CS416

Introduction to Computer Science II

Spring 2018

8 Trees and Graphs

Chapter 15

Previously in 416

- Abstract Data Types
  - Specification
  - Stacks, Queues, Lists
  - Dictionary
- Concrete Data structures
  - Implementation
  - Lists, arrays, hash tables

Preview

- *Program State*
- Tree Abstract Data Type
- Tree data structures
  - binary trees
  - n-ary trees
- Tree algorithms
  - Tree algorithm complexity
  - Quadtrees
- Graphs

Spatial Data Structures

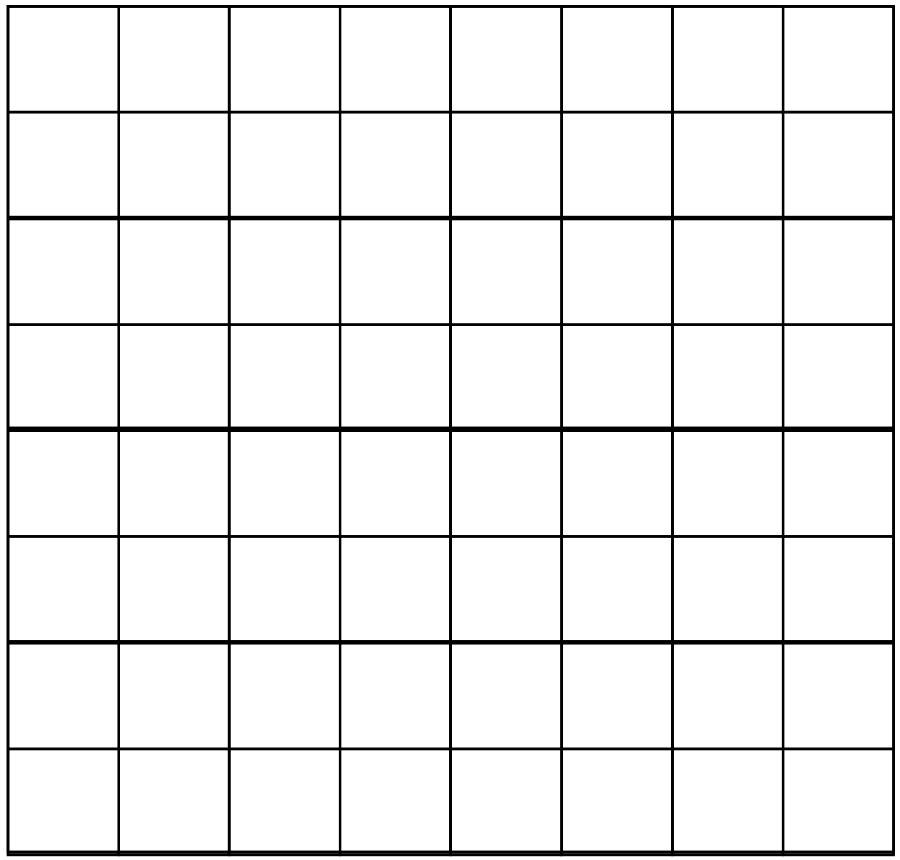
- Lots of applications contain data that has inherent spatial characteristics
  - Simulations of physical phenomena (fluid dynamics, e.g.)
  - Computer Aided Design (CAD)
  - Architectural applications
  - Earth-based sensor and image data
  - Computer gaming
  - much, much more

