Dominik Czech 100529977

Search in the PACMAN Environment

1. Introduction

This project explores the implementation and empirical analysis of various search algorithms within the classic Pacman environment. Using the knowledge I gathered during the Artificial Intelligence course I will test how these algorithms perform in guiding the legendary yellow icon - Pac-Man through various mazes to achieve his ultimate goal: food dots.

The environment used for purposes of this work is a modified version of the Pacman software provided by UC Berkeley (http://ai.berkeley.edu). Which we will further modify in part 2 of the project.

This project report will be split into 4 parts:

1. Introduction
2. Description and explanation of the tasks performed within the project.
3. Conclusions
4. Personal comments

Table of contents

1. Introduction **1**

2. Description and explanation of the tasks performed within the project. **3**

2.1 Understanding of the search algorithms: 3

generalGraphSearch: 3

Search Algorithms: 5

Depth-first search (DFS): 6

Breadth-first search (BFS): 8

Uniform cost search (Dijkstra/UCS): 10

A* algorithm (A star) 12

2.2 Problem 1: Finding a fixed food dot 14

Part 1: Empirical study of the algorithms 14

Analytical explanation of the PositionSearchProblem 14

Comparison and analysis of the behavior of search algorithms 16

Part 2: Slight modification of the problem 41

Level 1: Let’s cut the corner. 43

Level 2: The Corridor again. Remember about the walls! 44

Level 3: They all can see it. What about the corner? 47

Level 4: The Caves. Will they all find the optimal way? 50

Comparison of the A* heuristics for diagonal movement problem. 53

Is the Manhattan distance always the way to go then? Will it always find the best solution? The optimality problem. 55

2.3 Problem 2: Eating all the food dots. 57

2.3.1 Understanding of the implementation of the problem: 58

2.3.2 Analytical explanation of the FoodSearchProblem: 58

2.3.3 Understanding the algorithms used by agents to solve the problem: 60

Summary of the agents and the predictions of performance: 66

2.3.4 Comparison of the behavior of the searchAgents. 67

Level 1: TinySearch maze. Outstanding performance. 67

Side quest: Optimizing the mazeDistance heuristic. 68

Level 2: Small search. Expanding the size of the maze. 72

Level 3: Open space, open search. 75

Level 4: TrickySearch. Ouch, there is a wall. 76

Level 5: Greadiness. The problem of agent 3. 79

Level 6: Final boss. The bigSearch maze. 82

2. Description and explanation of the tasks performed within the project.

This section will be split into 3 parts:

- Understanding of the search algorithms:
    - depth-first search
    - breadth-first search
    - Dijkstra (uniform cost search)
    - A* with Euclidean distance as a heuristic function
- Problem 1: Finding a fixed dot
    - empirical study of algorithms
    - slight problem modification (introduction of diagonal movement and a new admissible heuristic)
- Problem 2: Eating all the food dots
    - understanding of the code that implements the problem to solve
    - analytical explanation of the implementation
    - comparison of the behavior of three different search agents

2.1 Understanding of the search algorithms:

To understand the given algorithms, we need to analyze their structure. All of them revolve around the *generalGraphSearch* function, let's take a look at its structure.

generalGraphSearch:

a) Introduction:

The generalGraphSearch is the foundational framework used by the search algorithms to explore the search space of the given problem. It is highly adaptable, the performance of generalGraphSearch and the order in which the states of the problem are explored depends on the choice of the data structure provided in the parameters.

b) Inputs:

- problem - represents the search problem that needs to be solved, has to be an instance of SearchProblem class
- structure - represents the data structure used for the exploration of the problem, it has to support methods push() and pop()
- greedy - an optional parameter, used only for the A* search algorithm, indicates if the search should be greedy or not (false by default)

c) Initialization:

- pushes the start state of the problem into the structure
- initializes an empty list of nodes that have already been visited

d) Loop operation:

This is the core mechanism of the function. It is based around a while statement, checking if there are still elements remaining to be explored. The loop has an additional exit case, if we find the goal state. The performance of this function depends on the structure given in the parameters (the structure is strictly related to the algorithms).

```
#the following pseudocode follows the Python indexation of arrays, that is
the first element of the array can be accessed by index 0 and the last by
index -1


#START OF LOOP
#__________________________________________________________________________
While structure is not empty:

      path = structure.pop() #retrieve the path from the data structure

      #depending on the data structure, the popped path might be the
      #first explored one (breadth-first-search)
      #last explored one (depth-first-search)
      #one with the lowest cost of actions needed to achieve it (Djikstra)
      #one with the lowest sum of the cost of actions needed to achieve it
      #and the heuristics function (A*)

      #path is a list of tuples with the following elements:
      #path[x] = (state_x, action, cost)
      #state is represented as a tuple of numbers (position_x, position_y)

      curr_state = path[-1][0] #retrieve the current state of the problem


      if (isGoalState(curr_state)): #checking if current_state is goal


      #once we have found a goal state, we return the actions processed to
      #achieve it and we exit the function

            actions = [] #initialize the list of actions

            #for every tuple in the path
            for i in 0..length(path[i])-1:

                  #append the action to the list of actions
                  actions.append(path[i][1])
                  #path[i][1] = (state_i, action, cost)[1] = action
            return actions #return the list of actions and exit the
function

      #if the current_state isn't the goal state we check if it has been
already visited in other path to avoid redundant calculations

      if (curr_state not in visited):

            #mark that the state has been visited
            visited.append(current_state)

            #iterate through every possible successor of the current state
            for (successor in getSuccessors(curr_state)):

            #if the successor hasn't been visited in a different path
                  if(successor not in visited):

                        #copy the path to the successor
                        succ_path = copy(path)
                        succ_path.append(successor)

                        #push the path into the structure
                        structure.push(succ_path)

#__________________________________________________________________________
#END OF LOOP
```

e) Outputs:

- If the goal state was found, the function returns the path from the start state to the goal state
- If the goal state wasn't found, the function returns False

Search Algorithms:

Now, as we have understood the generalGraphSearch, we can go in depth into specific search algorithms. In this subsection I will explain the idea behind the algorithms and the way the algorithm got implemented.

Depth-first search (DFS):

