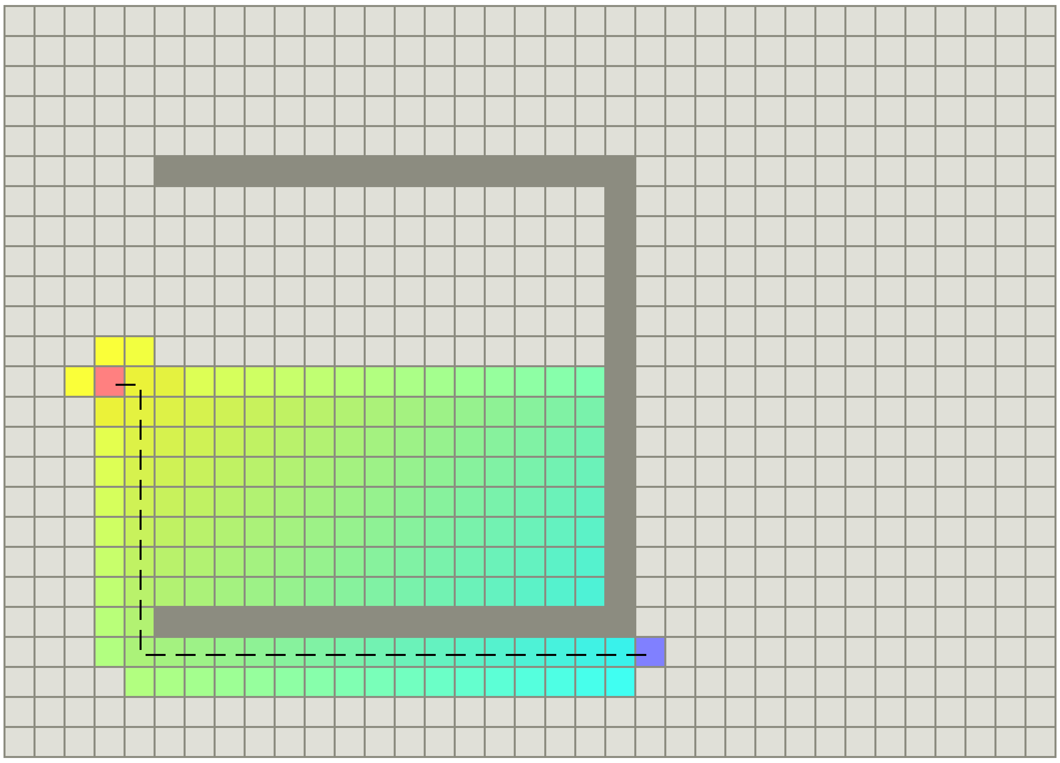
Artificial Intelligence Coursework Part 1

A* Search

A* is a computer algorithm which is widely used in pathfinding and graph traversal. The algorithm efficiently plots a walkable path between multiple nodes, or points, on the graph. On a map with many obstacles, pathfinding from points A to B can be hard. A robot, for instance, without getting much other direction, would proceed until it encounters an obstacle. However, the A* algorithm provides a heuristic into a normal graph-searching algorithm. The A* algorithm does this by planning at each step, so a more optimal decision is made. A* is an extension of Dijkstra's algorithm with some attributes of breadth-first search(BFS).

Like Dijkstra, A* operates by creating a lowest-cost path tree from the beginning node to the target node. For every node, A* uses a function f(n) which provides an estimate of total cost of path by using current node. Therefore, A* is a heuristic function that differs from an algorithm in that a heuristic is more of a estimate and is not fully correct. A* expands paths which is already less expensive by using the function f(n) = g(n) + h(n). f(n) is equal to total estimated cost of path through node n, g(n) is equal to cost so far to reach node n and h(n) is the estimated cost from n to goal. H(n) is the heuristic part of cost function which indicate that h(n) is like a guess.