Spatial Search

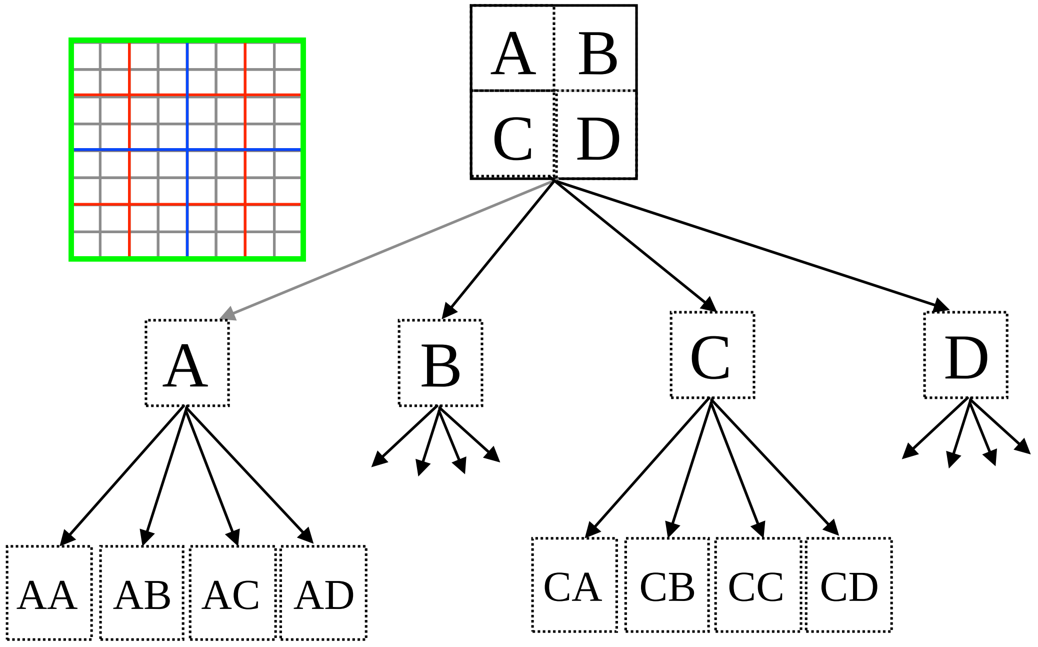
- Spatial applications often need to ask:
  - Where are all the objects or data that satisfy some condition?
  - Given a spatial region, what objects are in that region, or what characteristics do the data in that region have?
- Questions such as these can be facilitated with <u>hierarchical spatial data structures</u>

2D Hierarchical Spatial Data Structures

- Partition the application data space into 4 quadrants
- Partition each quadrant again into 4 quadrants
- and again
- upto a specified limit.
- But what good is it?
  - Can be basis for efficient spatial search algorithms



Quadtree Data Structure



- The quadtree is an efficient tree data structure for representing 2D spatial data
  - Each internal node of a quadtree has 4 children
    - each child represents 1/4 of the region occupied by its parent

Storing Point Data in a Quadtree

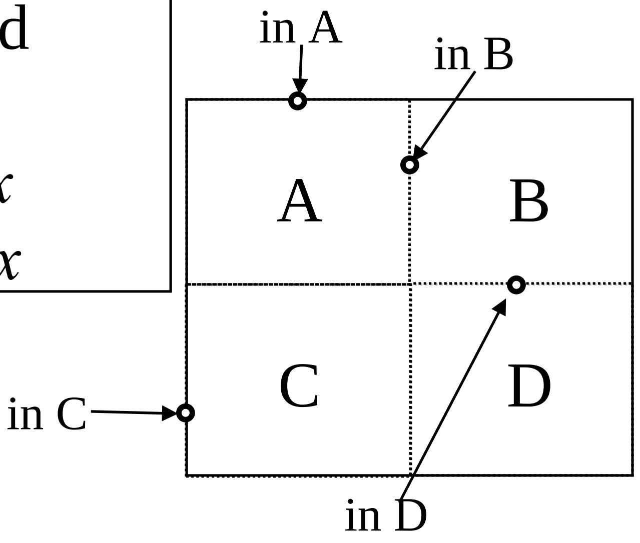
- Scattered point data maps naturally to a quadtree
  - Each data object has an *x,y* location <u>and</u> 1 or more *scalar variate* values at that location, such as pressure, temperature, etc.
  - Nodes often maintain point count, *min/max/average* values of variates in region

```
void addPoint( qnode, data ):
update min/max data for this qnode
if ( qnode is a leaf )
   add data to data vector for leaf
else
   child = subtree containing data
   addPoint( child, data )
```

Easy test on point's (*x,y*) compared to child boundaries:

$$xmin \le x < xmax$$

$$ymin < y \le ymax$$



These tests encapsulate a convention for edges:
-- a point on a top or left edge is **in** the cell

Quadtree Search: Find points in a region

- Find all points in *Region*, *r*, defined by (x1,y1) to (x2, y2)

```
Vector<Point> v = new Vector<Point>();
// quadtree already has points in it
Region        r = new Region( x1, y1, x2, y2 );
findPoints( qtree.root, r, v );
```

```
void findPoints( qnode, region, results ):
if ( intersection( region, qnode ))
   if ( qnode is leaf )
      for each point in qnode.points
         if point in region
            results.add( point )
   else
      for each child of qnode
         findPoints( child, region, results )
```

