CSC 421 Assignment 1

Cameron Long, V00748439

1 Introduction

These functions will solve a graph starting from City A, to find a path to City Z. This path can be the path of least resistance, depending on randomized distance data inputted to the system, or just a simple graph traversal to find the goal.

The state-space will consist of all possible routes through the cities. though not that many are reachable without reaching a goal state.

The successor functions will be slightly different depending on the type of search function. For uninformed searches, it will be whichever edge shows up first, depending on the search methodology. For the informed searches, it will traverse the cities based on closest distance to cities and cost estimates to the goal.

For the informed searches, it will use the Manhattan distance to estimate the distance to the goal. As long as this estimate is never over the true cost to reach the goal, the heuristic is considered heuristic. Since the Manhattan distance will relax the constraints, and ignore cities in the way, allowing the graph to traverse the cities freely. it will underestimate the true cost. Taking a detour to another node will always increase the cost of the function.

Additionally, we can remove the constraint that the function has to travel in x,y coordinates to the goal. (That we have to move on a grid) and we can use the Euclidean distance to the goal to create an admissible heuristic. Similar to the Manhattan distance, we have removed constraints to the goal, finding the shortest straight line path. Any deviation from this path will add to the length of this optimal path. Thus, it is admissible.

2 Experimental Results

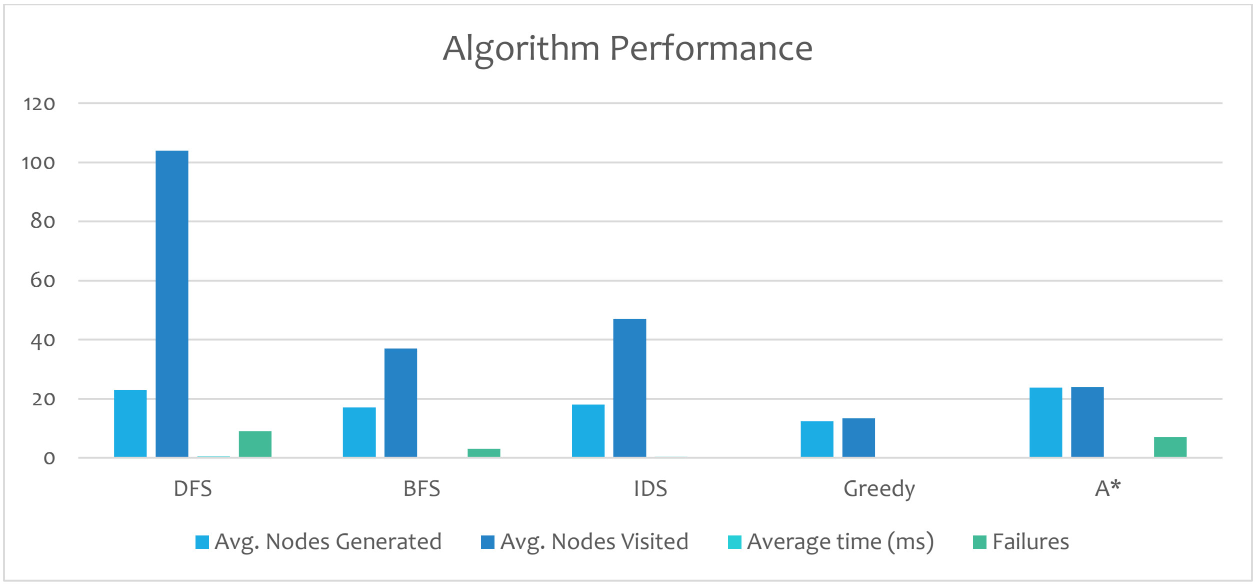

Figure 1: Algorithm Performance

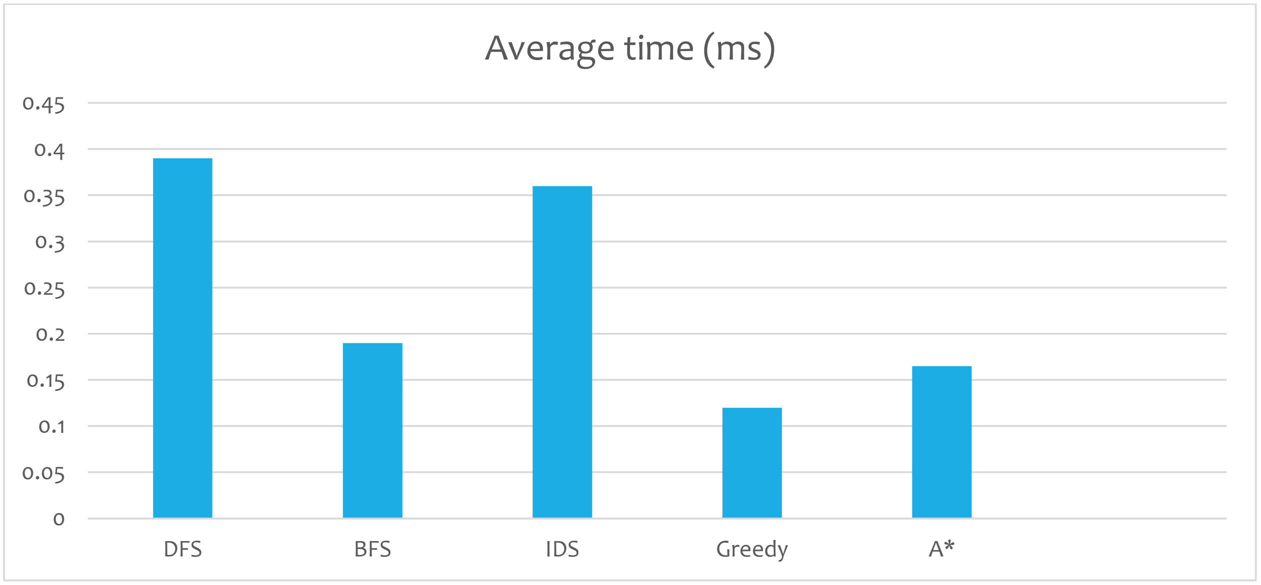

Figure 2: Response Time of Search Algorithms

	Avg. Nodes Generated	Avg. Nodes Visited	Average time (ms)	Failures
DFS	23	104	0.39	9
BFS	17	37	0.19	3
IDS	18	47	0.36	0
Greedy	12.39	13.39	0.12	0
A*	23.72	24	0.165	7