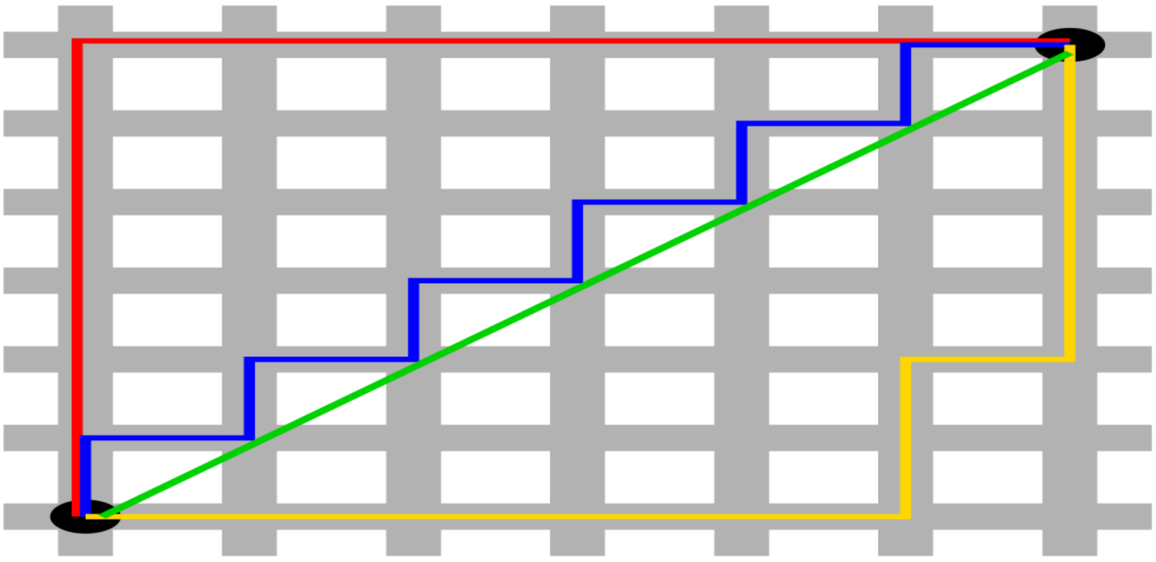
A* Search Example:



A* algorithm starts at the red node and considers all neighbour nodes. Once the list of adjacent nodes has been populated, it filters out those that is inaccessible which are walls, obstacles or out of bounds.   A* then picks the node with lowest cost that is the estimated f(n). This process is recursively repeated until shortest path has been located to the blue node. The computation of f(n) is done within a heuristic which normally provide decent results. Calculation of h(n) can be done in many ways such as using the Manhattan distance from node n to goal or if h(n) = 0 where A* becomes Dijkstra's algorithm, that is guaranteed to find a shortest path. The heuristic function should be admissible, that indicate it can never overestimate the cost to reach the goal. Both Manhattan distance and h(n) = 0 is admissible.

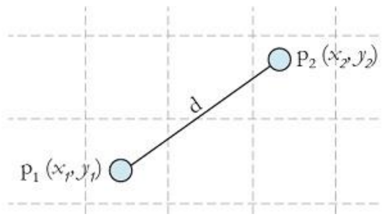
Using a good heuristic is crucial in determining the performance of A*. The value of h(n) would ideally equal the exact cost of reaching goal. This is, however not feasible as we do not know the path. However, we can choose a technique which would provide us the exact value some of the time, such as when traveling within a straight line with no obstacles. This would lead to a perfect performance of A* in such a case. We want to select a function h(n) which is less than the cost of reaching the destination node. This would enable h to work accurately, if we select a value of h which is larger, it would result to a quicker but less accurate performance. Therefore, it is normally the case which we choose a h(n) which is less than the real cost.

The Manhattan Distance Heuristic



This technique of computing h(n) is known as the Manhattan technique as it is computed by calculating the total amount of squares moved horizontally and vertically to reach destination square from current square. We ignore diagonal movement and any obstacles which could be in the way. The formula is h = |Xstart – Xgoal| + |Ystart – Ygoal|. The heuristic is exact whenever our path follows straight lines. A* would locate paths which is a combinations of straight line movements. Occasionally, we could prefer a path which tends to follow a straight line directly to the final node.