Quadtree Search: find max value of attribute

- Find the point with the maximum temperature

```
Data dummyMin = new Data( Float.MIN_VALUE )
Data maxTemp  = findMaxTemp( qtree.root, dummyMin )
```

```
Data findMaxTemp( qnode, maxSoFar ):
if ( qnode is leaf )
   maxSoFar = maxLocalTemp( maxSoFar, qnode.points )
else
   for each child of qnode
      if ( maxSoFar.temp < child.maxTemp )
         maxSoFar = findMaxTemp( gnode.child, maxSoFar )
return maxSoFar
```

```
Data maxLocalTemp( maxData, points )
for each point in points
   if ( point.temp > maxData.temp )
      maxData = point;
return maxData
```

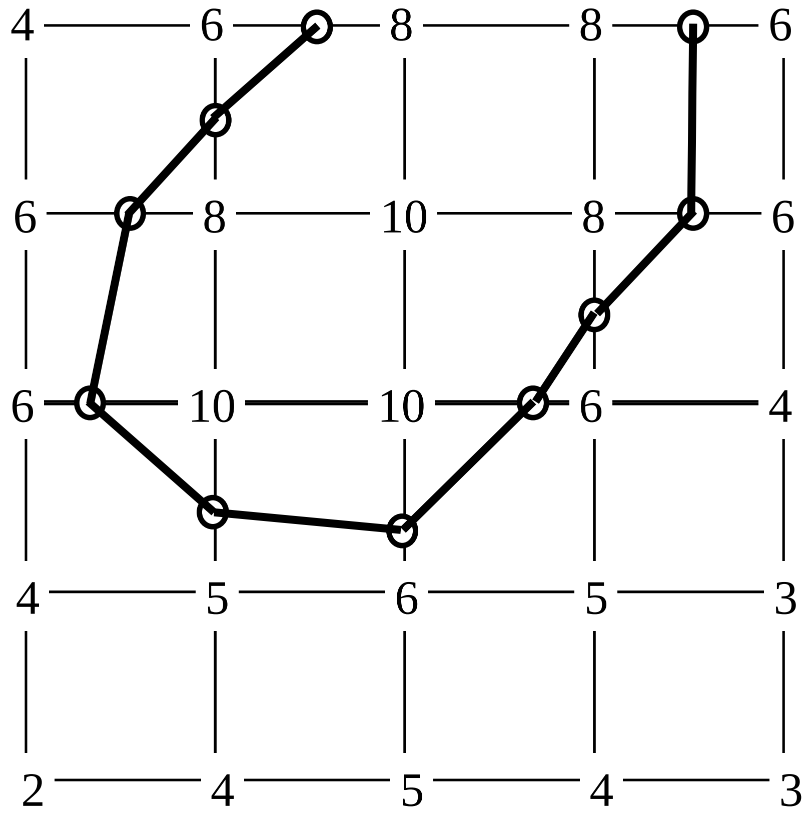
2D Scalar Data

- 2D scalar data can be stored in a 2D array
  - Each point is a sample representing the height, or temperature or pressure at a specific point in a 2D continuous space
- Can display as a color plot with one pixel (or block of pixels) for each data point where color is related to the data value
- Can also display with isolines, representing equal values in the data (contours)
  - Contour display can be significantly speeded up using a quadtree

Isolines (Contours)

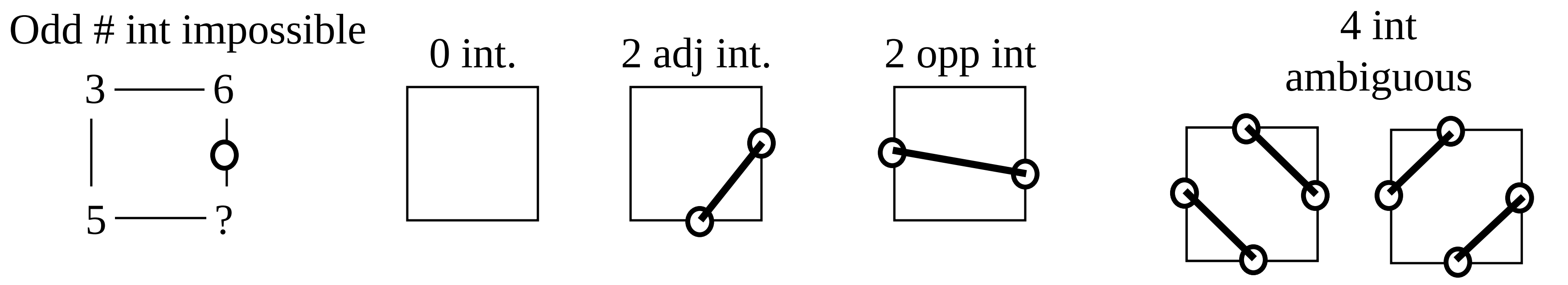
Choose an isovalue, e.g., 7
For each edge of each cell
        if corners straddle 7
            interpolate along edge to estimate location of 7
        connect points in cell

