ARTIFICIAL INTELLIGENCE – Pathfinding COURSEWORK

Davide Pollicino
40401270@live.napier.ac.uk
School of Computing, Edinburgh Napier University, Edinburgh -AI (SET09122)[1]

Abstract. In this paper is the official report for that pe first coursework in the AI module. In this coursewor our goal is to discuss a path finding algorithm that allows a robot to navigate through a series of cavern connected between them, finding the shortheath with a cave A and a cave B, with an optimal solution terms of time and space complexity. The only information provided in input to our algorithms will be: a set coordinates and an adjacency matrix, to know the connection of each cave with other caves.

Keywords: BFS Algorithm (Brest First Search), Davide, Pollicino, Artificial, Intelligence, Dijkstra, A*, Graph, Shortest Path Finding.

My Algorithm Choice for this project

The algorithm that I want to use for this coursework is the A* algorithm, explaining below, why the A algorithms have in the worst case a smaller or equal time complexity of any other path finding algorithm like Dijkstra.

Algorithms Confrontation

1.1 Best First Search (BFS)

BFS is a pathfinding algorithm with a O (V+E) time complexity. It explores the graph exploiting the most promising next node according to a heuristic valuation. The heuristic is used to establish the likelihood of the choice of the next node, to evaluate if that node will help get closer to our goal (destination) or no. In order to choose the possible best node, we need at least one common metric, used in all the graphs. This metric used in this specific coursework is the Euclidean distance between each vertex V. BFS has O(V+E) time complexity – it is perfect for unweighted graphs; It basically checks the level by level; It works using a queue.

The use of BFS will not always allow to find the shortest path and in particular graphs with lots of leaf end vertex, and managing many backtracking operations would decrease the performance of the algorithms.

1.2 A* (A Star)

A* is a BFS algorithm that can be considered as an extension of the Dijkstra algorithm. A* is widely used in many applications like: path finding in maps, social networks and games. It offers an overall improvement in terms of time and space complexity, thanks to the use of a heuristic metric. In order to use A* efficiently, two rules must be always adopted: the heuristic metric must be common in all the graphs, and the cost to reach the destination vertex (goal) must **never be overestimated.**

The time complexity of A* depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) *d*: $O(b^d)$, where *b* is the branching factor (the average number of successors per state).[2]

1.3 Dijkstra

It is another path finding algorithm. The problem with Dijkstra is that this algorithm will find the shortest path, without paying attention to where we are going while traversing a graph. Of course, in another context, like for example the analysis of the United Kingdom maps if all its route or a test case with more than 1.000.000.000 caverns, Dijkstra would start to have will be relatively slower than A*. In a situation when our graph would be an extreme fitted network (for example graphical representation of the roads In London or Manhattan), Dijkstra will not be optimal. In terms of performances, Dijkstra will perform worse or equal to A*exploiting lots of memory for the tracking of nodes that at the end of the path finding process will not be even nearly correlated to the path found.

Project Requirements and Project Analysis

A robot has to navigate through a series of small underground caverns connected by straight tunnels. Some tunnels can only be navigated in one direction. The robot is given a map of the caverns and tunnels which is given as the coordinates of the center of each cavern, plus a binary matrix showing which caverns can be reached from which other caverns.

- The user must be able to indicate the name of the input file, without the file format (".cav");
- The output of the program must be stored in file with the following format: <name_input_file.csn>

- The program must be able to find an optimal path (if the path exists), in less than 1 minutes, satisfying a test case with 5000 caverns involved in our graph.

Algorithm Logic Implementation

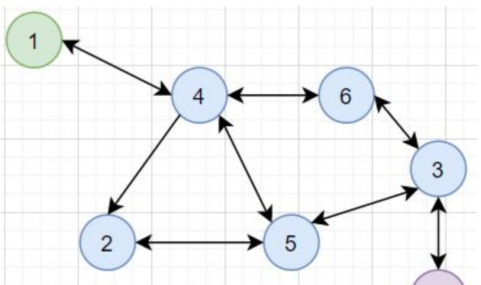
In A* we **don't** have just a brute force approach to search a path, we have an heuristic function that allows us to figure out the best possible path. For starting, we have a starting node, that creates an open set, managed by a priority_queue; The first step is to insert the starting node inside the priority queue; During our path searchin, we will consider the function **F(n) = G(n) + H(n);**

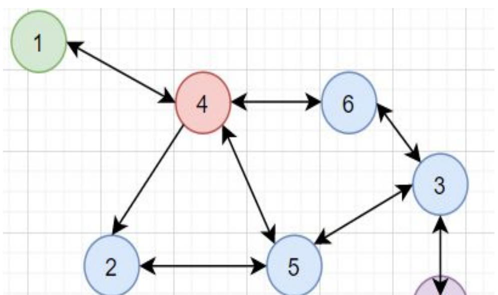
- **H(n):** is the heuristic function that gives an estimate of the distance between the current node and the destination/goal node.
- **G(n):** current shortest distance node between the start node and the destination node;
- **F(n):** A lower F score means that could be closer to our destination;

