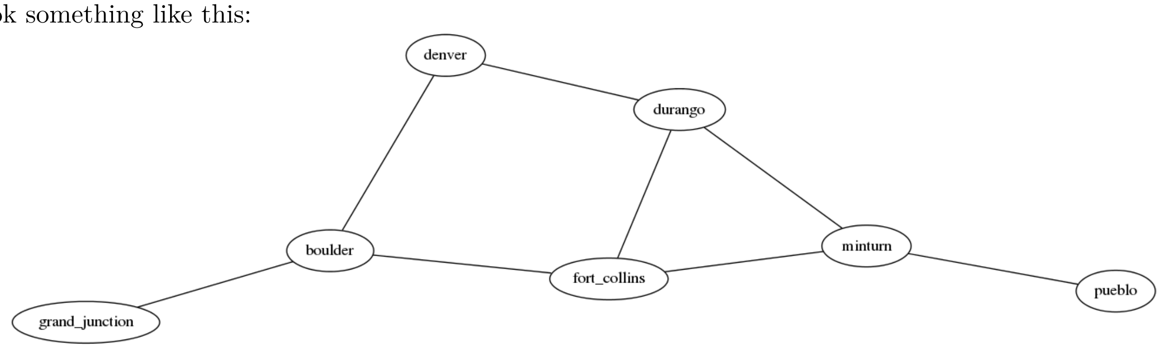
GRAPHS

CS 200 RECITATION 13

Before we start, we will do course evaluations

Graphs

Abstract data type with nodes and edges, kind of like a tree except that graphs do not have the hierarchy that trees do. In general trees are viewed as a subset of graphs (they are a graph with extra rules). Graphs look something like this:



Nodes and edges, like a tree, but with out any overall ordering. These edges have no direction, so this graph is undirected. A directed graph's edges have direction (indicated with an arrow), and can only be traversed in the correct direction.

Terminology

- Adjacent - two vertices (nodes) are adjacent if they are directly connected by an edge
- Incident - an edge is incident on two nodes (the ones it touches)
- Degree - of a vertex, the number of edges incident upon it (how many edges connect to this vertex)
- Self Loop - an edge which connects a vertex to itself
- Simple Graph - no self loops, no two edges connect same vertex pair
- Multigraph - may have multiple edges between same vertex pair
- Psudograph - multigraph with self loops.
- Cycle - repeatable path through the graph.

Traversals

- Depth first search: go as deep as you can visiting nodes along the way, then backtrack to catch any missed nodes.

- Psudocode:
- ```
  dfs(in v:Vertex)
    s - stack for keeping track of active vertices
    s.push(v)
    mark v as visited
    while(!s.isEmpty()) {
      if (no unvisited vertices adjacent to the vertex
      on top of the stack) {
        s.pop() \\backtrack
      else {
        select unvisited vertex u adjacent to vertex on
        top of the stack.
        s.push(u)
        mark u as visited
      }
    }
  ```