Note: this example is unrealistically simplified, but the complete algorithm still isn’t very complicated.



Cell Isoline Cases

- Each cell has 4 edges, each edge may or may not have 1 contour intersection point.
- In principle, that's 16 potential cases, but some are impossible; others are rotationally equivalent; just 5 unique cases:



Ambiguous cases resolved by looking at neighbors and their neighbors - not straightforward

Quadtree in Isoline Algorithm

- Isovalues typically occupy a very small % of the area
- If quadtree nodes store *min* and *max* values for all of the region defined by the node, the processing of **lots** of cells can be eliminated.

```
void drawContour( qnode, isoValue ):
if ( qnode == null ||
     qnode.max < isoValue ||
     qnode.min > isoValue )
  return
if ( qnode is leaf )
  draw IsoContour in qnode area
else
  for each child of qnode
    drawContour( child, isoValue )
```

Storing Objects into a Quadtree

- Have spatial objects stored in array or hashtable or other linear data structure.
- Want to find collisions efficiently
- Build quadtree
  - For each object, identify all leaf regions that contain part of the object
  - Traverse octree comparing bounds of node with bounds of the object
- Each leaf contains list of objects in it

Finding Collisions

- Consider a 2D arcade like game (ex. Space Invaders)
  - Have many targets scattered throughout the game region
  - Have a small number of moving objects (cannon balls)
  - For every frame, must compare every moving object to every other object (moving or stationary) for possible collision
  - If the region is partitioned by a quadtree, however, we can greatly reduce the overhead of the algorithm.

Quadtrees Aid Collision Detection

```
void newFrame():
  update moving objects' positions
  testCollide( quadtree.root )
```

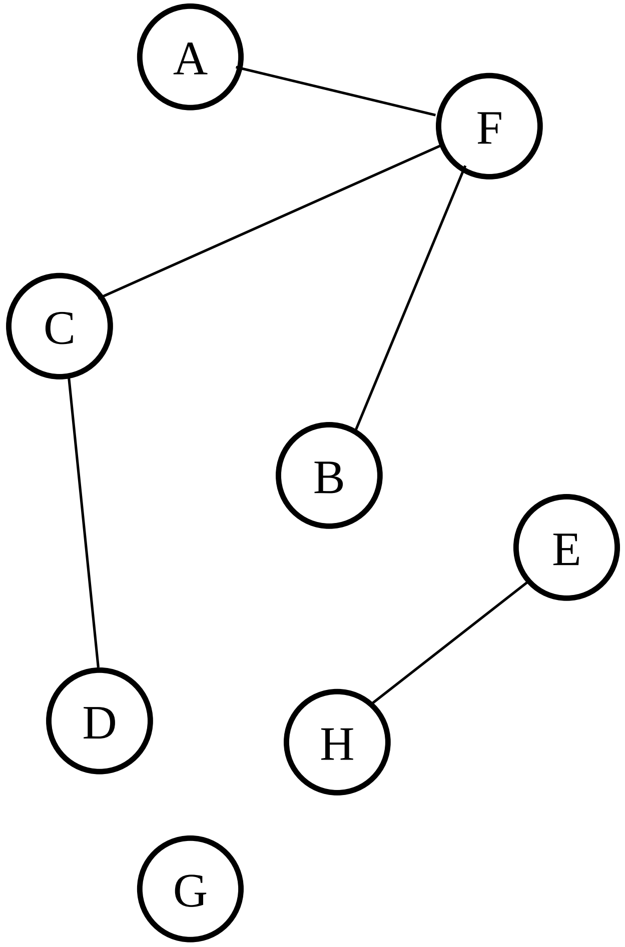
```
void testCollide( qnode ):
for each moving object
  if obj intersects qnode
     if qnode is leaf
        testObjects( obj )
     else
        for each child of qnode
           testCollide( child );
```

Note: we don't really need a quadtree for this if the regions are equally spaced. Instead, we could just use a fixed partitioning of the space.

