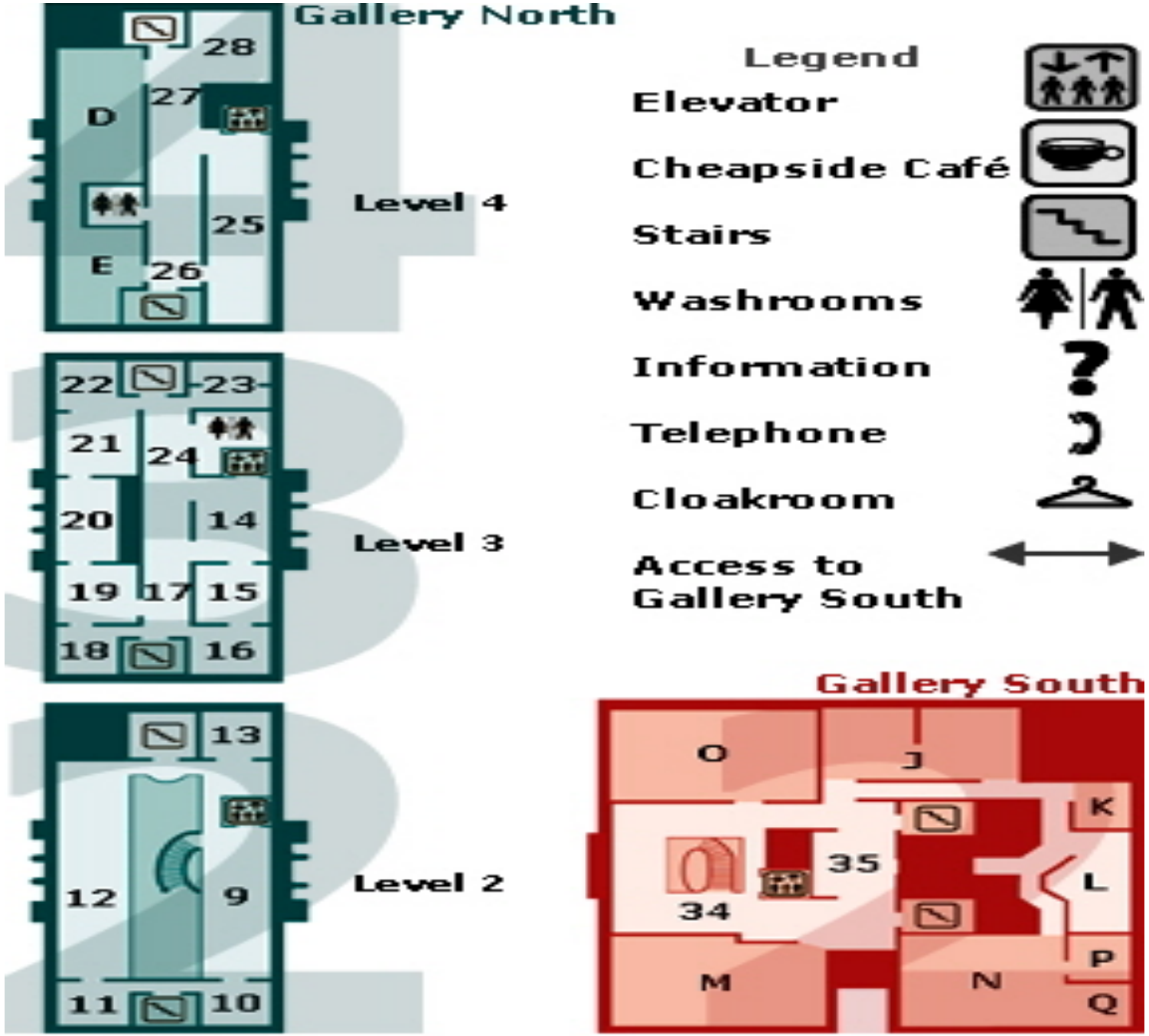
Minimum Spanning Tree Problem

Input: A *weighted* graph $G = (V, E)$ with edge weights w: $\rightarrow IN$.

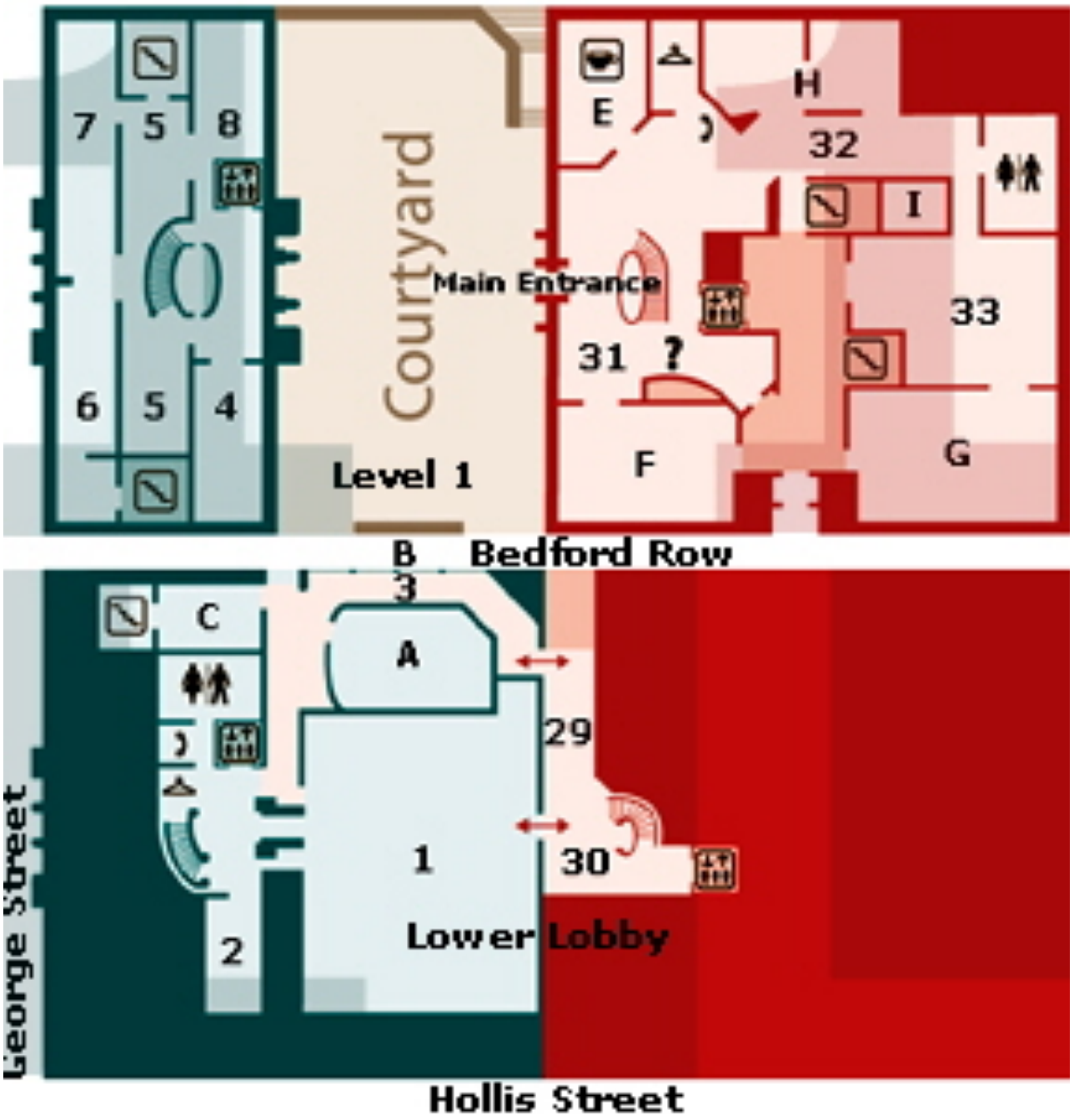
Output: A *minimum spanning tree* $T=(V, E')$, that is T is a *connected subgraph* of G, T is *acyclic* and T is *shortest*.



