Graphs

Graph \( G = \{ \text{vertex}^*, \text{edge}^* \} \) \hspace{1cm} // \text{object}^* \equiv 0 \text{ or more objects}

edge connects 2 adjacent (neighbor) vertices \hspace{1cm} // \text{vertex} \equiv \text{node}

directed graph has edges traversable in 1 direction
undirected graph edges can be traversed in either direction

weighted edge has information (label or value)
e.g., distance, cost, difficulty…

path between vertices (start to stop || source to destination …) is an (edge*) that when traversed starts at the source and ends at the destination

\( (\text{edge}^*) \equiv \text{ordered set of edges} \)

cycle exist in graph when 2+ paths connect 2 vertices

connected graph there exists a path from any vertex to another vertex

complete graph each pair of vertices in graph share an edge
Adjacency Matrix

A graph of n nodes has a square \([n, n]\) matrix for connected nodes. For connected node \(i\) and node \(j\), value in cell \([i, j]\) \(!= 0\). The matrix is redundant (symmetrical) and can be sparse. Nodes stored in collection size n.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>c</th>
<th>f</th>
<th>m</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>f</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>m</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Adjacency List

nodes stored in collection (JFC: Arraylist, TreeMap, HashMap) where each node has list of references to adjacent nodes

‘a’ is the vertex and ‘A’ is a pointer to ‘a’
Graph traversal (search) visits all vertices connected to traversal starting vertex.

class Vertex {
    ...
    // sorted || unsorted Collection ??
    Collection <Vertex> adjacent
    boolean marked()
    void mark()
    ...
}

Depth-first search (DFS)

dfs (Vertex v) {
    v.mark() // mark ≡ visited
    for (Vertex av : v.adjacent)
        if (! av.marked()) dfs(av)
}
**BFS**

Breadth-first search (BFS)

```java
bfs (Vertex v){
    Queue <Vertex> q = new Queue<Vertex>()
    v.mark()
    q.enqueue(v)    // arrive
    while (! q.empty()) {
        v = q.dequeue()    // depart
        for( Vertex av : v.adjacent) {
            if (! av.marked()) {
                av.mark()
                q.arrive(av) }
        }
    }
}
```

bfs( x)    // visit adjacent counter-clockwise
x, a, f, c, m
BFS() – setSimModel()

To use simulationFramework separate the initialization of your "algorithm" into the setSimModel() and simulateAlgorithm() methods.

Its necessary to distribute the checkStateToWait() method calls after each "step" of algorithm you want to visualize.

Method checkStateToWait() causes the "scene" to be redrawn.
    permanent Markers can be used for nodes
    temporary Markers can be used to show visits

```java
public void setSimModel()
    creates a Map<Node> class (aMap)
    for all the graph's nodes
    that sets each node's adjacency collection
    get the starting node for traversal
```
Map<Node>

ArrayDeque<Node> q = null;

public void mark(Node n) { // mark the node }

public void initialize (Node startNode)
    ArrayDeque<Node> q = new ArrayDeque<Node>();
    startNode.mark() // marking adds temporary drawable
    q.add(startNode)

public Node bfs()
    Node aNode;
    if (q.isEmpty()) // need to test halting condition
        return null; // done with search
    else // expand search
        aNode = q.remove()
        for (Node nNode : aNode.getAdjacents())
            if( ! nNode.isMarked() )
                nNode.mark() // change color size to visualize
                q.add(nNode)
        return aNode; // not done yet
BFS() – simulateAlgorithm()

```java
public synchronized void simulateAlgorithm()
    Node tNode;
    // initialization - code before loop
    aMap.initializeBFS(begin)  // begin is a Node
    // simulation loop
    while (runnable())  // becomes loop of algorithm
        tNode = aMap.bfs()  
        if (tNode != null)  // simulate step in search
            checkStateToWait()  // pause animation
        else  // search is done
            setSimRunning(false);
        animatePanel.setComponentState(f,f,f,f,t)
    return;
```
Spanning Trees

Cycle is a path in a graph that begins and ends at the same vertex.