Table 1: Algorithm Performance

In this implementation, it is not unsurprising that the Depth First search struggled to keep up with the other algorithms. In the worst case, Depth first must iterate through the adjacency matrix several times, as it takes the lowest city (in alphabet terms) connected to a city. It makes traversing much more of the graph much more likely.

The A* search had several failures in trying to reach the goal with a reasonable cost. This could be due to a number of different reasons, likely due to the algorithm getting stuck in cost loops or other programming bugs. Ideally it should have a similar performance cost as the Greedy search, but in this experiment it is not the case.

The Iterative Depth search proved to have the close to the slowest runtime, at least in terms algorithm latency. However, the search function found the goal the most reliably, edging a basic breadth-first search in terms of errors.

The Greedy algorithm has proven to be the fastest and most reliable search algorithm in these results. This could be due to the speed of the implementation; however, the greedy search requires much more pre-processing of the data in order to be ran. In these results, the pre-processing times are not shown.

3 Implementation Overview

This program was chosen to be written in Java. The uninformed searches are all part of the same Java class, "adjacenymatrix" named after its data implementation (incorrectly). This is due to all three algorithms taking in the same input, of an adjacency matrix of the 26 cities. To begin, the program will generate a random instance of the adjacency matrix and process it. Each search is a method written for the class.

The informed search strategies are a little more complicated. The searches have been implemented as separate classes, extending from a master search class. Ideally, this class could be implemented into both subclasses. In the initial design of this system, the master class had more functions, that were eventually absorbed elsewhere.