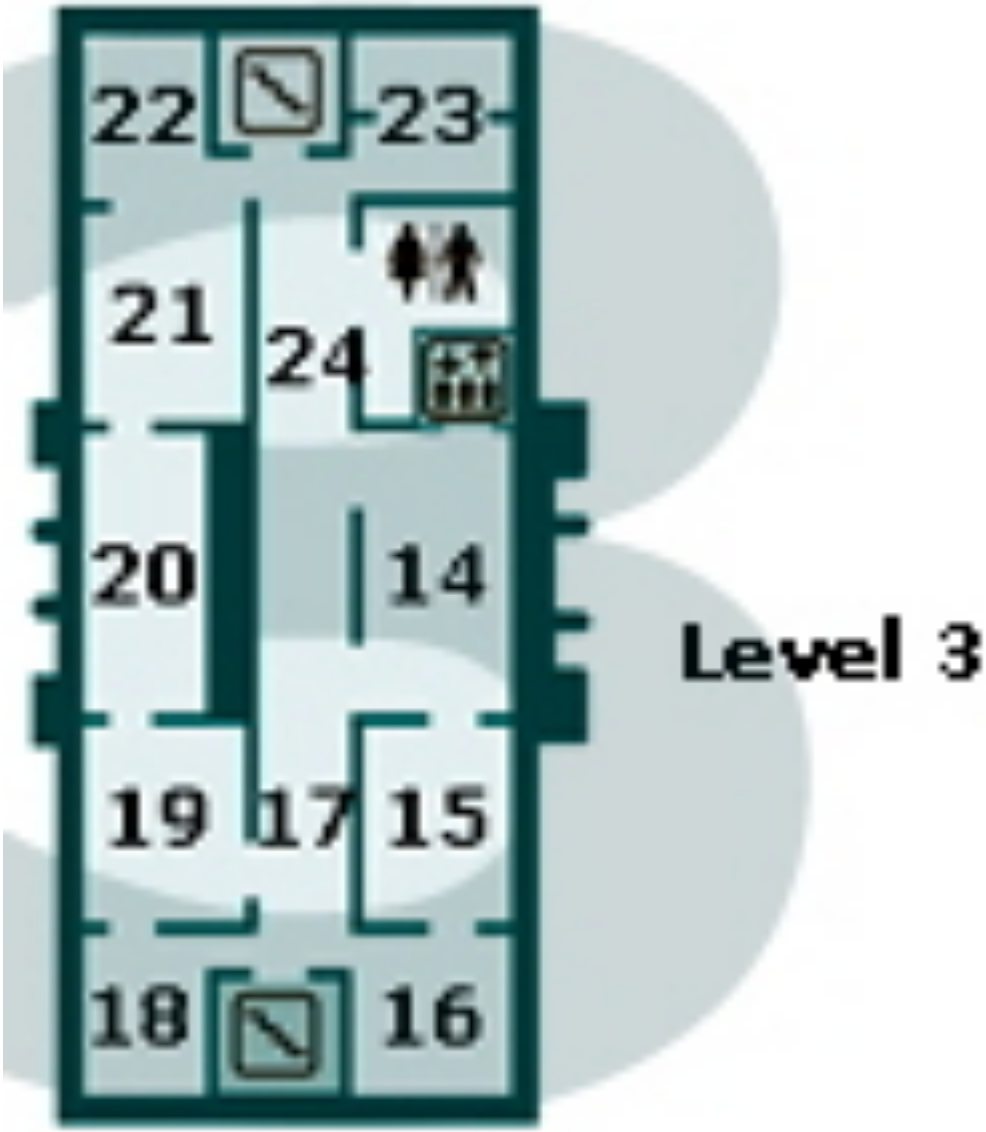
Gallery Problems



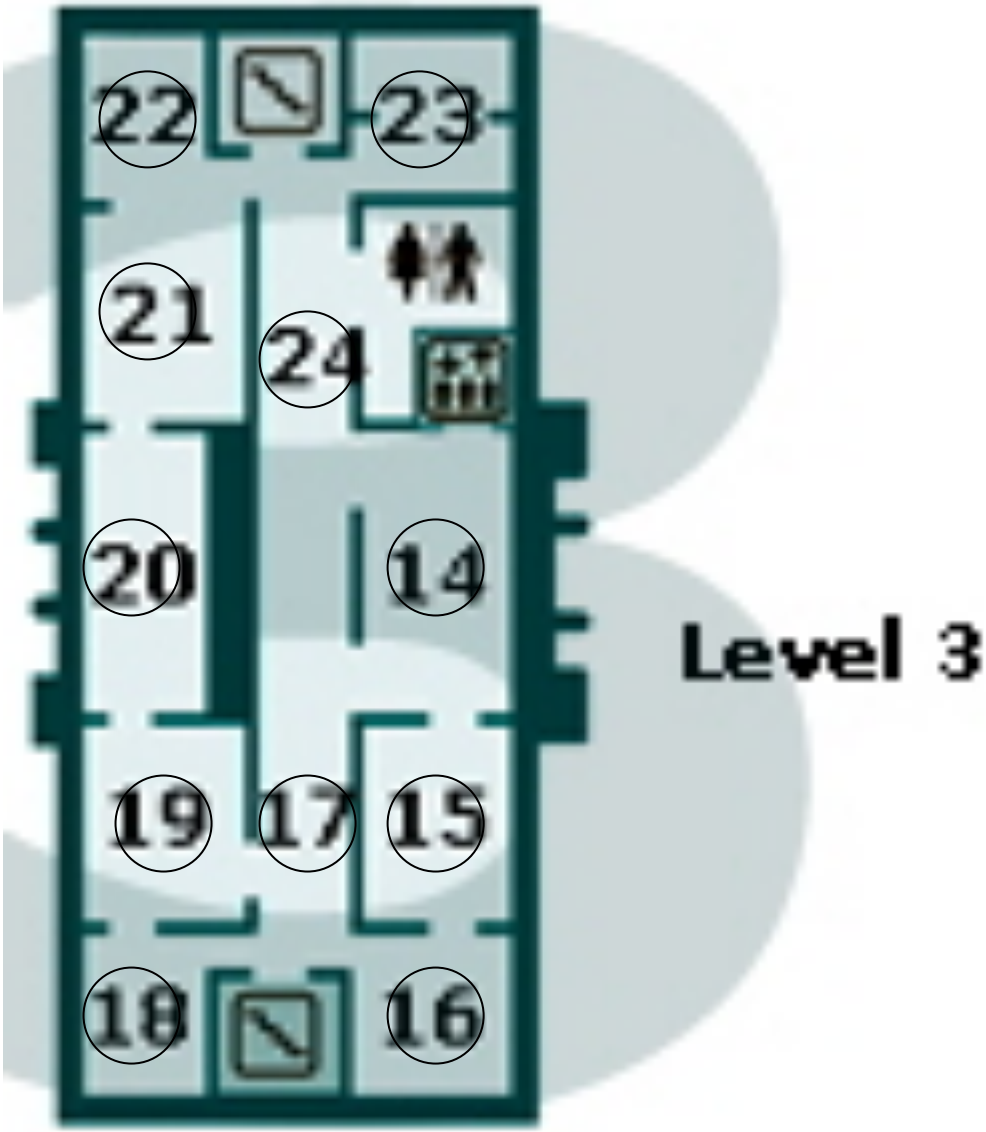
Gallery Problems



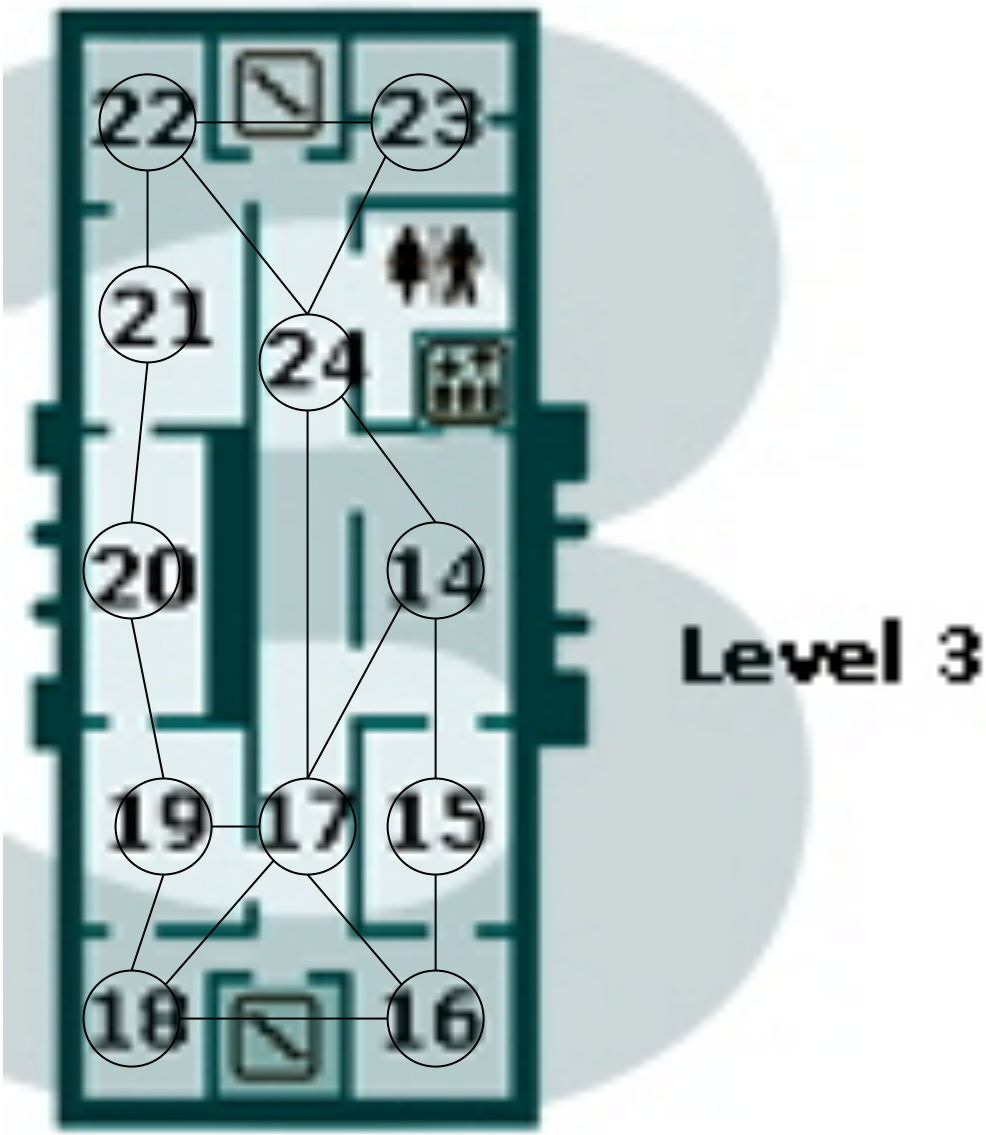
Gallery Problems



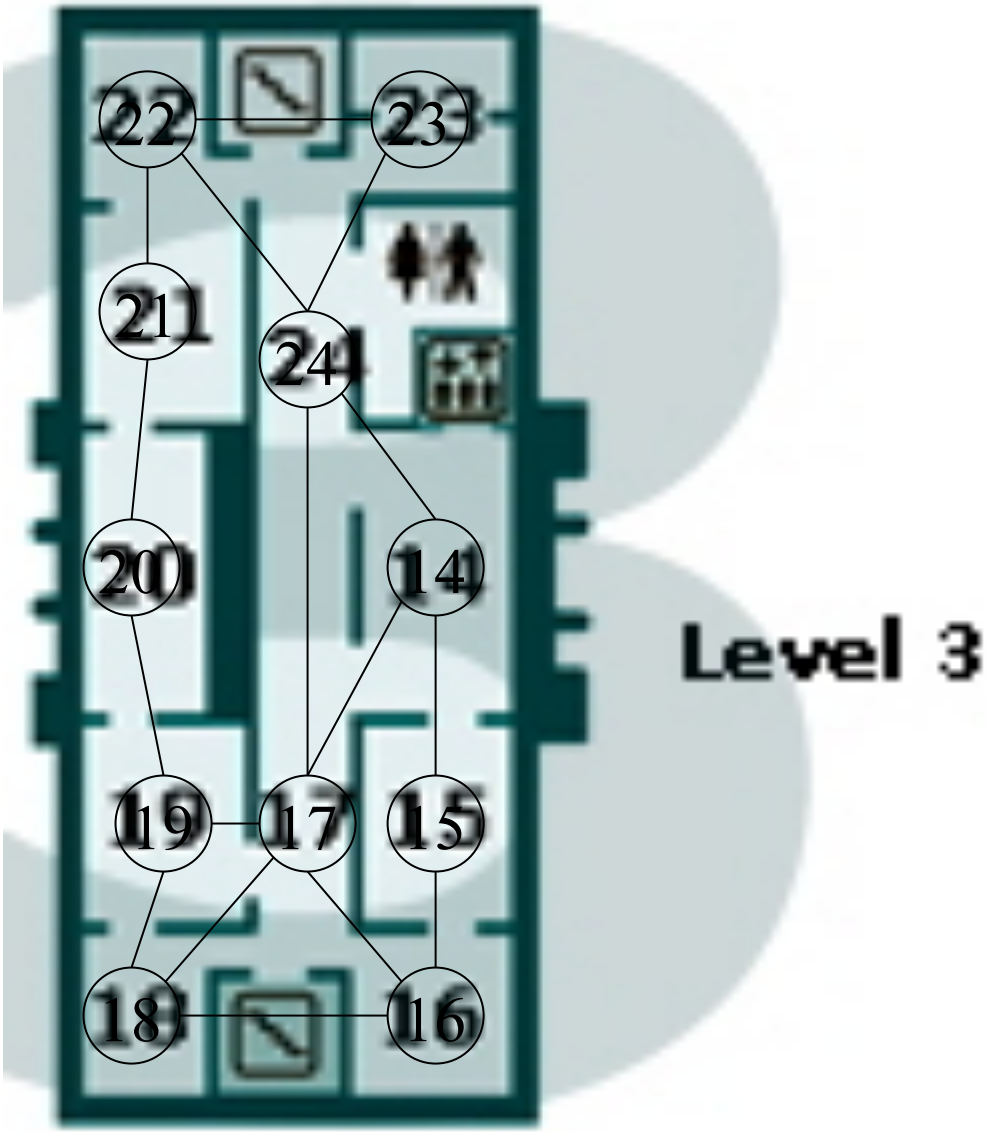
Gallery Problems



Gallery Problems



Gallery Problems



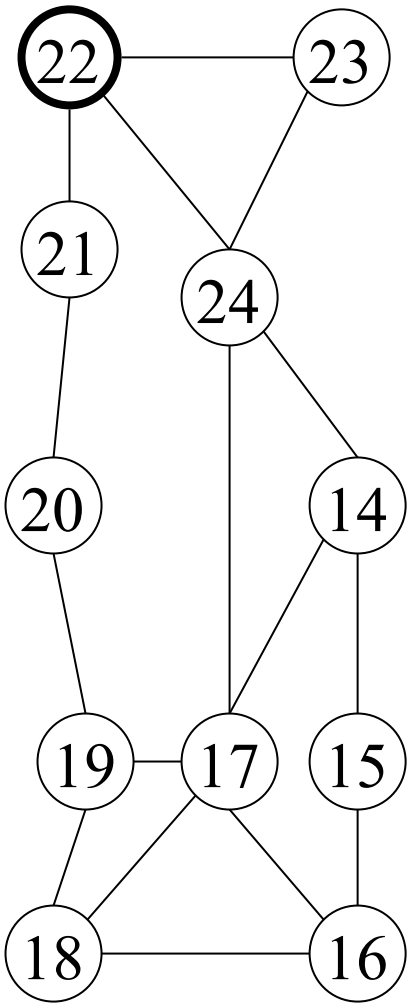
Vertex Cover Problem

Input: A graph $G = (V, E)$ with n vertices and m edges.

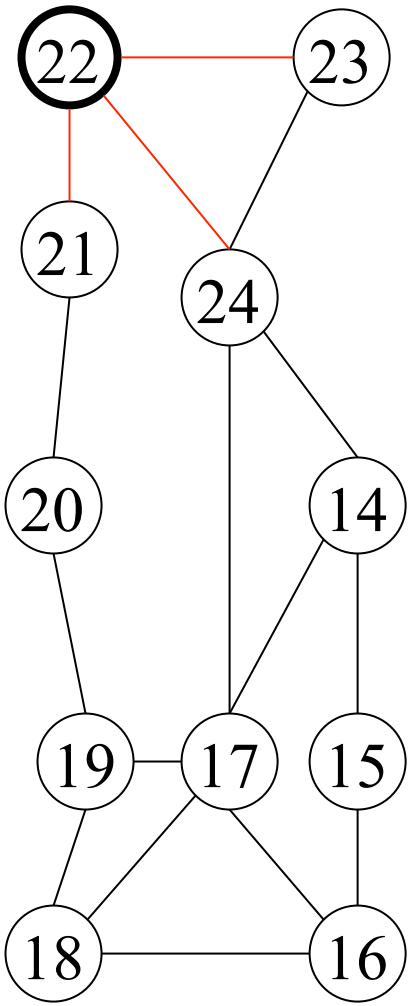
Output: A **minimum** number of vertices that can be chosen such that the vertices cover at least one of the two endpoints of each edge.