A connected graph must have at least number of vertex $- 1$ edges

$$n_E \geq (n_V - 1)$$

if $n_E = n_V - 1$ and connected, there are no cycles

A graph with $n_E > (n_V - 1)$ has at least one cycle

A connected graph with no cycles is a tree.

A spanning tree of a connected graph (G) is a subgraph of G containing all of G's $n_V$ and is a tree.
DFS and BFS search algorithms can be adapted to remove cycle edges to make spanning trees.

```java
dfsTree (Vertex v) {
    v.mark()
    for (Vertex av in v.adjacent)
        if( ! v.marked() ) {
            markEdge(v, av)
            dfsTree(av)
        }
}
```
Minimum Spanning Tree

Assume graph G with weighted (labeled) edges (wEᵢ). weights are costs for "action between vertices"

If graph is a spanning tree, cost of tree = \( \sum wEᵢ \) for \( nV - 1 \) edges

Minimum spanning tree of G has the lowest cost.

Graph \( g, mst \)  // \( g \) exists w/ \( nV \), \( mst \) is empty
  // Vertex v in in Graph \( g \)
PrimsMinimumSpanningTree ( Graph \( g \), Vertex \( v \)) {
  \( v \).mark()
  mst.add(\( v \))  // add vertex w/o connecting edge
  while ( \( g \).containsUnMarkedVertices() ) {
    Vertex nextV =
    \( g \).minCostUnMarkedVertexFromMarkedVertex()
    nextV.mark()
    mst.add(nextV, \( v \))  // add w/ edge(nextV, v)
  }
}
Euler developed graphs studying the Seven Bridges of Königsberg problem (now Kaliningrad). Is there a path to completely cross all 7 bridges once that connect 4 land masses?

An Euler circuit exists if a path begins at vertex $V_i$ and passes through every edge once to end at $V_i$.

All $V$s must have a even degree for a Euler circuit to exist
Complexity – V(G)

Psychological complexity of software design is a function of the paths (circuits) through a program (and other metrics).

Program represented as a graph if directed edges -- cyclomatic number estimates complexity:

\[ V(G) = nE - nV + 2 \text{ ( # connected components)} \]

V(G) is approximated well by the number of logical predicates (boolean expression evaluation / "ifs" – tests) in a program.

Program path changes only on a logical predicate.

V(G) measures modularity: 3 – 7 good design + 10 needs work ...

V(G) correlates highly with: LOC, other software metrics (Halstead, ...), programmer's rating of code.
Navigation graphs (NG)

Information about a traversable paths in a scene can be stored as navigation nodes, way points, path ...

- nodes have adjacency and "cost" to traverse terrain

Nodes are in a connected graph

NGs based GPS "apps"

- paths for maps, business logistics, cars, smartphone / tablets apps

NGs in Games

- reduce collision tests, objects constrained to paths can't collide
- limited visibility – can "see" n nodes from current node.

Movable objects translation (and orientation) is interpolated along, or constrained by path.

Nodes can be placed by designer, equally spaced from a terrain grid, or by other space partitioning technique (Quad tree).
Navigation graph

Adjacency Matrix bitmap

Navigation nodes with at most 8 adjacent nodes for traversal.

625 possible nodes, 553 in terrain, edges = 0.005% = 1828 / 625^2

Adjacency matrix is very sparse – matrix 99.005% empty
Path Finding

Assume a NG graph for movement in a scene. Characters move from node to node. Path = (node_i)

Each node has a location, cost (distance values), collection of its possible 8 adjacent nodes and a reference to it's pathPredecessor.

For example vertical and horizontal moves to adjacent nodes in a "regular geometry NG" can have a cost, or score, of say 10 and diagonal of 14 (sqrt(2*10^2)). Cost is the distance to travel.

