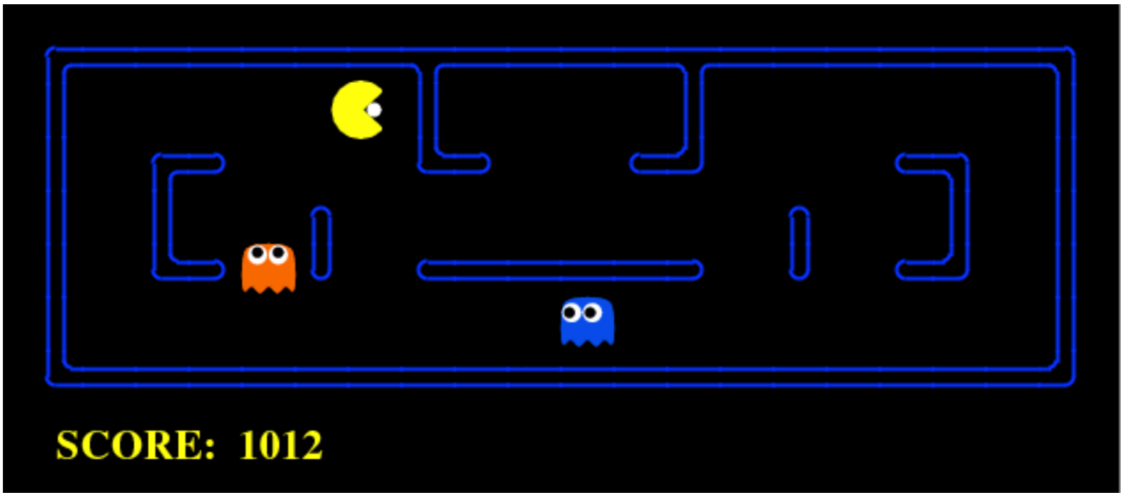
# FINAL PROJECT ASSIGNMENT

## PATH SEARCH FOR THE PAC-MAN GAME

*Artificial Intelligence*

---




**Rafael Morales Palacios - 100429019**
**Jordi Pérez Pastor - 100429127**

# INDEX

**Understanding search algorithms** **3**
tinyMazeSearch(problem) 3
generalGraphSearch(problem, structure, greedy=False) 3
depthFirstSearch(problem) 3
breadthFirstSearch(problem) 3
uniformCostSearch(problem) 3
nullHeuristic(state, problem=None) 3
aStarSearch(problem, heuristic=nullHeuristic) 3

**Problem 1: Finding a fixed food dot** **4**
Understand the code 4
Execution analysis 5
Execution time 5
Expanded nodes 5
Path cost 6
Maze where depth-first search finds optimal solution expanding less nodes than A* (with Manhattan distance heuristic) 7
Maze where breadth-first search finds optimal solution expanding less nodes than A* (with Manhattan distance heuristic) 7

**Problem 2: Eating all the food dots** **8**
SEARCH AGENTS 8
AGENT 1 8
AGENT 2 8
AGENT 3 8
COMPARING BEHAVIOR 8
Execution time 9
Expanded nodes 10
Path cost 11

**Conclusion** **12**

**Personal comments** **12**

# 1. Understanding search algorithms

**tinyMazeSearch(problem)**

This algorithm returns directly the solution for the search problem in the predefined tiny maze layout.

**generalGraphSearch(problem, structure, greedy=False)**

This algorithm is used by the different search algorithms possibilities in order to compute their solution. To do this, it needs the search problem to be solved and a data structure which can use the functions .push() and .pop() (i.e. a stack or a queue). First, the algorithm pushes into the structure the root node of the problem as well as the action taken (as it is the first “Stop”) and the cost to get to that node from the root (0 in this case). A loop is done until the structure is empty where the path is going to be evaluated. If the current state is the goal state, then the function returns the actions to get from the root node to that state; if not, then it checks whether the current node has been visited already or not (that means if it is in the “visited nodes” list). If it has not been visited, then it is marked as visited and the algorithm evaluates its successors. For each of them, the path to get to them is pushed into the structure to be evaluated. If the search fails, the algorithm returns False as error.

**depthFirstSearch(problem)**

This algorithm is for searching. The algorithm starts at the root node and explores as far as possible each branch before backtracking. It does this process until it gets to the goal node.

It uses the generalGraphSearch as a stack structure.

**breadthFirstSearch(problem)**

This algorithm is for searching. The algorithm starts at the root node and explores all the nodes at the same depth level. Onces it finishes with all the nodes that are from the same level it continues with the next level of depth.