The Euclidean Distance Heuristic



The Euclidean distance heuristic is little bit more accurate than the Manhattan counterpart. If we attempt to run both heuristics simultaneously on the same maze, the Euclidean path finder favours a path along a straight line. The Euclidean distance heuristic is more accurate, but it is also slower as it needs to explore a bigger area to locate the path. The heuristic formula is h = SquareRoot(((Xstart – Xdestination)(^2) + (Ystart – Ydestination)(^2)))

Advantages and Disadvantages of A* Search

The advantage of A* search is that it guides an optimal path to a goal if the heuristic function h(n) is admissible, indicating it never overestimates actual cost. For example, since airline distance never overestimates actual highway distance and Manhattan distance never overestimates actual moves within a gliding title.

The advantage of A* search is that A* makes efficient use of the provided heuristic function within among all shortest-path algorithms using the provided heuristic function h(n). A* algorithm expands the fewest number of nodes.

The disadvantage of A* search is that it requires a large amount of memory because the whole open list should be saved, A* search is very space-limited in practice and is no more practical compare to best-first search algorithm on current machines. For example, while it can be run successfully on the 8-puzzle, it exhausts available memory within a matter of minutes on the 15-puzzle.

Hill-Climbing Search

Hill Climbing is heuristic search used for mathematical optimization issues. Given a huge set of inputs and a good heuristic function, the algorithm attempts to locate a sufficiently good solution to the issue. This solution could not be the global optimal maximum. Mathematical optimization problems indicate that hill climbing solves the problems where we required to maximize or limit a provided real function by choosing values from provided inputs. For example, travelling salesman problem where we required to limit the distance travelled by salesman. Heuristic search indicate that search algorithm could not locate the optimal solution to the problem. However, it would provide a good solution within reasonable time. A heuristic function is a function which would rank all the possible alternatives at any branching step within search algorithm depending on available information. The function helps to select best route out of possible routes.

Hill climbing search is a variant of generate and test algorithm. The generate and test algorithm is:

1. Make possible solutions.
2. Test to see if this is the expected solution.
3. If the solution has been located quit else go to step 1.

Therefore, Hill Climbing search is called a variant of generate and test algorithm because it takes the feedback from test procedure. Then this feedback is utilized by the generator within deciding the next move within search space. Hill-Climbing also uses the greedy approach which mean that any point within state space, the search moves within that direction only that optimizes the function cost with the hope of locating the optimal solution at the end.

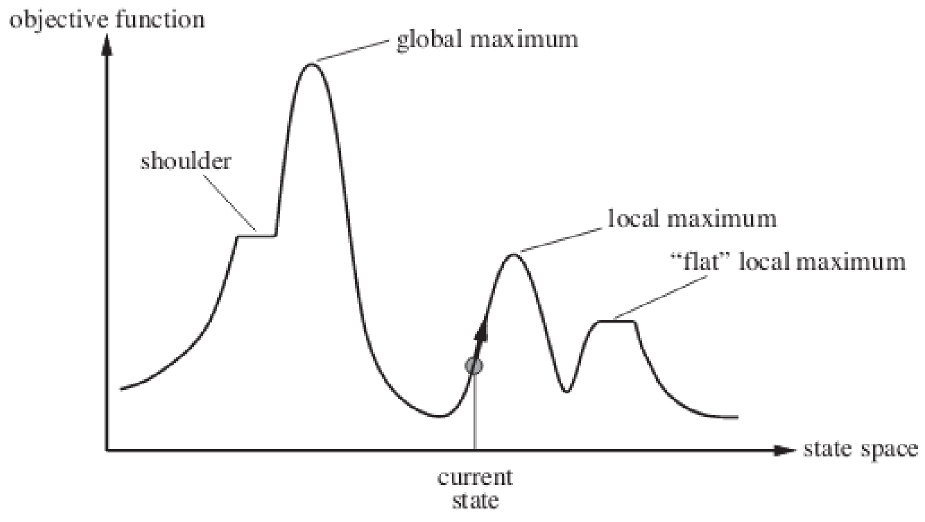
Types of Hill Climbing

Simple Hill climbing examines the adjacent nodes one by one and selects the first adjacent node that optimizes the current node as next node.

