CS4006 Project Analysis

Description

A* is an optimization search algorithm. It is employed to find the shortest path between a start and end with minimum cost. This program performs A* search on an eight-by-eight grid to find the shortest path from a start point to an end point while avoiding obstacles. It finds the shortest way around any obstacle. The program prints out an 8x8 grid with a random letter object positioned at random inside the grid. These letters are I, T, J and L. The user then enters co-ordinates for the start and end points. You will then be asked if you wish to proceed and find the shortest path to the endpoint around the obstacle. This is achieved with A star algorithm. The path is then displayed once the endpoint is reached and the program is finished.

Point Class

The **data fields** for the Point class are : an integer x for the X coordinate in the grid, an integer y for the Y coordinate in the grid, an integer gn for the steps taken to get to this point, an integer hn which represents the Manhattan distance to a given point, an integer fn for the point (hn plus gn) and finally, a parent Point which will give the program a *linked list* of Point approach where one Point points to its parent Point.

The **constructor** takes an integer X and an integer Y for the Points - X and Y coordinates.

- **Method setParent(Point p)** takes a Point p as a parameter and sets the parent of the point to p.
- **Method getParent()** returns the parent of the point.
- **Method setGn(int gn)** takes an integer gn as a parameter. Sets the gn value of the point to gn.
- **Method setHn(Point p)** sets the hn value by getting the Manhattan distance to the point passed in. The Manhattan distance is calculated by getting the absolute value of the difference in x between this point and p, the absolute value of the difference in y between this point and p and adding these together.
- **Method setFn()** sets the fn of the point by getting the total of gn plus hn.
- Three **methods getGn(), getHn(), getFn(),** return the gn, hn and fn integer values of the point respectively.
- Two **methods setX(int x)** and **setY(int y),** set the X and Y values of the point respectively.
- Two **methods getX()** and **getY(),** return the X and Y values of the point respectively.
- **Method isEqual(Point p),** returns Boolean true if this point's x and y coordinates are the same as the point passed in.
- **Method toString()** returns a string with the points data in a easily read layout.
- **Method compareTo(Point p)** over-rides the Comparable class method compareTo. This method determines how the Collections sort method works. It will sort arraylist of points by fn if two points being compared have the same fn, else it will compare them by their hn values.
- **Method equals(Object o)** over-rides the Object class method. If the object passed in is a Point then the method only returns true if the point passed in has the same coordinates as this point.

If the object isn't a point or if either of the coordinates dont match this points coordinates it will return false.

is18227023 Class

The **data fields** for this class are : a 2D array of type char called mat, a start Point, an end Point, a current Point, a scanner, an ArrayList of type Point called open, an ArrayList of type Point called closed and finally an ArrayList of type Point called neighbours.

The **constructor** initializes the 2D array of type char to a square grid of size eight by eight and the other data fields.

- Our first **method** randomPos() returns a random integer between 0 and 7.
- **Method shapeFits(int tlX, int tlY, int sizeX, int sizeY)** returns true only if tlx (aka "top left x") is greater than 0 so it's not on the edge of the grid and tlx + sizeX is less than 7, this means that if the random position for X plus the width of the shape passed in is less 7 so it's not on the right hand side edge of the grid. Also, this must be true for tlY and sizeY too. This is the same except for going down the board it should be greater than 0 so it's not at the top of the grid edge and less than 7 so it's not at the bottom grid edge.
- **Method fillI()** creates an I letter obstacle in the grid in a random location. Firstly, it creates four integer values. tlX which is called in the **randomPos()** method which generates a random X value anywhere in the grid between 0 and 7. tlY also calls on **randomPos().** sizeX which is equal to 1, this means the width of the shape in the X value going across the grid. sizeY which is equal to 4, this means the height of the shape in the Y value going down the grid. Using the **shapeFits()** method we check if the random X and Y values plus the size of the shape are greater than 0 on the X and Y axis and less than 7 on the X and Y axis, we check if our shapes fits in the grid. If yes, print the Object in its randomly assigned position with '*' for the object. If no, call on the **fillI()** method recursively so it tries inserting again.
- **Method fillT()**, **fillJ()** and **fillL()**, which creates T, J and L objects respectively. The code is the same as the **fillI()** method above.
- **Method isEqual(int x, int y, char value)**, this method checks if the x and y in the 2D grid are equal to a character value. Returns true if they are.
- **Method inGrid(int x, int y)** , returns true if x and y values are greater than or equal to 0 in the XY axis, and if the x and y values are less than or equal to 7 in the XY axis in the grid. Checks if a point is in the grid including the edges.
- **Method setStartAndEndIndexes(int sx, int sy, int ex, int ey),** can throw an exception. Therefore, wherever it is used it must be surrounded by a try catch. This method sets the start and end points. It calls on the **inGrid()** method to check if sx and sy are in the grid between 0 and 7 in the XY axis. If yes, continue otherwise throw Exception "Incorrect index positions". It then checks if the sx and sy are not equal to a * value, ie. checks if coordinates sx and sy do not conflict with coordinates of an obstacle(letter). If there is no conflict, set start equal to a new point with sx and sy as its x and y coordinates. Exception "Shape exists here. Try again" thrown if sx and sy fall on same coordinates as the shape. It then checks the same for ex and ey, if all is

