Natalia Alonso
COMP 282 Fall 2013
November 11, 2013

Project 2 – Treasure Map

Phase 1 – Graph Generation

All waypoints will be stored in a HashMap. Each waypoint will have an ArrayList called *neighborList* which will store all neighbors of type *Point*. Each waypoint will also have a variable called *neighbors* which will indicated how many neighbors each waypoint has. This will be used to iterate through *neighborList*. An ArrayList is used because it we only need to add few values once and its retrievals are done in constant time.

Phase 2 – Interactive Navigation

The *destination Point* is held in a global variable. Both the start and end points are marked on the map. An PriorityQueue is used for an open *Point* list (*openList*). A HashMap is used for a closed *Point* list (*closedList*). The PriorityQueue will hold Nodes. The PriorityQueue was chosen because it is implemented as a min-heap, so the minimum value will always be at the front of the queue. It compares each Node based on its *distance* value. Each Node has x and y values (*xvalue,yvalue*), a *distance* variable (*distance*), and a back *Point* (*back*) which indicates which *Point* was processed to find the Node. The HashMap will have a *Point* as a key, and it's back *Point* as the value. A HashSet called *HSOpenList* keep a running list of the points in the *openList*.

The start *Point* is the first *Point* to go into the *openList.* A *Node* is created for the *Point* with x and y values, *distance* to the previous node (which is 0 in the case of the starting *Point*) plus the heuristic value (the straight-line *distance* from the *Point* to the *destination*) plus the *distance* from the start to the previous node (which is 0 in the case of the starting *Point*).

The algorithm begins by taking the minimum value in the queue, the front of the queue, and processing it's neighbors. It is then put into the *closedList* once it is processed. A gray maker is added to indicate the *Point* is in the *closedList.* A *Node* is created for each neighbor. If the neighbor is not the *destination*, and is not already in the *openList* or *closedList*, it is placed into the *openList.* This continues for each neighbor in *neighborList.* A white marker is added to indicate it is in the *openList*.

If the neighbor is the *destination*, the algorithm exits the loop. A Stack of Points *pathStack* is created. This stack will hold the path for the *player.* A Stack is chosen because it is a LIFO data structure. The points will be added going back to the start *Point*, and will then be popped in order back to the *destination*. Because the loop exits when a neighbor is the *destination,* the *destination* is the first *Point* to go into *pathStack.* Then the *current Point* is added, as it was the node previous to finding the *destination*. From there, while the *current Point* is not the starting *Point*, each *back Point* is added to the stack. *current* gets *back* value until *current* is *start.* While *pathStack* has Points on the path to the *destination*, *player* is updated to each *Point* in *pathStack.* If the *Point* has gold, a cost, or a map, a message is displayed to the user and status is updated. Once the *pathStack* is empty, the *destination* will have been reach, a message is displayed to the user and status is updated, and the algorithm returns. All lists are cleared.

Phase 3 – Treasure Maps

If the *player* encounters a treasure map on the way to the *destination*, it halts the *current* path finding and makes the location of the treasure the *treasureDestination*. While the *player* is looking for treasure, it ignores all other treasure maps on its way. The *player* finds a path to the *treasureDestination* by the same method as its final *destination*. All lists are cleared and the *start* becomes the *player's current* position. The algorithm determines the path to the *treasureDestination* and the *player* walks to the *Point*. Once the *player* reached the *treasureDestination*, it begins to search for a path to its final *destination*. All lists are cleared, the start becomes the *player*'s *current* position. If the *player* encounters another treasure map it repeats the process of finding a path to the treasure, and then resuming its journey to the final *destination*. I chose to make the *player* extra greedy; if there is a treasure map at the *start* or final *destination*, the *player* will find the path in order to collect it. The *player*'s paths throughout the map remain marked (drawables are not cleared) after finding a treasure map, so the user can see all the places the *player* visits and what routes were taken (shown in red).

Classes

The WayPoint class holds all the values for each WayPoint. Each value is extracted from the file one by one and the constructor uses these values for each of the WayPoint attributes. In addition to setting and getting given attributes, the WayPoint class has functions which change the various WayPoint attributes. The setMapVisit() function sets the “treasure map's” coordinate values to 0 to show that the *player* has already seen the treasure map. The setVisited() function marks the wayPoint as visited and is used to determine whether or not the *destination* has been reached.

The *player* class holds all the attributes for the *player*. Each *player* is initialized with a set amount of *strength* and *wealth*, and its *x*- and *yvalue* are determined by the user. *Strength* and *wealth* can be retrieved and changed using their appropriate functions; get- , set-, and subStrength(), and get-, set-, and subWealth(), respectively. If the *player* is currently searching for treasure, its *treasure* attribute is set using setTreasure(), which toggles the boolean value of *treasure*.

The Node class is for the openList in the A* algorithm. Each Node has an *x*- and *yvalue*, a *distance* value determined during the path finding algorithm, and a *Point back* which is the *Point* the Node was a neighbor to and was processed from. The Nodes are compared based on their *distance* value in the algorithm using the getDistance() function. The function getBack() returns *back;* it is used to create the path.