Steepest-Ascent Hill climbing first examines all the adjacent nodes and then selects the node closest to the solution state as next node.

Stochastic hill climbing does not examine all the adjacent nodes before deciding what node to select. It just selects an adjacent node at random and decides whether to move to that adjacent node or to examine another adjacent node. This depends on the amount of enhancement in that adjacent node.

State Space diagram for Hill Climbing



State space diagram is a graphical representation of the group of states our search algorithm able to reach vs value of our objective function that we wish to maximize. The X-axis denotes the state space such as states or configuration our algorithm could reach. Y-axis denotes the values of objective function corresponding to a certain state. The best solution would be the state space where objective function contains maximum value which is also known as global maximum.

Different Regions in the State Space Diagram:

Local maximum is the state that is better than its adjacent state however there exists a state that is better than it such as global maximum. Global maximum state is better as value of objective function is higher than its neighbours. Global maximum is the best possible state within the state space diagram. This is due to the fact at this state, objective function has highest value. Flat local maximum is a flat region of state space where adjacent states contain same value. Ridge is a region that is higher than its neighbour but itself has a slope. Ridge is a special kind of local maximum. Current state is the region of state space diagram where we are currently present during the search. Shoulder is a plateau which contains an uphill edge.

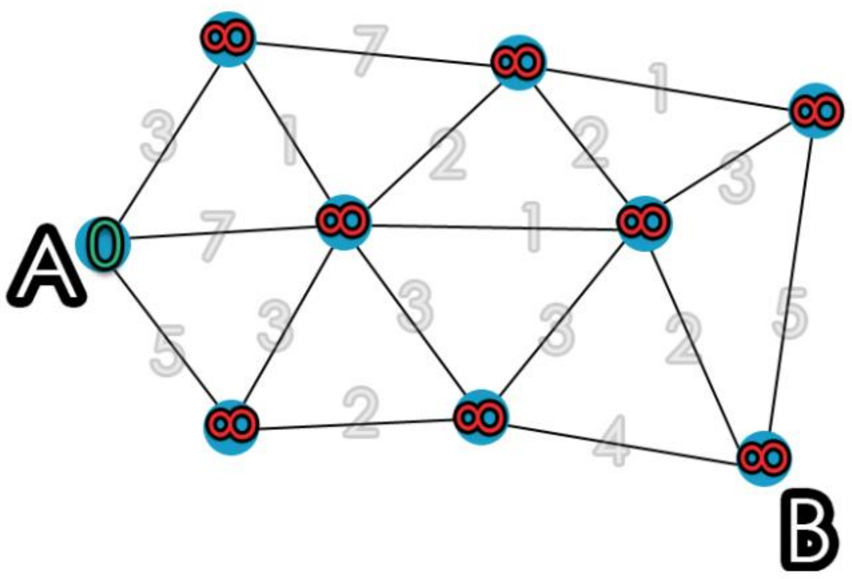
Advantage and Disadvantage of Hill-Climbing Search

The advantage of Hill-Climbing search is that Hill-climbing search is helpful to solve pure optimization problems where the aim is to locate the best state depending on the objective function. Also, the algorithm requires less conditions compare to other search methods.

The disadvantage of Hill-climbing search algorithm is that it uses local information, so the algorithm is easily can be fooled. The algorithm does not maintain a search tree, so the current node data structure required to only record the state and its objective function value. This would cause the algorithm to assumes that local enhancement would lead to global improvement.

Dijkstra's Shortest Path Algorithm

Dijkstra algorithm is an algorithm that locate the shortest path from a beginning node to a destination node within a weighted graph. The algorithm makes a tree of shortest paths from the beginning vertex, the source, to all other points within the graph. The graph can either be directed or undirected. One stipulation to using the algorithm is that graph required to contain a nonnegative weight on every edge. Dijkstra algorithm locates a shortest path tree from one source node, by building a group of nodes which contain a limited distance from the source.