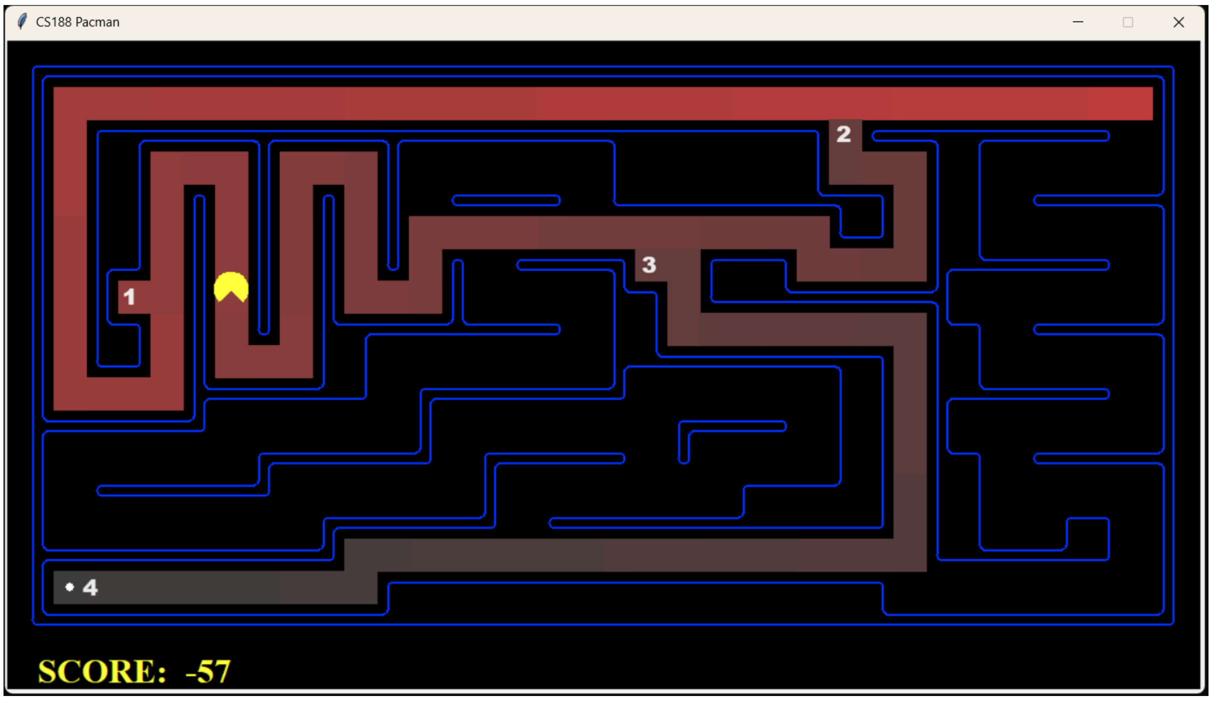


Figure 1: Depth-first search in a medium sized maze

a) Introduction

Depth-first search is one of the fundamental search algorithms used to solve graph traversal problems. It revolves around following a certain path of a problem until the solution is found or the new possible branches of the path are exhausted (there are no more of them).

Figure 1 can help us understand the order of exploration of the states of the search problem. As we can see, there were little to no branches of the path explored (namely 4). The first one has finished, because it found a dead end (number 1), second one stumbled upon an already explored state (number 2), third one got stuck in the corner (number 3) and finally, the fourth one has found the goal state (number 4). From the following heat-map we can also find out that the priority for exploring successors in the DFS algorithm is horizontal first and vertical second, however there is not a single point where we can see if going west has higher priority than going east, neither for north and south.

b) Implementation

The DFS algorithm has been implemented using a stack data structure due to the nature of its pop() and push() functions.

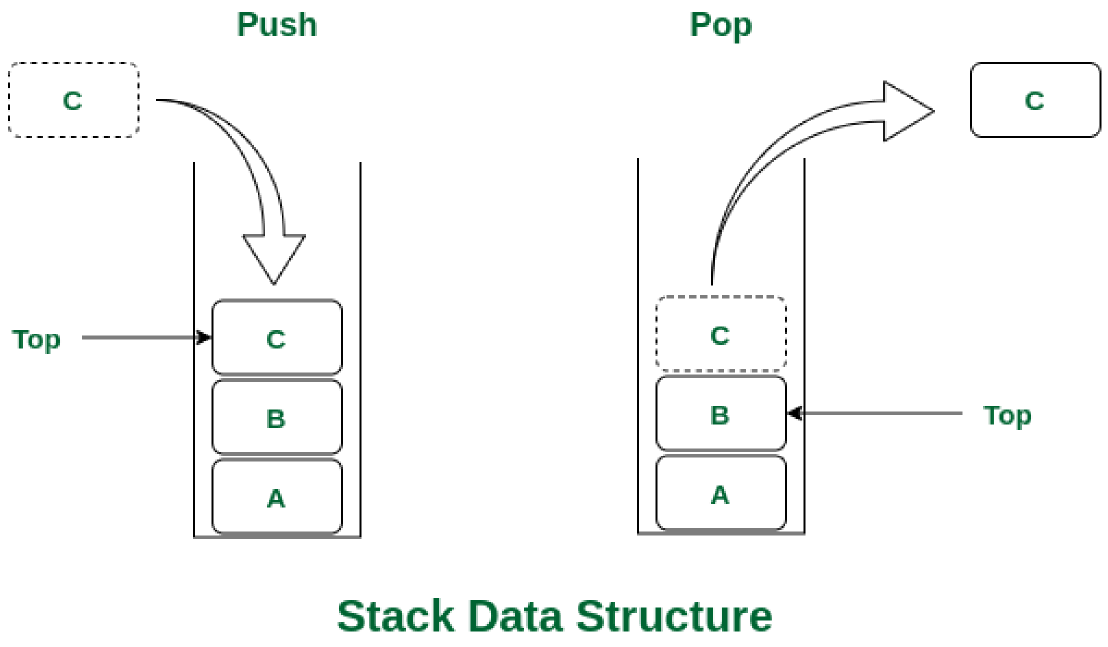


Figure 2: Stack Data Structure.
Source: https://www.geeksforgeeks.org/stack-meaning-in-dsa/

Push() pushes the new element on top of the stack, while pop() retrieves the element from the top and deletes it from the stack. Therefore in the generalSearchAlgorithm, the latest successors of the currently explored state are pushed to the top of the stack and the path with the last successor pushed is being retrieved, therefore the explored path is the last successor of the previous state, meaning it traverses depth-first.

c) Advantages and disadvantages

Likewise every algorithm DFS has its pros and cons, here are some of them:

Advantages:

- Low memory usage: As the algorithm focuses on one path and follows it until the end, the amount of paths in the memory is significantly lower than compared to BFS, especially in mazes without too many branches
- Great in traversing deep structures: Due to its nature, DFS finds paths requiring going deep into the maze way faster than BFS (with less memory required too)

Disadvantages:

- Inadmissible: Doesn't always find the optimal path (especially in mazes with big amount of branches)
- Might take a lot of time to reach the goal state if the goal was close to the start, but the branch that lead to the final state wasn't prioritized by the algorithm
- Incompleteness: Might never find a solution if the maze is infinite or big enough it will explore one way until the end of the memory space

Breadth-first search (BFS):

