Game AI - Assignment 2

Team: Absinthe Wu (Mengling) and Benjamin Dunbar

1. Implement Dijkstra algorithm to calculate the shortest distance from any two nodes in a level. Input and output formats should be consistent with PacMan 6.2/data/mazes and PacMan 6.2/data/distances.

We created our dijkstra algorithm in a new class under pathFinding/Dijkstra.java. Our class offers the ability to take a file path and name, like data/mazes/b, and will create an output file at the location distances_for_data_mazes_b. The input for the maze to analyze is taken via a console input when running the game. In order to run with this configuration, simply uncomment the line below "//Uncomment to run dijkstra.". When the executor main function is now run, the program will ask for input in the console. The easiest way to test a file would be to place it in data/mazes and then give the name data/mazes/yourFileName. The .txt extension is not needed.

In the Dijkstra algorithm itself, we created a array of ints based on the size of the maze node list. After we read from the file into a Vertex class which we created (similar to Node but with all of the data we needed and no extra data), we loop through each vertex and then run a dijkstra search over the whole graph from that vertex. After the dijkstra search finishes it will return a list of Vertices which all have their distance and previous values filled in. At that point the array of ints is filled in with values. When all of the values are written we created an output file as described above.

The dijkstra search which we created follows the pseudocode given in the lecture. We take the List<Vertex> in the file and set all of the distances to infinity and the previous Vertex to null. The start node is set to have a distance of 0 and all nodes are added to a sorted Q. Then we run a while loop which has a condition of the queue not being empty. We will take and visit the node in the queue with the lowest distance and then analyze its neighbors to determine if a better distance is found for any neighbors. If a better distance is found then the neighbors distance and previous fields are updated. When the Q list is empty, then all best values have been found and it returns a list of processed nodes. If a route to a node was not found the distance is set to -1.

Our testing for the dijkstra file tests with unit tests and functionality tests. We were able to match all 4 of the provided distance files in 1 minute and 7 seconds with our approach. Our unit tests will test the logic of the dijkstra algorithm itself, by testing a small graph we created. We test a short node to node route on this graph and we also check that when no route is found the distance given is -1. Our testing suite can be found at: test/pacman/pathFinding/DijkstraTest.java.

2. Implement A to let ghosts decide moves when chasing Ms. PacMan. Your code should not depend on any pre-computed results. Instead, A* should run efficiently in real-time. Describe your implemented heuristics for calculating.*

We created our A* algorithm within the MyGhost class. Each time getMove is called, our code will loop through each of the ghosts. Like the starter code, for each ghost it will only calculate a move if the game states that a move is needed, and a ghost will run away from Ms. Pacman if it is edible or Ms. Pacman is near a power pill. When a move is needed and the ghost is not running away, we implemented an A* approach that will allow the ghosts to efficiently chase Ms. Pacman. Our ghosts will always chose to chase rather than give a chase percentage, this makes it easier to predict moves for game state testing. Our code can be seen at MyGhost.getNextMoveAStar() and MyGhost.aStarAlgorithm(). The getNextMoveAStar function will take the nodes from the graph, the start and end indexes based on ghost and pacman positions and convert these into a List<Vertex> and start and goal Vertex objects. It then calls aStarAlgorithm, which will populate the Vertex.previous fields in order to find a path from the start to the goal. getNextMoveAStar uses these previous vertex fields to loop back through the path and then determine the direction the ghost has to take in order to start following the path.

Our aStarAlgorithm function is the area where A* is actually computed. We created a Vertex class similar to a node which stores an index, x position, y position, distance, heuristic, previous vertex, and neighbor indexes. (We decided to make a new class for this because it suited our needs better than the existing Node class. While it would be possible to extend the node class, we wanted to demonstrate our understanding of the algorithms with our own code.) Our A* works similar to a dijkstra algorithm, but uses a Manhattan Heuristic while calculating an estimated cost for any given node. Our A* algorithm also uses an openList and closedList of vertices. When a node is visited it is removed from the openList and put in the closed list and

then its children are analyzed. For each child, if a better cost is found then the previous field is updated to represent the current node and the better cost is applied. If this happens and the child is not in a list yet, then it is added to the open list. If a better cost is found and the child is in the closed list, then it is put back in the open list. These updates to the open and closed lists are the second change from our dijkstra code. Finally, the open list is sorted based not only on distance, but using the distance to a location AND the heuristic for that location.

We chose a manhattan distance for our heuristic value because the ghosts can not move diagonally, so the delta x distance and delta y distance summed together should give the most simple and accurate estimation of how far a node will be from the goal.

To run our A* implementation on the game, simply uncomment the two lines below the comment in Executor.java which reads “//UNCOMMENT TO RUN A* ghosts”. These should be lines 44 and 45 of the Executor.java file of our submission.