Both types of searches draw from a list of class objects called Cities, that store adjacent city data as well as compute shortest paths and Euclidean costs. The representation of adjacencies is a list of cities. Each City also has a Position object, which is a x and y coordinate on a 100x100 graph of the cities. No two cities can occupy the same spot.

The test class generates all of my initial state data. Each City must look into its adjacency matrix and generate a list of positional data of all the cities near it. Then, the City object can process this data and make decisions for the searching algorithms. Once this is done, the testing algorithms simply calls the searching algorithms.

4 Code Listings

4.1 Uninformed Searches

```java
/*
 *
 * CSC 421 Assignment 1
 * Fall 2015
 * Cameron Long
 * V00748439

 *
 */
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.PriorityQueue;
import java.util.Stack;

public class Adjacenymatrix {

      /**
       * @param args
       */
      //global variables
       private int Vertices;
       private int Edges;
       private boolean[][] matrix;
       private static int alphabetno = 26;
       int depthbound = 1;
       static int[] visited = new int[26]; // list of visited cities
       static int nodesvisited =0;
       static int nodesgenerated =0;
       static int totalnodesv = 0;// total visited nodes for statistics
       static int totalnodesg = 0;//total generated nodes for statistics
       static long totaltime = 0;
       static long endTime;
       static long startTime;
       static int failures = 0; //how many failures has the algorithm had?


     static int depth = 0;

     int maxDepth = 1;
```

```
boolean goalFound = false;
public Adjacenymatrix(int vertices){
      //make empty matrix with 26 vertices
      this.Vertices = vertices;
      this.Edges = 0;
      this.matrix = new boolean[Vertices][Vertices];
      diagonalMatrix();
}

public void diagonalMatrix(){
     // generate zeroes in the diagonal
      for (int i = 0; i<26; i++){

            // cities cannot be adjacent to themselves
            matrix[i][i] = false;

      }

      randomAssignment();

}

public void randomAssignment(){

      //populate remaining graph with 260 0's and 390 1's

      int lcount = 390;

      while (lcount>0){

      int x =  (int)(Math.random()*26);
      int y =  (int)(Math.random()*26);

      //Around 60% of the graph will be 1's

      if(x == y){

            matrix[x][y] = false;
            //place cities in matrix
      }

      else if(matrix[x][y] == false){

            matrix[x][y] = true;

            lcount--;
            //Don't waste 1's on spaces already adjacent
      }

      }

}
public void dfs(){

      //Depth-first searching of the adjacency matrix

      Stack<Integer> search = new Stack<Integer>();
```

```
            search.push(0);
            for(int i=0; i<26 ; i++){


                  for (int j=0; j<26 ; j++){

                 //System.out.println("I: " + i + "J: " + j);
                 // Print the system state for debugging

                 if (nodesvisited > 676){

                       failures++;

                       return;
                       // if the stack is stuck in a loop, terminate and
indicate failure

                 }
                  if (matrix[i][j] == true && j == 25){

                        System.out.println("Found the city: Z");
                        System.out.println("Complexities:   " +
nodesgenerated  + " " + nodesvisited);
                        return;
                        //if goal city is found, exit and print state
                  }
                  if(matrix[i][j] == true){

                        if(!search.contains(j)){
                             //Graph traversal of nodes with children
                        //System.out.println("Traversed: " + j);
                        search.push(j);
                        if(visited[i] == 0){
                              nodesgenerated++;
                              visited[i] = 1;
                              //indicated the node has been visited before
                        }

                        nodesvisited++;
                        i = j;
                        j = 0;
                        }

                  }

                  if (j == 25 && matrix[i][j] != true){

                        j = (Integer)search.pop();
                        nodesvisited++;
                        //if the goal hasn't been been found for a given row
                  }

            }

            }
      }
     public void bfs(){

           //Breadth first search implementation with a prioirity queue.

           PriorityQueue<Integer> queue = new PriorityQueue<Integer>();

           for (int row=0 ; row<26; row++){


                 //iterate through adjacency matrix
                 for (int column = 0; column < 26 ; column++){

                        if (matrix[row][column] == true && column == 25){

                              System.out.println("Found the city Z:");
                              System.out.println("Complexities:   " +
nodesgenerated  + " " + nodesvisited);
                             // nodesvisited = nodesgenerated = 0;

                             // System.exit(0);
                              // if City Z has been found, terminate
                                   for (int j = 0 ;j< 26; j++){