Usually it follows an order when exploring nodes of the same level as alphabetical or numerical order.

**uniformCostSearch(problem)**

This algorithm is for searching using Dijkstra. It computes the least cost path for the problem starting from the root node and ending in a goal state. To do that it uses the generalGraphSearch algorithm with a queue that takes priority into account.

**nullHeuristic(state, problem=None)**

It returns value 0 receiving the arguments state and problem (which is set as None). It is only used for the node before the goal.

**aStarSearch(problem, heuristic=nullHeuristic)**

This algorithm is for searching by taking the nodes with the least cost of the sum between the heuristic and regular cost. Two functions are defined to track these costs and the search problem is solved by using the generalGraphSearch algorithm with a queue that takes priority into account.

# 2. Problem 1: Finding a fixed food dot

**Understand the code**

The state space of the problem is defined by the position in the map of the agent in a tuple (x,y).
The initial state is the position where the agent starts, which depends on the start parameter defined before solving the search problem. The goal state is the position in the map where the fixed food dot is located, which is also defined before solving the search problem.
The operators for this problem are: getSuccessors, move, isGoalState, getCostOfActions.

getSuccessors
Conditions of applicability: state is valid and is not part of the walls
Results of application: list of valid successors (which are not walls) with the actions to be taken to get there and the action cost

move
Conditions of applicability: state is valid and is not part of the walls
Results of application: move agent to valid successor (which are not walls)

isGoalState
Conditions of applicability: position of agent is equal to the position of goal
Results of application: goal=True if it is equal and goal=False if the position of the agent is not the goal.

getCostOfActions
Conditions of applicability: get the list of actions in the path
Results of application: if the actions are not valid returns max cost, if they are valid, return the cost of the path following those actions.

Regarding the four regular mazes, they were designed in different sizes with different possible paths to get to the goal.

MAZE1
The first maze is the smallest of them and there is only one way for the pacman to access the area where the goal is.

MAZE2
For the second maze we design a bigger map with different possible paths.

MAZE3
This maze is similar to the second one but slightly bigger.

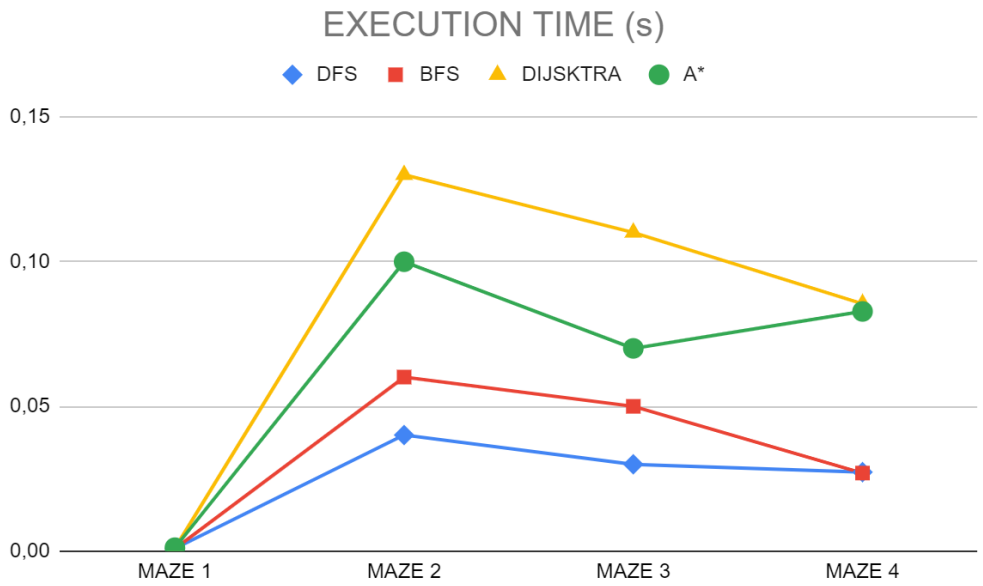
MAZE4
This maze is the biggest of them and there are several paths for the pac-man to get to the goal.

**Execution analysis**

For this section of the project we executed the different search algorithms for each of the mazes and took note of the results that the program returned.