Euclidean distance =
```
H = sqrt ( (current_cell.x – goal.x)2 +
            (current_cell.y – goal.y)2 )
```

Apply A* In our cavern path search

Consider the node 1 as starting point and node 7 as the designation point. The **Fig.1** represents the initial state of the graph. Here, the starting node is inserted inside the priority queue. Immediately after, the algorithm will move on to node 4**(Fig.2)**, visiting then node 2 **(Fig.3)** and immediately after node 5**(Fig.4).** After this, the algorithm will have to choose between the node 4 and 3; If A* would visit again the node 4 for the second time **(Fig.5**), it will apply the backtracking and go back to the node 5, to avoid visit multiple times the same node; From the node 5, the next visited node will be the node 3**(Fig. 6)**; From node 3, the path finder will visit also the node 7**(Fig. 7)**, considered destination node; At this point, the algorithm will stop to search a path and print the shortest path found.





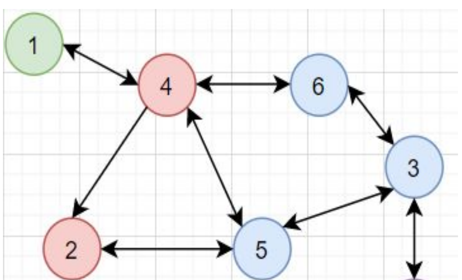
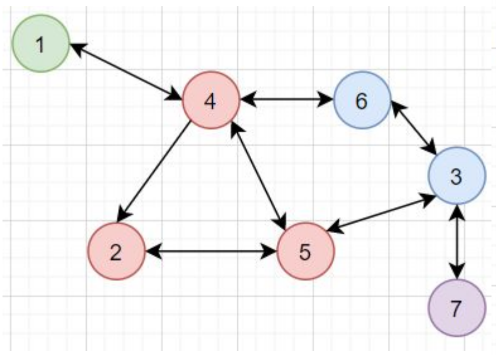
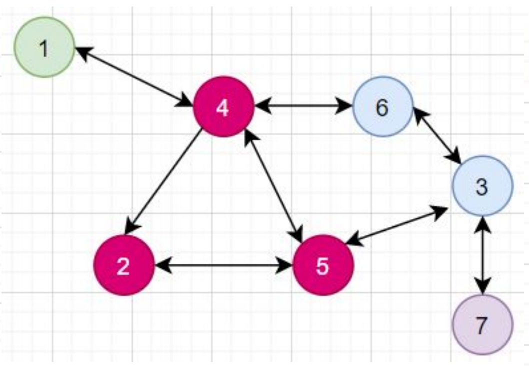


Fig.1 Fig. 2 Fig. 3





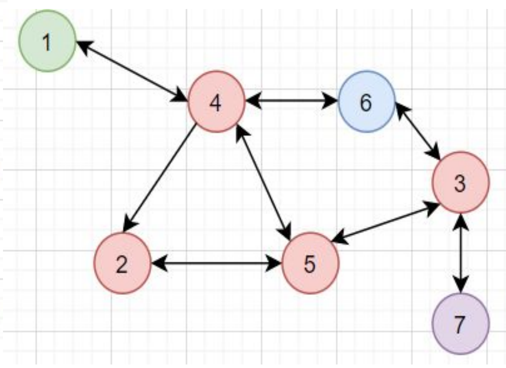
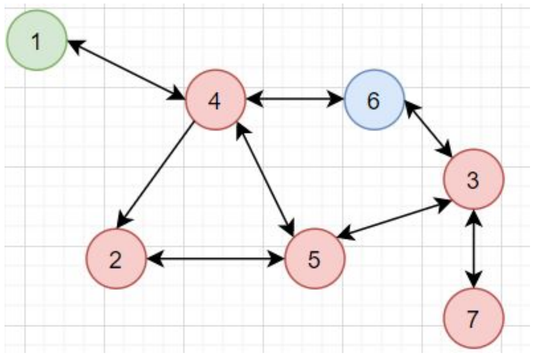