The graph has vertices, or nodes, denoted within the algorithm by v or u. The graph also has weighted edges which connect two nodes: (u, v) denotes an edge, and w(u, v) denotes it weight. The weight for each edge is represented in a grey colour within the diagram.

Advantages and Disadvantages of Dijkstra

The advantage of Dijkstra shortest path algorithm is once it has been carried out you can find the least weight path to all permanently labelled nodes.

The advantage of Dijkstra shortest path algorithm is that the Dijkstra's algorithm has a order of n(^2) so it is efficient enough to use for relatively big problems.

The disadvantage of Dijkstra shortest path algorithm is that the algorithm is not able to operate with negative weight arcs.

Instructions on how to run program

Run the program, the 2nd mode would run straight away and display the final route and its final information. If you want to step through search one by one, press keyboard key 'a' into the frame until the final route is found and its information displayed on the console.

Source Code

ReadCaverns.java

```
import java.awt.event.KeyAdapter;

import java.awt.event.KeyEvent;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.util.Deque;

import java.util.LinkedList;

import javax.swing.JFrame;

import javax.swing.JTextField;


public class ReadCaverns {


        //Checked if current cave and neighbour cave is connected or not

        public static Boolean connected(int cave1, int cave2) {
```

```
        return Connected[cave1 - 1][cave2 - 1];
}



//boolean 2D array
private static boolean[][] Connected;

public static void main(String[] args) throws IOException {



        BufferedReader br = new BufferedReader(new FileReader("input.cav"));


        // Read the line of comma separated text from the file
        String buffer = br.readLine();
        System.out.println("Raw data : " + buffer);


        //close buffered reader
        br.close();


        // Convert the data to an array
        String[] data = buffer.split(",");


        // Now extract data from the array - note that we need to convert from
        // String to int as we go
        int noOfCaves = Integer.parseInt(data[0]);


        //print our number of caves
        System.out.println("There are " + noOfCaves + " caves.");


        //Linked list of caves
        LinkedList<Cave> caves = new LinkedList<Cave>();
```

```
Connected = new boolean[noOfCaves][noOfCaves];

//Set coordinates to each cave and add cave to the caves list
for (int cave = 1; cave <= noOfCaves; cave++) {
            int x = Integer.parseInt(data[2 * cave - 1]);
            int y = Integer.parseInt(data[2 * cave]);
            caves.add(new Cave(x, y, cave));
}

// Build connectivity matrix

// Declare the array

// Now read in the data - the starting point in the array is after the
// coordinates

//When two caves are connected o each other, the connected 2D array is set to 1
int start = noOfCaves * 2 + 1;
for (int i = 0; start + i < data.length; i++) {
   Connected[i % noOfCaves][i / noOfCaves] = Integer.parseInt(data[start + i]) == 1;
}



Dijkstra dijkstra = new Dijkstra();
LinkedList<Cave> listOfCaves = new LinkedList<Cave>();

//Dijkstra method is computed
dijkstra.Compute(caves, caves.getFirst(), caves.getLast());
```

```
                    //Total distance is printed out
                    System.out.println("Total Distance is " + caves.getLast().getDistance());


                    //Caves are pushed to the path and added to list of caves linked list from last cave to
start cave
                    Deque<Cave> path = new LinkedList<>();
                    Cave cave = caves.getLast();
                    while (cave != null) {
                       path.push(cave);
                       listOfCaves.add(cave);
                       cave = cave.getPredecessor();
                    }


                    //The shortest route is printed out in one go
                    for (Cave currentCave : path) {
                              System.out.println("Cave number is " + currentCave.getNodeNumber() + "
The coordinates is " + currentCave.getX() + "," + currentCave.getY() + " current distance is "
                              + currentCave.getDistance());
                    }


                    class MyKeyListener extends KeyAdapter{


                              int i = listOfCaves.size();
                              //Key pressed method
                              public void keyPressed(KeyEvent evt){


                                 i--;
                          //Search stepped through by pressing the 'a' key to advance by one step. Each
information about current state of search is displayed at each step.
                                 if(i > 0 && evt.getKeyChar() == 'a'){
                                        System.out.println("The cave number is " +
listOfCaves.get(i).getNodeNumber() + " The coordinates is " + listOfCaves.get(i).getX() + "," +
                                        listOfCaves.get(i).getY()  + " The current distance is " + 
listOfCaves.get(i).getDistance());
                                    }
                                    else if(i == 0 && evt.getKeyChar() == 'a'){
                                            System.out.println("The cave number is " + 
listOfCaves.get(i).getNodeNumber() + " The coordinates is " + listOfCaves.get(i).getX() + "," + 
                                        listOfCaves.get(i).getY()  + " The total length of route is " + 
listOfCaves.get(i).getDistance());
                                    }
                                }
                    }

                    //Text field
                    JTextField component = new JTextField();

                    //Key listener
                    component.addKeyListener(new MyKeyListener());

                    //frame:
                    JFrame f = new JFrame();
                    f.add(component);
                    f.setSize(300, 300);
                    f.setVisible(true);
            }
}
```