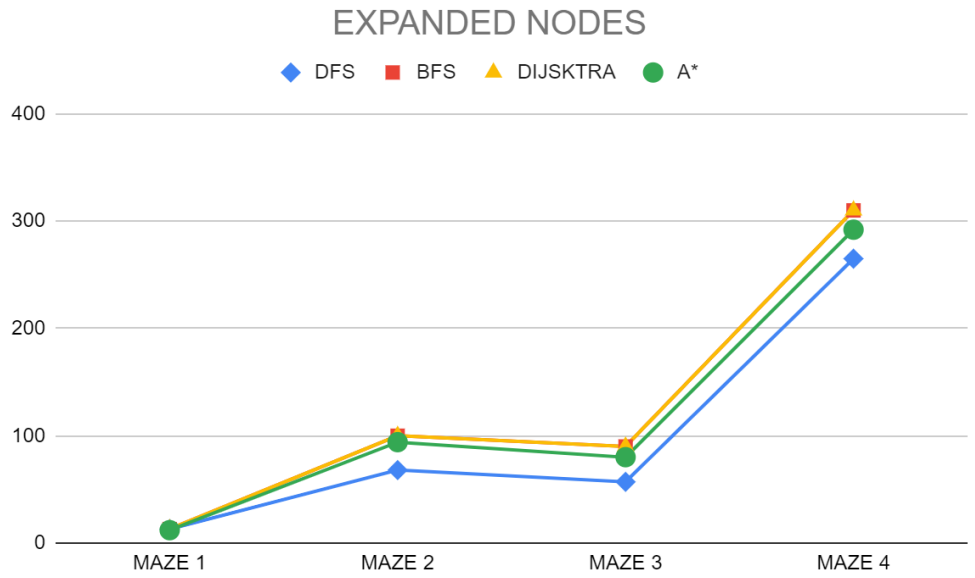
- **Execution time**

  This part measures the time that each algorithm took to obtain the solution for each of the mazes. We can see that for the first maze, all the algorithms took very few time (less than 0.002 seconds), being the one that took the smallest time to complete as expected. The second one was the maze that took more time for each one of the algorithms, which is interesting to compare with the execution of the third maze because, as explained before, they are similar but maze3 is bigger. In the fourth algorithm the execution times are almost the same for the DFS and BFS and for Dijsktra and A*.



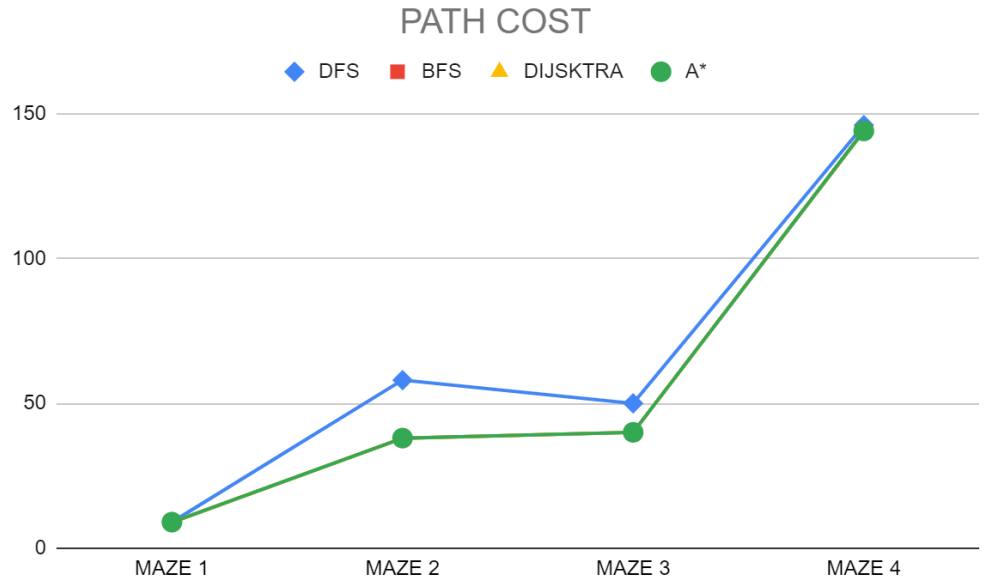

- **Expanded nodes**

  Regarding the nodes expanded for each of the algorithms in the mazes designed. The graph shows a relation between the size of the mazes and the expanded nodes (the bigger size, the more expanded nodes are needed to find the solution). But this relation breaks when comparing maze2 and maze3 because, in spite of being bigger, it expands some nodes less.




- **Path cost**

  If we take a look at the path cost for each algorithm's execution, we can see that BFS, Dijsktra and A* always get the same cost (for these mazes), which is the optimal. Also, DFS's cost is bigger than the rest for all mazes. If we compare the graphs for expanded nodes and path cost we can see a relation between them as the tendency is to increase with the size, except for maze3.



