Project Write-Up

MET CS 526 - Data Structures and Algorithms

Nathanael Thompson

# Pseudocode

## Algorithm 1

*Input:* an start graph node and a destination graph node, which will always be the node with the lowest direct distance (dd(0))

*Output:* Sequence of all traveled nodes, shortest path and sum of edges along path

Conduct a depth first traverse by calling ddTraverse, keeping useWeight to false to indicate dd(v) priority

// ---- ddTraverse ---- //

*Input*: a map graph, a start node, a positional list, a map, an end node and a true / false indicator

*Output:* a depth first traversal of the map graph stored in a map priorities stored in a priority heap

Store start node in positional list

**If** start node (u) equals the end node

Exit function by printing blank line

**Else if** (u) has one incoming edge

Recursively call ddTraverse with the one incoming edge as the start node

**Else**

**For** edge outgoing from (u)

Set empty vertex equal to vertex opposite edge (v)

**If** priority is w(u,v) + dd(v) (useWeight set to true)

Add w(u,v) + dd(v) and vertex element to heap

**Else** (useWeight set to false)

Add dd(v) and vertex element to heap … **END IF**

**If** (v) not in positional list

Add (v) to map … **END IF** … **END FOR LOOP**

Set new vertex to minimum vertex in heap

**If** minimum vertex not in positional list

Recursively call ddTraverse with minimum vertex as start node

**Else**

For edge outgoing from (u)

Set empty vertex equal to vertex opposite edge (v)

If vertex not in positional list

Recursively call ddTraverse with undiscovered node as start node … **END IF**

//--- end ddTraverse---//

Store the call to shortestPath into new positional list of vertexes

// --- shortestPath --- //

*Input:* an adjacency map graph, a start node, end node and map populated from DFS traversal

*Output:* positional list of the path from start to end nodes and sum of edges

Instantiate a new positional list

**If** edge of destination vertex (v) is not null

Set (v) equal to an empty vertex … **END IF**

**While** new vertex is not the origin vertex (u)

Set new edge object equal to new vertex edge

Add edge to positional list

Increment path length global variable by edge amount

Set new vertex equal to node opposite the followed edge … **END LOOP**

Return positional list

// --- end shortestPath --- //

Call the summary output function

// --- summaryOutput --- //

*Input:* starting vertex and positional list containing output from shortestPath call

*Output:* sequence of nodes traversed to the destination, the shortest path, and shortest path length all displayed on the consul.

Print “Sequence of all nodes:”

**For** vertex in global path storing all nodes

Print vertex + comma

**If** vertex equals the destination

Break loop … **END IF** … **END FOR LOOP**

Print “Shortest path:” + start node

**For** edge in input list containing shortest path node sequence

Print “ -> “ + edge element … **END FOR LOOP**

Print “Shortest path length: “ + global variable shortestPathLength

// --- end summaryOutput ---//

End of Algorithm 1

## Algorithm 2

Algorithm 2 is nearly identical to algorithm 1. What distinguishes the two algorithms is the true or false indicator when calling the ddTraversemethod. If true, then the ddTraverse will traverse based on edge weight + direct distance. If false, the method will prioritize based on direct distance only. The outputs are of course are different on some occassions, but the implementation is the same.

# Data Structure Descriptions

**Adjacency Map Graph:** This is an implementation of the Graph ADT with the primary distinction that outgoing edges from each vertex are stored in a map. Vertex objects storing a string element are placed into a positional list, while edge objects are stored in maps as keys to corresponding vertex keys. The primary vertex list reference corresponds to a secondary map, where the keys are the vertices adjacent to the vertex in the list and the values are the edges connecting the vertices. When the insertEdge function is called, a map object is created and subsequent insertions add to the vertices’ corresponding map objects. Subsequent end vertexes are stored as keys and edges are stored as values, while the map belongs as a reference to that vertex. The maps themselves are implemented as ProbeHashMaps, and are therefore unsorted.

**Map:** The implementations are ProbeHashMaps, which are unordered but a table of map entries allows for efficient search and storage. A single value is stored for each key, and an index of “buckets” (index returned by hash function) is maintained. Searching for a key occurs until found or a null value in table is returned. There are two maps in this implementation: the first storing only string and integer key / value pairs, and the other a vertex and edge key / value pair. This allowed me to have one map function store graph items, while the other could store the direct distance items, both for O(1) expected access later on.