Given a source and goal node,

the cost from the source to a current node is known.

the cost to the goal from the source or current node is not known.
Dijkstra Vs A*

Dijkstra’s and A* are “greedy” algorithms
- make the locally optimal choice at each stage (don’t backtrack)
- not symmetrical $Distance_{ij} \neq Distance_{ji}$

Dijkstra’s finds the shorter path than A*
- node’s cost = distanceToSource

A* finds a short path faster than Dijkstra’s
- node's cost = distanceToSource + distanceToGoal (heuristic)

Heuristic, or “best guess” is an approximation applicable to the problem

A* uses Euclidean distance as a heuristic to estimate the cost to the goal from any “current” node.
Path algorithm

Collection open, closed // open set PQ ordered on min cost
// nodes: current, path predecessor, goal
Node current = startNode, goal = stopNode
calculate and set current's cost
current's predecessor = null
open += current // open can't be empty
while (open ! empty)
  current = open's lowest cost node // get next path node
  if current == goal then path complete, done
  else // ! done. keep looking for goal
    closed += current // do not use again
    foreach of current's adjacent nodes
      if (adjacent node !in open && !in closed)
        calculate and set adjacent node's cost
        adjacent node's predecessor = current
        open += adjacent node
  if (current == goal)
    path += traversal back on path predecessors
  else path does not exist
example path (take home)

Find path from "i" to "h" using Dijkstra and A*
Bold numbers in adjacency matrix are actual distance values (labels)
Italic values in adjacency matrix are "hueristic" distance values
I used a ruler to determine heuristic distance values ...
Dijkstra’s Vs A* path finding

Dijkstra path found  
size = 20  
distance = 514.984  
max open = 48  
max closed = 467  
time = 116.829 milliseconds

A Star path found  
size = 20  
distance = 516.129  
max open = 47  
max closed = 123  
time = 70.219 milliseconds
Asymmetrical Paths

Path source to destination:
Dijkstra path found size =  20  distance = 514.588
max open =  30  max closed = 231 time = 57.129 milliseconds

Path destination to source:
Dijkstra path found size =  20  distance = 514.984
max open =  48  max closed = 467 time = 130.181 milliseconds

Path source to destination:
A Star  path found size =  20  distance = 535.440
max open =  27  max closed = 110 time = 31.989 milliseconds

Path destination to source:
A Star  path found size =  20  distance = 516.129
max open =  47  max closed = 123 time = 62.317 milliseconds
Dijkstra Vs A*

<table>
<thead>
<tr>
<th></th>
<th>size</th>
<th>distance</th>
<th>time (msec)</th>
<th>open</th>
<th>closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>26.75</td>
<td>690.48</td>
<td>46.49</td>
<td>37.00</td>
<td>448.13</td>
</tr>
<tr>
<td>A*</td>
<td>27.13</td>
<td>701.93</td>
<td>24.47</td>
<td>60.13</td>
<td>195.13</td>
</tr>
</tbody>
</table>

A terrain's 8 “edge” paths: upperLeft to lowerRight corner, ML to MR
Dijkstra’s ≈ 2 % shorter and ≈ 90% slower (different waypoints than P2)
Search & paths

Validate waypoint.txt data file.
For each node in graph w/ map search for map with:

<table>
<thead>
<tr>
<th></th>
<th>DFS</th>
<th>BFS</th>
<th>Dijkstra</th>
<th>A*</th>
</tr>
</thead>
<tbody>
<tr>
<td>marked nodes</td>
<td>277</td>
<td>130</td>
<td>132</td>
<td>49</td>
</tr>
<tr>
<td>used nodes</td>
<td>114</td>
<td>111</td>
<td>113</td>
<td>35</td>
</tr>
<tr>
<td>path length</td>
<td>115</td>
<td>11.06</td>
<td>11.06</td>
<td>11.10</td>
</tr>
<tr>
<td>time (msec.)</td>
<td>118</td>
<td>522</td>
<td>1547</td>
<td>435</td>
</tr>
</tbody>
</table>

What conclusions can be drawn?
What constraints should be acknowledged about this data?
Other sources / references

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

Spanning tree

Animation of Prim’s spanning tree algorithm
http://students.ceid.upatras.gr/~papagel/project/prim.htm

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

GraphStream  A Dynamic Graph Library (in java)
http://graphstream-project.org/

JUNG  Java Universal Network/Graph Framework
http://jung.sourceforge.net/