Project 3 Report: Graph Modeling and Graph Algorithms by Vaddanak Seng 4/26/2016

The graph is made of a set of vertices and a set of edges. **Figure 1** shows an example of a graph input. It is a 3 by 3 matrix of values. The set of 9 cells represents the set of vertices (Jim's trampolines). The numbers represent the distance of a horizontal and/or vertical edge (Jim's moves). The corresponding story for this project is that the cells are trampolines and the edges are paths to the next trampolines and that Jim starts at vertex 0 (first index of array with 9 elements) and needs to find a path to a destination vertex (vertex 8 in **figure 1**). **Figure 2** shows the graph construction that includes the vertices and directed edges with weight. The edges for each vertex can be drawn only horizontally and vertically, and also an edge is only possible if the distance (or number of moves) does not go outside of the matrix, that is, an edge should connect with another vertex to be a valid edge. The result is a finite graph because there are a finite number of vertices. It is a simple graph because there is 0 or 1 edge between vertices and no self-loop. It is a connected graph because all vertices are connected. It is a directed graph because a correct path always starts at vertex 0 and the next move must progress by following the edge in a leftward/rightward horizontal direction or in a upward/downward vertical direction to the next vertex and cannot backtrack along the same edge. The moves are attempted by following a specific direction as indicated by the arrow head and the labeled weight of each edge for the purpose to eventually find the goal vertex. The graph is weighted because each edge has a distance value. So, the graph that is used to model the problem input is finite, simple, connected, directed, and weighted.

Figure 1. Sample graph input data with 3 x 3 vertices located in input.txt file.

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

Figure 2. Directed graph shows vertex ids and weighted edges based on sample input data.

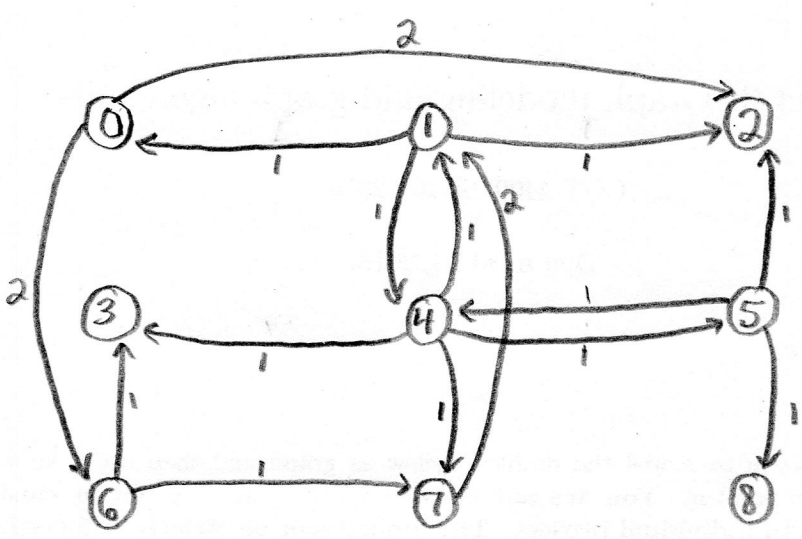

Figure 3. Dijkstra's algorithm used to solve the problem.

```
Input:  V – set of vertices
Input:  E – set of edges
Input:  s – start vertex
Output: S – shortest path

For each vertex v in V:
        set parent to null
        set distance to infinity
Set distance[s] to 0
Add all vertices to priority queue Q
Create queue S to hold vertices in shortest path

While Q is not empty:
        v ← extract vertex with minimum distance from Q
        add v to queue S
        For each vertex u adjacent to vertex v:
                If distance[u] > distance[v] + edge_weight(u,v):
                        distance[u] ← distance[v] + edge_weight(u,v)
                        parent[u] ← v
```

When the input data is loaded (**figure 1**), the elements in the **3 x 3** matrix are stored in a single dimensional array. The position of the elements corresponds to the index of the array. For example, in **figure 1**, the element in the first position of the matrix is stored at index 0 of the array and the last element in the matrix is stored at index 8. This allows for constant time lookup. Also, the index of the array corresponds to the vertex identifier (its id or label or name).

Next, the adjacency list is created to model the graph. The adjacency list is a vector container that holds vertices corresponding to the elements in the input matrix, and also, it holds extra vertices that are used for diagonal moves. (The latter extra vertices are for personal experimentation and so will not be alluded further.) The index of the vector also corresponds to the vertex identifier for constant time lookup. Each node in the vector is considered a **root** node because any other vertices that are adjacent to it will result in a new node that is linked to the root node. Thus, each vertex in the vector is the head node and its neighbors are linked to it in a linked-list data structure. For each root node, we look up the distance value in the input matrix and store it. For each root node, we scan the input matrix (using proper offset as dictated by the distance value) to find its adjacent vertices along horizontal and vertical directions, which gets added as new links where the root vertex/node is the head of the linked-list data structure.

A different vector container was used to create a set of vertices. The index of the vector corresponds to the position in the input matrix and id of the vertex that is stored at that index location. For convenience, the data structure that represents and stores state information for the vertex (**class Node**) also stores the information of its neighbors as a set of pointers. The neighbors were obtained by lookup in the adjacency list.

Another vector container was used to create a set of edges. An edge is represented by **class Edge**. Since this is a directed graph, the data structure for an edge contains information on the source vertex and the destination vertex in addition to the weight (distance) of the edge. The information for an edge was obtained by lookup in the adjacency list.

Once we have the set of vertices V and the set of edges E, we can execute the Dijkstra algorithm

(**figure 3**) on G(V,E). The vertices are initialized with distance set to infinity and parent set to null. The starting vertex is a given input and its distance is set to 0. All the vertices are added to a priority queue Q that orders the vertices based on its distance. A different and empty queue S (not ordered) is created that will hold the vertices representing the shortest paths. Each element in Q is examined until Q is empty, at which point, we will have the shortest path to each vertex in the graph. For each iteration of **while** loop, the vertex v with the smallest distance in Q is removed. Vertex v is added to queue S. We will then visit each neighbor of v. If a neighbor u has a distance greater than the distance of v plus the edge weight from v to u, then we set the distance of u to the sum of distance of v and edge weight from v to u. Also, we update the parent of u to vertex v. That completes one iteration. We continue each iteration in the same way until Q is empty.

Figure 2 shows the possible moves that Jim can make. The content of queue S for horizontal and vertical moves is: **0 6 2 3 7 1 4 5 8** . This set of vertices represents the shortest path to each vertex. Vertex 8 is Jim's goal. Some moves reach a dead end. For example, **figure 2** shows that starting at 0, we can move to 6, and then move to 3, from which there is no next more, ie a dead end. To determine the shortest path for Jim when he starts at vertex 0 and wants to reach vertex 8, the back-link, as provided by the parent stored for each vertex, is used. So, we need to start with vertex 8 and look up the parent for this vertex. Vertex 8's data structure shows a parent data member with value of 5. We go to vertex 5 and lookup its parent, which is 4. We repeat the process and the sequence of lookup results in **8 → 5 → 4 → 1 → 7 → 6 → 0** . The parent of vertex 0 is null, which is the sentinel that is used to determine that it is the starting vertex. Thus, Jim's shortest path starting at vertex 0 and getting to vertex 8 is the reverse, that is, **0 → 6 → 7 → 1 → 4 → 5 → 8** .