Searching a Wumpus World in Java

To start off I created the int’s that would be used as North, East, South and West. Because I decided to use one big array for the world the North and South direction would be set to 5 and -5, and East and West to 1 and -1. I liked to imagine these as the numbers on a snakes and ladders board so if you wanted to go one square down you would have to go five back. This idea of a game board helped me when setting up the walls and searching algorithm to avoid them.

I originally tried to created the World with a 2d array set out like world[4][4] = new Room, But I ended up having problems progressing through the parent and visited functions in my Depth limited search function so I decided to scrap it and go with just one big array of 25 values when I realized it would be much simpler to do. I Created a ArrayList for the nodes I have already visited because ArrayList’s are the easiest to manage in terms that their default value is null. I then Created a public class Room to represent each square in the dungeon, I gave the room a setWumpus method and a isWumpus method to initialize and recognize a wumpus, I also created a Wall function to set wall’s that the search would have to go around. I then created the world using the WumpusDungeon method that creates the World via a for loop which creates a Room in each of the array nodes. It then sets walls in their respective positions as specified on the map, and then the wumpus’ also using the attributes of the Room class.

The starting position is then passed to my wumpusSearch method, which is a stepping through style method to traverse through the world up unto the size of the array. It then uses a for loop to set up a haveVisited variable to make sure Rooms aren’t being checked more than once. Once this is done we are ready to run DepthLimitedSearch, I initially was going to use a breadth first search to find the closest wumpus but after re-reading the lecture slides I decided to go with DepthLimitedSearch because of its ability to limit how deep to go so it would only search the 25 Room world and not delve any deeper than the world size. DepthLimitedSearch uses a ArrayList I decided to call stack, stack is to manage the current position of the hunter. I have the successor state function after the current stack position has been resolved, The successor state function is used as a kind of step to continue the searching of the dungeon, it works by moving the current position by one room to continue searching on. The Stack is being continually updated so that the most updated room is being used.

The rest of my DepthLimitedSearch is basically just checking each Room one by one branching out slowly from the original starting point. I set four variables, west and east which I made the current position plus one square to either side, and north and south which is the current position up or down one. I then made a northSouth variable and a eastWest variable which will be used with a for loop to determine which direction should be taken. These are used in a bunch of if statements that check that the current position isn’t off the map and that there isn’t a wall in the way. If any of these if statements are wrong it the search stops and goes and searches the other direction. I feel like I could have been more efficient with the handling of my if statements but I didn’t want to let it fail somehow so I made them quite thorough. If all the previous if statements are satisfied then the current position is moved to the proposed position, the have visited variable is incremented and then the location that was taken is added to the stack. I have tried to make it more efficient by eliminating incrementing by using a for loop with the NorthSouth, westEast direction deciders. I felt like I could have made it more efficient again with the if statements but I couldn’t have done much more due to unfamiliarity with the algorithm.

The first Wumpus it finds stops the search by just returning the current position. Now that the search is done the directions to get there are printed out in via the end of the wumpusDungeon method, the directions are printed once all the nodes that have been visited are stored, the directions then reads the direction from the previous node until all of the visited nodes have been printed. The directions are recognized by checking whether the value is 5,-5,1,-1 away from the previous room, these values represent the global variables for north , south, east and west and can therefore be checked in an if statement and printed out depending on the direction. Because I put all of the methods that are to be output in wumpusDungeon method this is the only method that needs to be put into WumpusRun application file.

Because I have written all of the specific map information in the WumpusDungeon method it is really easy to change to replicate a different map, just change the current position variable to have the search start at different locations( I have tested this and it works) it is also just as easy to change the location of the wumpus’ and walls and the algorithm should work just as good as the original case. I could have put wumpusSearch and DLS together in one method but I felt like It would be to messy and hard to understand and mark. Overall the program searches a predefined dungeon, uses Breadth-first-search to find the closest wumpus and prints out the directions a hunter would take to get to it.

Appendix

package Assignment2;

/\*\*

\* My wumpus application class that is used to create the wumpus world

\*/

public class WumpusRun{

public static void main(String[] args){

WumpusDungeon wumpusWorld=new WumpusDungeon();

}

}

package Assignment2;

import java.util.\*;

/\*\*

\* the main class that sets up and initializes the wumpus dungeon with walls and

\* wumpus', It then searches the dungeon for the closest wumpus and prints out the directions to get to it.