Graphs

(trees with loops!)

- Trees are a (very) special case of a more generic data structure called a ***graph***
- A graph G is a set of nodes (or vertices) and a set of edges (or arcs)that link a pair of nodes together
  - $G = (N, E)$
  - $N = \{n_0, n_1, \ldots n_{N-1}\}$
  - $E = \{e_0, e_1, \ldots e_{E-1}\}$
    - where each $e_k = (n_i,n_j)$ for some n_i,n_j in N.

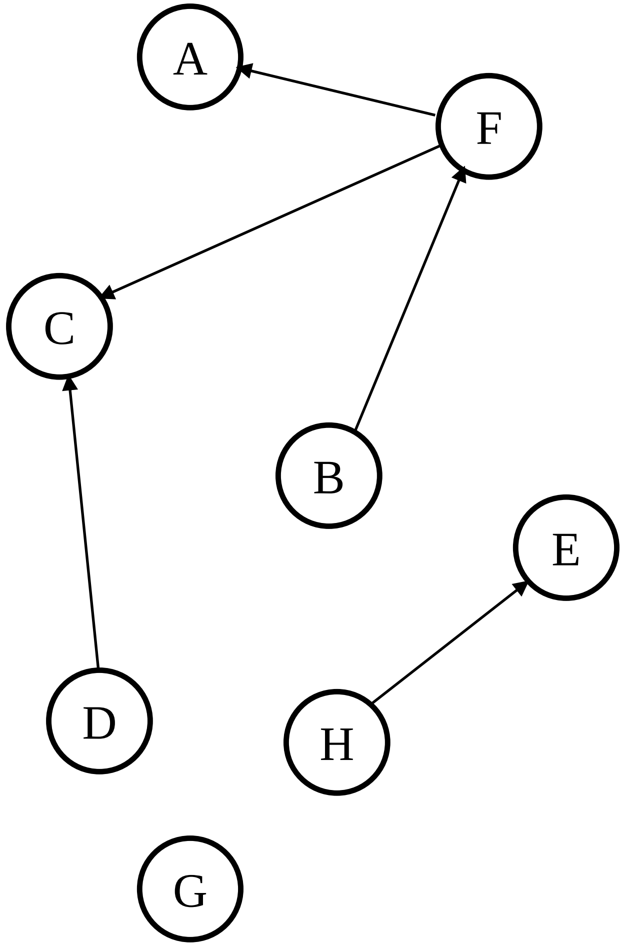


This is graph has disconnected components

Directed Graph

What exactly is an <u>edge</u>?

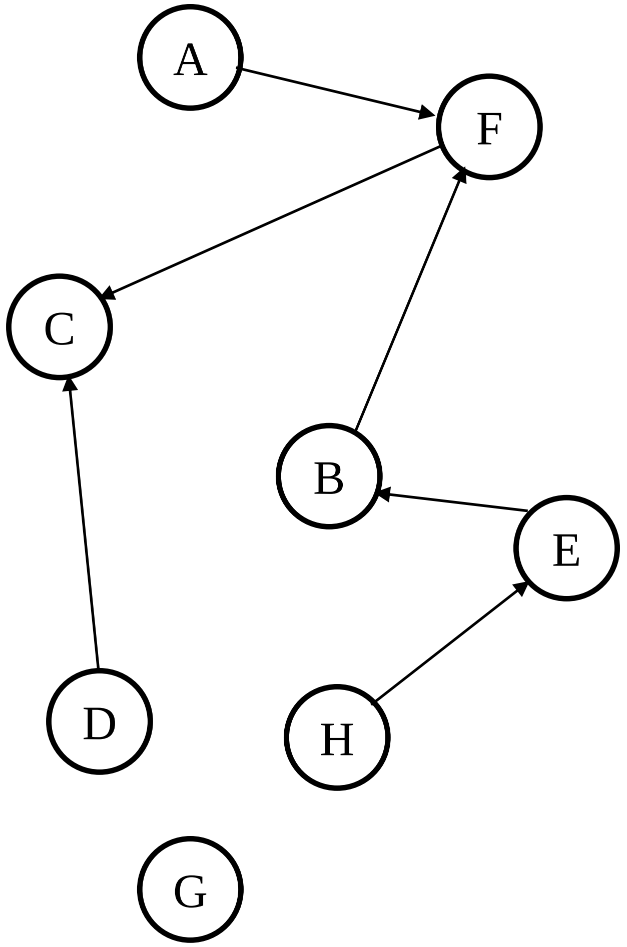
- If there is an edge between nodes A and B, does it represent a *symmetric* relationship?
  - Can you "go" from A to B **and** from B to A?
    - if yes, it is an *undirected* graph
    - if no, it is a *directed graph*