- Breadth first search: visit every adjacent node, then go one level deeper and repeat
  - Psudocode:
  - ```
    bfs(in v:Vertex)
      q - queue of nodes to be
      processed
      q.enque(v)
      mark v as visited
      while(!q.isEmpty()) {
        w = q.dequeue()
        for (each unvisited vertex u
        adjacent to w) {
          mark u as visited
          q.enqueue(u)
        }
    }
    ```

Implementation

3 ways to implement a graph ADT:

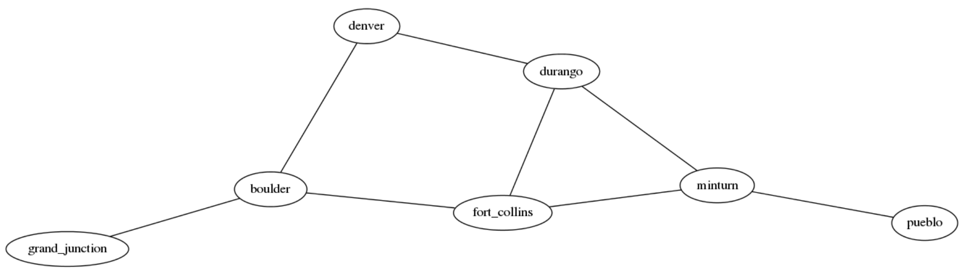
- Adjacency matrix
  - use a matrix to store which nodes are connected by an edge.
	first	second
first	0	1
second	1	0
  - first is connected to second
  - and second is connected to first
  - matrix[i][j] tells you if there is connection between i and j
  - for today, the row label will be the 'from ' node and the column label will be the 'to' node. This is not set in stone, but I had to pick one.
- Adjacency list

    - each node keeps a list of outgoing edges (if undirected, it's just a list of edges). This stores edge data on the nodes as opposed to the above matrix which stores it independently of the nodes.

- Nodes and References (like out trees from before, not used much)
    - variant of the adjacency list.

Exercise

Your exercise for today is to convert my skeleton code (from the course website) into a Graph data structure, based on an adjacency matrix. Details of the skeleton:

- **MainClass.java** - the familiar main class, used for testing. I was not able to make a good randomization system so I had to hard code the testing a bit. Tries to build this graph:
    - 

    - MainClass also has a method for you to make your own graph.

- **GraphADT.java** - is the graph class. It's generic type T specifies what type of data the nodes store (String in this example), T must implement the Comparable interface. Also stores a list of nodes separate from the adjacency matrix.
    - This list of nodes is used to build the matrix, and it's indices match the matrix's row and column labels. So nodes[i] and matrix_row[i] refer to the same node (matrix_row is data gotten from the matrix). This is a somewhat risky design decision on my part, since it invites programming errors in the long run, but it is much simpler than the alternative.

- **AdjacencyMatrix.java** - the adjacency matrix used by GraphADT, is a Vector of Vectors (a 2d Vector). Vector is a list type similar to ArrayList

I've tried to document the skeleton code a bit better, so it should be easier to figure out how to use it.

What you need to do:

- implement both versions of the addEdge method
- implement the DFS method, depth first search which traverses the whole tree
    - hint: use java's stack class
- implement the BFS method, breadth first search which traverses the whole tree
    - hint: use a java's linked list class, which implements the queue interface but the method names are different: offer() is enqueue, poll() is dequeue, peek() is peek.
- implement the test2 method in MainClass to build and test your own tree (print it out, DFS, BFS)

Grading

- Sign the attendance
- turn in your code via checkin:
  - Monday: ~cs200/bin/checkin R13L01 R13L01.tar
  - Tuesday: ~cs200/bin/checkin R13L02 R13L02.tar
  - Wednesday: ~cs200/bin/checkin R13L03 R13L03.tar
  - Thursday: ~cs200/bin/checkin R13L04 R13L04.tar
- don't worry if your not finished, just turn in what you have and add a readme file saying that you ran out of time. This will not cost points.

Sample Output:

```
Graph ADT. Adjacency matrix:
                denver  boulder  fort_collins  durango  grand_junction  minturn
pueblo
          denver  [0, 1, 0, 1, 0, 0, 0]
         boulder  [0, 0, 1, 0, 1, 0, 0]
    fort_collins  [0, 0, 0, 1, 0, 1, 0]
         durango  [0, 0, 0, 0, 0, 1, 0]
  grand_junction  [0, 0, 0, 0, 0, 0, 0]
         minturn  [0, 0, 0, 0, 0, 0, 1]
          pueblo  [0, 0, 0, 0, 0, 0, 0]

Depth first search:
[denver, boulder, fort_collins, durango, minturn, pueblo, grand_junction]

Breadth first search:
[denver, boulder, durango, fort_collins, grand_junction, minturn, pueblo]

graph cities{
   denver -- boulder
   denver -- durango
   boulder -- fort_collins
   boulder -- grand_junction
   fort_collins -- durango
   fort_collins -- minturn
   durango -- minturn
   minturn -- pueblo
}

test2 not implemented
```