Pathfinding in Artificial Intelligence

Louis Boursier
40404293@napier.ac.uk
Edinburgh Napier University - Module Title (SET09122)

Abstract

This document is a report for the first coursework in the artificial intelligence module[1]. For this coursework, we had to develop a path finding algorithm to solve a maze-like problem. The goal is to go from a start cave to an end cave passing in other caves, with the shortest path, and in a reasonable amount of time. The only information we have about the graph is the connection is the coordinates of the nodes. We also had to identify when there is no possible path. In this document, I will introduce three different algorithm, and explain how they perform in general and in this scenario.

Keywords – Louis, Boursier, Artificial, Intelligence, Pathfinding, Dijkstra, A*, Best First Search, BFS

1 Algorithms presentation

1.1 Best First Search

BFS is an informed path finding algorithm, because it uses graph information to perform. It explores graph by expanding the most promising node according to a heuristic. The heuristic is suppose to represent the likelihood of this node being the best choice to reach the goal. Even though the heuristic is specific to the problem, a common one is to use the distance from the goal to assess the score of a node. This means that we have to know some information on the problem to solve in order to figure out a good heuristic. This type of algorithm is called a greedy best first search, or pure heuristic search, because it only uses a heuristic to score a node.

$$score(node) = heuristic(node) \tag{1}$$

1.2 Dijkstra

In this part, we consider Dijkstra as an algorithm to find the shortest path between two node, and not as an algorithm building a shortest-path tree. Contrary to BFS, Dijkstra algorithm score a node by the global cost of all the edges cost to reach the node. Dijkstra is called a uniform cost search, because it does not use a heuristic. Dijkstra is also an informed path finding algorithm.

$$score(node) = totalGraphCost(node) \tag{2}$$

1.3 A*

Here, A* can be seen as a fusion between BFS and Dijkstra. A* is an improved version of Dijkstra thanks to the use of a heuristic, as seen in the BFS algorithm. If the heuristic is admissible, we can be sure that A* is at least as efficient as Dijkstra [2].This means that we should never overestimate the cost to reach the goal in our heuristic. But it will perform better in most of the cases. Also, A* can be seen as Dijkstra algorithm when its heuristic is equal to zero. Due to its good performance, A* is widely use in path finding, games and even natural language processing.

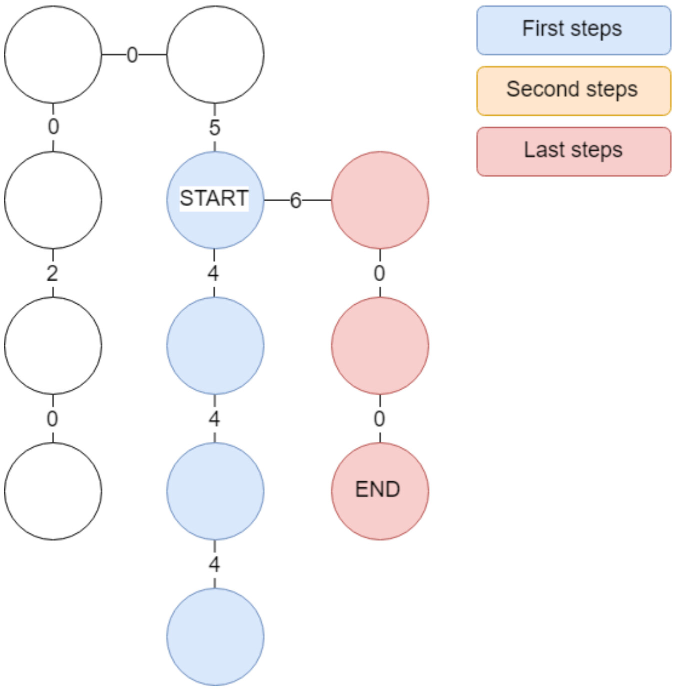

Figure 1: **Best First Seach** - BFS will simply investigate the node for which the heuristic is the best. In this case, the heuristic used is the Manhattan distance. Note that we started by the blue node below the starting node, but we could have chose the red node on the right of the starting node as the Manhattan distance cost is the same (3 units). In that case, the algorithm would have performed better.

$$score(node) = totalGraphCost(node) + heuristic(node) \tag{3}$$

2 Algorithms evaluation

2.1 Best First Search

BFS is a near optimal algorithm as it does no guarantee to find the shortest path. However, it can be advantageous in some situations. For example, in an almost