true it creates a temporary point P which is equal to a new point with ex and ey as its x and y coordinates. It then checks that P is not equal to the start, ie. that ex and ey are not the same as sx and sy. If true, then set end to a new point with ex and ey as its x and y coordinates. It then calls on the **setPoints()** method for end and start to set E character and S character respectively in the grid. We then calculate gn, hn and fn for start. We also set the parent for start to start. If coordinates p.x and p.y is equal to sx and sy, throw new Exception “Your start can't be the same as your end”.

- **Method setPoints(Point p, char value)**, puts a character(value) in a point on the grid.
- **Method fillArray(char value)**, this method goes through the 2D grid with two for loops and sets each value to a given character(value).
- **Method display(),** this method displays the 2D array in a grid like fashion with 0-7 along the top and 0-7 down the side. It also prints a ‘|’ down the right-hand side to close off the grid.
- **Method run()** runs the code. We have 4 integers a, b, c and d which will hold the start x y coordinates and end x y coordinates respectively. We call on the method **fillArray()** to fill the array with just spaces (as blank entries). It then generates a random integer between 0 and 3 and stores it in variable r, which is used in the switch case to choose one of the letters to put in the grid. The loop code is surrounded by a try catch. The loop runs while run is true. It will first try to **display()** the grid. It then requires the user to input a value start x and start y and a value for end x and end y using the scanner. It stores these values in a, b, c, d respectively. We call on the method **setStartAndEndIndexes(a,b,c,d)** which sets our start and end points. It then calls on **display()** again to redisplay the grid now with ‘S’ and ‘E’ in it to show the start and end points. It will then ask the user would they like to run the algorithm. If yes, the algorithm is run, then it displays the grid with the path shown and sets run to false to exit out of the while loop. If no, run is set to false and the loop will no longer run.
- **Method checkOpenPos(int x , int y)** checks if a given coordinates is free from characters.
- **Method neighbours(Point p)** returns an ArrayList of the neighbours for a given point. It only returns a point if it is blank and if it is in the grid.
- **Method printPath(Point p)** goes from the point passed into its parent and sets the grid position equal to some character until the parent is null i.e. it has reached the start point as this point has no parent.
- **Method search()**, In our A star search we begin by clearing the open and closed lists, setting the gn, hn and fn of the start point and putting current equal to start. The main loop runs while the open list isn’t empty. The hn to the end point and the fn of each point in open is set. The open list is then sorted from the smallest fn to the largest fn and current is set equal to the first point in this now sorted open arraylist. Currents gn is set as one more than the previous current’s gn and the hn and fn are set also. This point is added to closed and removed from open. A check is now performed to see if we have reached the end point and if so, the path is filled in in the matrix by the printPath method and the main while loop is broken. For each of the neighbours to the current point the gn is set as one more than the currents gn (as the neighbours are always one step away from the current). If a neighbour is in closed, then this neighbour is skipped and the next iteration of the for each loop will begin. Otherwise, if this neighbour’s gn is less than the current’s gn *or* if open doesn’t contain the neighbour, we set the hn fn and set the parent of the neighbour as the current point indicating that the algorithm went from current to

neighbour. The neighbour is added to open if it is not already in it. After the main loop is broken, the start point is entered again as the printPath method writes over it.

Authors:

Arnas Juravicius 18257305

Cyiaph McCann 17233453

Dylan Kearney 18227023

Oisin McNamara 18237398