$$V' \subseteq V : \forall \{(a,b)\} \in E : a \in V' \vee b \in V'$$

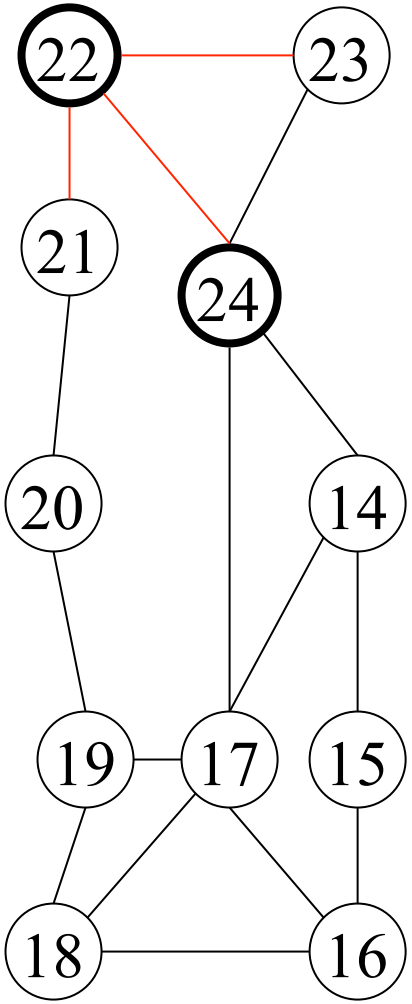
Vertex Cover



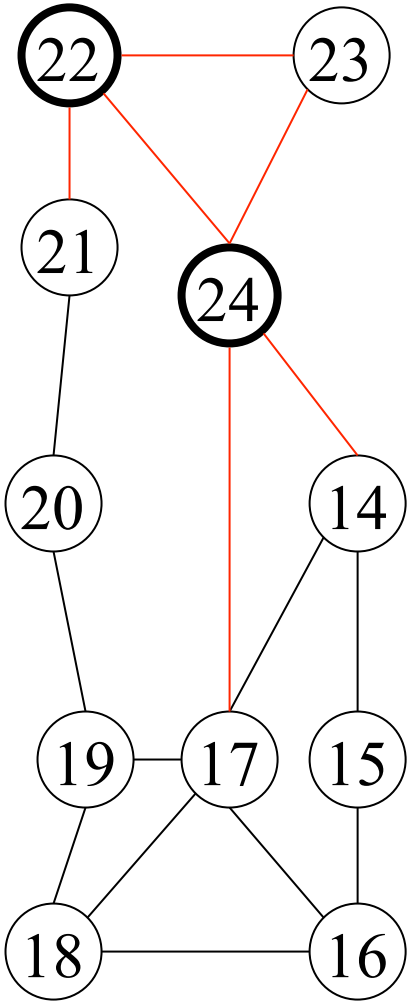
Vertex Cover



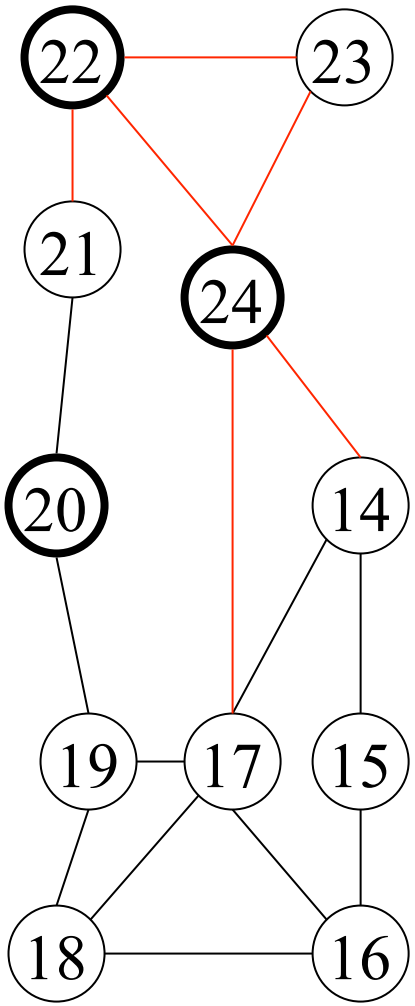
Vertex Cover



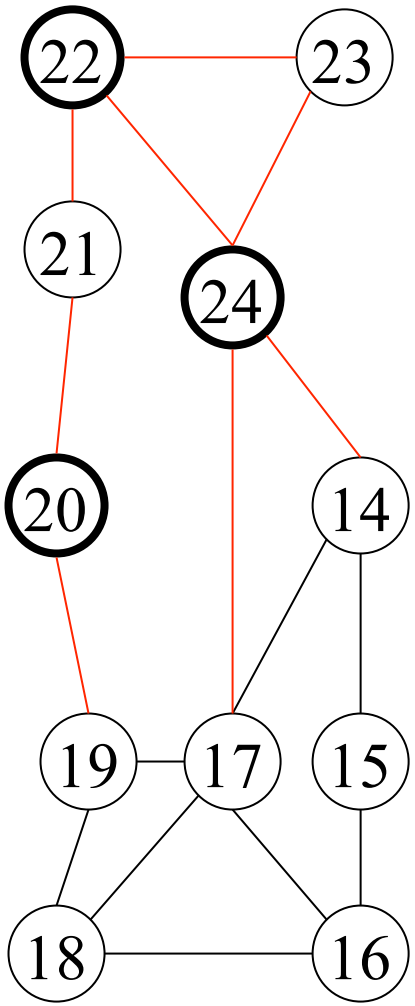
Vertex Cover



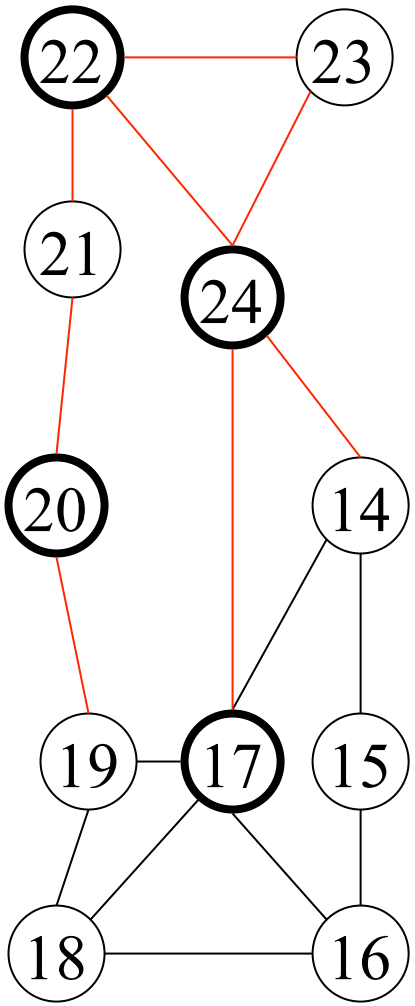
Vertex Cover



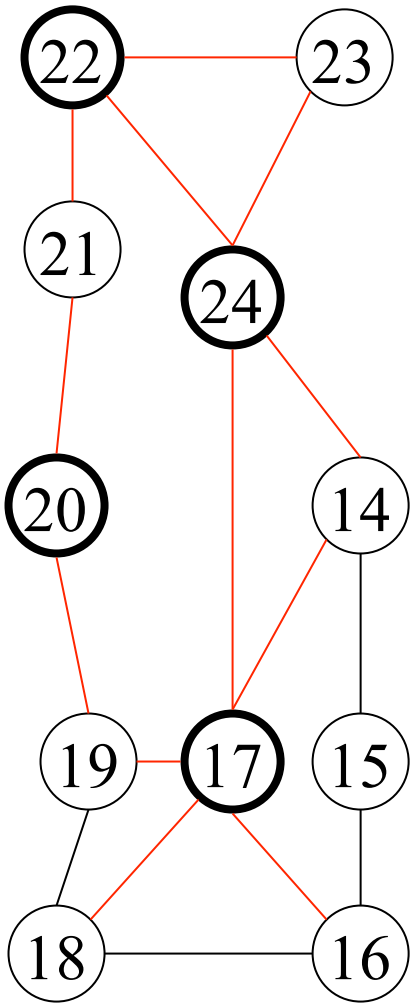
Vertex Cover



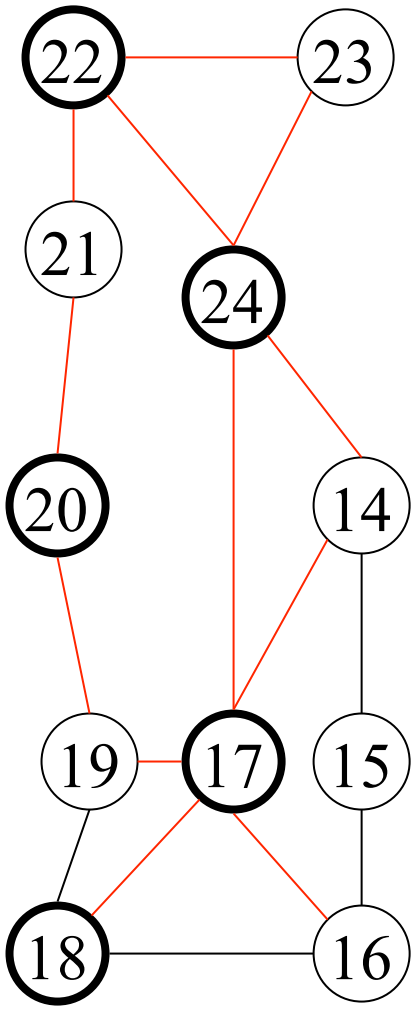
Vertex Cover



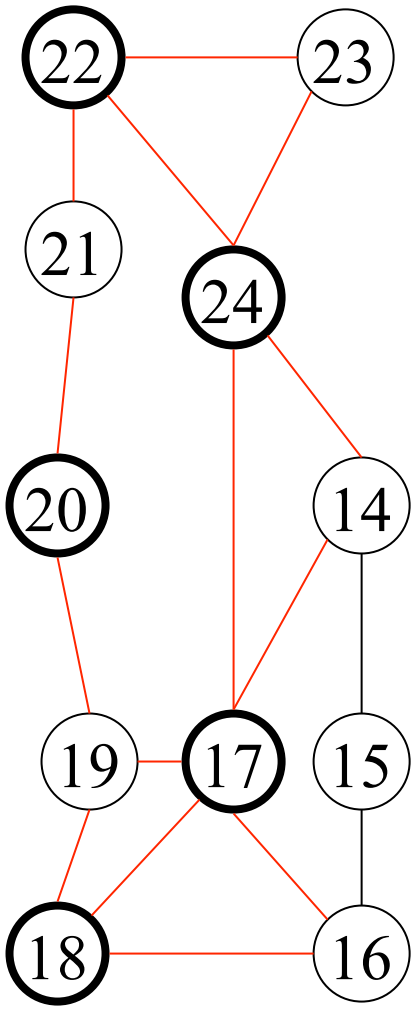
Vertex Cover



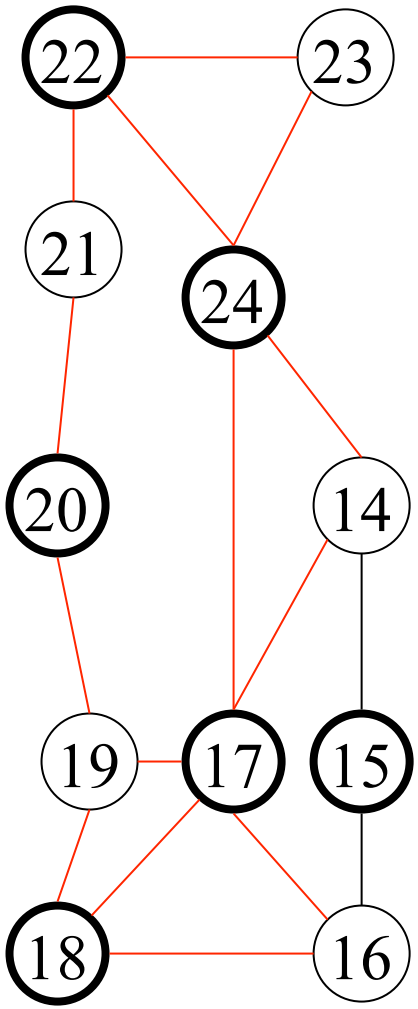
Vertex Cover



Vertex Cover



Vertex Cover



Vertex Cover

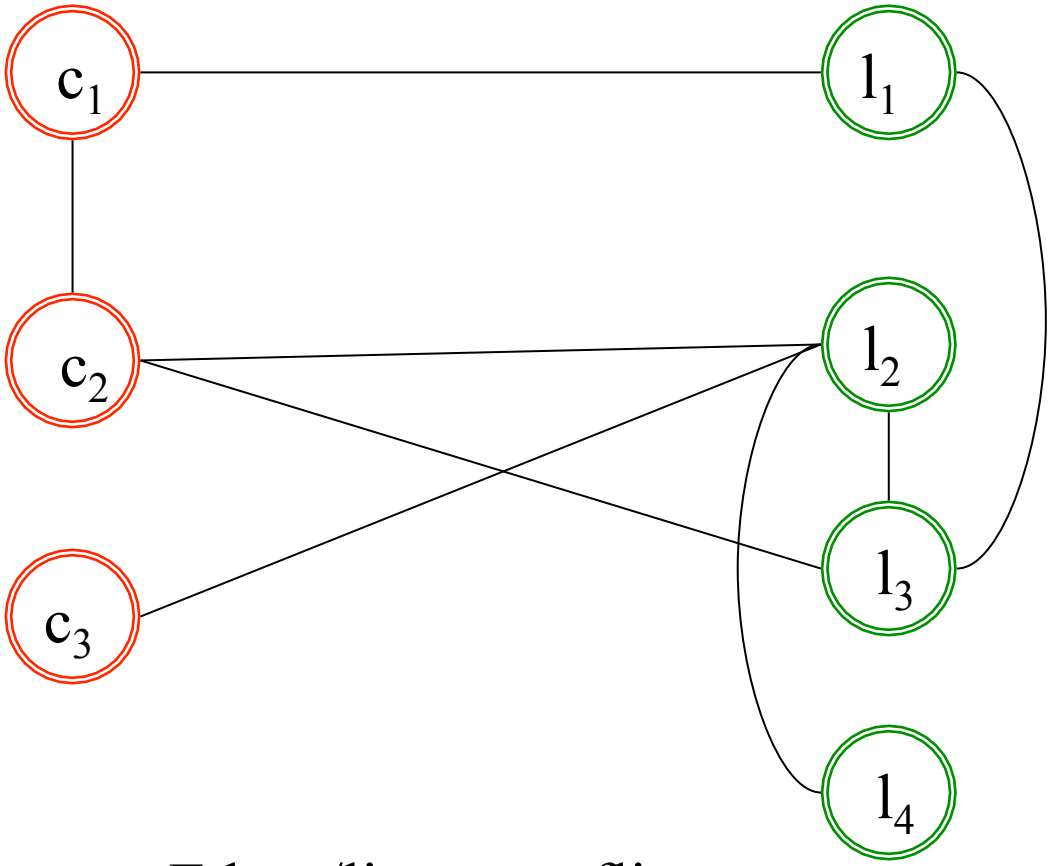
Everyday Decisions
Conflict Graphs

Task

A graduate student wants to schedule his/her term as follows:

- What courses can be taken?
- What conferences can be attended such that no exam is going to be missed?
- The student wants to participate in as many activities as possible.

conferences courses



Edges/lines: conflicts

The Scheduling Task

Input: *n* activities, and *m* conflicting activity pairs

Question: What is the maximum number of activities that can be chosen such that no conflict is contained between the chosen activities?

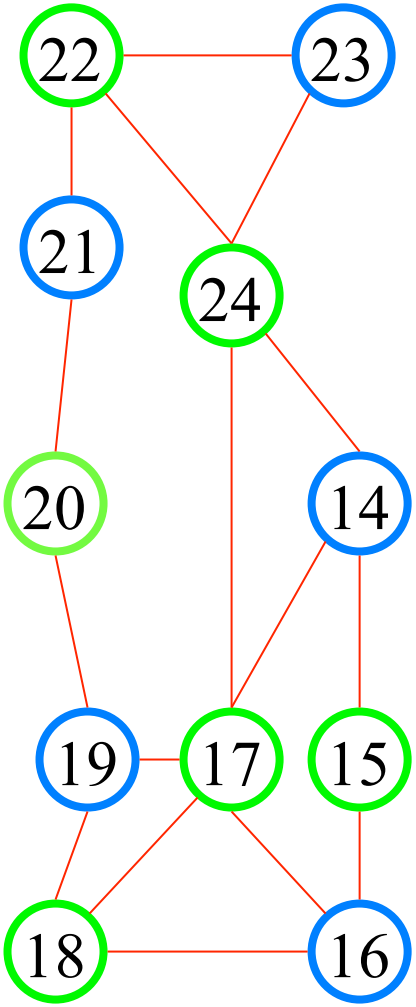
More formally, the scheduling task can be formulated as the Independent Set Problem

Independent Set Problem

Input: A graph $G = (V, E)$ with n vertices and m edges.

Output: A **maximum** number of vertices that can be chosen such that no pair of chosen vertices is connected by an edge in G.