**Maze where depth-first search finds optimal solution expanding less nodes than A* (with Manhattan distance heuristic)**

The idea of this maze consists of a maze where there are several paths that the Pacman can take to get to the goal, but the optimal one is the first. That is why the DFS algorithm expands less nodes, it takes that path directly without checking the rest as A* would do during its execution to compare different possibilities.
This maze is included in the layouts-student folder as maze5.lay

| MAZE 5 | | | |
|---|---|---|---|
| ALGORITHM | EXECUTION TIME (s) | EXPANDED NODES | PATH COST |
| DFS | 0,00292 | 31 | 31 |
| A* | 0,01163 | 72 | 31 |

**Maze where breadth-first search finds optimal solution expanding less nodes than A* (with Manhattan distance heuristic)**

This maze is impossible to design because in the best case, both algorithms will expand the same number of nodes. This is due to the fact that the BFS is a particular case of the A* which there is no heuristic.
The algorithm A* (with Manhattan distance heuristic) is the efficient way we know to obtain the optimal path in terms of expanding nodes.

# 3. Problem 2: Eating all the food dots

The state space of the problem can be defined by the position of the Pacman and the food grid containing a mark in each position that represents whether there is a food dot or not in that place. The Pacman position is a tuple (x,y) that locates the agent in a specific point of the map.
The initial state of the problem is the starting position of the pacman and the food grid having the starting amount of food dots marked.

**SEARCH AGENTS**

- **AGENT 1**

  This agent follows an A* algorithm where the state is defined as the position of the Pacman in the maze and the map of foods in the maze. The heuristic used is the largest distance between the position of the Pacman and each remaining food using Manhattan distance. The heuristic is not admissible because it does not take into account that the Pacman can eat other food dots while getting to that other location.

- **AGENT 2**

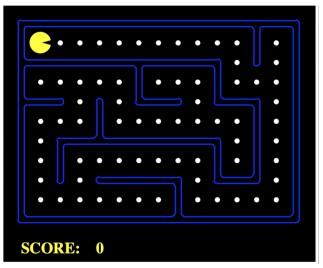
  Agent 2 follows an A* algorithm similar to the one in Agent 1. In this case, the heuristic used is also the largest distance between the position of the Pacman and each remaining food but using the real distance in the maze (taking into account walls). The heuristic here has the same problem as in Agent 1: it is not admissible because it does not take into account that the Pacman can eat other food dots while getting to that other location.

- **AGENT 3**

  This agent follows a path based on an algorithm that takes the shortest path from the agent to any food. This path will probably take less time to be computed than the other agents' path but it is not guaranteed to be the shortest.

**COMPARING BEHAVIOR**

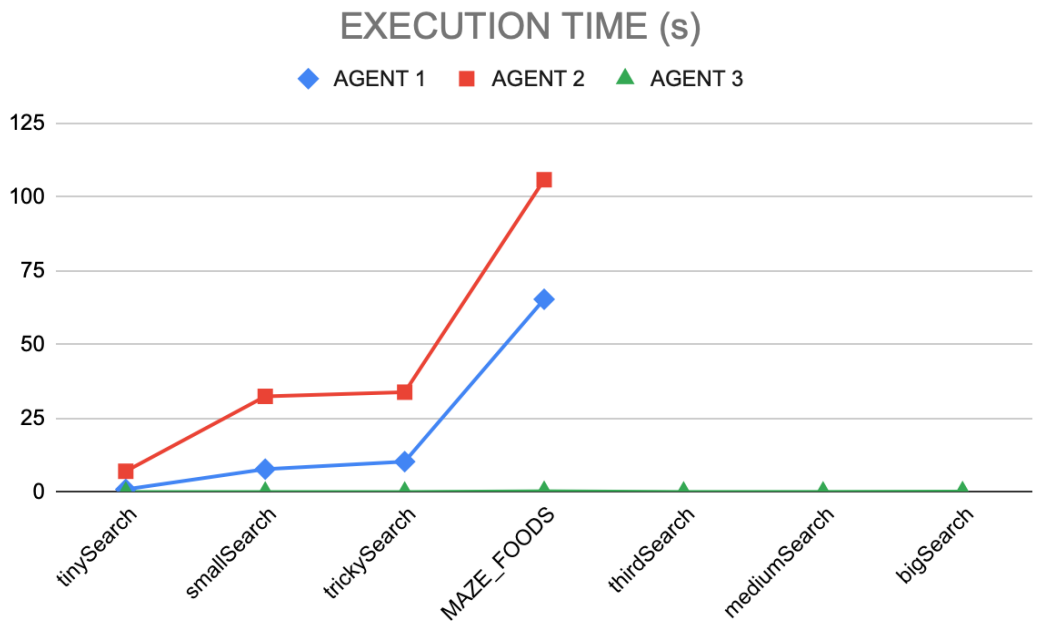
We have made our own maze which we have called "MAZE_FOODS" (MAZE_FOODS.lay); with size 11x15 filled up with food-dots. This maze is between trickySearch and thirdSearch in terms of size.