                                         if (visited[j] == 1){

                                               nodesgenerated++;
                                               //calcukated how many unique
nodes were visited
                                         }

                                   }

                             return;

                        }
                       if(nodesvisited > 676){

                             // if stuck in a search loop, indicate failure
                             for (int j = 0 ;j< 26; j++){

                                   if (visited[j] == 1){

                                         nodesgenerated++;

                                   }

                             }

                             failures++;// up failure count for stats
                             return;
                       }

                       if(matrix[row][column] == true){
                             if(!queue.contains(column)){
                             //queue a node if not already in the queue and
indicate it has been visited
                             queue.add(column);
                             visited[column] = 1;
                             nodesvisited++;
                             }

                       }



                 }
                 // travel to next node
                 row = queue.remove();
                 nodesvisited++;

           }





     }

           //Iterative Deepening Search with Stack
   public void depthLimitedSearch(int source, int goal)
   {
       Stack<Integer> stack;



       while(!goalFound){
       //while the goal hasn't been found: do
       int element, destination = 1;

       //start by pushing the start node
       stack = new Stack<Integer>();
       stack.push(source);

       depth = 0;

       //start max depth at 0


      // System.out.println("\nAt Depth " + maxDepth);

     //  System.out.print(source + "\t");

       //printing for debugging

       while (!stack.isEmpty())

       {

           element = stack.peek();
           //What's on top?
           while (destination <= 25)

           {

               if (depth < maxDepth)
               {
                 //terminate past max depth
                   if (matrix[element][destination] == true)

                   {

                       stack.push(destination);
                       nodesvisited++;
                      //mark that a node has been visited for stats
                       visited[destination] = 1;

                       System.out.print(destination + "\t");
                       // debugging
                       depth++;

                       if (goal == destination)

                       {
                             //found the goal and terminate
                           goalFound = true;
                           for (int i = 0; i < 26 ; i ++ ){


                                   if (visited[i] == 1){

                                   nodesgenerated++;

                             }

                           }
                           return;

                       }
                       //breadth - first on children
                       element = destination;
                       //row = column of adjacency matrix
                       destination = 1;
                       // start columns at 1
                       continue;

                   }

               } else

               {

                   break;

               }

               destination++;
               //keep looping through the adjacency matrix
           }
           //avoid nullstack errors
           destination = stack.pop() + 1;
           nodesvisited++;
```

```
            //if pop, you've moved one level up
            depth--;

        }
        maxDepth++;
        //restart from new depth
        }


    }
 // reset the variables for multiple trials in a loop
    public static void reset(){
          visited = null;

            visited = new int[26];




    }

      public static void main(String[] args) {

            // Main for calling the searching algorithms. Keeps track of
execution time and seperate varibles.

            //The searches must be ran seperately.
            int i = 0;
            failures = 0;
            while(i <100){
            // TODO Auto-generated method stub
            Adjacenymatrix test;
            test = new Adjacenymatrix(alphabetno);
            //System.out.println(test.toString());

            reset();

            startTime = System.nanoTime();
            //test.bfs();

            test.depthLimitedSearch(0, 25);
            endTime = System.nanoTime();

            totalnodesv += nodesvisited;

            totalnodesg += nodesgenerated;

            nodesvisited = 0;

            nodesgenerated = 0;

            totaltime += (endTime - startTime);



            totaltime += endTime -startTime;
            i++;
            }

            System.out.println("Complexities:   " + totalnodesg/100  + " " +
totalnodesv/100 + " " + totaltime/1000000 + " " + failures);


            //test.bfs();
            //test.ids();


            //test.depthLimitedSearch(0, 25);
            //System.out.println("\nGoal Found at depth " + depth);

            //System.out.println("Complexities:   " + nodesgenerated  + " " +
nodesvisited);
            System.exit(0);

      }

}
```