Independent Set and Vertex Cover



The Maximum Independent Set is the complement of the Minimum Vertex Cover

Graph Representations

- Node centric
  - Edge-list structure with vertex and edge objects
  - Adjacency matrix (i.e., 0 or 1 entries)
  - Adjacency list
  - Labeled adjacency matrix (i.e., edge labels entries)
  - Labeled adjacency list
- Edge centric
  - Doubly connected edge list
  - Computational Geometry and Computer Graphics
- Face centric
  - Computational Geometry and Computer Graphics

Edge-List Structure

- A vertex v of G storing an element o is explicitly represented by a vertex object
- The vertex objects are stored in a container V
  - Vector: vertices are numbered
  - Dictionary: each vertex is identified by its key
- The elements of container V are the vertex positions of graph G

Edge-List Structure: Vertex Objects

- The vertex object for a vertex v storing element o has instance variables for
  - A reference to o
  - Counters for the number of incident undirected edges, incoming directed edges, and outgoing directed edges
  - A reference to the position of the vertex-object in container V

Edge-List Structure

- An edge *e* of *G* storing an element *o* is explicitly represented by an edge object
- The edge objects are stored in a container *E*
  - list, vector, dictionary
- The elements of container *E* are the edge positions of graph *G*

Edge-List Structure: Edge Objects

- The edge object for an edge *e* storing element *o* has instance variables for
  - A reference to *o*
  - A Boolean indicator of whether *e* is directed or undirected
  - References to the vertex objects in *V* associated with the endpoint vertices of e or the origin and destination vertices of *e*
  - A reference to the position of the edge-object in container *E*

Adjacency-List Structure

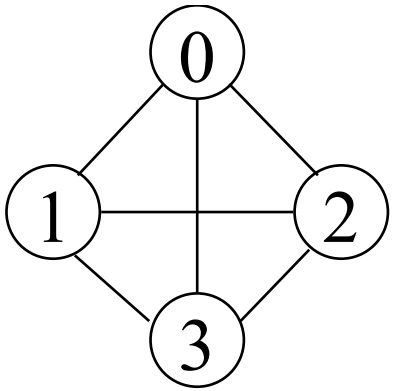
- Extends the edge-list structure, adding extra information that supports direct access to the incident edges (and to the adjacent vertices) of each vertex
- Edge-vertex incidence relation
  - Accessible from edge list and from vertex list

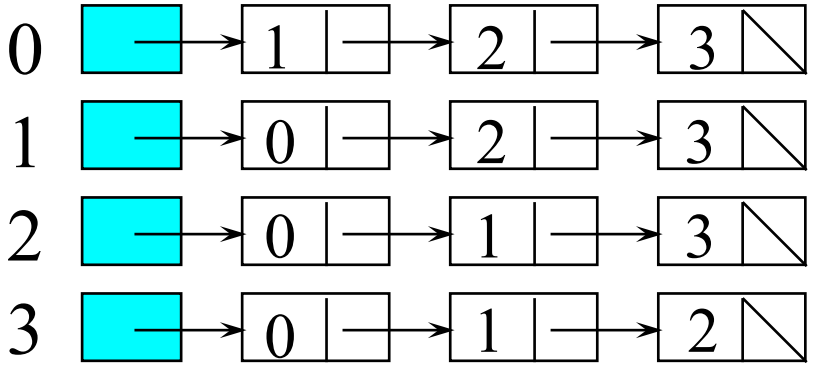
Adjacency-List Structure: Vertex Objects

- The vertex object for a vertex v storing element o extends the vertex object of the edge-list structure and additionally has an instance variable for
- A reference to the *incidence container* $I(v)$, which stores the references to the edges incident on v
- For directed edges use $I_{in}(v)$ and $I_{out}(v)$

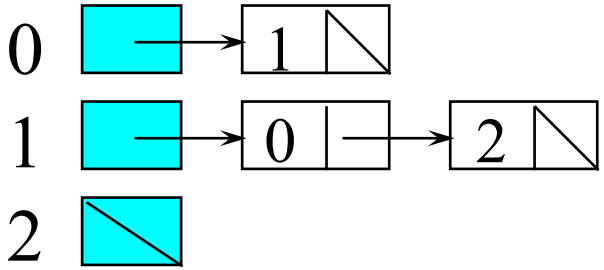
Adjacency-List Structure: Edge Objects

- The edge object for an edge e storing element o extends the vertex object of the edge-list structure and additionally has an instance variables for
- References to the positions of the edge (u,v) in the incidence containers I(u) and I (v)

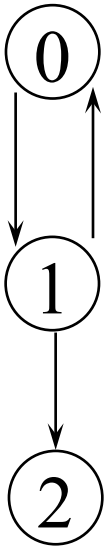

0
1
2
3

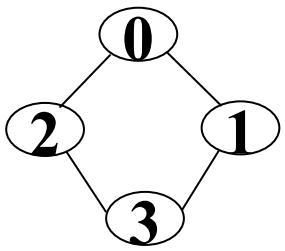

0
1
2
3
1
2
3
0
2
3
0
1
3
0
1
2

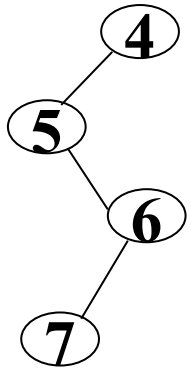
G_1

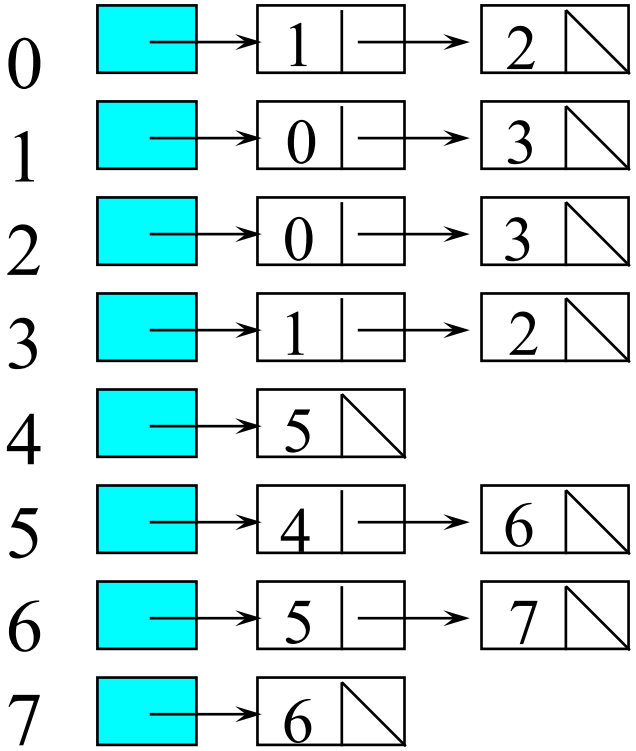

0
1
2
1
0
2

G_3


0
1
2


0
2
1
3


4
5
6
7

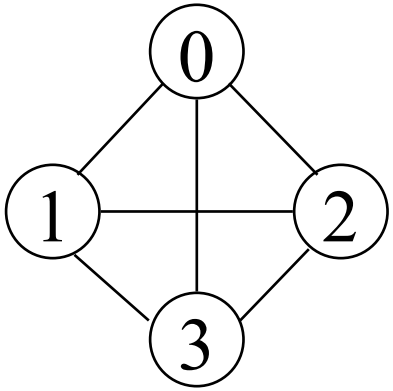

0
1
2
3
4
5
6
7
1
2
0
3
0
3
1
2
5
4
6
5
7
6

G_4

Adjacency-Matrix Structure

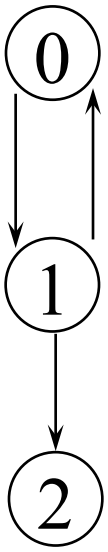
- Vertices are numbered 0,1, .., $n-1$
- (i,j) denotes the edge between vertex with number i and vertex with number j
- The Graph is represented by an $(n \times n)$-array A such that $A[i,j]$ stores a reference to edge (i,j), if (i,j) exists, and null otherwise
- If (i,j) is undirected, store reference in A$[i,j]$ and A$[j,i]$
- The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric
- A 1 along the diagonal indicates a self-loop

Adjacency Matrices



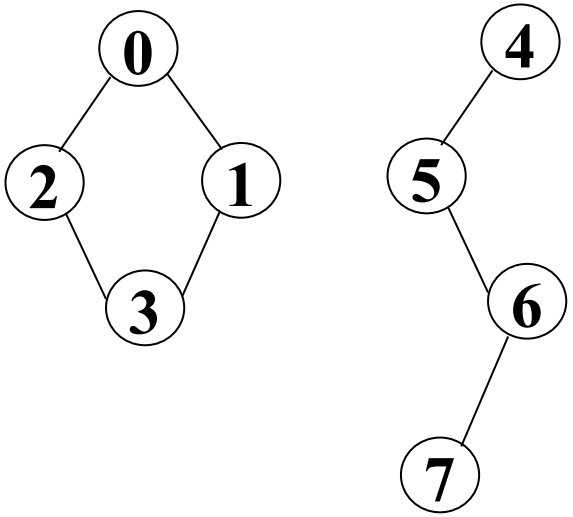
$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1



from $\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$ to

G_2



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_3

Useful Computations on Adjacency Matrices

- The outdegree of vertex k is the sum of the adjacency matrix elements in row k
- The indegree of vertex k is the sum of the adjacency matrix elements in column k
- The degree of a vertex k is the sum of the adjacency matrix elements in row or column k

The Graph ADT

- A graph is a positional container of elements that are stored at the graph's vertices and edges
- The *positions* in the graph are its vertices and edges
- We can store elements in a graph at either its edges or its vertices or both

Positional Graph Methods

- size()
- isEmpty()
- elements()
- positions()
- replaceElement(*p*,*o*) [*p*: position, *o*: object]
- swapElements(*p*,*q*) [*p*,*q*: positions]

General Graph Methods

- numVertices(): Return the number of vertices in G
- numEdges(): Return the number of edges in G
- vertices(): Return an iterator of the vertices of G
- edges(): Return an iterator of the edges of G
- aVertex(): Return an arbitrary) vertex of G

Graph Methods with Vertex and Edge Positions as Arguments

- degree(v): Return the degree of v
- adjacentVertices(v): Return an iterator of the vertices adjacent to v
- incidentEdges(v): Return an iterator of the edges incident upon v
- endVertices(e): Return an array of size 2 storing the end vertices of e
- opposite(v,e): Return the endpoint of edge e distinct from v
- areAdjacent (v,w): Return whether vertices v and w are adjacent

Graph Methods for Directed Edges

- directedEdges(): Return an iterator of all directed edges
- undirectedEdges(): Return an iterator of all undirected edges
- destination(*e*): Return the distination of the directed edge *e*
- origin(*e*): Return the origin of the directed edge *e*
- isDirected(*e*): Return true if and only id the edge *e* is directed

Graph Methods for Directed Edges

- inDegree(v) : Return the in-degree of v
- outDegree(v): Return the out-degree of v
- inIncidentEdges(v): Return an iterator of all the incoming edges to v
- outIncidentEdges(v): Return an iterator of all the outgoing edges from v
- inAdjacentVertices(v): Return an iterator of all the vertices adjacent to v along incoming edges to v
- outAdjacentVertices(v): Return an iterator of all the vertices adjacent to v along outgoing edges from v

Graph Methods for Updating

- insertEdge(*v,w,o*): Insert and return an undirected edge between vertices *v* and *w,* storing the object *o* at this position.
- insertDirectedEdge(*v,w,o*): Insert and return a directed edge fro vertex *v* to vertex *w,* storing the object *o* at this position.
- insertVertex(*o*): Insert and return a new (isolated) vertex storing the object *o* at this position.

Graph Methods for Updating

- removeVertex(*v*): Remove vertex *v* and all its incident edges
- removeEdge(*e*): Remove edge *e*
- makeUndirected(*e*): Make edge *e* undirected
- reverseDirection(*e*): Reverse direction of directed edge *e*
- setDirectionFrom(*e*,*v*): Make edge *e* directed away from vertex *v*
- setDirectionTo(*e*,*v*): Make edge *e* directed into vertex *v*

Graph Traversals

- Explore the graph $G = (V, E)$ by examining all of its vertices and edges systematically
- A traversal that can be done in $O(|V| + |E|)$ time is efficient
- Systematic procedure for exploring a graph

For undirected and directed graphs

- DFS (Depth-First Search)
  - Iterative using a stack
  - Recursive using the run-time stack
  - Algorithm design technique: backtracking
- BFS (Breadth-First Search)
  - Iterative using a queue

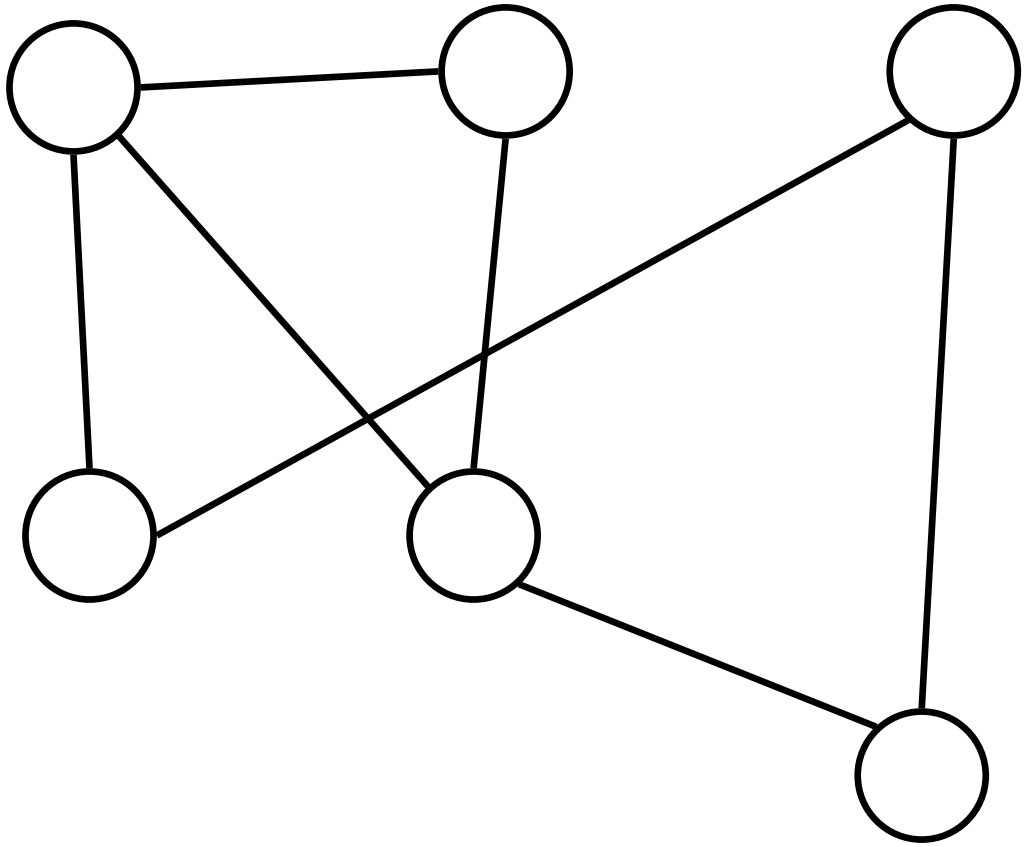
Depth First Search (DFS)