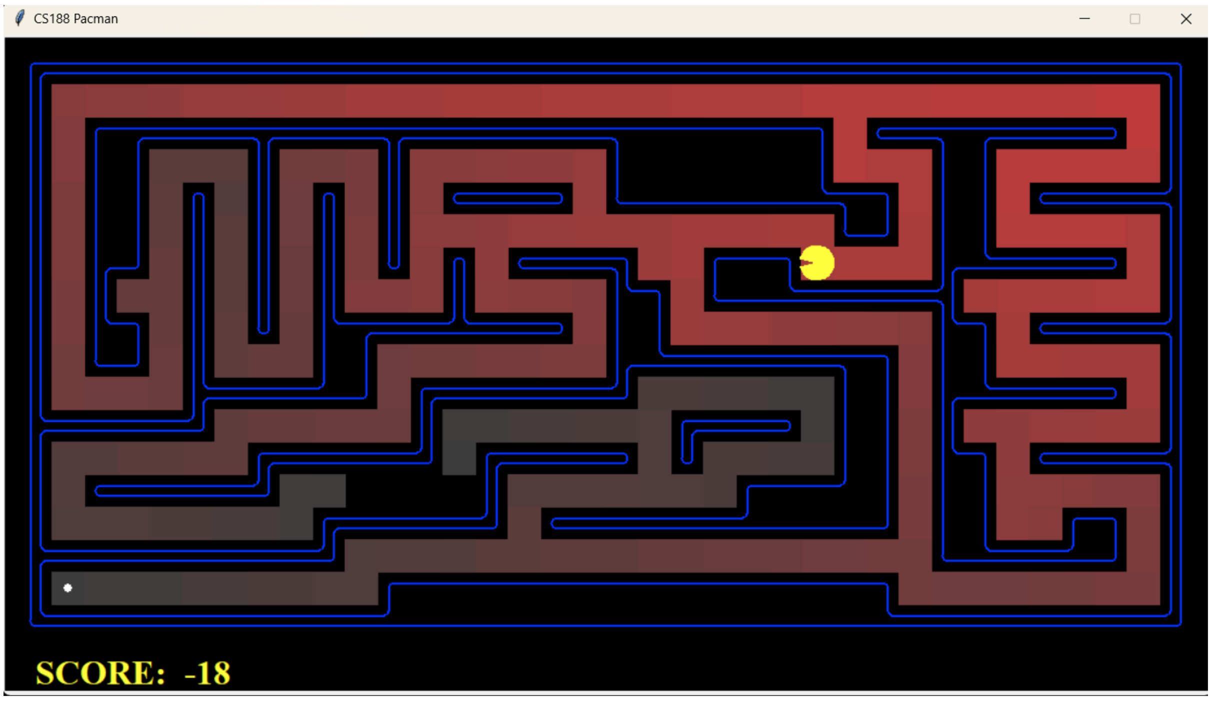


Figure 3: Breadth-first search in a medium sized maze

a) Introduction

Breadth-first search is the core algorithm when it comes to finding the shortest path in not waged graph search problems. It starts with the initial node at depth 0 and explores all of its successors (which are depth 1). The next iterations go through all the nodes at depth 2 and so on. This way it is guaranteed to find the shortest path to the final state, however it requires a lot of different paths to be explored at once.

In Figure 3 we can see that only available game states haven't been explored before finding the shortest path. By looking at the heat map we can very well estimate the actual cost required to travel to the explored nodes (assuming that all the costs of traversing from node to adjacent node are equal).

b) Implementation

The BFS algorithm has been implemented using the Queue data structure. Queues allow us to push (enqueue) a new element at the rear of the queue and pop (dequeue) the element from the front. That way every new successor is getting pushed to the end of the data structure, therefore the first enqueued one is also getting explored first and we can perform a breadth-first search.

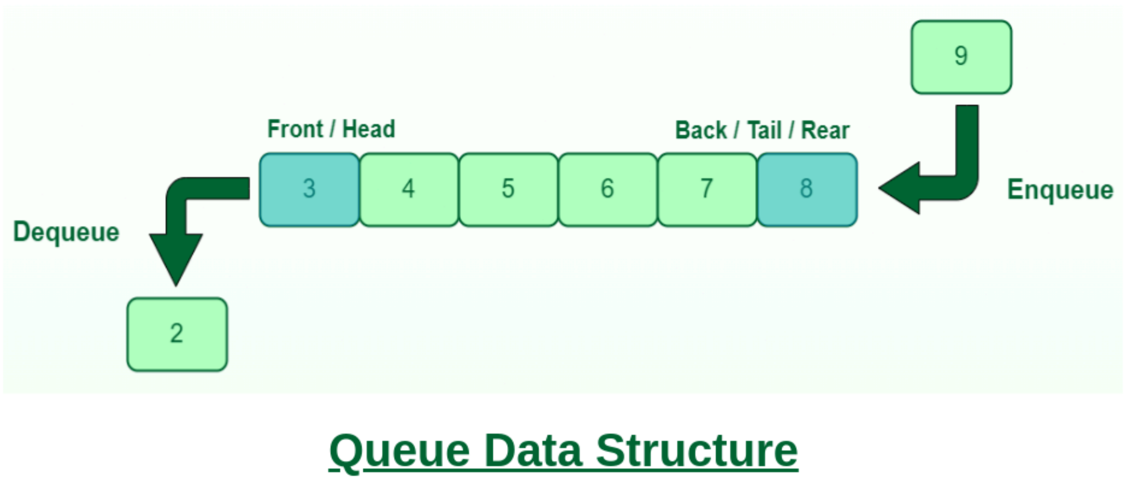


Figure 4: Queue Data Structure
Source: https://www.geeksforgeeks.org/what-is-queue-data-structure/

c) Advantages and Disadvantages

Breadth-first search is pretty much the opposite of DFS, the strong sides of DFS will be the weak sides of BFS and vice versa.

Advantages:
- Admissible: always finds the shortest path to the goal state if the cost of every action is equal.
- Completeness: always finds a path to the goal state if there is one
- Predictable: the behavior of DFS is predictable, the algorithm will behave more or less the same, no matter how complex is the maze or how many branches it has

Disadvantages:
- Doesn't take into the consideration the weights of the actions, only the adjacency
- High memory usage: Since each level of the search saves a different path, and BFS explores all the discovered nodes, it stores a huge amount of different paths at the same time. In big problems we might run out of memory.

- Bad in traversing deep structures: If the goal state is deep into the maze BFS has to expand most of the nodes to find a solution
-

Uniform cost search (Dijkstra/UCS):

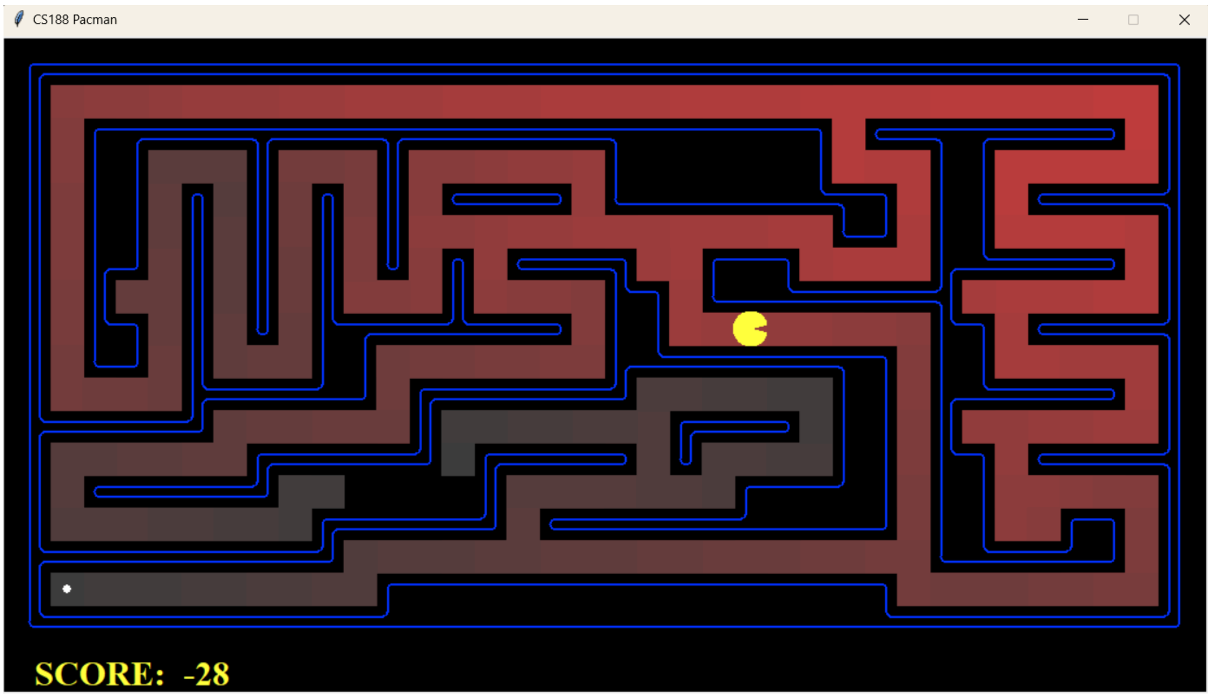


Figure 5: Dijkstra algorithm in a medium sized maze

a) Introduction

The Dijkstra algorithm serves the same purpose as the breadth-first search, but for weighted graph searches. It starts the search in the root node and then expands the nodes with the lowest cost of the path to get to them. If the algorithm will find a shorter way to get to a node already connected the cost of the path to the node will be updated to be the lower one.