- **Execution time**
  In this table we can see the execution time for each of the three agents depending on the mazes provided (as well as an extra one we designed) ordered from smallest to biggest. There is a tendency for the execution time to increase with the size of the maze where they are tested, which makes sense if we take into account that the bigger the maze is the more food dots are in these cases (except for smallSearch and trickySearch mazes, where there are a similar amount of food dots). It is important to remark that in the case of medium to big mazes the values for Agent 1 and Agent 2 could not be calculated as for these agents the time to finish the path was extremely large.

| EXECUTION TIME (s) | | | |
|---|---|---|---|
| MAZE | AGENT 1 | AGENT 2 | AGENT 3 |
| tinySearch | 0,90424 | 7,03279 | 0,01149 |
| smallSearch | 7,74506 | 32,41001 | 0,01694 |
| trickySearch | 10,29100 | 33,82800 | 0,01299 |
| MAZE_FOODS | 65,30991 | 105,81095 | 0,26347 |
| thirdSearch | * | * | 0,03790 |
| mediumSearch | * | * | 0,05697 |
| bigSearch | * | * | 0,16200 |

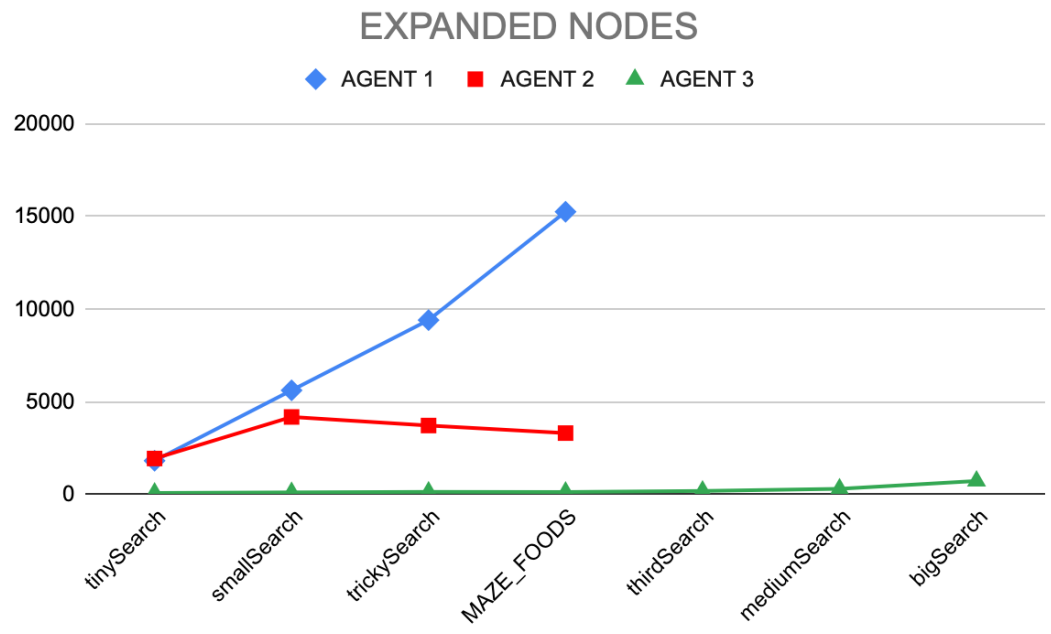
* This value has not been calculated because the process would take a very big amount of time (hours or days…)



- **Expanded nodes**
  Regarding the nodes expanded for each execution of the agents in each maze we can see that, while for agents 1 and 3 they increase in number with the size of the maze, for agent 2 it does not. This happens because agent 2 takes into account the walls of the maze to calculate its heuristic and discriminates them from the actual path and the other agents do not take walls into account so the number of actual expanded nodes is not reduced by discriminating walls.