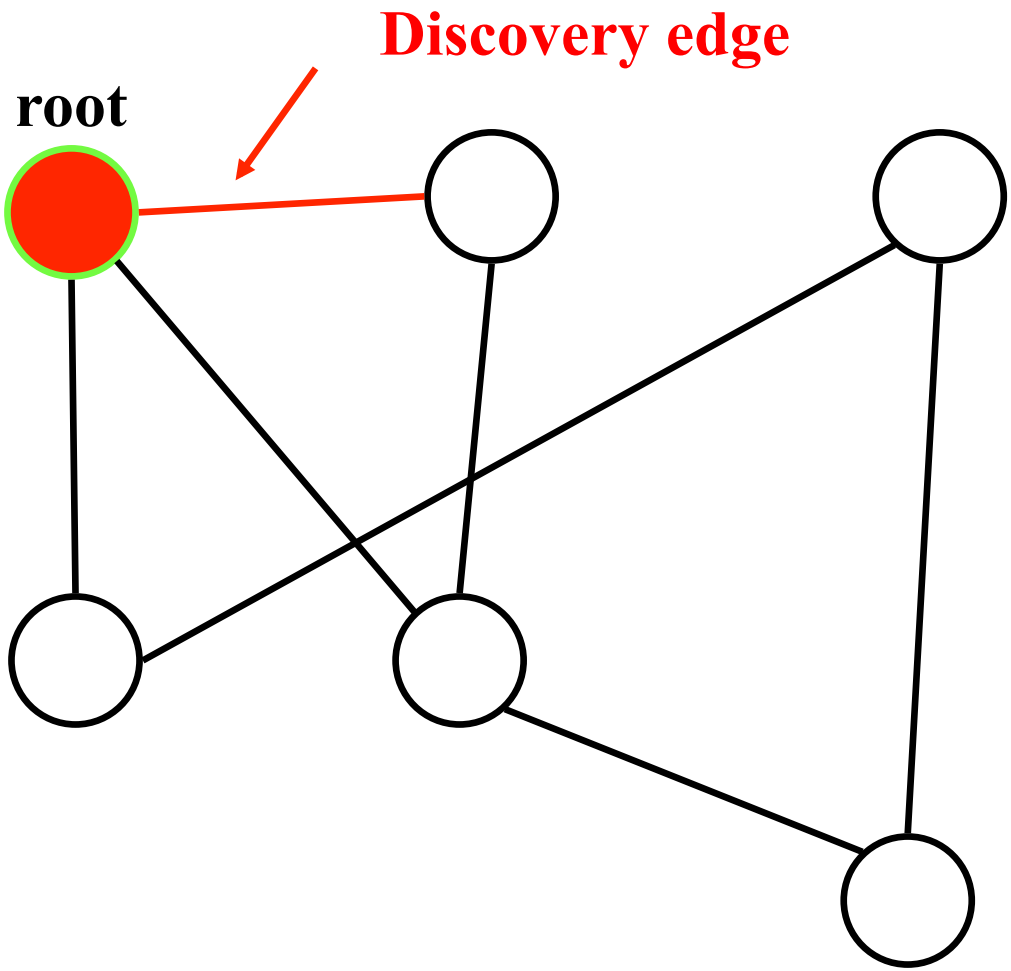
- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph
- Depth-first search is to graphs what Euler tour is to binary trees

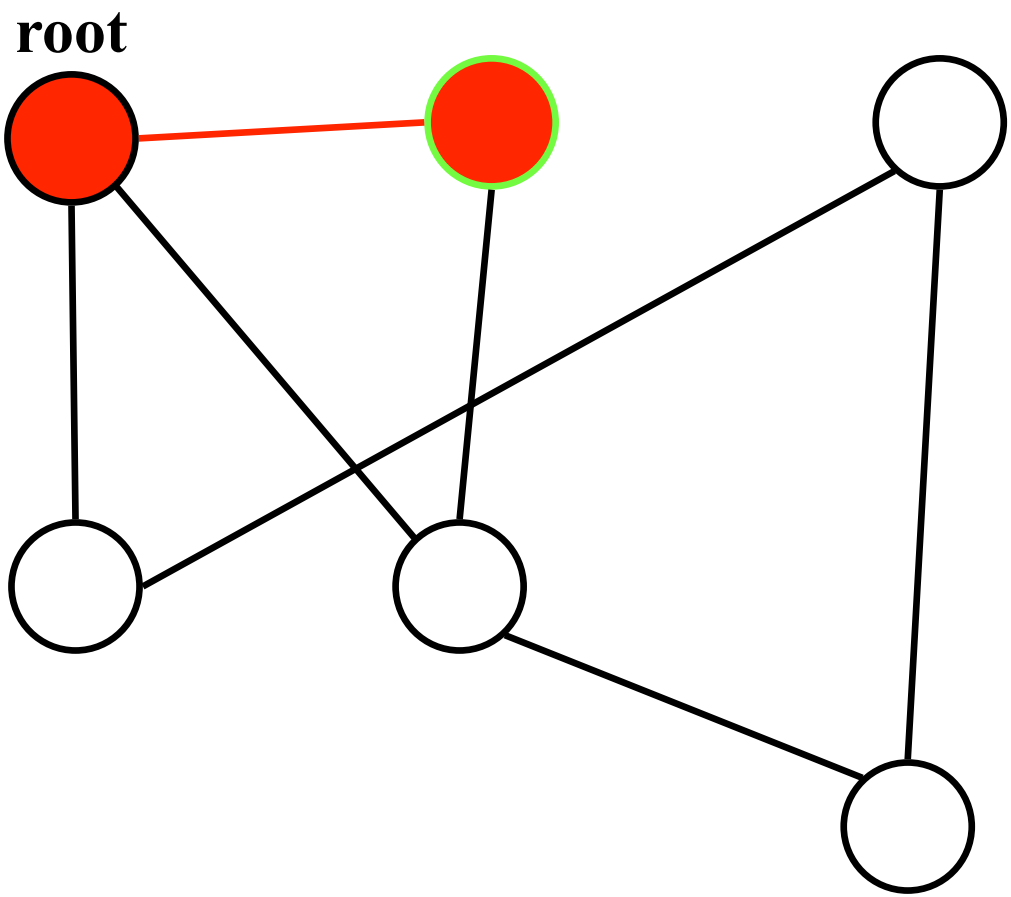
Depth First Search (DFS)

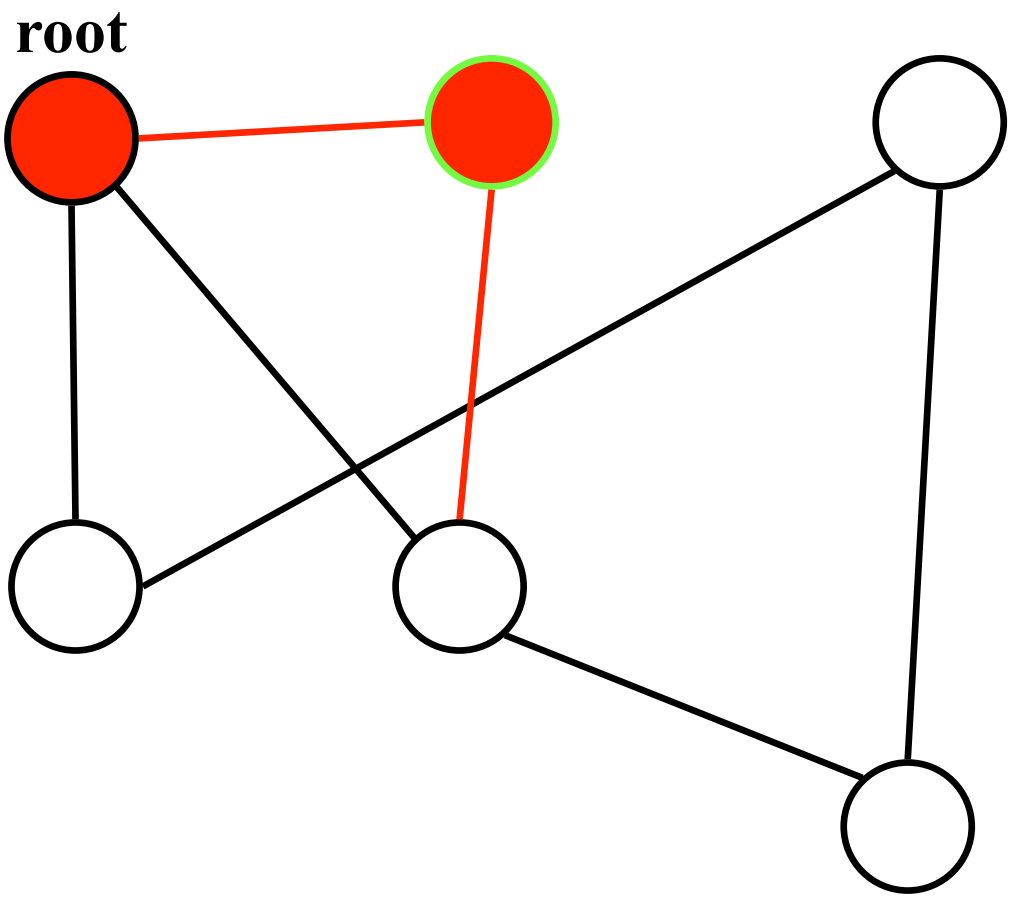
- Start at a fixed vertex *v* and mark *v* as visited
- From v, select an arbitrary incident edge (*v, u*) not already explored
- If *u* has not been visited yet, we proceed from that node recursively in a depth first manner
- Otherwise, backtrack to vertex *v* and explore *next* incident edge