Both heat map and the path taken by pacman on figure 5 are the same as the one in the breadth-first search, because if the costs of each transition between nodes are the same (in this case every action has a cost of 1) it acts the same way as the BFS.

b) Implementation

The UCS has been implemented with the help of a Priority Queue based on a heap implemented over a list using the *heapq* library functions. It uses the heappush function to append elements to the heap, keeping it invariant. The priority used in the function is the cost, the lowest cost element being popped the first. The implemented data structure also has an update function which is supposed to update the cost of an already inserted element, however the generalGraphSearch is only using the pop() and push() functions.

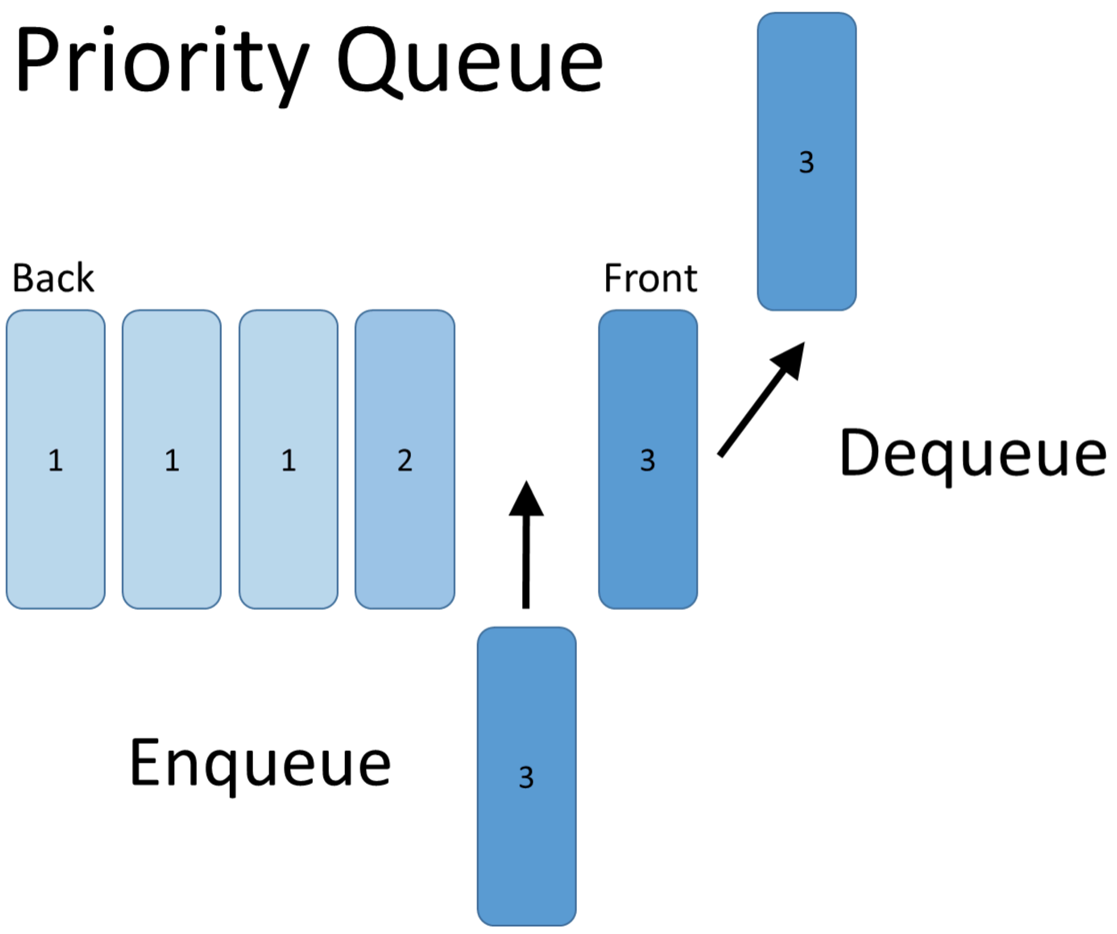


Figure 6: Priority Queue
Source: https://netmatze.wordpress.com/tag/data-structure/

c) Advantages and Disadvantages

Since Dijkstra algorithm is a solution to the problem of weighted costs of actions of the BFS algorithm, it shares most of the advantages and disadvantages with BFS:

Advantages:
- Admissible - even if the cost of the actions is not uniform
- Completeness

Disadvantages:
- High memory usage
- Bad in traversing deep structures
- Requires a data structure supporting priority - more complex than DFS

A* algorithm (A star)

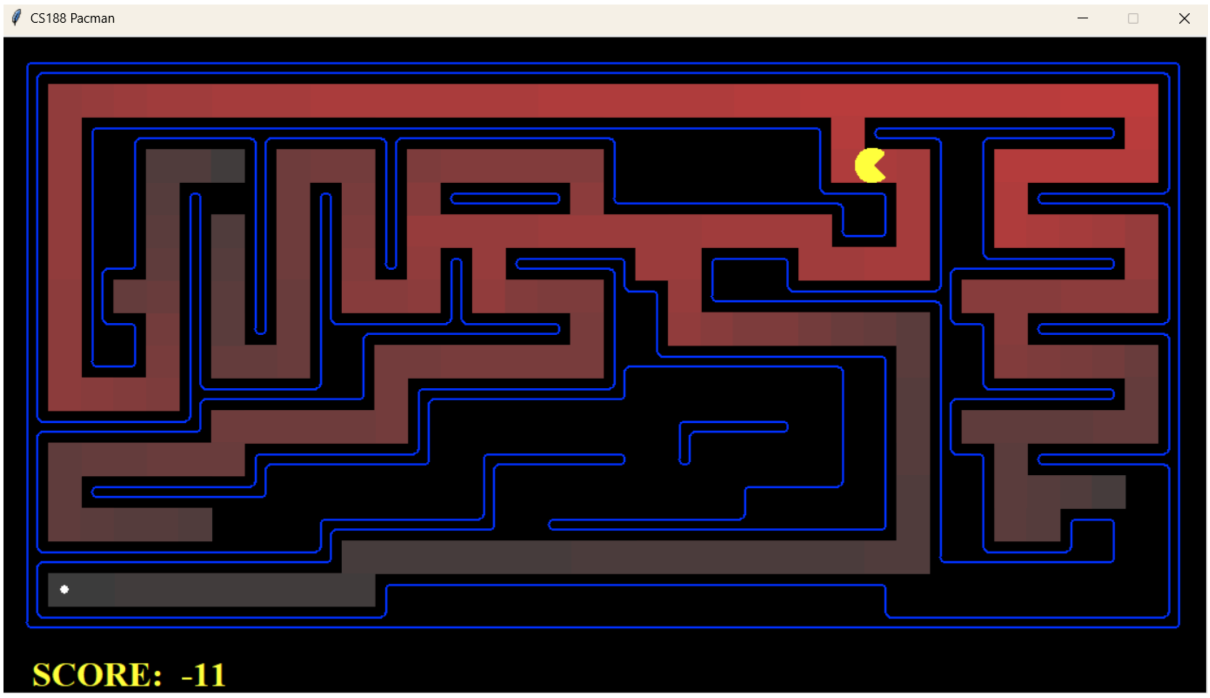


Figure 6: A* with Euclidean distance as a heuristic in the medium sized maze

a) Introduction

A* Algorithm is a widely adopted search known for its efficiency and effectiveness in finding the shortest path through graphs. It works similarly to the Dijkstra algorithm in a way that it explores the most promising path options in the first priority. Instead of the cost to get to the next node it uses a combination of the Dijkstra cost with the estimated cost to get to the goal state from the potentially expanded node (heuristics function). If the heuristics function is admissible, that is the estimated cost is lower than the actual cost of the shortest path to get to the goal state.