Arrow heads on the edges imply a *directed* graph

Graph Cycles

- This graph has no cycles
  - No matter where you start, there is no path along the edges back to the start.
- Let's add a more edges
  - D to H is ok
  - B to D creates a cycle: BEHD



Directed Graph Examples

- Organization chart where some employees report to multiple managers
- A knowledge dependency graph:
  - need to know A in order to understand B
- A maze "access" graph:
  - From room A, you can go to rooms B, Q, and Z

Magic Dungeon Maze

- Suppose you want to write an adventure game with *n* rooms in a maze of unknown physical shape
  - The maze is defined only by a set of Room objects with a name and a set of *edges* that identify direct access from one Room to another.
  - An input file such as the one on the right could be used to define mazes of this sort in a generic way
    - This one defines the example graph shown previously

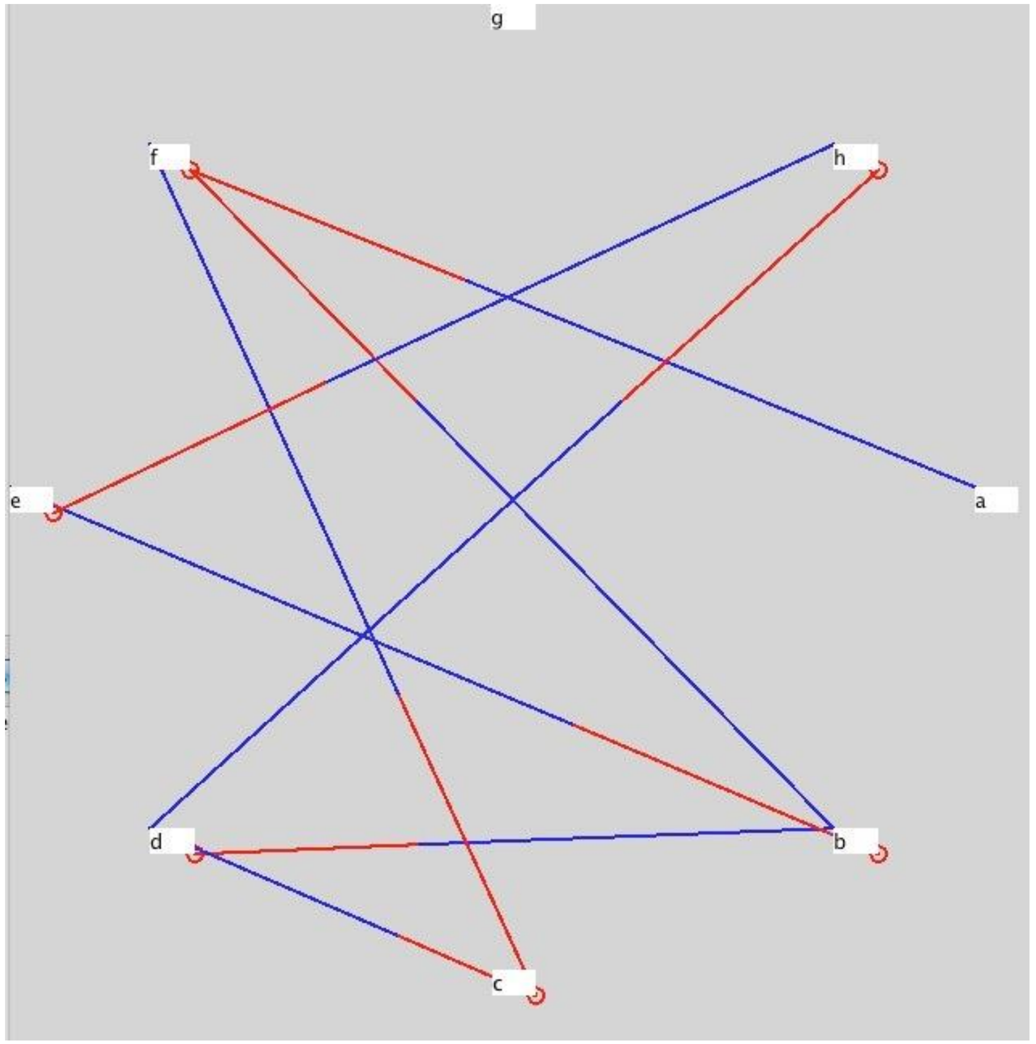
```
room A
room B
room C
room D
room E
room F
edge A F
edge F C
edge D C
edge B F
edge B D
edge E B
room G
room H
edge D H
edge H E
```