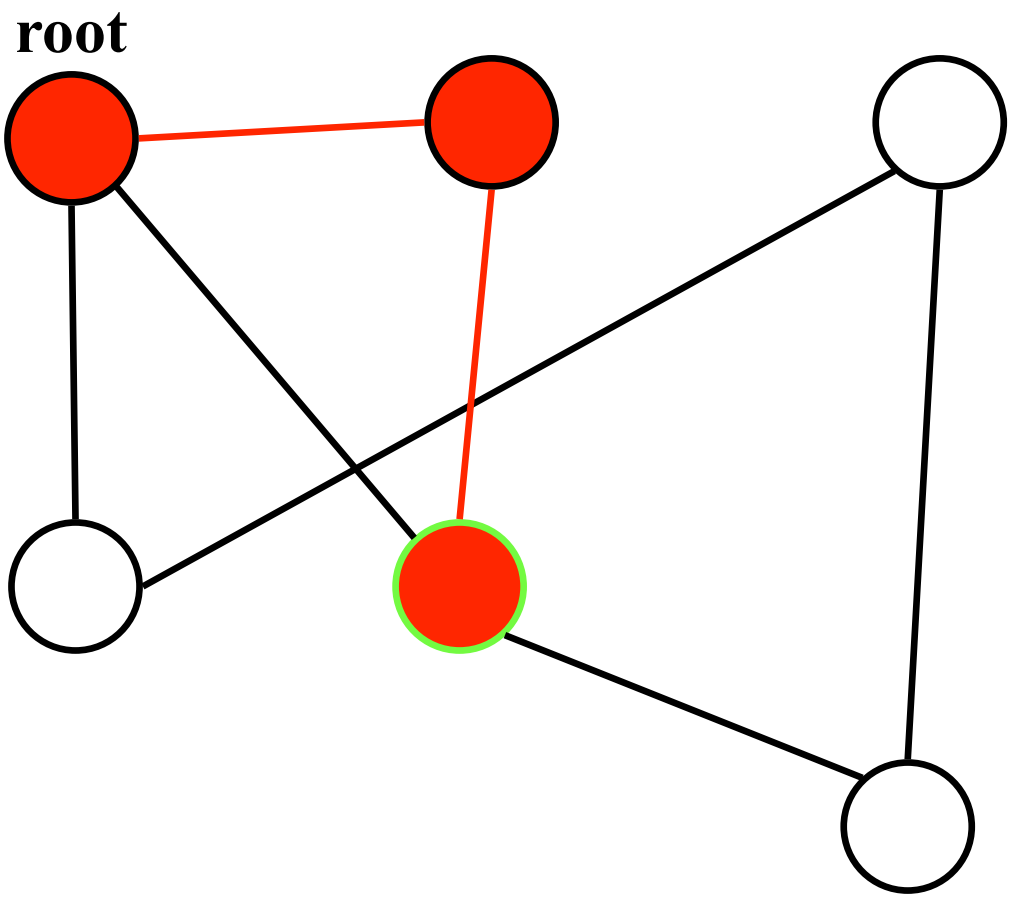
DFS Example

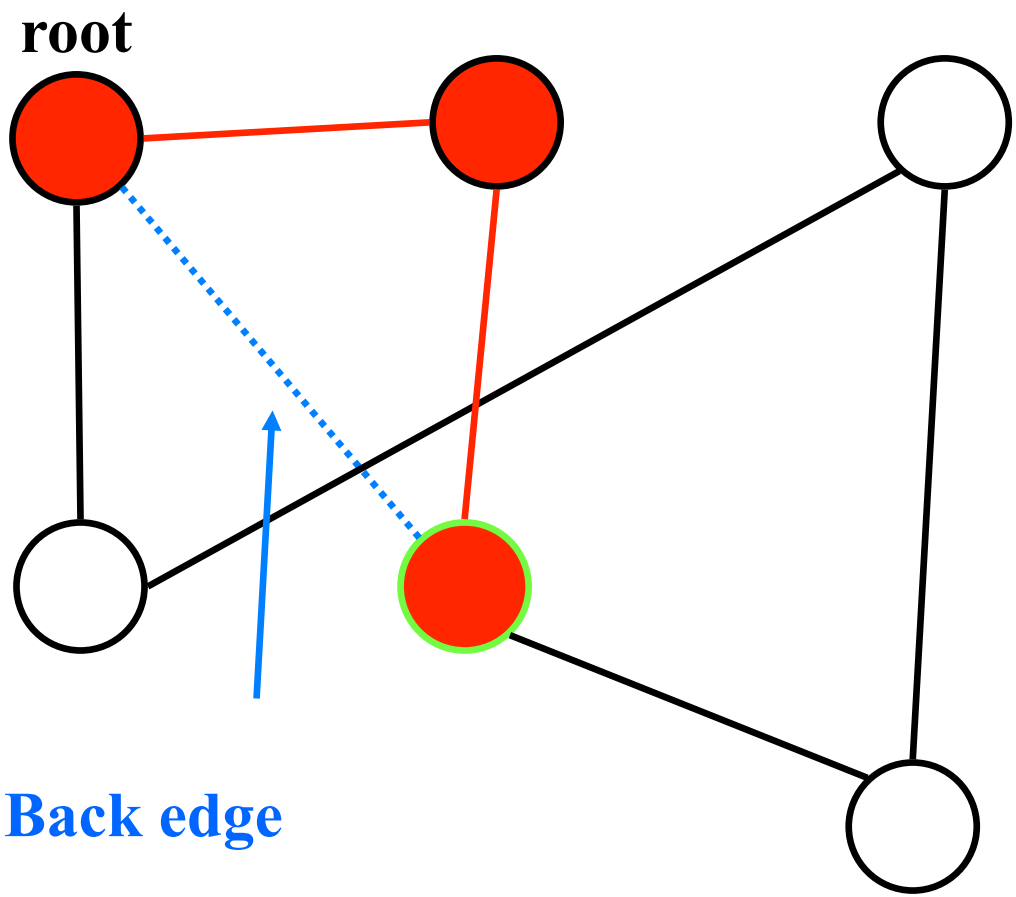


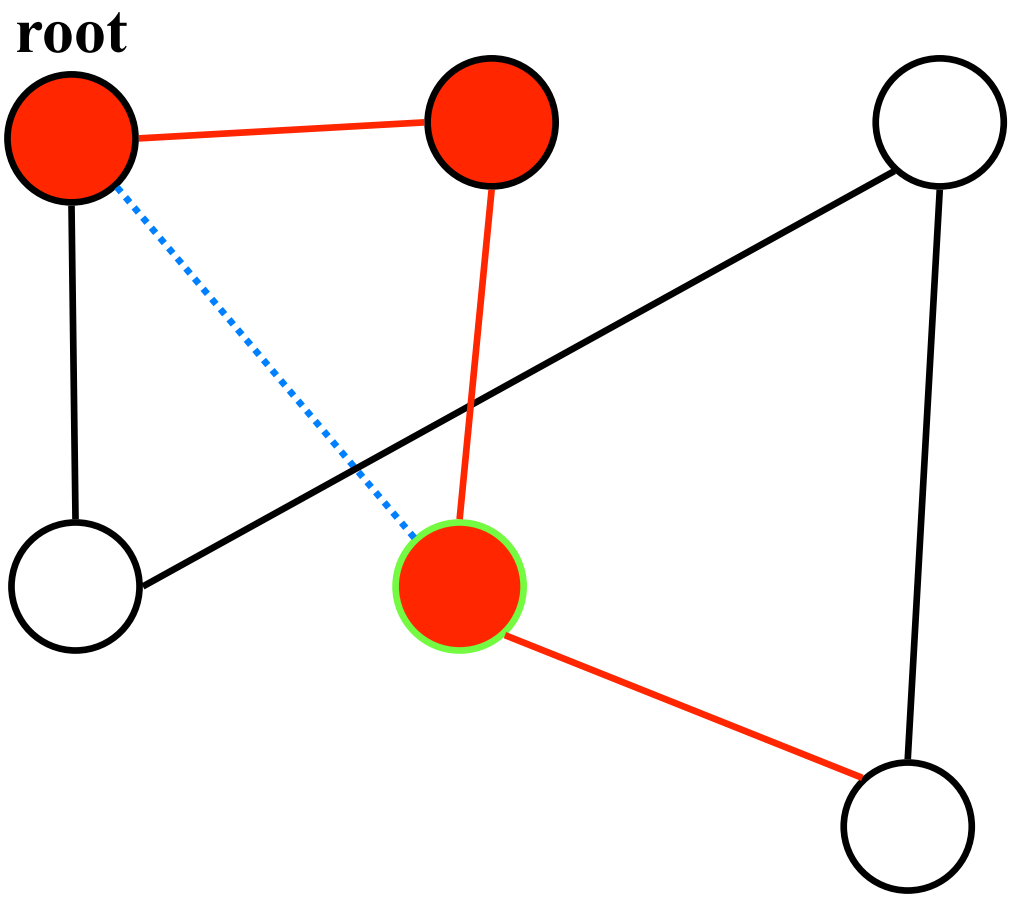

Discovery edge
root

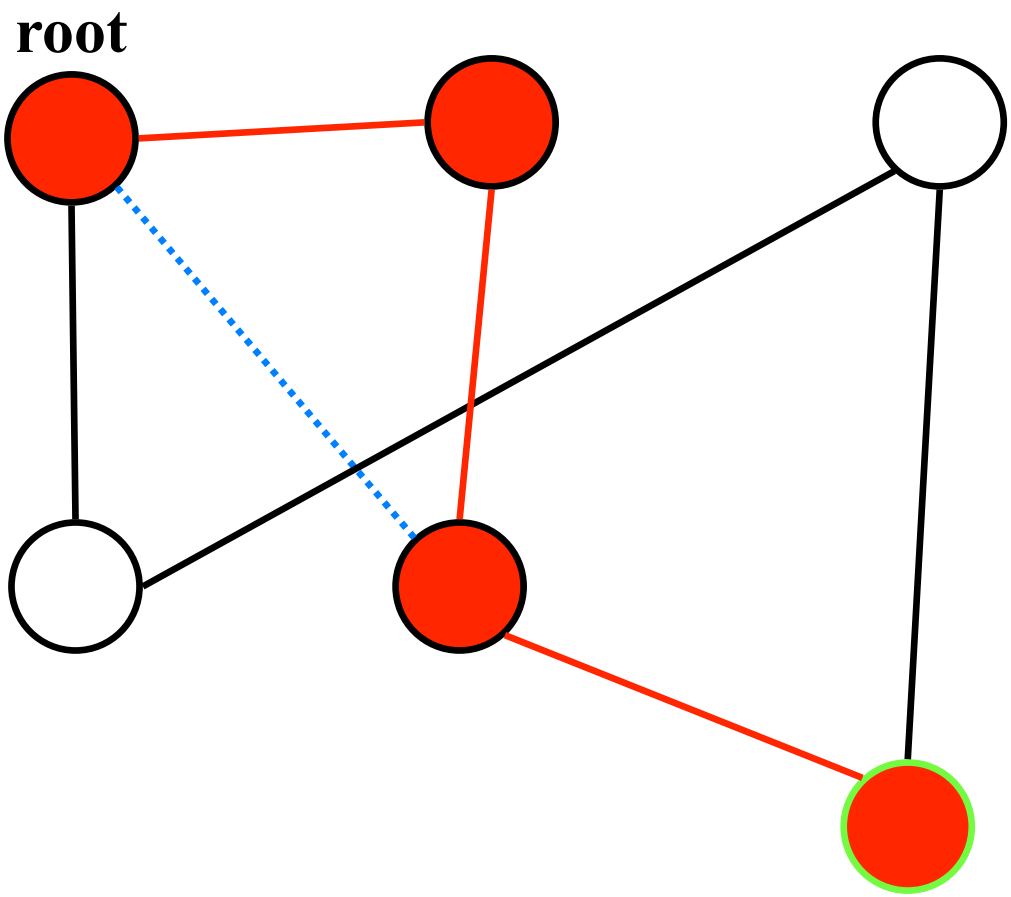

root

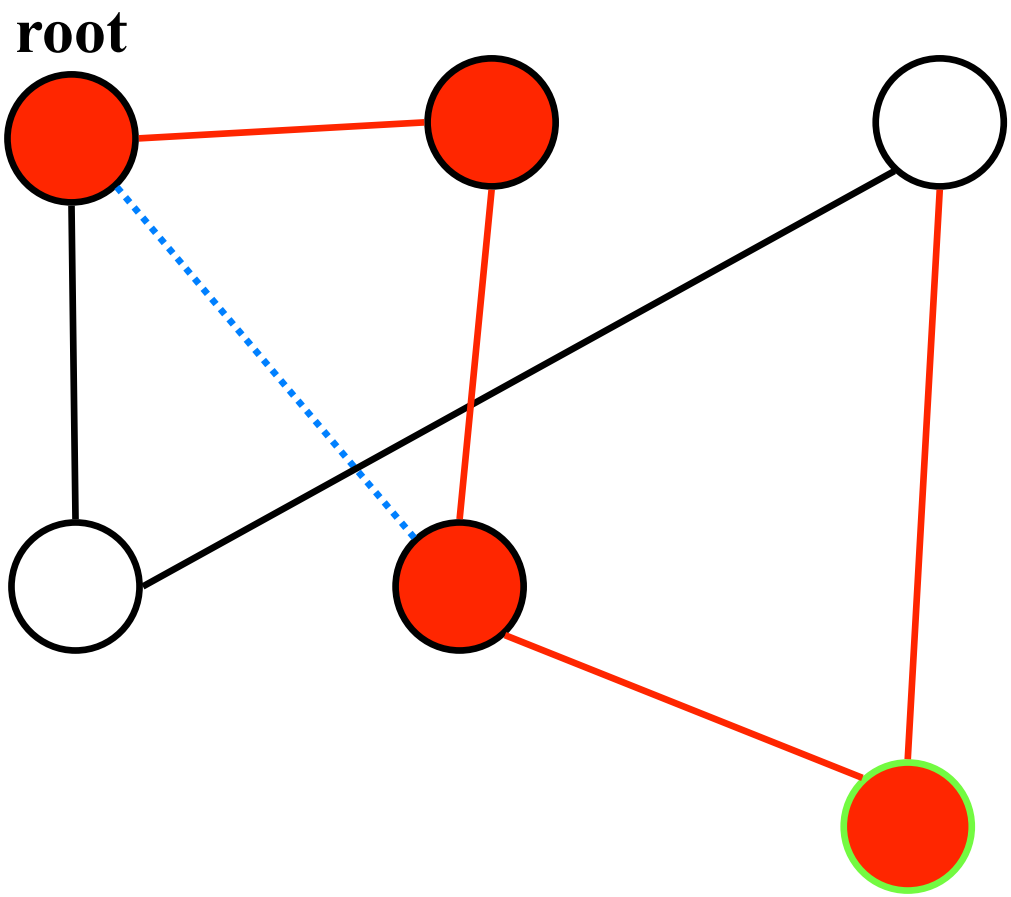

root

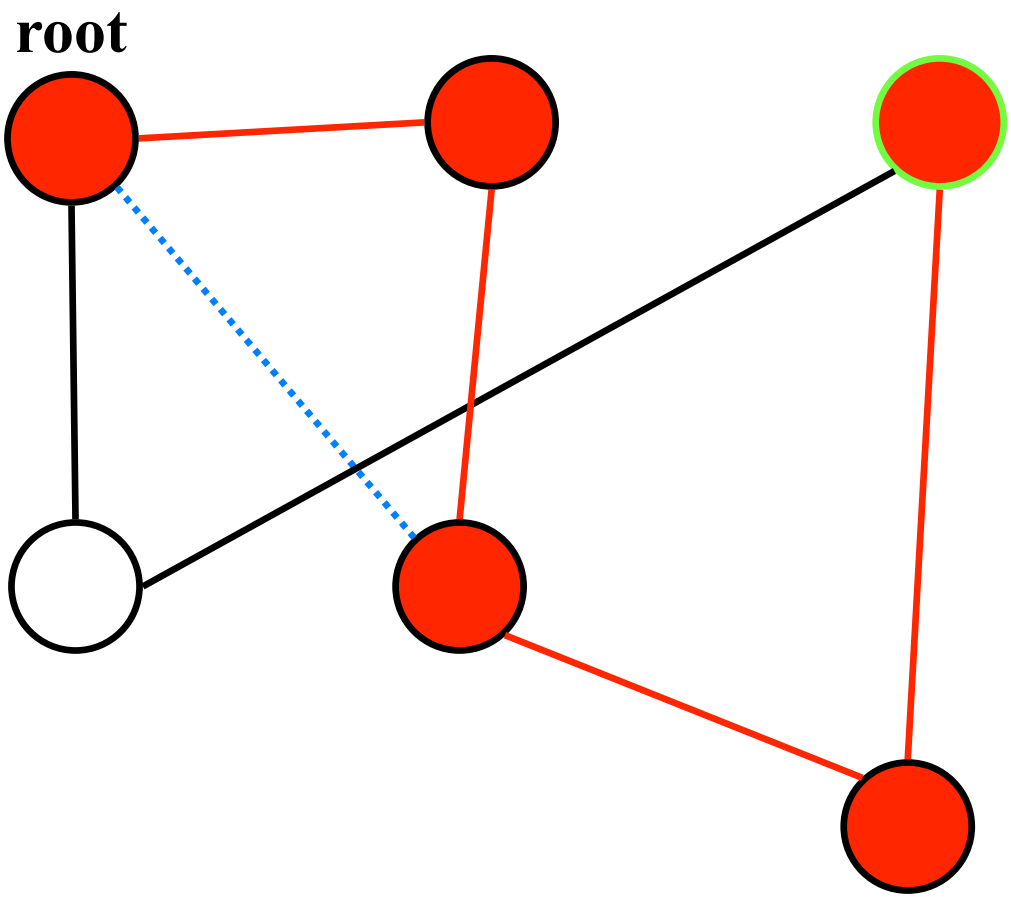

root

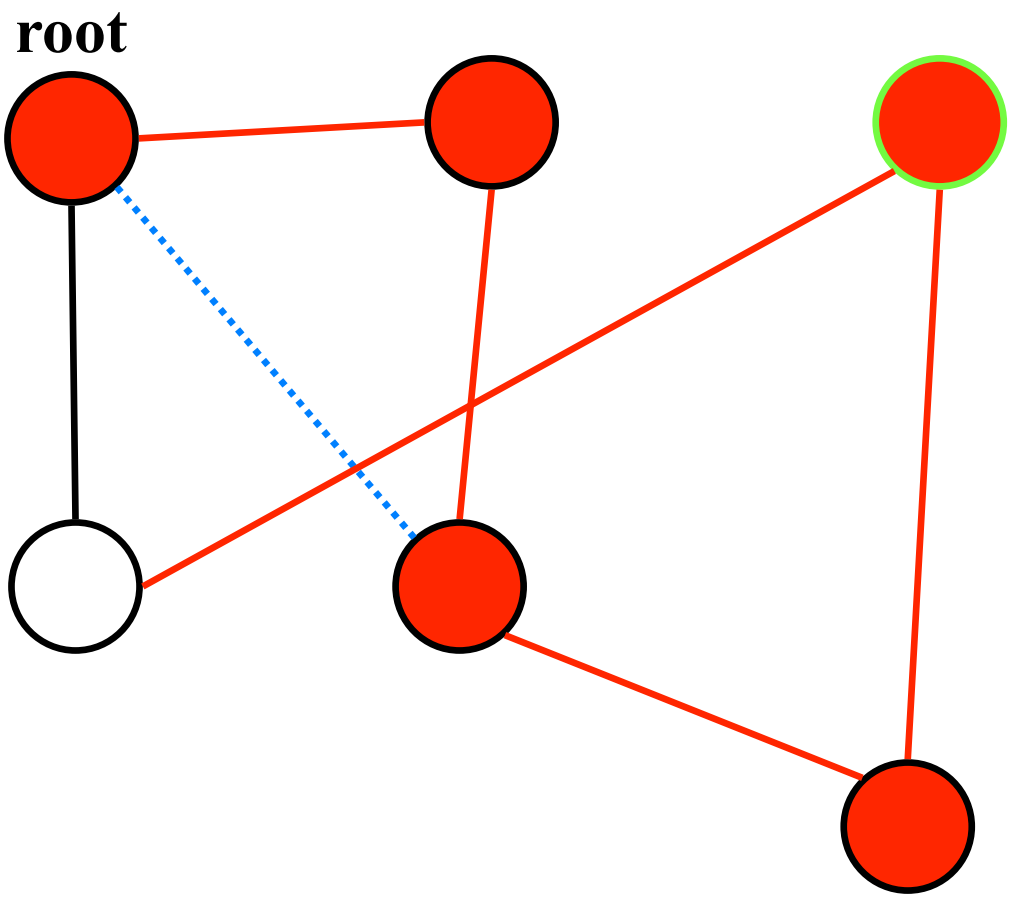

root
Back edge

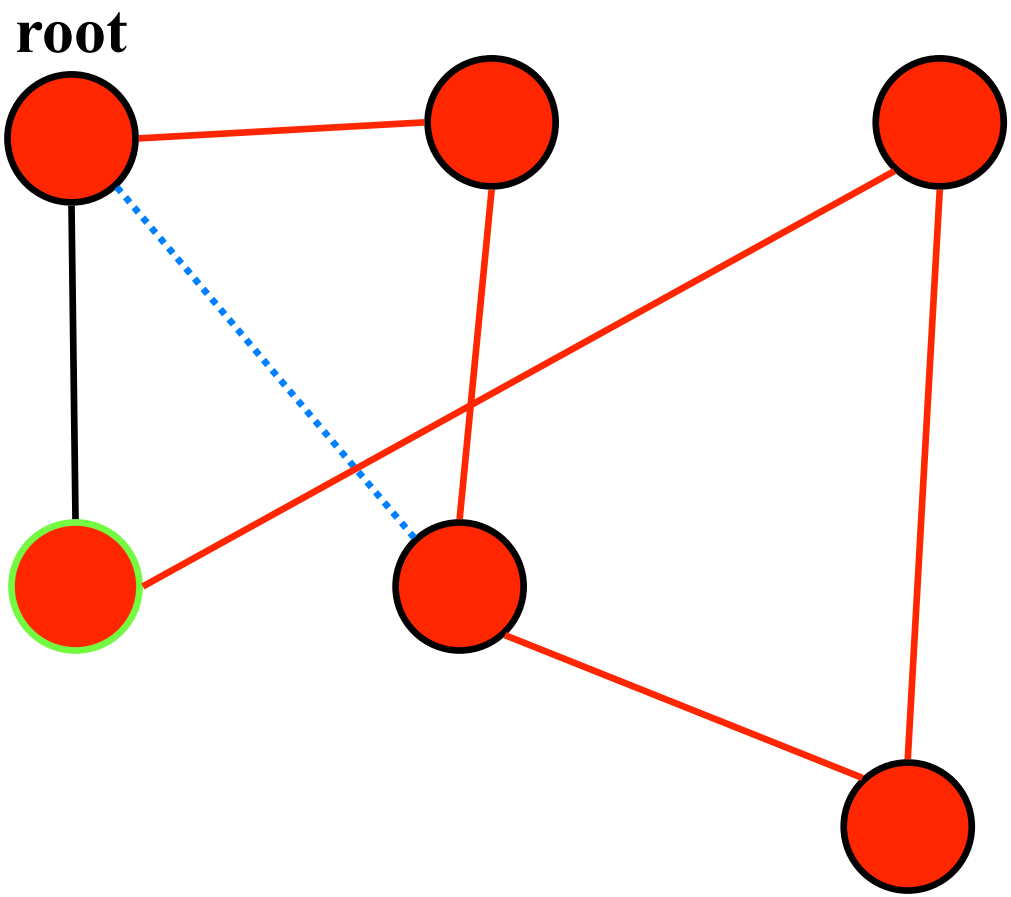

root

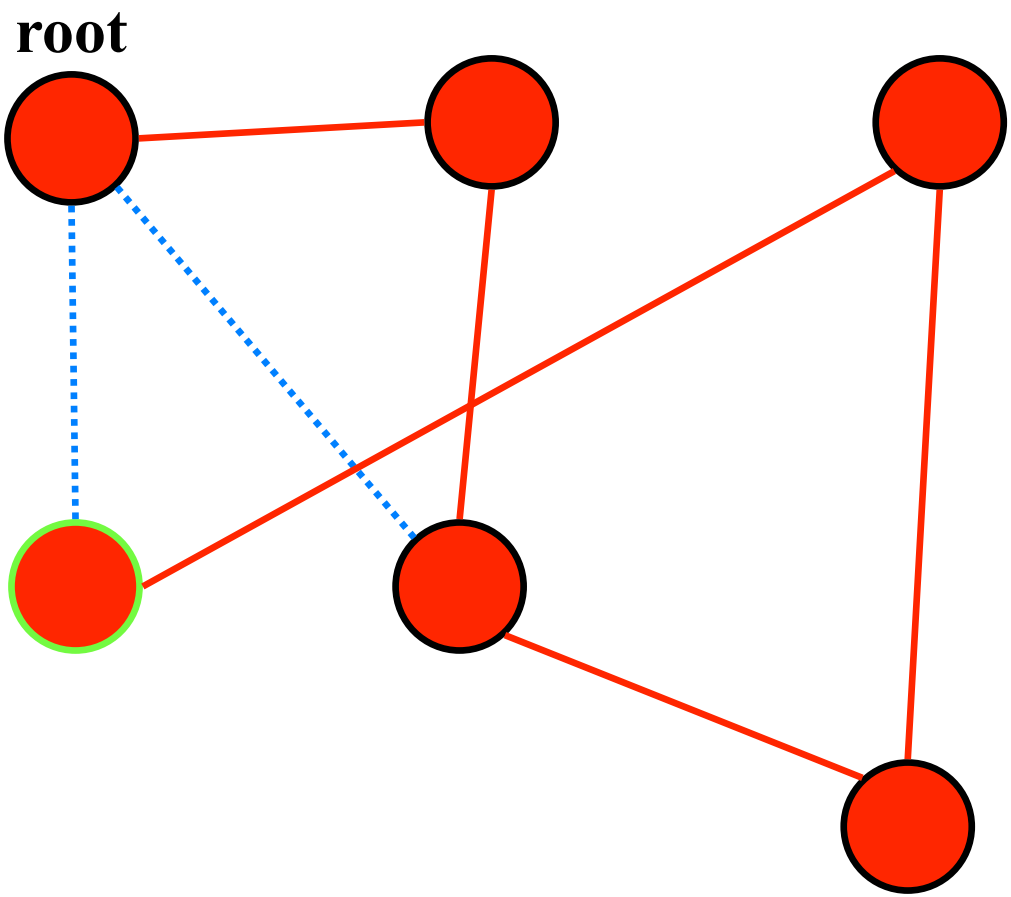

root


root


root


root


root


root

Algorithm DFS

Input: A graph G and a vertex v of G

Output: A labeling of the edges in the connected component as discovery edges and back edges

Algorithm DFS

```
Algorithm dfs(Node v)
   mark v as visited
   for each node w adjacent to v do
      if w has already been visited then
         mark edge (v, w) as back edge
      else
         mark edge (v, w) as discovery edge
         dfs(w)
      end
   end
end
```

DFS Time Complexity Analysis

- Every vertex is examined exactly once
- Every edge is examined at least once and at most twice
- **Theorem.**
  - The time complexity of DFS traversal for a graph G = (V, E) is
    $O(n + m)$ for $|V| = n$ and $|E| = m$

Applications of DFS

- Explore graph (e.g., to visit all nodes and edges)
- Web crawling, Web spiders
- Answer connectivity and reachability questions
- Path finding
- Testing whether G is connected
- Computing a spanning forest/tree for a connected component
- Check whether a graph is acyclic (i.e., test for back edges)
- Compute a cycle if it exists