The function used to calculate the cost used in A* algorithm is:
$f(x) = g(x) + h(x)$
where:
g(x) - cost required to get from the current state to the potentially expanded node
h(x) - heuristics function - estimated cost to get from the potentially expanded node to the goal state

In figure 6 we see that the A* algorithm with Euclidean distance as the heuristics function has found the shortest way to get to the goal state and has expanded less nodes than the BFS and Dijkstra algorithms due to the better 'intuition' of the direction in where the goal node is.

b) Implementation

Likewise the Dijkstra algorithm, the A* has been implemented with a Priority Queue based on a heap. This time though, the priority variable is a list of the priority functions (the main priority function and the tiebreakers). In our case, the main priority function is the cost to get to the node + heuristics function from this node and the tiebreaker is just the heuristic function, because we estimate that this node is closer to the goal state, therefore we assume that we're gonna find the solution faster.

In the generalGraphSearch there is a variable called greedy given as a parameter, however it has no usage in the provided code. A greedy search using heuristics function (Hill Climbing Algorithm) is similar to the Dijkstra algorithm, however instead of the cost needed to get to a state we use just the heuristics function to define which nodes we are going to use first. It could also be implemented on the Priority Queue with function, where:

$f(x) = h(x)$

The Hill Climb Algorithm unfortunately isn't admissible (it doesn't always find the best solution) however it usually finds the solution faster than the A* algorithm.

c) Advantages and Disadvantages

Advantages:

- Efficiency: A* is highly efficient because it intelligently combines the cost to reach a node with the estimation of the cost to reach the goal from that node. That combination allows the algorithm to focus on the most promising paths.
- Optimality: If the heuristic function ($h(x)$) is admissible, the algorithm always returns the shortest path to the goal
- Flexibility: The behavior of A* algorithm might differ significantly, depending on the choice of the heuristic function

Disadvantages:

- Effectiveness and optimality depend on the choice of the heuristic function
- Additional computation required to calculate the heuristic function
- Sometimes it might be hard to find an accurate heuristic function

2.2 Problem 1: Finding a fixed food dot

In this subsection we will focus on a problem of collecting a fixed dot in a maze. We will split it into 2 parts:

- Part 1: empirical study of algorithms
    - understanding of the code that implements problems to solve
    - analytical explanation of the problem
    - comparison and analysis of the behavior of the search algorithms
- Part 2: slight problem modification
    - introduction of diagonal moves
    - implementation of new admissible heuristic
    - comparison of different heuristics using A*

Part 1: Empirical study of the algorithms

Analytical explanation of the PositionSearchProblem

a) State space:

The search state is characterized by:

- Pacman position - the current position of Pacman
- Goal state - position of the food dot
- Walls grid - Coordinates of all the walls
- Pacmans orientation (last action done) - the direction Pacman is facing
- Score - the score of the game

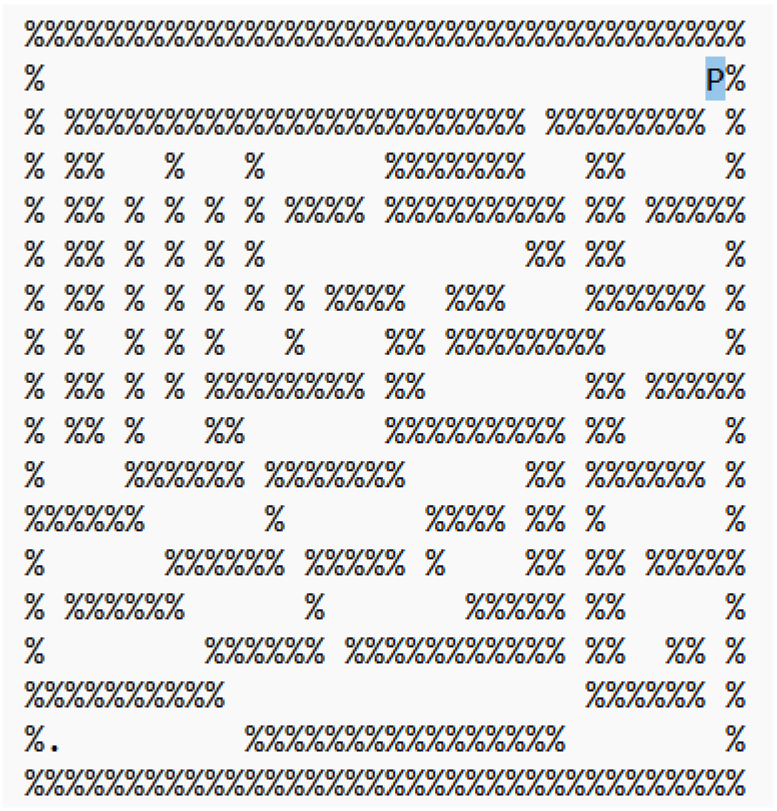
When a state is getting expanded all of its legal successors are getting generated by a code resembling the following pseudocode:

```
successors = [] #initialize an empty list of successors

#for each possible move
for direction in [NORTH, SOUTH, EAST, WEST]:
    x, y = current_state #get the coordinates of the current state
    dx, dy = directionToVector(direction) #find the value of the move

    next_x, next_y = x + dx, y + dy #calculate the position after moving

    if not walls[next_x, next_y]: #if the new position is not in a wall
        nextState = (next_x, next_y) #it is a valid state
        cost = costFunction(nextState) #calculate the cost
        #add the state to the space states
        successors.append(nextState, direction, cost)
```

Where
- directionToVector(direction) is a function that returns a vector corresponding to the difference in the moved coordinates given a direction
- costFunction(node) is a function that calculates a cost of moving to the node

b) Initial and Goal States

Initial state is the starting position of pacman. It might be predefined in the initialization of the SearchProblem instance or read from the layout of the maze. By default it is read from the layout file, defined by the location of the letter ‘P’.



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                 P%
% %%%%%%%%%%%%%%%%%%%%%%% %%%%%%%% %
% %%   %   %      %%%%%%%   %%     %
% %% % % % % %%%% %%%%%%%%% %% %%%%%
% %% % % % %             %% %%     %
% %% % % % % % %%%%  %%%    %%%%%% %
% %  % % %   %    %% %%%%%%%%      %
% %% % % %%%%%%%% %%        %% %%%%%
% %% %   %%       %%%%%%%%% %%     %
%    %%%%%% %%%%%%%      %% %%%%%% %
%%%%%%      %       %%%% %% %      %
%      %%%%%% %%%%% %    %% %% %%%%%
% %%%%%%      %       %%%%% %%     %
%        %%%%%% %%%%%%%%%%% %%  %% %
%%%%%%%%%%                  %%%%%% %
%.         %%%%%%%%%%%%%%%%        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 8: Example of the indicator of the initial state and goal state, mediumMaze layout

Goal state is defined in a similar way although there is no possibility of predefining the goal state. The goal state is always read from the layout file (it is being indicated by the “ . ”) symbol. There is an additional restriction in the structure of the PositionSearchProblem that ensures that there is only one goal state if the layout is not valid, the goal state is always going to be in the position (1, 1).

c) **Operators**

Operators are the set of possible actions. In our case it is a set of all the directions in which pacman can move. In the original state of the problem the possible operators are:
North, East, West and South.
The operators are only applicable if the state after the action has been added into the state space during the evaluation of getSuccessors(current_state).