Drawing Graphs

- Displaying arbitrary graphs "cleanly" is a very **hard** problem, both programming-wise and with respect to complexity analysis
- Lots of *heuristic* approaches have been used
  - some of these are pretty good
  - an easy (but not so great) approach:
    - display representations (JLabel?) for all the nodes in a big circle on the screen
    - draw lines between labels to represent edges.

Circle Rep of Maze

- Nodes are JLabels
- Arcs are Line2D.Double
  - Blue end is start node
  - Red end is end node



Directed Graph Data Structure Representation

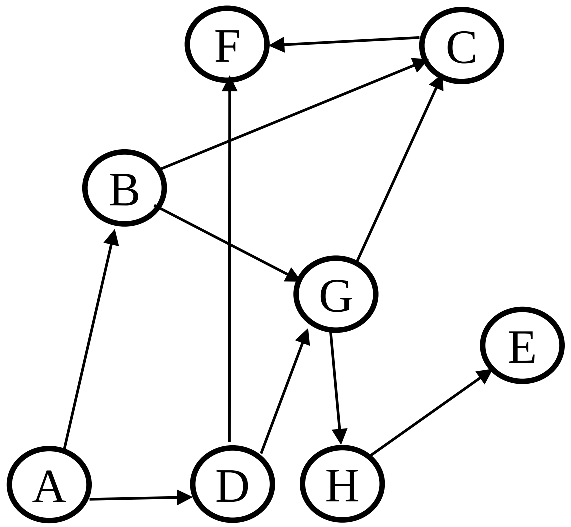
- There are lots of options for internal representation of graphs and performance can be an issue
- *Node* and *Edge* classes are obvious candidates
  - An *Edge* should contain references to start and end *Nodes*
  - A *Node* could contain an array, or *Vector*, or *ArrayList* of *Edge* objects
- A *Path* object might be convenient
  - could be defined as a collection of *Nodes* <u>or</u> *Edges*

Common Directed Graph Algorithms

- find any acyclic path from node A to node B
- find the shortest path from A to B
- find all acyclic paths from A to B

- Let's first try to find any acyclic path in a Directed Acyclic Graph, a DAG

findPath for a DAG



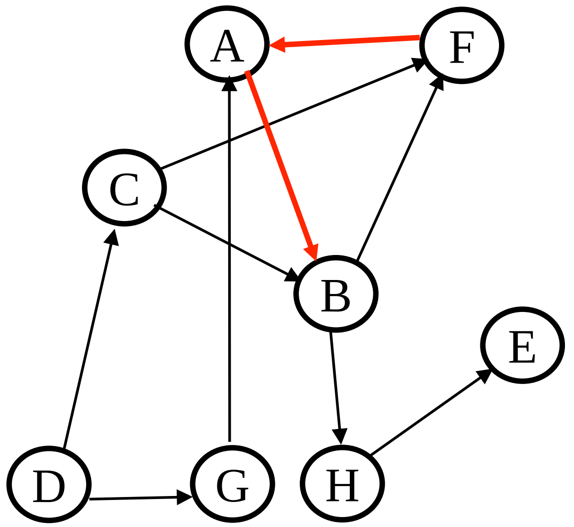
```
Path findPath( Node start, Node end )
if ( start == end ) // base case 1
  path = new Path
  path.add( start )
  return path
EdgeSet edges = start.getEdges
if edges == null
  return null  // 2nd base case
for each edge
  // try going down this edge
  path = findPath( edge.end, end )
  if path != null
    append start to front of path
    return path // 3rd base case
return null   // 4th base case
```

```
Path findPath(A,H):
  A+findPath(B,H)
    B+findPath(C,H)
      C+findPath(F,H)
        null
      null
    B+findPath(G,H)
      G+findPath(H,H)
      G+H
    B+G+H
  A+B+G+H
```

```
Path findPath(H,A):
  H+findPath(E,A)
    null
  null
```