Fig.4 Fig.5 Fig.6



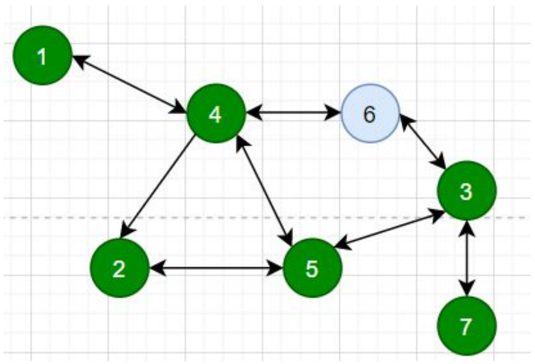


Fig. 5 Fig. 6

Implementation Cavern Path Search

```
public static void main(String[]args){

          List<String> file;

          String content;

          Cave[] caves;

          if(args.length != 1){

              System.err.println("Mind to provide the input file name without it
    extension");
              return;

          }

          String fileName = args[0];
```

```
        Path path = FileSystems.getDefault().getPath("./", fileName +
EXTENSION);
        try {
            file = Files.readAllLines(path);
            content = file.get(0);
        } catch (IOException e) {
            System.err.println("Error while reading file");
            return;
        }
        String[] values = content.split(",");
        int cavesNumber = Integer.parseInt(values[0]);
        caves = new Cave[cavesNumber];
        int offset = 1; // Used to ignore the first cave
        for(int i=0 ; i<cavesNumber ; i++){
            caves[i] = new Cave(Double.parseDouble(values[i*2+offset]),
Double.parseDouble(values[i*2+offset+1]), ""+(i+1));
        }
        offset = 1+2*cavesNumber; // we jump the cave's number and the cave's
locations
        for(int i=offset ; i<values.length ; i++){ // creating the connections
between caves from the matrix
            int startingCaveIndex = (i-1)%cavesNumber;
            int endingCaveIndex = (i-offset) / cavesNumber;
            if(values[i].equals("1")){
                caves[startingCaveIndex].addLink(caves[endingCaveIndex]); // add
connection
            }
        }
        Cave startingCave = caves[0];
        Cave endingCave = caves[cavesNumber-1];
        ArrayList<Cave> toExplore = new ArrayList<>();
        ArrayList<Cave> explored = new ArrayList<>();
        toExplore.add(startingCave); // we start the exploration
        startingCave.setDistanceFromStart(0);
        do{
            currentCave = toExplore.get(0);
            for(Cave link : currentCave.getLinks()){ // we explore each links in
the current cave
```

```
                // if the cave we discover hasn't been explored, we add it to the
exploration list
                if(!explored.contains(link) && link != currentCave){

                    if(!toExplore.contains(link)){

                        toExplore.add(link);

                    }

                    double distanceToThatLink = currentCave.getDistance(link) +


currentCave.getDistanceFromStart();



                    if(distanceToThatLink<link.getDistanceFromStart()){

                        link.setDistanceFromStart(distanceToThatLink);

                        link.setClosestCave(currentCave);

                    }

                }

            }

            toExplore.remove(currentCave); // the current node has been explored,
we don't care about it anymore
            explored.add(currentCave); // so that we don't work on that node
again
            // We sort the exploration list by their distance from the starting
point
            // We can pass from A* to Dijkstra by removing the heuristic

            // The heuristic is the distance from the current cave to the end
cave
 Collections.sort(toExplore, (o1, o2) -> (int) ((o1.getDistanceFromStart() -
o2.getDistanceFromStart());
        }while (currentCave!=endingCave && toExplore.size()>0);

        String solution = "";

        double distance = 0.0;

        if(explored.contains(endingCave)){ // print the shortest path backward

            do{

                solution = currentCave.getId() + " " + solution;

                if(currentCave.getClosestCave()!=null){

                    distance +=
currentCave.getDistance(currentCave.getClosestCave());
                }

            }while((currentCave = currentCave.getClosestCave())!=null);
            }else{

               solution = "0";

         }

            //System.out.println(solution + " - Distance = " + Math.round(distance));

            writeUsingOutputStream(solution.trim(), fileName + ".csn");

        }
```

What we if we will not consider the Euclidean Distance(PART B

In a scenario where the Euclidean distance between any vertex will **not be used** as common metric used to determine the best path between two nodes of the graph, it will not always be possible to use an path finding algorithms based on an heuristic evaluation like A*.

To satisfy high performance requirements, it would be better to use the **Uniform-Cost search algorithm.** The U.C.S (Uniform Cost Search), can be considered as the Dijkstra algorithm for large Graphs.[3]

UCS is a variation of the Dijkstra algorithm, where instead of inserting all vertices into a priority queue, we insert only source (starting node), then one by one insert when needed. In every step, we check if the node already exists in the priority queue to avoid visiting the same node multiple times. This variant of Dijkstra is used for graphs with lots of vertices and edges that are difficult to represent in the memory. The uniform-Cost search is highly used in AI.

Algorithm FOR USC

- Insert Root Node (starting node) into the queue
- Repeat till the queue is not empty
    - Remove the next element with the highest priority from the queue
    - If the node popped from the queue is the destination node, then print the cost and the path and exist
    - Else: insert all the children/neighbours node of the removed elements into the queue with their cumulative costs as their priority;

Complexity of the Uniform Cost Search Algorithm

The time complexity used to find the minimum cost between the start and destination node is: O(m ^ (1+floor(l/e))) where,

- **m** is the maximum number of neighbor a node has
- **l** is the length of the shortest path to the goal state
- **e** is the least cost of an edge

Conclusion of the Uniform Cost Search

Uniform Cost Search is used to find the path from root node to destination node with the lowest cumulative cost in a weighted search space where each node has a different cost of traversal. It is similar to Heuristic Search but no Heuristic information is being stored which means h=0.