The result of application of an operator is a new position of pacman and incrementation of the total cost of the path traveled.

The cost of the application of an operator is calculated using the costFn function, which is 1 for every action by default, but can be adjusted during the initialization of the instance of PositionSearchProblem.

Comparison and analysis of the behavior of search algorithms

In this section we will focus on analyzing the differences and similarities of different searching algorithms: DFS, BFS, Dijkstra, A* with Euclidean Distance as the heuristics function, A* with Manhattan Distance as the heuristics function. However, because in this part of the project the cost of an action is always equal to 1, Dijsktra and BFS will act the same with Dijkstra taking a little bit more time to compile due to the implementation of a priority queue.

We will compare them in the 3 following categories:
- execution time
- expanded nodes
- path cost

To make the comparison I designed 4 mazes of incrementing complexity, designed to expose the weaknesses and strong parts of the algorithms.

Level 1: One way only. The comparison of the speed of proposed data structures.

This is the simplest designed maze. There is only one path with no detours available, therefore every algorithm will have the same amount of expanded nodes and the same path. The only difference will be the execution time of the search algorithms. The expected results are that A* algorithms will be the slowest, due to the computation needed to calculate the heuristics functions. Every heat map will look the same, therefore I will only include one example of it.

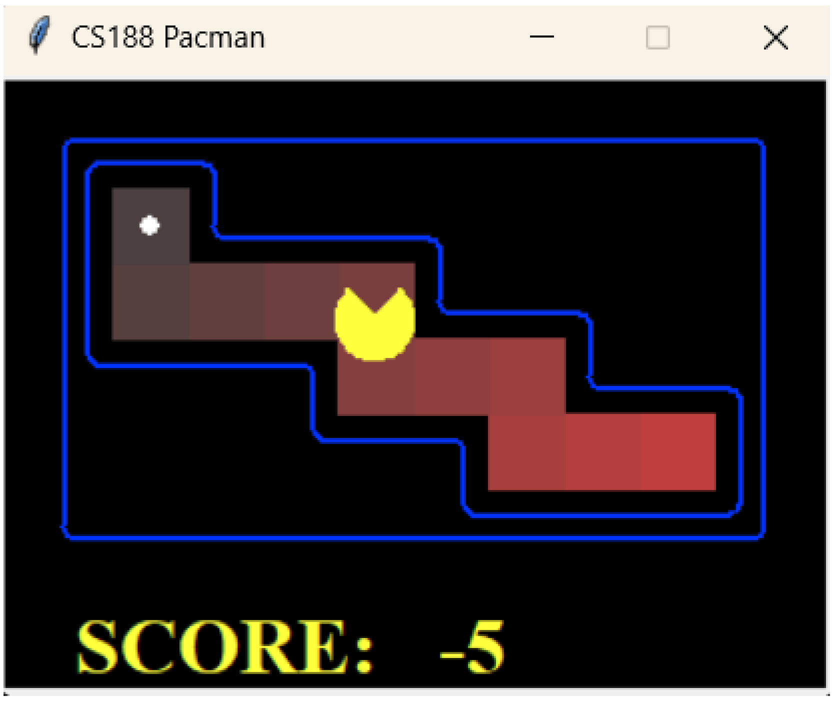


Figure 9: oneWay maze

Because this maze is really simple, the execution time will be an average of 10 separate computations.

	DFS	BFS	Dijkstra	A* Euclidean	A* Manhattan
execution time (ms), % of the best	2,537 110%	**2,291 100%**	2,64 115%	2,657 116%	2,491 109%
nodes expanded	**10**	**10**	**10**	**10**	**10**
path length	**10**	**10**	**10**	**10**	**10**

As we can see, the results don't match the expectations when it comes to the execution times. Although the priority queue search algorithms are slower than the standard queue, there is no big difference between any of the algorithms. That is great news, because we can assume that the data structures have a very similar complexity and the execution time only depends on the algorithms.

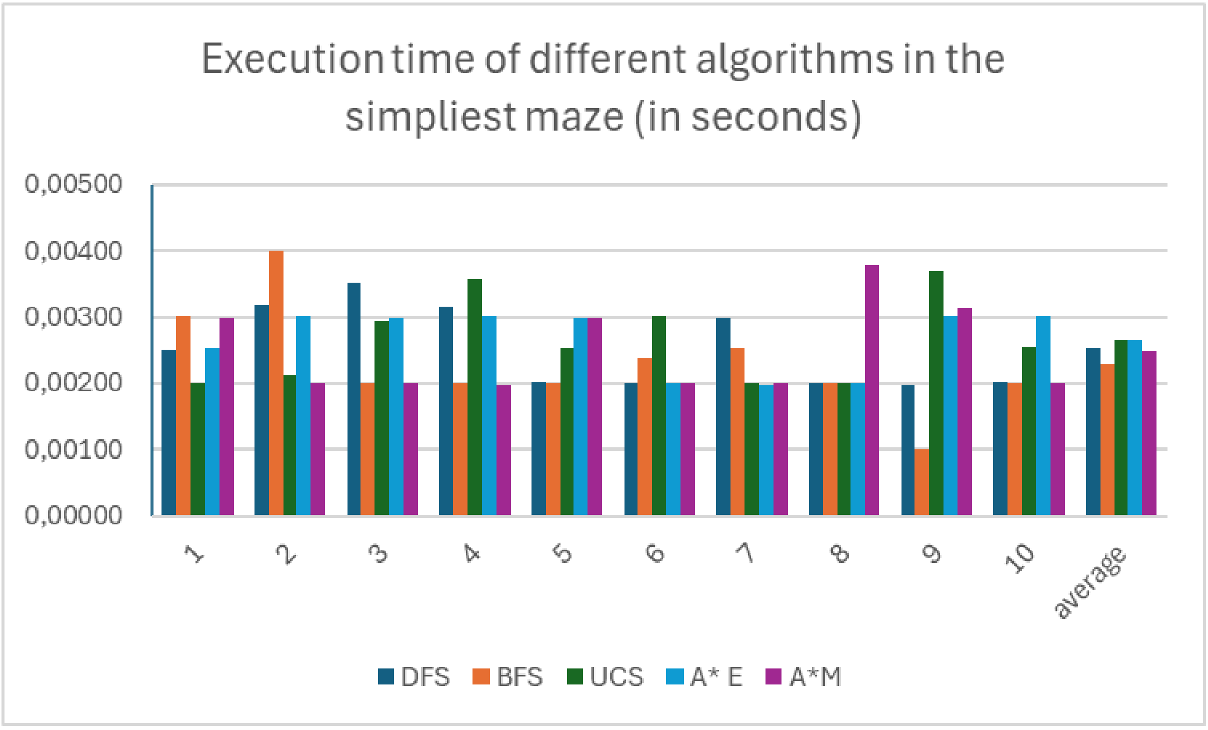


Figure 10: comparison of the execution times in the oneWay maze

Level 2: The corridorMaze. Which room did I leave my phone in?

This room layout was designed to show that even though the DFS and BFS algorithms have a completely different approach to the search problem, sometimes they can share the exact same performance in achieving the goal. Here is the layout of the maze and heat maps corresponding to different algorithms. We will go into the performance of algorithms later in this section.