Finally, our test suite for A* runs both unit and functionality tests and can be found in test/pacman/entries/ghosts/MyGhostTest.java. Our functional test applies a game state to a game and then ensures that the getMove algorithm returns the moves we expect.

Pseudocode Used:
Dijkstra (from lecture notes)

```
1  function Dijkstra(Graph, source):
2
3      create vertex set Q
4
5      for each vertex v in Graph:             // Initialization
6          dist[v] ← INFINITY                  // Unknown distance from source to v
7          prev[v] ← UNDEFINED                 // Previous node in optimal path from source
8          add v to Q                          // All nodes initially in Q (unvisited nodes)
9
10     dist[source] ← 0                        // Distance from source to source
11
12     while Q is not empty:
13         u ← vertex in Q with min dist[u]    // Source node will be selected first
14         remove u from Q
15                 Check if the node is the goal, if it is terminate the algorithm
16         for each neighbor v of u:           // where v is still in Q.
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]:               // A shorter path to v has been found
19                 dist[v] ← alt
20                 prev[v] ← u
21                          Also, you need to check if it is in the open list,
22     return dist[], prev[]  and update it if it is.
```

A* (from book pages 220 - 223)

```
def pathfindAStar(graph, start, end, heuristic):

  # This structure is used to keep track of the
  # information we need for each node
  struct NodeRecord:
    node
    connection
    costSoFar
    estimatedTotalCost

  # Initialize the record for the start node
  startRecord = new NodeRecord()
  startRecord.node = start
  startRecord.connection = None
  startRecord.costSoFar = 0
  startRecord.estimatedTotalCost =
    heuristic.estimate(start)

  # Initialize the open and closed lists
  open = PathfindingList()
  open += startRecord
  closed = PathfindingList()

  # Iterate through processing each node
  while length(open) > 0:

    # Find the smallest element in the open list
    # (using the estimatedTotalCost)
    current = open.smallestElement()

    # If it is the goal node, then terminate
    if current.node == goal: break

    # Otherwise get its outgoing connections
```

Page 1

```
    connections = graph.getConnections(current)

    # Loop through each connection in turn
    for connection in connections:

      # Get the cost estimate for the end node
      endNode = connection.getToNode()
      endNodeCost = current.costSoFar +
                    connection.getCost()

      # If the node is closed we may have to
      # skip, or remove it from the closed list.
      if closed.contains(endNode):

        # Here we find the record in the closed list
        # corresponding to the endNode.
        endNodeRecord = closed.find(endNode)

        # If we didn't find a shorter route, skip
        if endNodeRecord.costSoFar <= endNodeCost:
          continue;

        # Otherwise remove it from the closed list
        closed -= endNodeRecord

        # We can use the node's old cost values
        # to calculate its heuristic without calling
        # the possibly expensive heuristic function
        endNodeHeuristic = endNodeRecord.estimatedTotalCost -
                           endNodeRecord.costSoFar

      # Skip if the node is open and we've not
      # found a better route
      else if open.contains(endNode):

        # Here we find the record in the open list
        # corresponding to the endNode.
        endNodeRecord = open.find(endNode)

        # If our route is no better, then skip
        if endNodeRecord.costSoFar <= endNodeCost:
          continue;

        # We can use the node's old cost values
```

Page 2

```
        # to calculate its heuristic without calling
        # the possibly expensive heuristic function
        endNodeHeuristic = endNodeRecord.cost -
                           endNodeRecord.costSoFar

      # Otherwise we know we've got an unvisited
      # node, so make a record for it
      else:
        endNodeRecord = new NodeRecord()
        endNodeRecord.node = endNode

        # We'll need to calculate the heuristic value
        # using the function, since we don't have an
        # existing record to use
        endNodeHeuristic = heuristic.estimate(endNode)

      # We're here if we need to update the node
      # Update the cost, estimate and connection
      endNodeRecord.cost = endNodeCost
      endNodeRecord.connection = connection
      endNodeRecord.estimatedTotalCost =
        endNodeCost + endNodeHeuristic

      # And add it to the open list
      if not open.contains(endNode):
          open += endNodeRecord

    # We've finished looking at the connections for
    # the current node, so add it to the closed list
    # and remove it from the open list
    open -= current
    closed += current

  # We're here if we've either found the goal, or
  # if we've no more nodes to search, find which.
  if current.node != goal:

    # We've run out of nodes without finding the
    # goal, so there's no solution
    return None

  else:

    # Compile the list of connections in the path
```



```
    path = []

    # Work back along the path, accumulating
    # connections
    while current.node != start:
      path += current.connection
      current = current.connection.getFromNode()

    # Reverse the path, and return it
    return reverse(path)
```