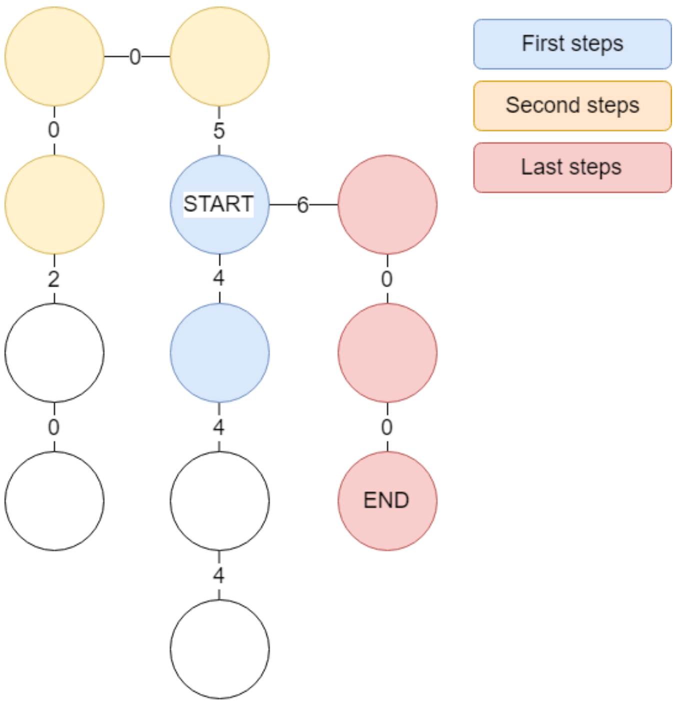

Figure 2: **Dijkstra** - We choose the node for which the cumulative cost is the lowest. Therefore we often change the path to investigate.

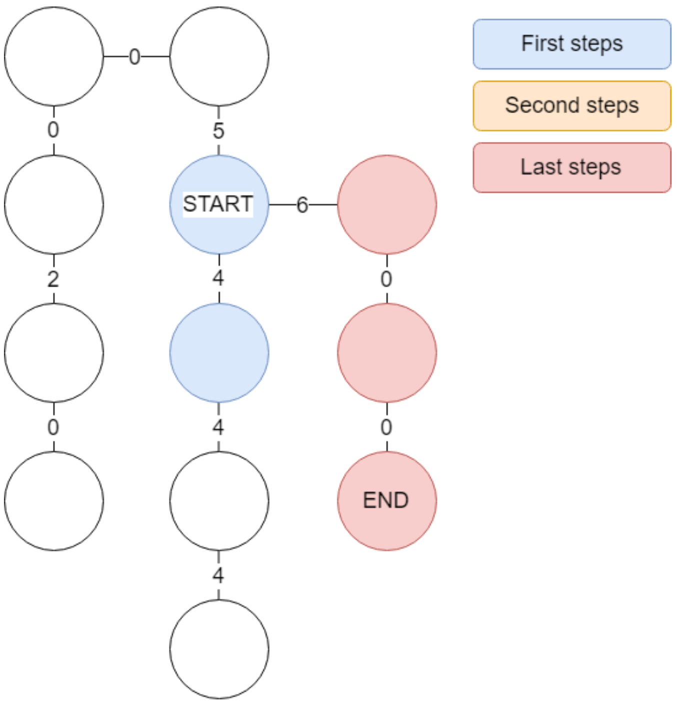

Figure 3: **A*** - We choose the node for which the sum of the cumulative cost and the heuristic is the lowest. We can see that A* takes advantage of the cumulative cost from Dijkstra, and the heuristic of BFS.

unobstructed environment, BFS will quickly find a solution because it will directly look towards the goal. But in a trap situation, BFS can lose a lot of time to persevere investigating a dead end.

In our scenario, the more possibilities to reach a goal there will be, the better BFS will perform. In the same idea, if the path from the start to the end is unobstructed, BFS will perform perfectly. On the other hand, if the scenario is design as a trap with dead end, BFS will perform badly. In general, it is likely that BFS perform better than Dijkstra, and even A* in some cases, in terms of time and memory used. But the path is unlikely to be optimal.

2.2 Dijkstra

Contrary to BFS, Dijsktra does not persevere in a dead end because it takes the total cost of a path to a node in consideration. This means that whenever the current node is further than an other node, the farthest node will be set aside. On the other hand, Dijsktra can do some useless move by investigating the closest node first, even though this node is at the opposite direction of the goal. This is due to a lack of heuristic in the algorithm.

In our scenario, Dijkstra will of course always find the shortest path if it exists. But it can lose a lot of time and memory by exploring useless nodes. The more open the graph will be, the worse Dijkstra will perform. However, for small problem like in our scenario, Dijkstra can found a solution in a reasonable amount of time. Dijkstra will always perform worse or equal to A*, because in this scenario, I have used the euclidean distance to the end for the heuristic. And this is an admissible heuristic.

2.3 A*

As said before, A* will always find the shortest path, faster or in the same time than Dijsktra. Because of the use of a heuristic, the space complexity is larger than Dijkstra however.

In our scenario, it is the algorithm I chose for my program. Indeed, it is a good compromise between the intelligent heuristic and the consideration of the total cost of the path used.

References

[1] E. N. University, “Artificial Intelligence SET09122 ,” 2018.

[2] P. J. Rina Dechte, “Generalized best-first search strategies and the optimality of a*,” *Journal of the ACM.*