**Heap Adaptable Priority Queue:** The heap priority queue maintains the laws of a binary tree with inputs formatted like a priority queue. The root of the heap is the highest priority (lowest value) object and lower priority heap objects are unordered as children. Heap priority queues are filled from the base then rearranged accordingly. Higher priority items are removed from the top or move up the heap after insertion. This data structure was particularly ideal for storing the minimum vertexes, since the higher priority items were the shorter distance vertices, and should be accessed first.

**Positional List:** The positional list was necessary for this implementation because it maintained the order in which the items were stored, allowing for sequential access of the path later on. A positional list is a modified implementation of a singly linked list, allowing for O(1) insertion of items by utilizing a positional entry reference. Searching a positional list is the same as other linked list implementations, in that it must be done iteratively.

# Appendix – Project.java file

/\*  
 MET CS 526 Data Structures and Algorithms  
 Project  
 Nathanael Thompson  
 \*/  
  
import java.util.\*;  
import java.io.\*;  
  
public class Project {  
  
 // constructors  
 private static AdjacencyMapGraph<String, Integer> *mapGraph* = new AdjacencyMapGraph<>(false);  
 private static Map<String, Integer> *directDistances* = new ProbeHashMap<>();  
 private static Map<Vertex<String>, Edge<Integer>> *forest* = new ProbeHashMap<>();  
 private static AdaptablePriorityQueue<Integer, Vertex<String>> *vertices* = new HeapAdaptablePriorityQueue<>();  
 private static PositionalList<Vertex<String>> *path* = new LinkedPositionalList<>();  
  
 // instance variables  
 final static String *matrixFile* = "src/graph\_input.txt"; // matrix file  
 final static String *distanceFile* = "src/direct\_distance.txt"; // direct distance file  
 static Integer *shortestPathLength* = 0; // inter to store total path length  
 static String *minDD*; // vertex reference for lowest direct distance  
 static String *AFFIRM* = "Y"; // continue variable  
  
 public static void main(String[] args) {  
  
 Project project = new Project();  
 try (Scanner input = new Scanner(System.*in*)) {  
  
 project.readMatrix(*matrixFile*);  
 *minDD* = project.readWeightsFromFile(*distanceFile*);  
  
 // Show graph connections  
 // System.out.println(mapGraph.toString());  
  
 Vertex<String> dest = *getVertex*(*minDD*); // set destination to minimum direct distance  
 boolean isComplete = false;  
  
 while (!isComplete) {  
 System.*out*.println("Enter a starting point: ");  
 String userInput = input.next();  
  
 if (*isVertex*(userInput)) {  
 Vertex<String> origin = *getVertex*(userInput);  
  
 // reset structures  
 *forest* = new ProbeHashMap<>();  
 *vertices* = new HeapAdaptablePriorityQueue<>();  
 *path* = new LinkedPositionalList<>();  
 *shortestPathLength* = 0;  
  
 *Algorithm1*(origin, dest);  
  
 // reset structures  
 *forest* = new ProbeHashMap<>();  
 *vertices* = new HeapAdaptablePriorityQueue<>();  
 *path* = new LinkedPositionalList<>();  
 *shortestPathLength* = 0;  
  
 *Algorithm2*(origin, dest);  
  
 } else {  
 System.*out*.println("Invalid node, would you like to exit? (Y/N)");  
 String newInput = input.next();  
  
 if (newInput.equals(*AFFIRM*)) { // user does not wish to continue  
 System.*out*.println("Goodbye");  
 isComplete = true;  
  
 } else {  
 System.*out*.println("Okay, here are the options to choose from: ");  
 for (Vertex<String> item : *mapGraph*.vertices()) // display vertices to choose from  
 System.*out*.print(item.getElement() + " ");  
 System.*out*.println();  
 }  
 }  
 }  
 } catch (IOException e) {  
 System.*out*.println("File read error");  
 }  
 }  
  
 */\*\*  
 \* Input: an start graph node and a destination graph node, which will always be the node with the  
 \* lowest direct distance (dd(0))  
 \* Output: Sequence of all traveled nodes, shortest path and sum of edges along path  
 \*  
 \* Algorithm 1 prioritizes the next node in the path by direct distance dd(v)  
 \** ***@param*** *origin start vertex on the graph (user input)  
 \** ***@param*** *dest end destination vertex on graph (dd(0))  
 \*/* private static void Algorithm1(Vertex<String> origin, Vertex<String> dest) {  
  