Dijkstra.Java

```
import java.util.LinkedList;
import java.util.Queue;

public class Dijkstra {
        //queue
        Queue<Cave> queue = new LinkedList<Cave>();

        //current cave
        Cave currentCave;

        @SuppressWarnings("unchecked")
        public void Compute(LinkedList<Cave> caves, Cave source, Cave destination){

                //First cave distance is set to 0
          source.setDistance(0);

          //Each cave that is not the first cave is set to max value. Every cave is added to the
queue
                for(Cave cave : caves){
                        if(cave != source){
                                cave.setDistance(Integer.MAX_VALUE);
                        }
                        queue.add(cave);
                }

                //Check that the queue is not empty
                while(!(queue.isEmpty())){

                  //current cave is set to the cave that has the lowest distance within the
queue
                        double minValue = Double.MAX_VALUE;
                        for(Cave cave : queue){
                                double value = cave.getDistance();
```

```
                    if(value < minValue){
                        minValue = value;
                     currentCave = cave;
                    }
                }

                //The current cave is removed from the queue
                queue.remove(currentCave);

                //While loop terminated if current cave is equal to the final cave
                if(currentCave == destination){
                   break;
                }



                for(Cave caveNeighbour : queue){
                    //Check if current cave and the next cave is connected
                    if(ReadCaverns.connected(currentCave.getNodeNumber(),
caveNeighbour.getNodeNumber())){
                        //distance between current cave and neighbour cave
                        double minDistance = (currentCave.getDistance() +
Math.hypot(currentCave.getX()-caveNeighbour.getX(), currentCave.getY()-
caveNeighbour.getY()));
                        //checks if the minimum distance is less than the
neighbour cave distance
                        if(minDistance < caveNeighbour.getDistance()){
                            //cave neighbour distance is set to the
minimum distance
                            caveNeighbour.setDistance(minDistance);
                            //cave neighbour is set to current cave
                            caveNeighbour.setPredecessor(currentCave);
```

```
                                        }
                                }
                        }
                }
        }
}
```

Cave.java

```
public class Cave {

        //integers:
   private int x;
        private int y;
        private int node_number;

        //double
        private double distance;

        //Cave
        private Cave predecessor;

        //constructor
        public Cave(int x, int y, int node) {
           this.x = x;
           this.y = y;
           this.node_number = node;
        }
```

```
//get and set methods:
public int getNodeNumber(){
        return node_number;
}

public int getX(){
        return x;
}

public int getY(){
        return y;
}

public void setDistance(double distance){
        this.distance = distance;
}

public double getDistance(){
        return distance;
}

public void setPredecessor(Cave cave) {
   predecessor = cave;
}

public Cave getPredecessor() {
        return predecessor;
    }

    public void setNodeNumber(int nodeNumber){
            this.node_number = nodeNumber;
    }

    public void setX(int x){
            this.x = x;
    }

    public void setY(int y){
            this.y = y;
    }
}
```