- Compute all connected components
- Compute a path between two vertices, if one exists
- Many more!!!

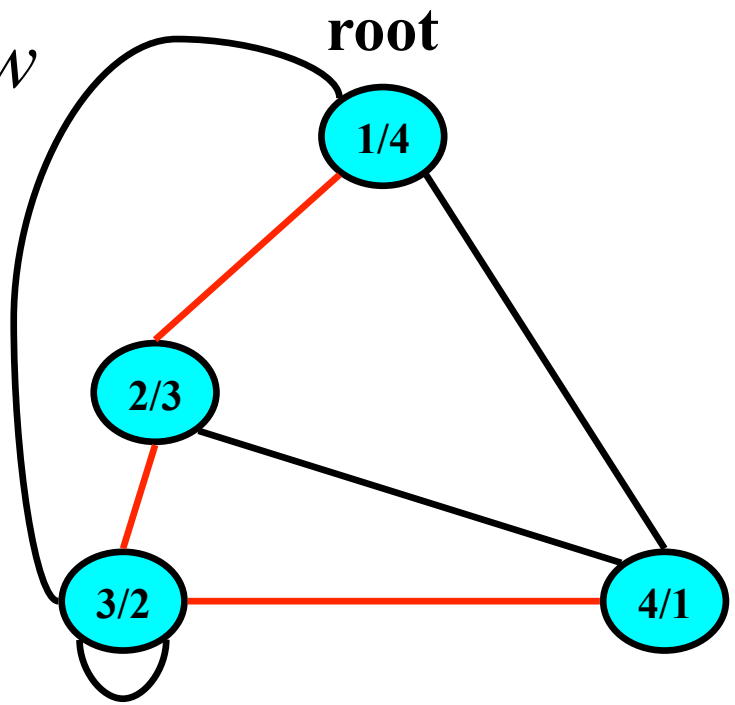
Depth First Spanning Forest

- **Discovery arc**
  - The arcs leading to unvisited nodes in the dfs traversal are called tree or discovery arcs and form a forest
- **Back arc**
  - A back arc goes from a node to one of its ancestors (including self-loops) in the spanning forest
- **Forward arc**
  - A non-spanning arc that goes from a node to a proper descendant (exclude self-loops) is called a forward arc
- **Cross arcs**
  - Arcs that go from one node to another node that is neither an ancestor nor a descendant is called a cross arc

- **Notes**
  - The spanning tree is not unique and depends on the starting node and in which order we explore the edges from a given node

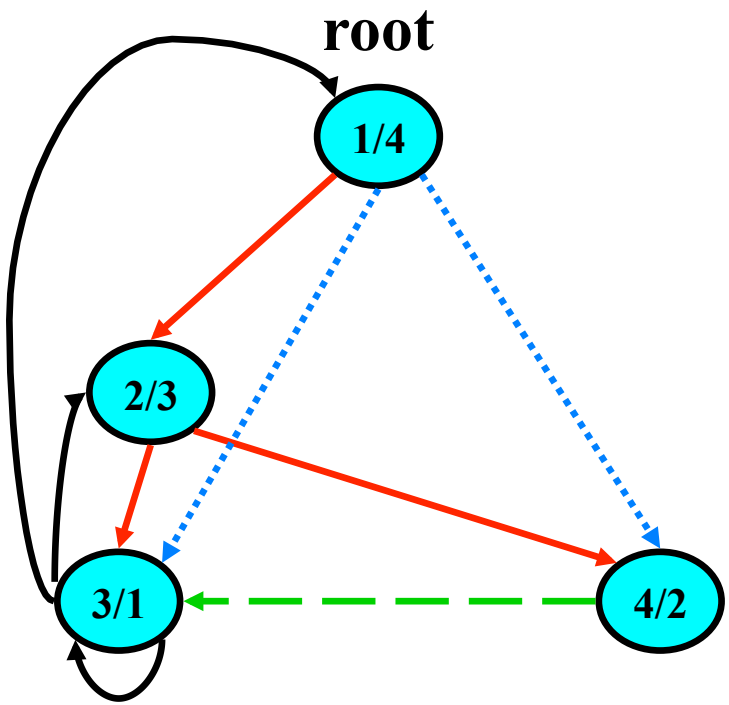
Characterization of Edges in Undirected Graphs

- Consider an arc from v to w
- **Discovery edge**
  - Leads to unvisited node
- **Back edge**
  - Leads to an ancestor
  - Pre[v] $\geq$ Pre[w]



Characterization of Arcs in Directed Graphs

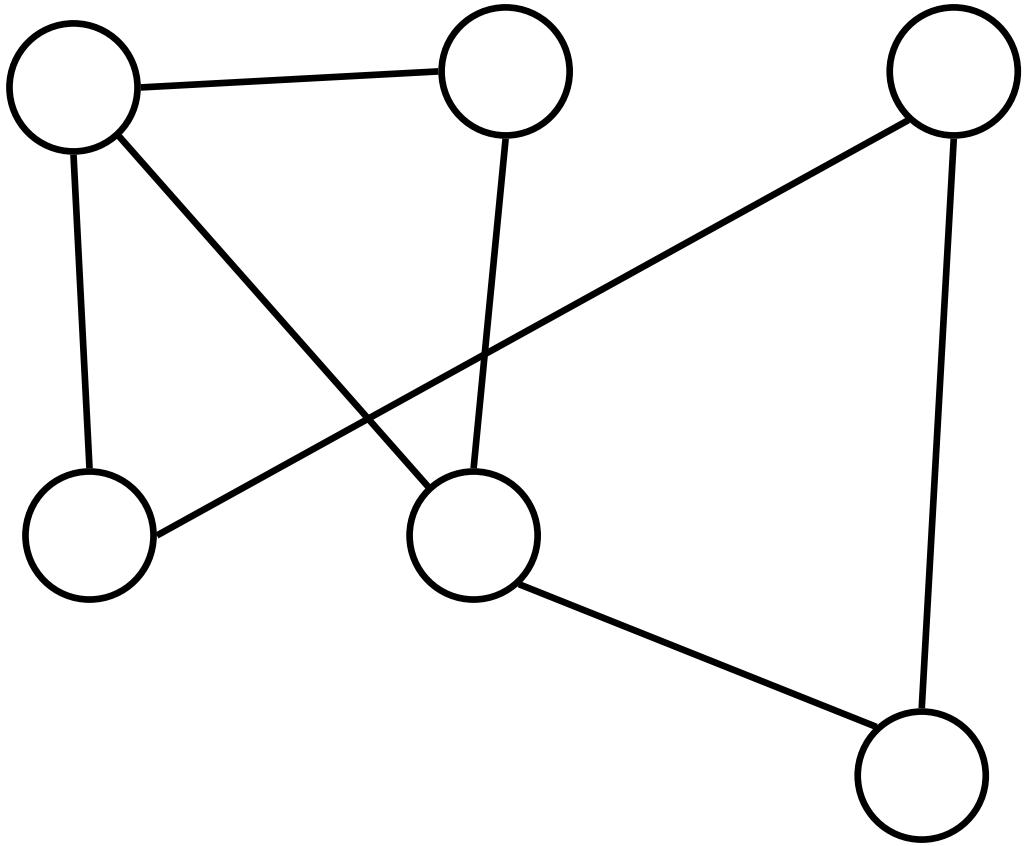
- Consider an arc from v to w
- **Discovery arc**
  - Leads to unvisited node
- **Forward arc**
  - Leads to a proper descendant
  - Pre[v] < Pre[w]
- **Back arc**
  - Leads to an ancestor
  - Pre[v] ≥ Pre[w]
- **Cross arc**
  - Leads to a sibling
  - Back arc: Post[v] ≤ Post[w] (equal for self-loops)
  - Cross arc: Post[v] > Post[w] (leads to an already explored sub-tree)