 *ddTraverse*(*mapGraph*, origin, *path*, *forest*, *minDD*, false);  
  
 PositionalList<Vertex<String>> alg1path = *shortestPath*(*mapGraph*, origin, dest, *forest*);  
  
 System.*out*.println("Algorithm 1");  
 System.*out*.println();  
 *summaryOutput*(origin, alg1path);  
 }  
  
 */\*\*  
 \* Input: an start graph node and a destination graph node, which will always be the node with the  
 \* lowest direct distance (dd(0))  
 \* Output: Sequence of all traveled nodes, shortest path and sum of edges along path  
 \*  
 \* Algorithm 2 prioritizes the next node in the path by weight plus direct distance w(u,v) + dd(v)  
 \** ***@param*** *origin start vertex on the graph (user input)  
 \** ***@param*** *dest end destination vertex on graph (dd(0))  
 \*/* private static void Algorithm2(Vertex<String> origin, Vertex<String> dest) {  
  
 *ddTraverse*(*mapGraph*, origin, *path*, *forest*, *minDD*, true);  
  
 PositionalList<Vertex<String>> alg2path = *shortestPath*(*mapGraph*, origin, dest, *forest*);  
  
 System.*out*.println("Algorithm 2");  
 System.*out*.println();  
 *summaryOutput*(origin, alg2path);  
 }  
  
 */\*\*  
 \* displays the sequence of notes traversed to the destination, the shortest path, and the  
 \* shortest path length which is the sum of edges along the path  
 \** ***@param*** *origin start vertex on the graph (user input)  
 \** ***@param*** *list positional list containing the shortest path sequence of nodes  
 \*/* private static void summaryOutput(Vertex<String> origin, PositionalList<Vertex<String>> list) {  
  
 System.*out*.print("\tSequence of all nodes: ");  
 for (Vertex<String> vertex : *path*) { // this list contains all nodes visited  
 System.*out*.print(vertex.getElement() + ", ");  
 if (vertex.getElement().equals(*minDD*)) // stops output when destination is reached  
 break;  
 }  
  
 System.*out*.println();  
 System.*out*.print("\tShortest path: " + origin.getElement());  
 for (Vertex<String> edge : list) { // this list contains the shortest path node sequence  
 System.*out*.print(" -> " + edge.getElement());  
 }  
  
 System.*out*.println();  
 System.*out*.println("\tShortest path length: " + *shortestPathLength*); // sum of edges along shortest path  
 }  
  
 */\*\*  
 \* Input: starts with a graph, starting node, and an initially empty positional list, map, end node,  
 \* and boolean confirming if which algorithm is given priority  
 \* Output: a positional list and map with all visited nodes, and a priority heap containing the  
 \* priority of the node based on the algorithm choice (weight + dd or just dd)  
 \*  
 \* This method traverses the graph using a depth first approach. If useWeight is true, the traverse  
 \* will prioritize weight + dd for moving to the next node, or else just the dd is considered.  
 \* Adapted from the DFS function in the textbook.  
 \** ***@param*** *graph a Adjacency Map Graph  
 \** ***@param*** *origin starting node (user input)  
 \** ***@param*** *known a positional list containing all visited nodes  
 \** ***@param*** *forest a map populated with vertex, edge key, value pairs  
 \** ***@param*** *end destination node  
 \** ***@param*** *useWeight true if weight + dd is priority and false if dd is priority  
 \*/* static void ddTraverse(Graph<String, Integer> graph, Vertex<String> origin,  
 PositionalList<Vertex<String>> known, Map<Vertex<String>, Edge<Integer>> forest, String end,  
 boolean useWeight) {  
 known.addLast(origin);  
  
 // check if destination is reached. If so, exit.  
 if (origin.getElement().equals(end)) {  
 System.*out*.println();  
  
 // option to recursively backtrack out of hole if dead end is reached  
 } else if (graph.inDegree(origin) == 1) {  
 *ddTraverse*(graph, graph.opposite(origin, graph.incomingEdges(origin).iterator().next()),  
 known, forest, end, useWeight);  
  
 // iterate through outgoing edges  
 } else {  
 for (Edge<Integer> edge : graph.outgoingEdges(origin)) {  
 Vertex<String> dest = graph.opposite(origin, edge);  
  
 // w(u,v) + dd(v) priority - insert node with priority into heap  
 if (useWeight) {  
 *vertices*.insert(*directDistances*.get(dest.getElement()) + edge.getElement(), dest);  
  