4.2 City Class

```
//City class for informed searches
//acts as a storage object for

import java.util.ArrayList;


public class City {
      //City data
      Position parent; // parent node when searching. Stores it's position
data
      final Position current; // Position data of current Node
      boolean goal; // is this the goal?
      int number;// number 0-26 of City
      double cost, Euclidean;// cost funstions
      boolean visited = false;
      int isave =0;
      Position closest = null;//closest city to current city
      ArrayList<Position> adjacencyList;// Copy of adjacency for processing
      ArrayList<Position> adjacencyList2;


      public City(int x, int y, int number){
            //initialize the city
            this.current = new Position(x, y);
            cost = 0;
            this.number = number;
            this.adjacencyList = new ArrayList<Position>();
            this.adjacencyList2 = null;
            Euclidean = Math.sqrt((Math.pow((26-x), 2)+Math.pow((26-y), 2)));
            if (number == 25){
```

```
                        goal = true;
                }
        }

        public Position findclosest(){

                //calculates the closest city, that has not been visited by the
algorithm

                // removes visited nodes from consideration
                adjacencyList2 = adjacencyList;
                double smallest = 500;

                if(adjacencyList2.size() == 1){
                        // if there's only one city left
                        closest =  new Position(adjacencyList2.get(0).x,
adjacencyList2.get(0).y);
                        //return it
                }

                for (int i = 0; i< adjacencyList2.size(); i++){

                Euclidean = Math.sqrt((Math.pow(Math.abs((current.x-
adjacencyList2.get(i).x)), 2)+Math.pow(Math.abs((current.y-
adjacencyList2.get(i).y)), 2)));

                if (Euclidean < smallest){

                        smallest = Euclidean;

                        isave= i;
                        //save position of closest city in list
                }

                }

                if (isave == adjacencyList2.size()){

                        return closest;

                }

                closest =  new Position(adjacencyList2.get(isave).x,
adjacencyList2.get(isave).y);



                return closest;


        }

        public void remove(){
                //remove nodes from consideration
                if (!(adjacencyList2.size() == 1))
                adjacencyList.remove(isave);
```

```
      }
}
```

4.3 Position Class

```
//Stores position data of the cities in the grid

public class Position {

      int x, y;
      //x,y coordinates
      public Position(int x, int y) {
            this.x = x;
            this.y = y;
      }
}
```

4.4 Informed Search Class

```
import java.util.ArrayList;
import java.util.List;

//Master Search class
//Contains some shared assets for both searches, so I didn't have to generate
them seperately

public class InformedSearch {
      int startx, starty;
      protected Position startNode;
      protected City[][] cities = new City[100][100]; //grpah of cities in 2D
space
      protected List<City> visitedList = new ArrayList<City>(); //visited
list of cities
      static int[] visited = new int[26]; // easier processing of generated
nodes
      //Create the necessary data structures
      public InformedSearch() {
            this.startNode = new Position(startx, starty);
      }


      public void createCities(String fileName){

 for (int i = 0; i < 26 ; i++){
                  //Create the positional data for the cities
                  int x =  (int)(Math.random()*100);
                  int y =  (int)(Math.random()*100);

                  if (cities[x][y] == null && i > 0){

                        cities[x][y] = new City(x, y, i);

                        //place cities in graph

                       // System.out.println("City Generated at :" + x + " "
+ y);
```

```
                    }
                    if (i == 0){
                          cities[x][y] = new City(x, y, i);
                          startx = x;
                          starty = y;
                          //if it's city 0 or "A", it's the start of the graph
                    }
              }
        }
       }
```

4.5 Greedy Search

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

//Greedy Best First Implmentation, from the Informed Search Class

public class GreedyBestFirst extends InformedSearch {
      public GreedyBestFirst(Position startNode, City[][] cities, List<City>
visitedList) {
            this.startNode = startNode;
            this.cities = cities;
            this.visitedList = visitedList;
            //Initialized Variables
      }

      public void greedy(){
             int nodesvisited =0;
            //System.out.println("Startnodes: " + startNode.x + " " +
startNode.y);