| EXPANDED NODES | | | |
|---|---|---|---|
| MAZE | AGENT 1 | AGENT 2 | AGENT 3 |
| tinySearch | 1812 | 1932 | 73 |
| smallSearch | 5611 | 4175 | 105 |
| trickySearch | 9402 | 3712 | 132 |
| MAZE_FOODS | 15251 | 3304 | 124 |
| thirdSearch | * | * | 181 |
| mediumSearch | * | * | 296 |
| bigSearch | * | * | 719 |

* This value has not been calculated because the process would take a very big amount of time (hours or days…)



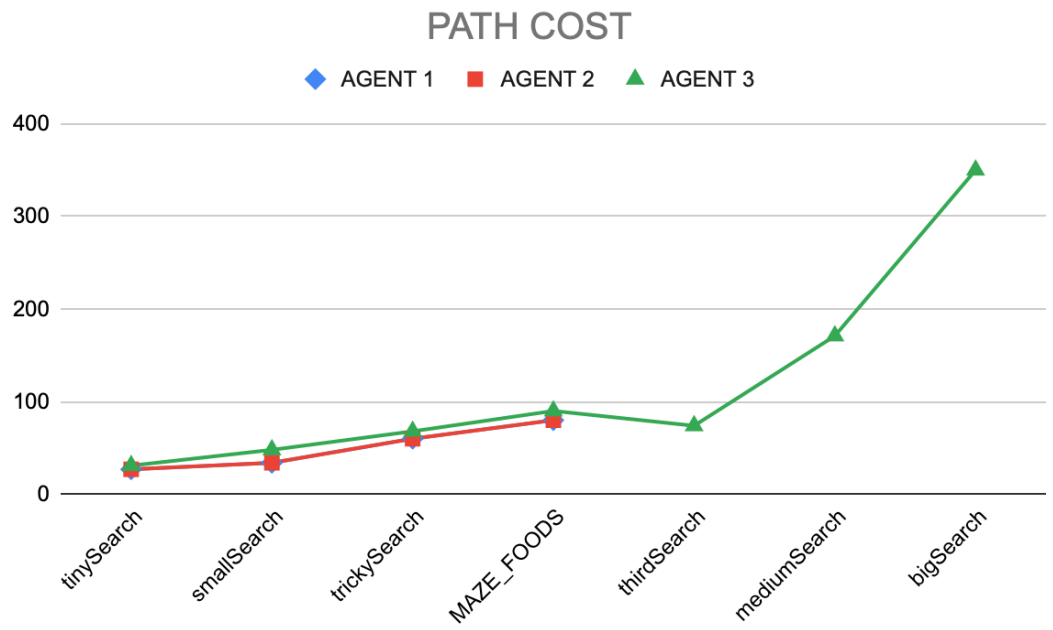
- **Path cost**

  In the following table we have the path cost for each of the three agents applied in all the mazes, ordered by size from smallest to biggest. As we can appreciate, the cost for the agent 1 and 2 are always equal, because, as we have explained before, both agents find the optimal path, so as the path is equal, the costs are equal too.

  On the other hand, we have agent 3 that his path normally costs more than the other agents. This is because this agent is not always returning the optimal path. Maybe in some cases the path returned is optimal, in which case, the path cost would be the same in all agents.

| PATH COST | | | |
|---|---|---|---|
| MAZE | AGENT 1 | AGENT 2 | AGENT 3 |
| tinySearch | 27 | 27 | 31 |
| smallSearch | 34 | 34 | 48 |
| trickySearch | 60 | 60 | 68 |
| MAZE_FOODS | 80 | 80 | 90 |
| thirdSearch | * | * | 74 |
| mediumSearch | * | * | 171 |
| bigSearch | * | * | 350 |

* This value has not been calculated because the process would take a very big amount of time (hours or days).




The agent 1 and agent 2 have the same plot because as we explained before, they have the same path cost.

# 4. Conclusion

Overall we can say that, even though we initially found some problems understanding the code for the calculation of the paths, we managed to finish the project successfully.

Also, in the development of the second problem we tried to use the agents 1 and 2 for mazes thirdSearch, mediumSearch and bigSearch but, due to the algorithms' execution time being huge (more than 4 hours), we could not get any result and we so we tried smaller mazes.

It is also remarkable that we designed a custom maze (*MAZE_FOODS*) for that last exercise, which all agents were able to solve correctly.

# 5. Personal comments

After all, we think that we have learned how to understand code from other people and how to test it as well as improved our understanding of Artificial Intelligence algorithms and its applications.