 // dd(v) priority - insert node with priority into heap  
 } else {  
 *vertices*.insert(*directDistances*.get(dest.getElement()), dest);  
 }  
  
 // populate map to help with shortest path traverse function  
 if (!*contains*(known, dest))  
 forest.put(dest, edge);  
 }  
 Vertex<String> minVertex = *vertices*.min().getValue(); // get minimum vertex from heap  
  
 // min vertex in heap is unvisited node so begin from new node  
 if (!*contains*(known, minVertex)) {  
 *ddTraverse*(graph, minVertex, known, forest, end, useWeight);  
 }  
 // fail safe to prevent infinite recursion between a 'valley' of minimum nodes  
 // looks for the next unknown node regardless of priority  
 else {  
 for (Edge<Integer> edge : graph.outgoingEdges(origin)) {  
 Vertex<String> dest = graph.opposite(origin, edge);  
 if (!*contains*(known, dest))  
 *ddTraverse*(graph, dest, known, forest, end, useWeight);  
 }  
 }  
 }  
 }  
  
 */\*\*  
 \* helper method to find vertex in positional list  
 \** ***@param*** *list positional list  
 \** ***@param*** *vertex vertex to search for  
 \** ***@return*** *true if vertex is in list, false otherwise  
 \*/* static boolean contains(PositionalList<Vertex<String>> list, Vertex<String> vertex) {  
 for (Vertex<String> vert : list) {  
 if (vert.equals(vertex))  
 return true;  
 }  
 return false;  
 }  
  
 */\*\*  
 \* Input: an adjacency map graph, a start node, end node and map populated from DFS traversal  
 \* Output: positional list of the path from start to end nodes.  
 \*  
 \* This method finds the shortest path from selected start node to end node, and calculates  
 \* the total path length. if destination is unreachable from the origin, or if only one  
 \* node is present, then an empty path is returned. May not always return the shortest path  
 \* Adapted from the textbook code constructPath  
 \* function  
 \** ***@param*** *graph an adjacency map graph  
 \** ***@param*** *origin start node  
 \** ***@param*** *dest end node  
 \** ***@param*** *forest map populated from DFS traversal where the origins were the same  
 \** ***@return*** *a positional list with the path from start to end nodes  
 \*/* private static PositionalList<Vertex<String>> shortestPath(Graph<String, Integer> graph,  
 Vertex<String> origin,  
 Vertex<String> dest,  
 Map<Vertex<String>, Edge<Integer>> forest) {  
 PositionalList<Vertex<String>> path = new LinkedPositionalList<>();  
 if (forest.get(dest) != null) {  
 Vertex<String> walk = dest;  
  
 // iteratively traverse the map until the origin is reached  
 while (walk != origin) {  
 Edge<Integer> edge = forest.get(walk);  
 path.addFirst(walk);  
 *shortestPathLength* += edge.getElement(); // increment the path length total  
 walk = graph.opposite(walk, edge); // point to next node  
 }  
 }  
 return path; // returns empty if one node is present or origin is unreachable  
 }  
  
 */\*\*  
 \* Input: a txt file with an n x n matrix containing vertices and edges to store into a map  
 \* Output: an undirected adjacency map graph with string vertices and integer edges  
 \*  
 \* reads data from matrixFile; stores vertices and edges into AdjacencyMapGraph  
 \** ***@param*** *filename matrixFile to read  
 \** ***@throws*** *IOException Error if file not found or other file read errors  
 \*/* private void readMatrix(String filename) throws IOException {  
 try (Scanner input = new Scanner (new File(filename))) {  
 int count = 0;  
 int vertIDX = 0;  
 while (input.hasNext()) {  
 Iterator<Vertex<String>> horizontalScan = *mapGraph*.vertices().iterator();  
  
 // read header row of matrix into array then store vertices positional list  
 if (count < 1) {  
 String[] topLine = input.nextLine().split("\\s+|[^A-Z]+"); // spaces and capital letters  
 for (String item : topLine)  
 if (!item.equals(""))  
 *mapGraph*.insertVertex(item);  
 count++;  
  
 // read remaining lines into array then store edges based on vertex reference  
 } else {  
 String[] line = input.nextLine().split("\\s+");  
 Vertex<String> verticalVertex = *getVertex*(line[vertIDX]);  
 for (int idx = 1; idx < line.length; idx++) {  
 Vertex<String> horizontalVertex = horizontalScan.next();  
  