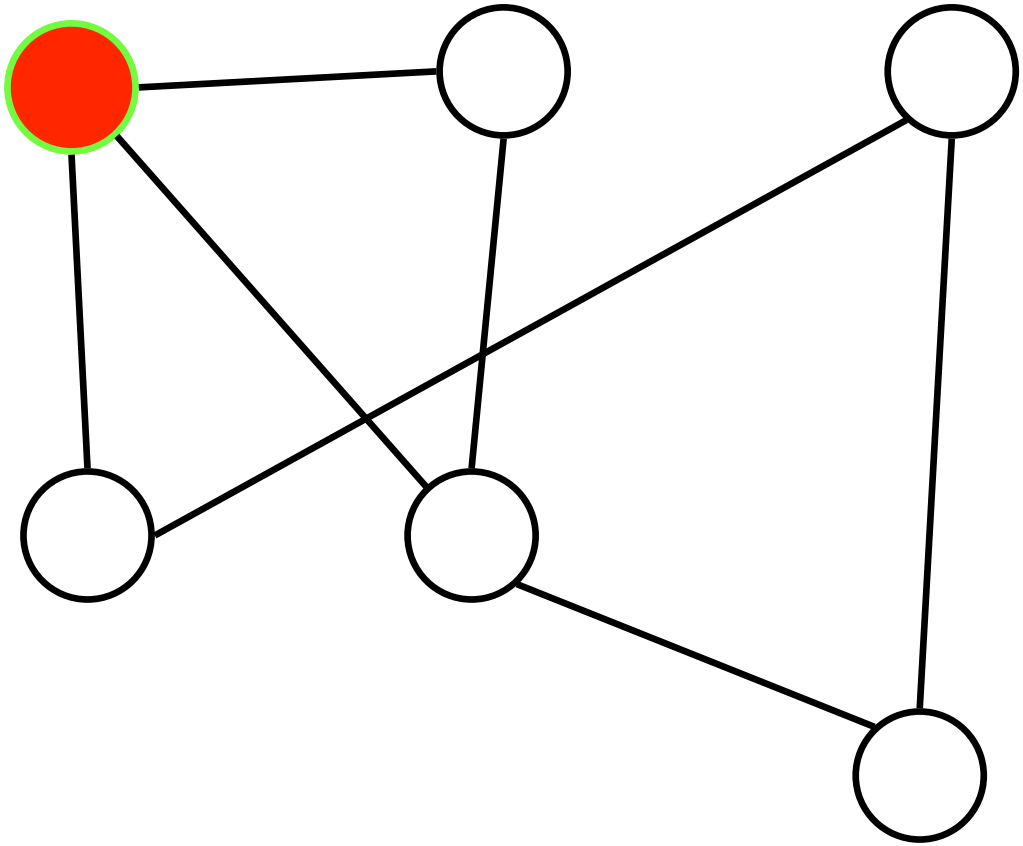
Breadth First Search (BFS)

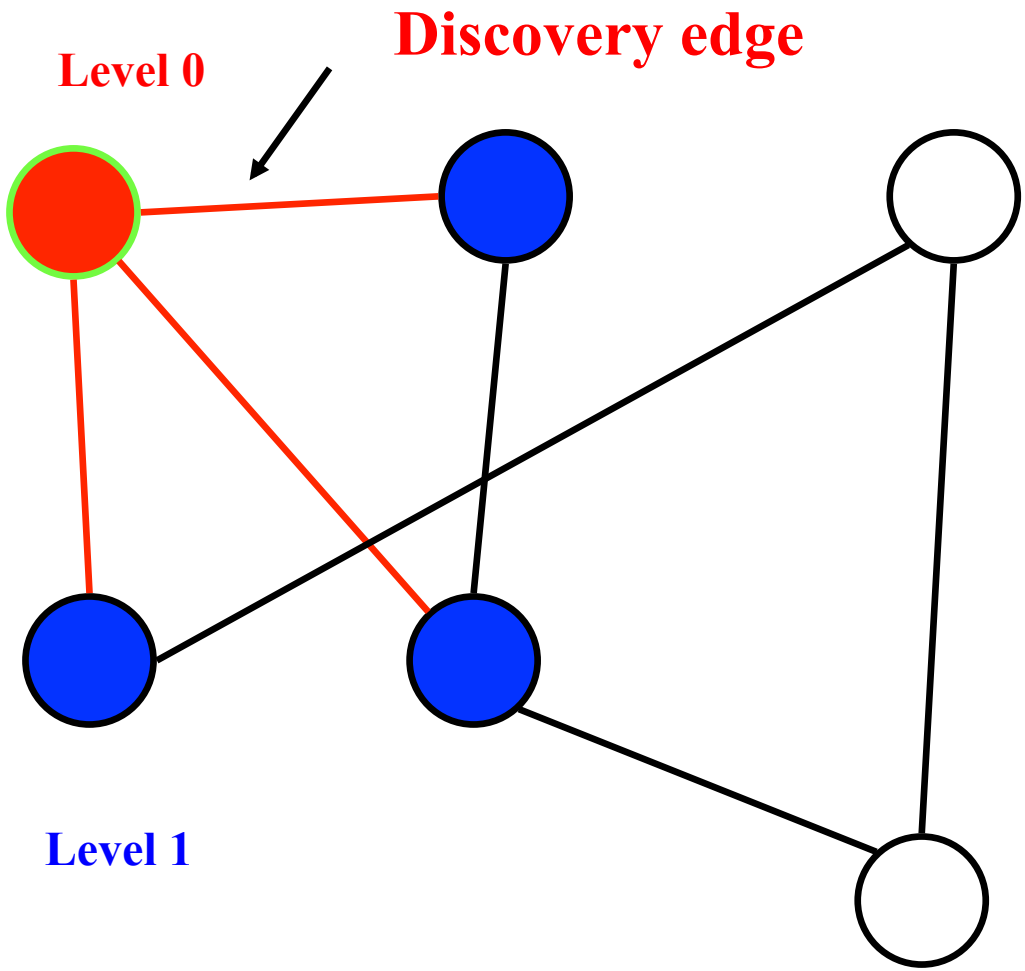
- Traverses a connected component of a graph and defines a spanning tree of it
- During BFS, the vertices are classified into layers or levels
- BFS is started at a fixed vertex *v* and puts *v* into level 0
- From there BFS visits all adjacent vertices and places them into level 1
- BFS then visits all the vertices adjacent to the vertices in level 1 that have not been visited yet and places them into level 2
- …

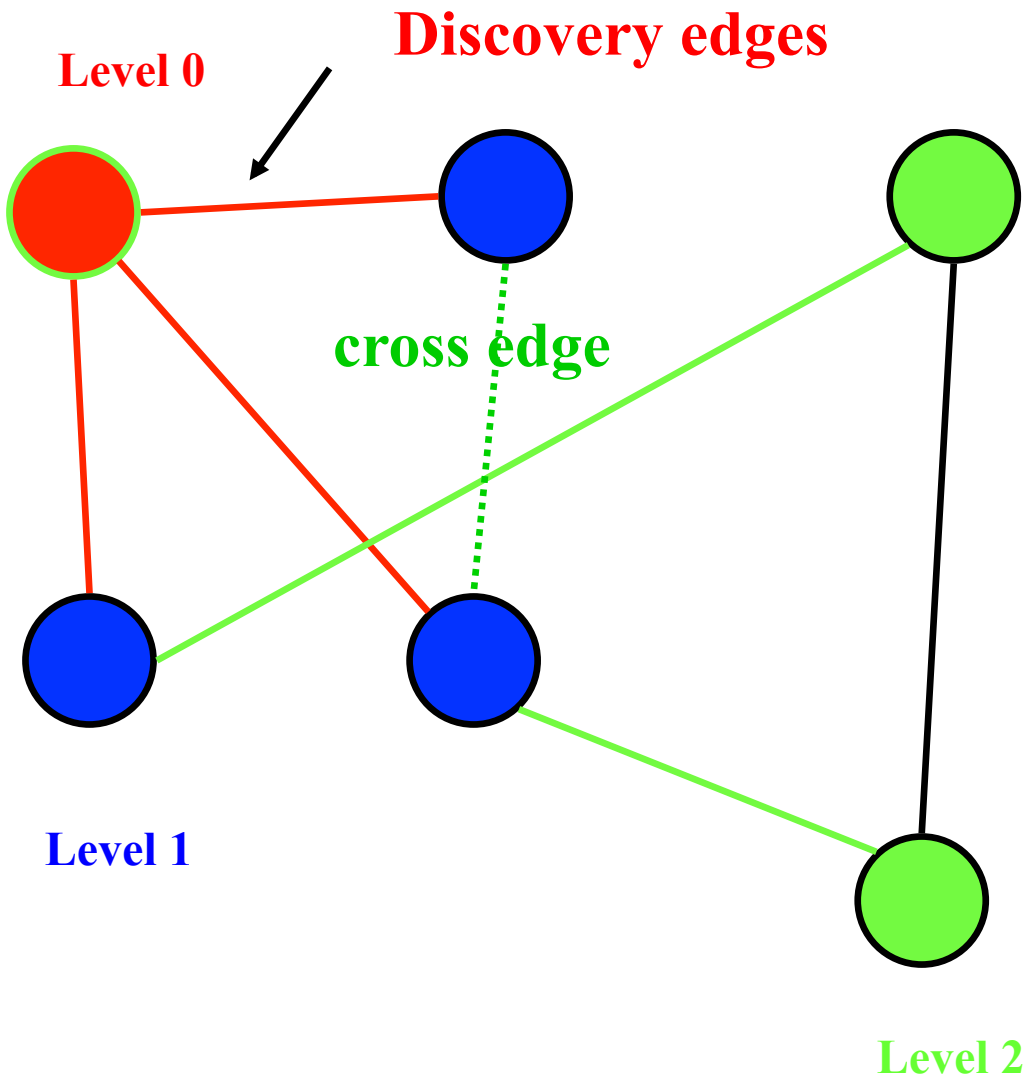
BFS Example

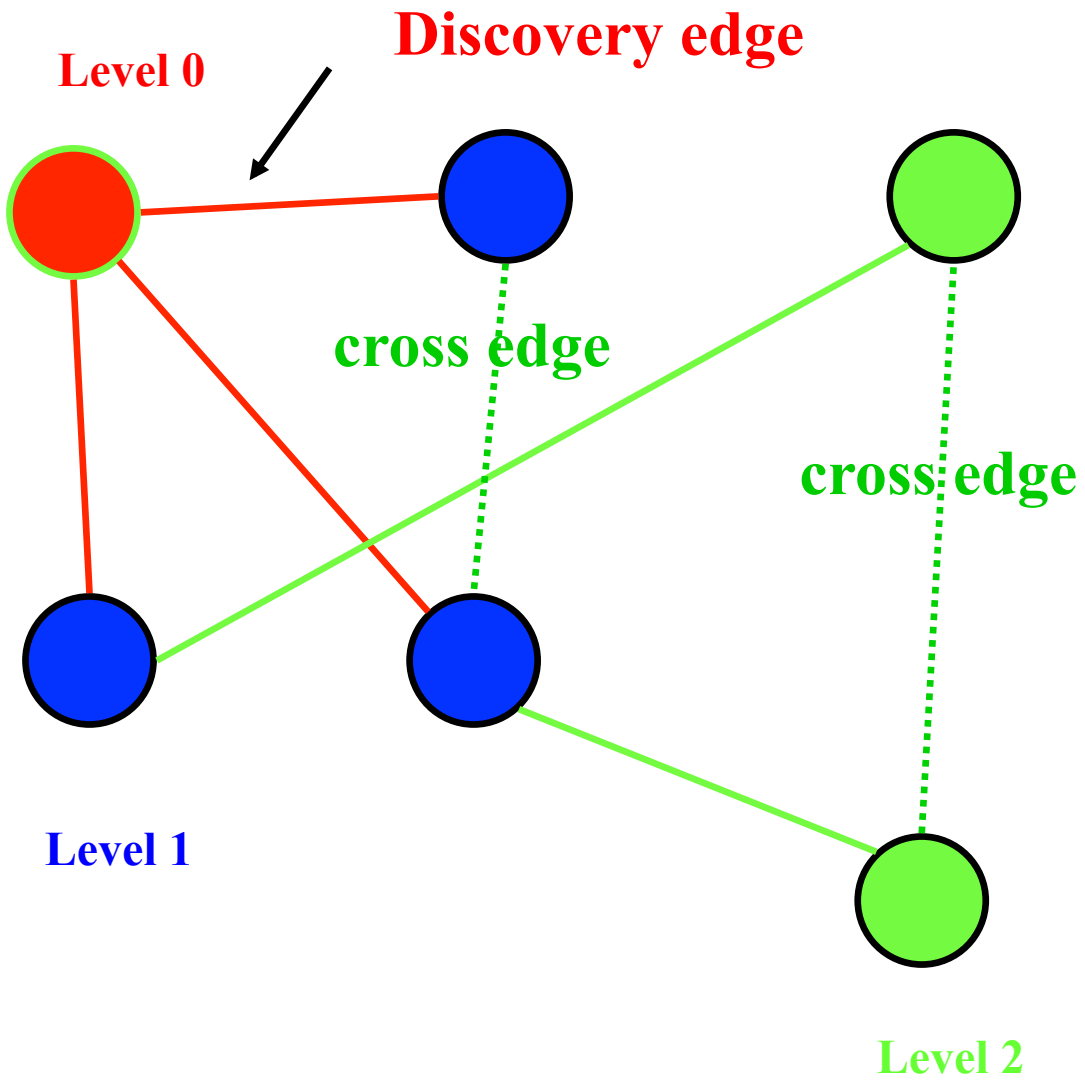


Level 0




Discovery edge
Level 0
Level 1


Discovery edges
Level 0
cross edge
Level 1
Level 2


Discovery edge
Level 0
cross edge
cross edge
Level 1
Level 2

Algorithm BFS

Input: A graph G and a vertex v of G

Output: A labeling of the edges in the connected component as discovery edges and cross edges

Algorithm BFS

```
Algorithm bfs(Node v)
   Queue Q;
   mark v as visited
   Q.enqueue(v)
   while not q.empty() do
      v = Q.dequeue()
      process(v)
      for each node w adjacent to v do
         if  w not visited yet do
            Q.enqueue(w )
          end
      end
   end
end
```

Time Complexity of BFS

- **Theorem.**
  - The time complexity of BFS traversal for a graph G = (V, E) is
    $O(n + m)$ for $|V| = n$ and $|E| = m$