            List<City> queue = new ArrayList<City>(); //store cities
            List<City> visitedList2 = new ArrayList<City>(); // store which
ones you've visited
            queue.add(cities[startNode.x][startNode.y]); // add the start to
the queue
            City closest;
            while(!queue.isEmpty()){

                  City r = queue.remove(0);
                  //remove top city
                  //System.out.println("City x-y:"+r.current.x+"-
"+r.current.y);
                  nodesvisited++;
                  //System.out.println(r.number);
                  if (r.goal) {

                        // if the goal, exit
                        //System.out.println("Found the Goal ");

                        System.out.println(" " + visitedList2.size() + " " +
nodesvisited);


                        //System.exit(0);

                        return;
                  }

                  visitedList.add(r);
                  if (!visitedList2.contains(r)){
                  // add the visited to the list for stats
                  visitedList2.add(r);}

                  r.visited = true;
                  closest = (cities[r.findclosest().x][r.findclosest().y]);//
find the closest euclidean distance
                  while(closest.visited == true){
                        try{
                  r.remove();// keep iterating through nodes that haven't
been visited
                        }
                        catch(NullPointerException e){

                              System.out.println("Failed to find goal");
                        }
                  closest = (cities[r.findclosest().x][r.findclosest().y]);

                  }


      if(!visitedList.contains(cities[r.findclosest().x][r.findclosest().y])
&& !queue.contains(cities[r.findclosest().x][r.findclosest().y])){

                        //moce to closest node and select the parent

                        closest.parent = r.current;

                        queue.add(closest);


                  }



            }
      }

}
```

4.6 A* Class

```
import java.util.ArrayList;
import java.util.List;

//Astar implementation with cost analysis
public class Astar extends InformedSearch {


            public Astar(Position startNode, City[][] cities, List<City>
visitedList) {
                  this.startNode = startNode;
                  this.cities = cities;
                  this.visitedList = visitedList;
                  //initialize the object
            }

      public void astar(){
            ArrayList<City> queue = new ArrayList<City>(); //list of nodes to
visit
            queue.add(cities[startNode.x][startNode.y]);// initialize start
state
            City closest = null; // store closest city
            List<City> visitedList2 = new ArrayList<City>();// store visited
cities
            int nodesvisited = 0;

            while(!queue.isEmpty()){

                  City r = queue.remove(0);
                  double cost = 0;
                  //add cost to goal
                  closest = (cities[r.findclosest().x][r.findclosest().y]);
                  //find the closest with least cost
                  while(closest.visited == true){
                        //System.out.println("Where am I hanging");
                        try{
                  r.remove();
                        }
                        //keep iterating through adjacency list to find
closest
                        catch(NullPointerException e){

                              System.out.println("Failed to find goal");
                        }// catch failure

                        closest =
(cities[r.findclosest().x][r.findclosest().y]);
                  }

                  if(r.parent != null){
                        //find total cost of movement
                        cost =
cities[r.parent.x][r.parent.y].cost+1+r.Euclidean;
                  }
                  //System.out.println("City x-y:"+r.current.x+"-
"+r.current.y);
```

```
                    if (!visitedList2.contains(r)){

                          visitedList2.add(r);
                          }// second visited list for stats

                    if(nodesvisited > 100){

                          System.out.println("Failure");
                          // catch failure
                          return;
                    }

                    visitedList.add(r);
                    r.visited = true;
                    nodesvisited++;
                    if (r.goal) {
                          //printSolution("astar", r);

                          //System.out.println("Cost: " + cost + " Astar
Completed");
                          System.out.print(" " + visitedList2.size() + " " +
nodesvisited);
                          //find goal and exit
                          //System.exit(0);
                          return;
                    }


      if(!visitedList.contains(cities[r.findclosest().x][r.findclosest().y])
&& !queue.contains(cities[r.findclosest().x][r.findclosest().y])){



                          queue.add(closest);
                          //add closest one to queue
                          closest.parent = r.current;
                          //set parent node in path
                          cities[r.current.x][r.current.y].cost = cost;

                          //total cost of running so far
                    }

      }
      }
}
```