 // check if edge is not zero and does not already exist in the graph  
 if (*toInt*(line[idx]) > 0 && !*edgeExists*(verticalVertex, horizontalVertex))  
 *mapGraph*.insertEdge(verticalVertex, horizontalVertex, *toInt*(line[idx]));  
 }  
 count++;  
 }  
 }  
 } catch (IOException e) {  
 throw new IOException(e);  
 } catch (NumberFormatException ex) {  
 System.*out*.println("Input matrix contains a non-integer edge value");  
 }  
 }  
  
 */\*\*  
 \* Input: a txt file with vertex references in the first column and corresponding edges  
 \* in the second column  
 \* Output: a map containing string representations of vertices and integer representations  
 \* of edges  
 \*  
 \* reads data from distanceFile and stores node and distance data into map directDistances  
 \** ***@param*** *filename name of file to be read from  
 \** ***@throws*** *IOException thrown if there is a file read error  
 \** ***@return*** *a string representation of the minimum direct distance object (dd(0))  
 \*/* private String readWeightsFromFile(String filename) throws IOException {  
 try (Scanner input = new Scanner (new File(filename))) {  
 int nodeIDX = 0, distIDX = 1;  
 while (input.hasNext()) {  
 String[] line = input.nextLine().split("\\s+"); // any spaces  
 try {  
 *directDistances*.put(line[nodeIDX], *toInt*(line[distIDX])); // read data into map  
 } catch (NumberFormatException e) {  
 System.*out*.println(String.*format*("Error at line: %s", line[nodeIDX]));  
 }  
 }  
 } catch (IOException ex) {  
 throw new IOException(ex);  
 }  
 return *getZeroDD*(*directDistances*); // return dd(0) string reference  
 }  
  
 */\*\*  
 \* gets the minimum direct distance from the direct distance map  
 \** ***@param*** *map a map containing the vertexes and their direct distances  
 \** ***@return*** *the minimum direct distance or else an empty string  
 \*/* private static String getZeroDD(Map<String, Integer> map) {  
 try {  
 for (String key : map.keySet()) {  
 if (map.get(key) == 0)  
 return key;  
 }  
 } catch (IllegalArgumentException e) {  
 System.*out*.println("Zero element not found in direct distance list");  
 }  
 return "";  
 }  
  
 */\*\*  
 \* determines if an edge exists between two vertexes in a map graph  
 \** ***@param*** *origin start vertex  
 \** ***@param*** *dest destination vertex  
 \** ***@return*** *true if an edge exists between the two vertexes or false otherwise  
 \*/* private static boolean edgeExists(Vertex<String> origin, Vertex<String> dest) {  
 for (Edge<Integer> edge : *mapGraph*.edges()) {  
 if (*mapGraph*.getEdge(origin, dest) == edge) {  
 return true;  
 }  
 }  
 return false;  
 }  
  
 */\*\*  
 \* converts a string value to an integer  
 \** ***@param*** *string representation of integer  
 \** ***@return*** *integer representation of string input value  
 \** ***@throws*** *NumberFormatException if string cannot be converted to integer  
 \*/* private static Integer toInt(String string) throws NumberFormatException {  
 try {  
 return Integer.*parseInt*(string);  
 } catch (NumberFormatException e) {  
 throw new NumberFormatException();  
 }  
 }  
  
 */\*\*  
 \* iterates through vertices until it finds a vertex matching the string or else  
 \* returns the first item in iterator  
 \** ***@param*** *string string to match vertex  
 \** ***@return*** *vertex matching input string or else the first vertex in the map graph vertices list  
 \*/* private static Vertex<String> getVertex(String string) {  
 Iterable<Vertex<String>> list = *mapGraph*.vertices();  
 for (Vertex<String> item : list)  
 if (item.getElement().equals(string))  
 return item;  
 return list.iterator().next();  
 }  
  
 */\*\*  
 \* determines if the string input is a vertex in the map graph  
 \** ***@param*** *string element to determine if a vertex  
 \** ***@return*** *true if there is a corresponding vertex in the map graph to the string, false otherwise  
 \*/* private static boolean isVertex(String string) {  
 Iterable<Vertex<String>> list = *mapGraph*.vertices();  
 for (Vertex<String> item : list)  
 if (item.getElement().equals(string))  
 return true;  
 return false;  
 }  
}