CS 526 Term Project

JEFFREY DALBY
4-22-2018

Introduction:

In determining my approach to this project I had two goals, one to create as simple of code as possible, and two to utilize the most "object-oriented" approach, since that is a skill I've been focusing on improving throughout the class. Because of this, I've come up with a fairly novel approach to the problem. The goal of the project is to find the path between two nodes using a given heuristic algorithm; however, as one examines the actual algorithm one can see that the method used to find that paths is identical aside from the way that the weighting of the paths is calculated.

I've chosen to accomplish this task by utilizing an Edge List vs a two dimensional array adjacency matrix (which feels more object oriented to me and more in line with the teaching of the class), and to pre-sort the edges for each node using a Priority Queue which natively is based on a min-heap. This allows for a very fast method of meeting all the objectives by splitting the task of determining the priority based on the heuristics and path finding into separate items.

Primary Data Structures:

GraphReader: The graph reader class handles all the work of reading in both the direct distance file, and the graph file and translating that into an edge list. The edge list is implemented as as ArrayList created at the time we read in the direct distance file. It then loops through this arraylist utilizing the graph file to read and assign edges to each vertex within the edge list. Because the edges are stored in a priority queue they are automatically sorted into the proper order at the time the list is created, saving us from having to determine the proper selection during the search. It has the following components:

- vertices- an arraylist to contain the edgelist as it is created.
- createEdgeList- main method to call to create the edge list
- readDistanceFile- method to parse through the distance file line by line and create our initial list of all possible vertices for the given file. This allows it to handle any size list of objects, with any string as the name of the vertex.
- readGraphFile- this method is dependent on the distance file being read first, and adds each edge as it is found, along with the edge weight, to each vertex that was created when reading the distance file.

Vertex: Each vertex in the edge list is stored as a vertex object, which has four major pieces.

- data- A string to store the name of the node
- visited- A Boolean to keep track of the visited status during our search
- directDistance- An int to store the direct distance as read from the direct distance file
- neighborList- The workhorse of the algorithm to choose the shortest path to the next edge, this is a priorityqueue which adds edges in an ascending order based on a comparator on the edge object.

Edge: The edge object implements Comparable which allows us to have a means of determining the sort order for the neighbor list. Edges have the following fields and method:

- vertex- a pointer to the vertex this edge connects to
- edgeWeight- the weight of this edge as read from the graph file
- compareTo- overridden method which compares the directDistance of the vertices of each edge when determining the location to insert the edge into the priority queue, per algorithm 1: "Among all nodes v that are adjacent to the node n, choose the one with the smallest dd(v)."

AltWeightedEdge: An extension of the Edge object that allows us to use a different weighting to determine the placement in the neighborlist. It consists of one override:

- compareTo- implements the comparison used in algorithm 2:
  "Among all nodes v that are adjacent to the node n, choose the one for which w(n, v) + dd(v) is the smallest."

Algorithm- Because we have presorted the neighbor list at the time of edge list creation the algorithm is just a modified recursive backtracking algorithm that finds a path between two points. It loops through the neighbor list in order (which has been

presorted from lowest priority to highest thus satisfying the constraints of the algorithm). We simply have to pass it the appropriately sorted list and it will find the correct path based on the heuristic sort. Algorithm has three fields and two methods:

- walkedPath- this stores a list of each node as we determine if we need to walk that path in order to get to the end node. It contains each element as it is backtracked, or when it is checked to see if we have visited it in the past.
- foundPath- contains the path that was found (in reverse order due to the recursive nature. We reverse this when displaying it).
- shortestPathLength- an int to track the length of the shortest path.
- findShortestPath- calls findPath and prints out the results
- findPath- recursive backtracking algorithm to find the path between nodes. See the pseudocode later in this document for more information.

Algorithm pseudo code: The pseudo code has two phases, one which does the sort at the time of edge list creation and one that finds the path.

- EdgeList creation-

  readDistanceFile-
  open direct distance file
  loop through each line
  split string into two parts separated by spaces
  create new vertex named after string in part 0, with direct distance of int in part 1
  add new vertex to edgelist
  close file

  readGraphFile-
  open graph file
  skip line one which contains labels
  for each vertex in the edgelist
  read line in graph file
  string line = split based on spacing
  for i = 1; i < line.length; i ++
  if integer value of line[i] > 0
  create new edge object with edge weight set to value
  add edge to vertex neighborlist (uses the edges comparable base on algorithm)
  close file;

- Recursive Path finding Algorithm:
  boolean findPath(starting vertex, ending vertex)

  if startpoint = end point return true since we found the end (this is the base case for recursion)
  loop through each workingEdge in the starting vertex's neighborlist (remember this is already in the sorted order of the algorithm)
  add workingEedge.vertex to list of vertex's that have been checked
  if workingEdge has not been visited
  set workingEdge visited to true
  if findPath(workingEdge, endpoint) – recursively call findPath to walk down this node looking for end
  add workingEdge to foundPath list since it can reach the end
  add workingEdge edge weight to shortest path length
  return true (which breaks us out of the recursion with a successfully found path
  return false (breaks this path since it cannot reach the end causing us to walk back up the tree until we loop to the next working edge to see if it has a path)