\*/

public class WumpusDungeon{

int N=5,S=-5,E=1,W=-1;

int size =25;

Room[] world;

ArrayList<Integer> traveled;

int[] haveVisited;

int currentPosition;

public class Room{

int wall=0;

int steps=-1,previousRoom=-1;

boolean wumpus=false;

public void setWumpus(){

this.wumpus=true;

}

public boolean isWumpus(){

return wumpus;

}

public void setWall(int wall){

this.wall=wall;

}

}

/\*WumpusDungeon method that initializes all of the dungeon to having Rooms per number in the array

\* it then sets each Room that needs a wall with it to the corresponding direction.

\*/

public WumpusDungeon(){

System.out.println("You are entering a wumpus dungeon");

haveVisited = new int[size];

world = new Room[size];

for(int i=0;i<size;i++){

world[i] = new Room();

}

world[3].setWall(E);

world[4].setWall(W);

world[20].setWall(S);

world[15].setWall(N);

world[21].setWall(E);

world[22].setWall(W);

world[16].setWall(E);

world[17].setWall(W);

world[13].setWall(S);

world[8].setWall(N);

world[14].setWall(S);

world[9].setWall(N);

world[4].setWumpus();

world[17].setWumpus();

world[20].setWumpus();

world[24].setWumpus();

currentPosition = 0;

traveled = new ArrayList<Integer>();

int closestWumpus = wumpusSearch(currentPosition);

System.out.println("The closest Wumpus is in Room "+closestWumpus);

for(int i=closestWumpus; world[i].previousRoom!=-1; i=world[i].previousRoom)

traveled.add(i);

traveled.add(0);

System.out.println("Directions to the closest wumpus are: ");

for(int i = traveled.size()-1; i > 0; i--){

int direction = traveled.get(i-1) - traveled.get(i);

if(direction ==E)

System.out.println("East");

else if(direction ==N)

System.out.println("North");

else if(direction ==S)

System.out.println("South");

else if(direction ==W)

System.out.println("West");

}

System.out.println("You have found a wumpus");

}

/\*\*

\* wumpusSearch is used to progress through the rooms in the wumpus dungeon

\* by setting closestWumpus to the result of a DepthLimitedSearch

\*/

public int wumpusSearch(int position){

int steps = 0,closestWumpus;

while(steps<size){

for(int i=0;i<size;i++){

haveVisited[i]=0;

world[i].steps=-1;

}

closestWumpus = DLS(position , steps);

if(closestWumpus!=-1)

if(world[closestWumpus].isWumpus()){

return closestWumpus;

}

steps++;

}

return 0;

}

/\*

\* DepthLimitedSearchMethod

\* I used some pseudocode i found on the internet to help me code this becouse i was previously

\* unfamiliar with it.

\*/

public int DLS(int start,int steps){

int sidestep = 5;

ArrayList<Integer> stack = new ArrayList<Integer>();

stack.add((Integer)start);

world[start].steps=0;

haveVisited[start]=1;

while(true){

int end = stack.get(stack.size()-1);

stack.remove((Integer)end);

currentPosition+=(end-currentPosition); //Successor State Function

if(stack.isEmpty())

if(currentPosition!=start){

for(int i=2;i<size; i++){

haveVisited[i]=0;

}

}

if(haveVisited[currentPosition]==1){

haveVisited[currentPosition]=2;

if(world[currentPosition].steps == -1)

world[currentPosition].steps= world[world[currentPosition].previousRoom].steps+ 1;

if(world[currentPosition].isWumpus()){

return currentPosition;

}

if(world[currentPosition].steps < steps){

for(int i=-1;i<= 1;i+= 2){

int west = currentPosition-1;

int east = currentPosition+1;

int westEast = currentPosition+i;

int north= currentPosition+sidestep;

int south = currentPosition-sidestep;

int northSouth = currentPosition+i \*sidestep;

if(currentPosition+i\*sidestep>=0 && currentPosition+i\*sidestep<size)

if( haveVisited[northSouth]==0)

if (northSouth != currentPosition+world[currentPosition].wall){

world[northSouth].previousRoom=currentPosition;

stack.add(northSouth);

haveVisited[northSouth]=1;

}

if(westEast != currentPosition +world[currentPosition].wall)

if( westEast>=0 && westEast<size)

if(haveVisited[westEast]==0){

stack.add(westEast);

haveVisited[westEast]=1;

world[westEast].previousRoom=currentPosition;

}

}

}

}

if(stack.isEmpty())

return -1;

}

}

}