DAG.findPath fails with cycle

Add a cycle!



```
Path findPath( Node start, Node end )
if ( start == end )
  path = new Path
  path.add( start )
  return path
EdgeSet edges = start.getEdges
if edges == null
  return null  // can't get to end!
for each edge
  // try going down this edge
  path = findPath( edge.end, end )
  if path != null
    append start to front of path
    return path
return null
```

```
Path findPath(C,H):
  C+findPath(B,H)
    B+findPath(F,H)
      F+findPath(A,H)
        A+findPath(B,H)
          B+findPath(F,H)
            . . . . .
            infinite loop
```

How can we *detect* the recursion loop?

- check if start is already in the path, **or**
- mark a node when it goes into the path; coming to a marked node is another *base* case for the recursion.

findPath v.3

```
boolean findPath( Path p, Node s, Node e )
if ( s is marked )
  return false
if ( s == e )
  p.add( s );
  return true
EdgeSet edges = start.getEdges
if edges == null
  return false  // can't get to end!
p.add( s )
mark s
for each edge
  // try going down this edge
  if findPath( p, edge.end, end )
    return true
unmark s
p.remove( s )
```

It may be easier to pass the path being computed as a parameter to the findPath method as in this variation.

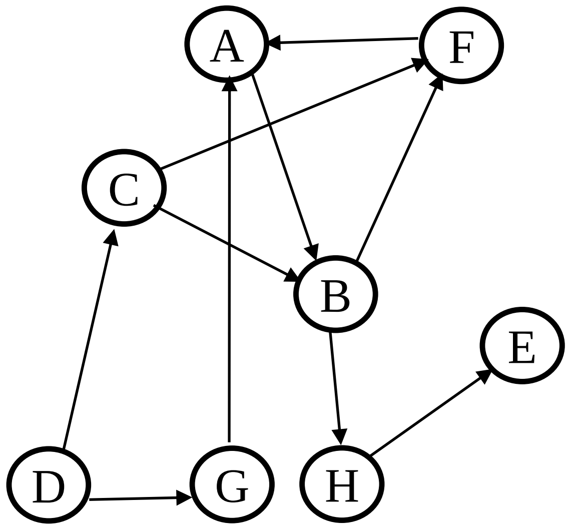
Stop recursion when we see we have a cycle.

Here we mark the node as visited.

If we didn't find a path with this node on it, we need to remove it from the path.

Note we "unmark" the node. This isn't strictly needed to find "any" node, but it is for most other search algorithms.

findPath v.3 Example



```
boolean findPath( Path p,
            Node s, Node e )
if ( s is marked )
  return false
if ( s == e )
  p.add( s );
  return true
EdgeSet edges = start.getEdges
if edges == null
  return false
p.add( s )
mark s
for each edge, e
  // go down this edge
  if findPath( p, e.end, end )
    return true
unmark s
p.remove( s )
```

```
findPath( path, C,H):
  findPath(B,H)
    findPath(F,H)
      findPath(A,H)
        findPath(B,H)
          B marked, fail
        no more A edges, fail
      no more F edges, fail
    findPath(H,H)
      H==H, return true
  return true
return true
```

path
C
CB
CBF
CBFA
CBFA
CBF
CB
CB
CBH
CBH
CBH

Graph Implementation

- Implementation of a graph data structure provides some performance challenges, especially when the number of arcs from nodes gets very large
- Suppose you want to find out if there is an edge from the current node to a specific other node.
  - You could store the edges as an array and search, but this could be expensive if there are hundreds of edges
  - Java provides a variation of a hash table that can help

Review

- Binary search tree
  - concepts
  - Tree implementation
- Recursion with trees
- Game trees
  - minimax algorithm
- Quadtrees
- Graphs

Next, in 416

- Advanced String processing
  - Regular expressions
- Applications to genetics