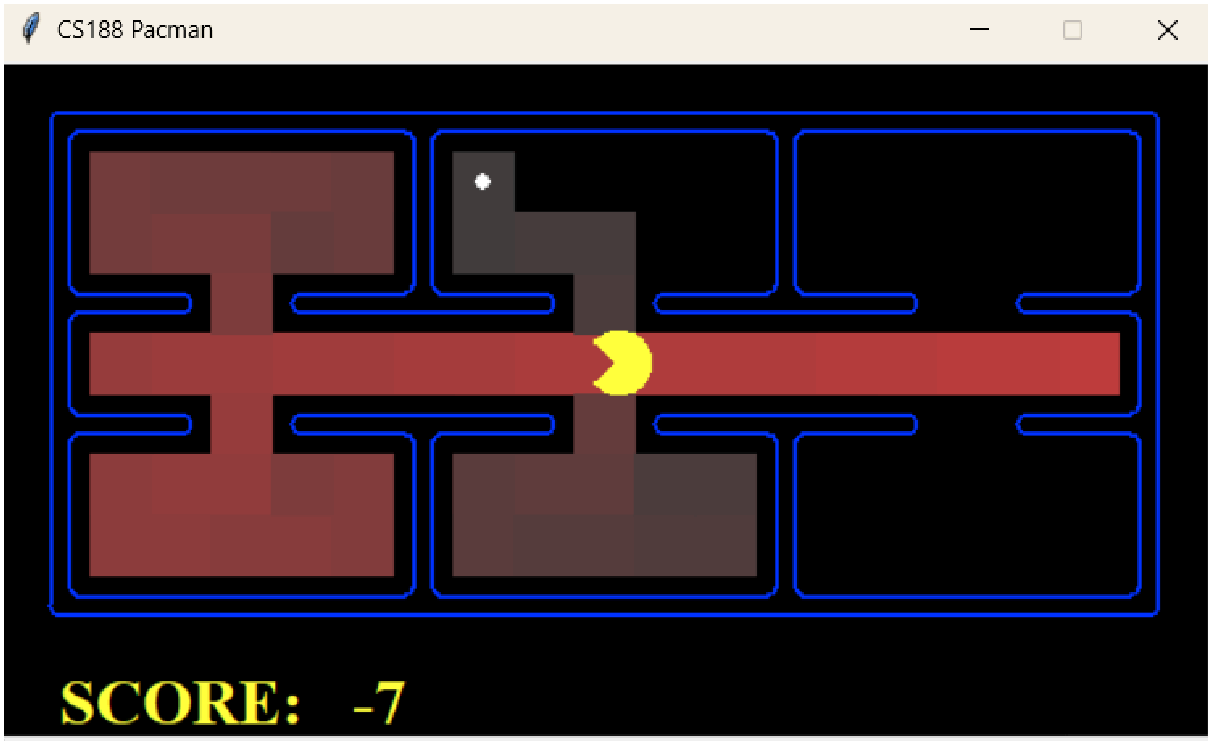


Figure 11: DFS in the corridor maze

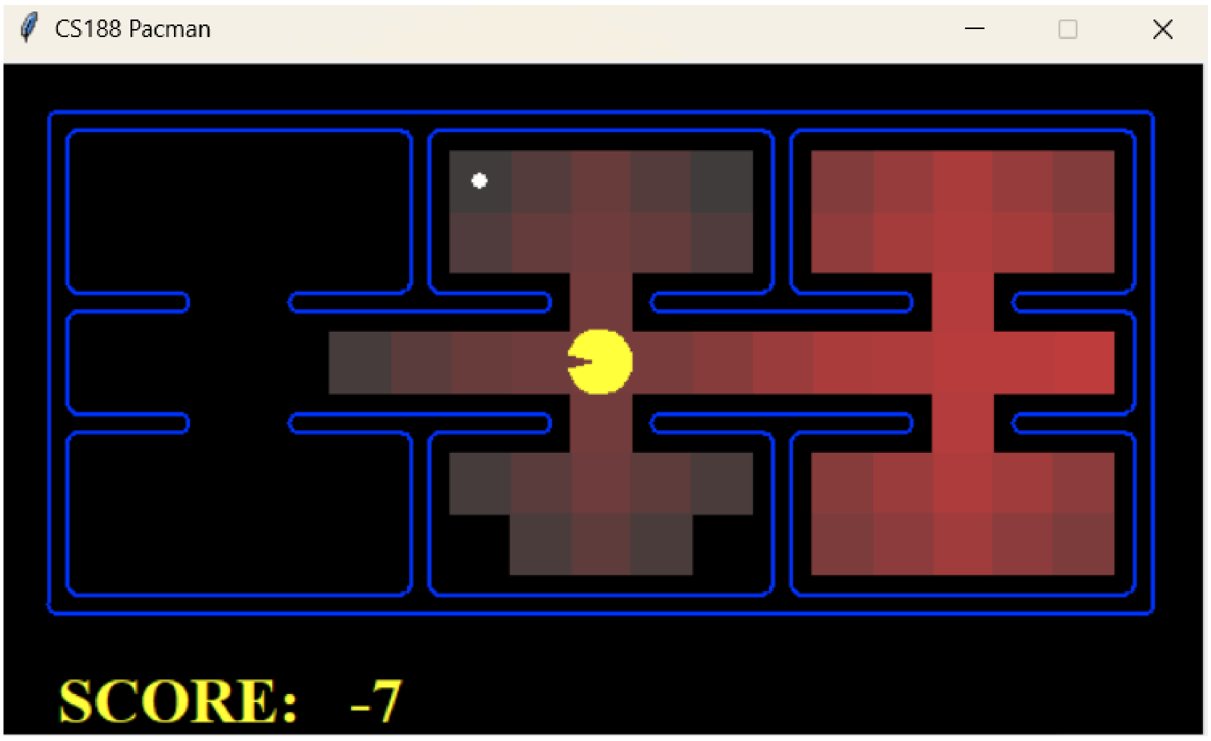


Figure 12: BFS in the corridor maze

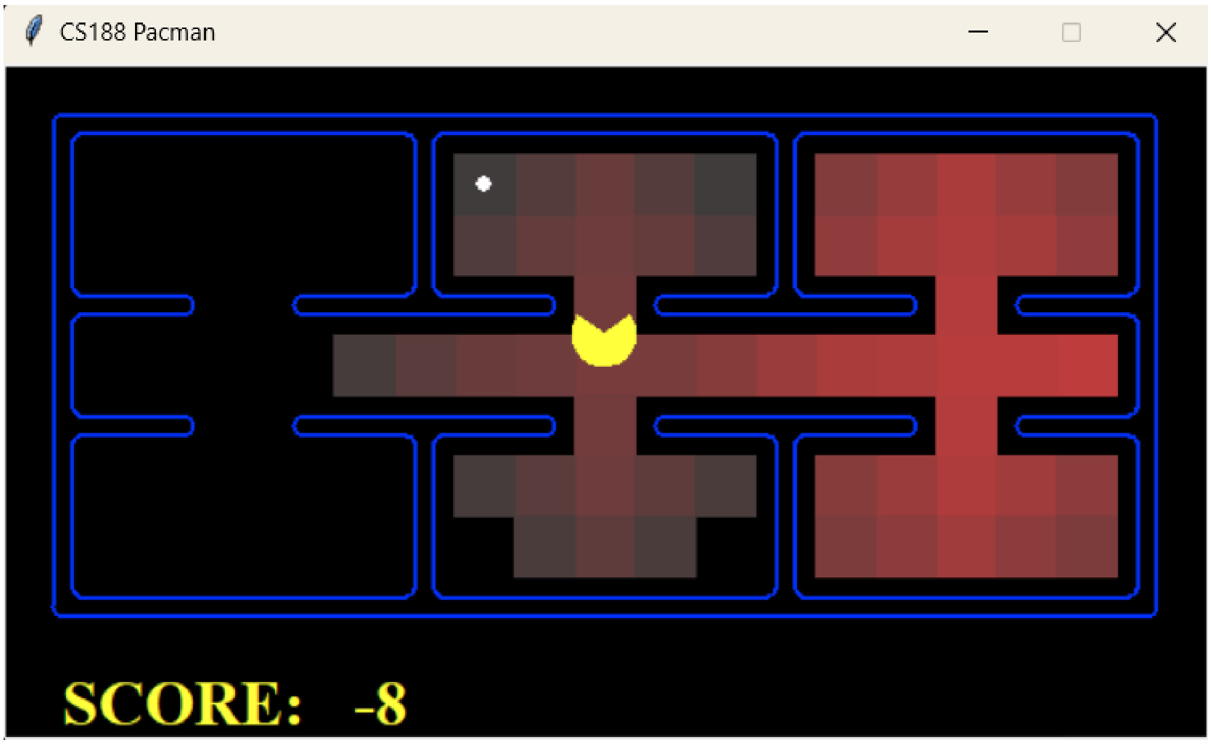


Figure 13: Dijkstra in the corridor maze

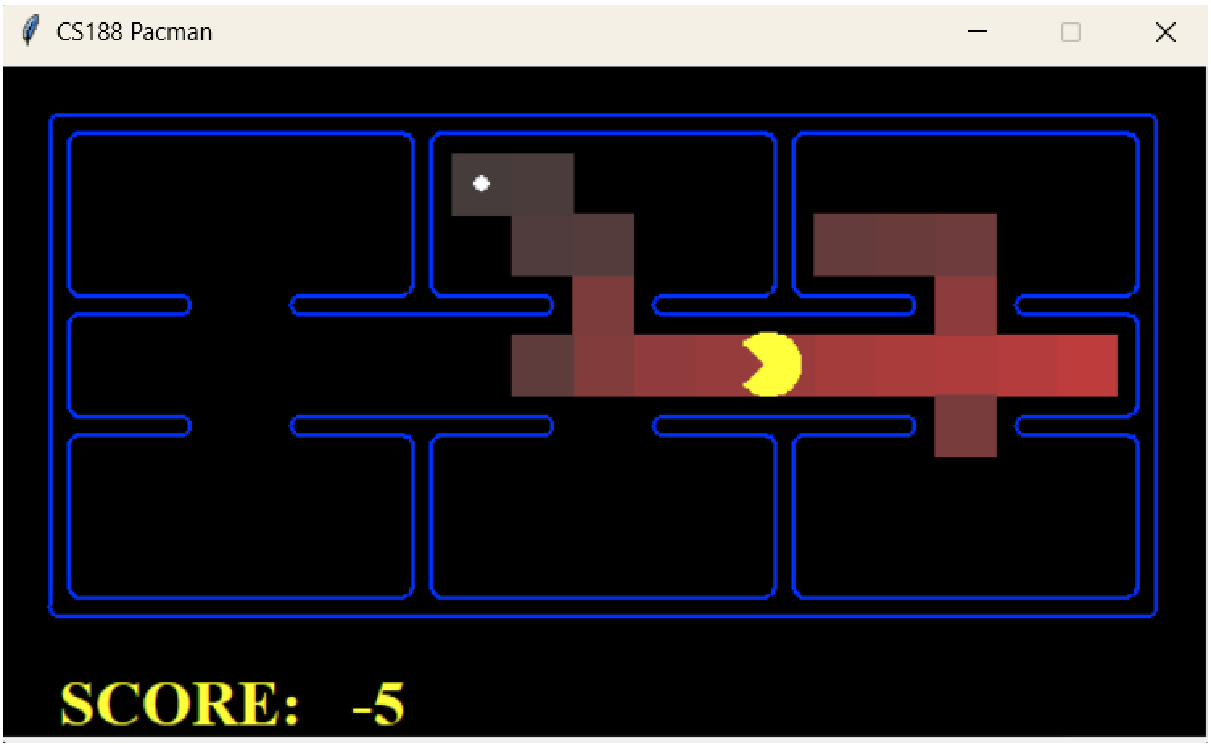


Figure 14: A* with Euclidean Distance in the corridor maze

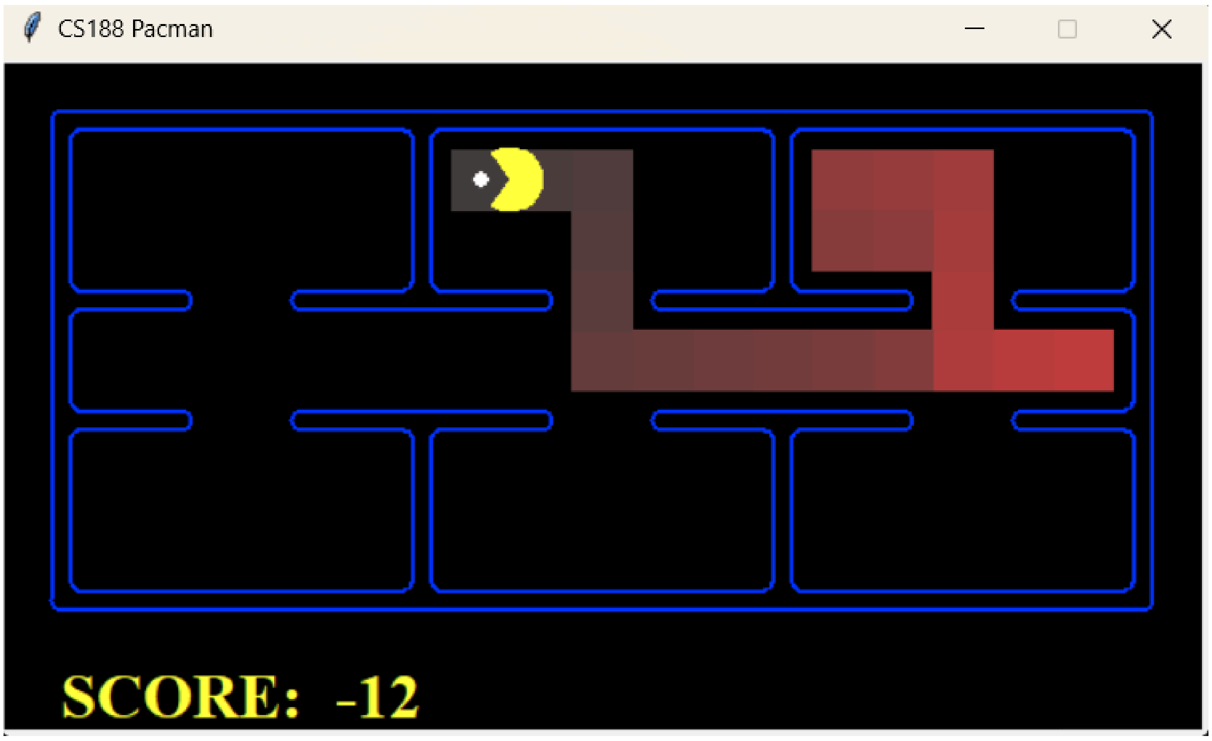


Figure 15: A* with Manhattan Distance in the corridor maze

This maze is a little bit more complex than the previous one. We can see that UCS and BFS share the exact same heat map as expected. The heat maps of DFS and A* algorithms are completely different than BFS and Dijkstra. A* algorithms shared similar approaches to expanding the nodes, however the Euclidean heuristics explored also the entrance to the room in the bottom right corner, due to the euclidean distance not changing that significantly with the difference of one step if we are far away from the problem, while the Manhattan one explored the top right room more.

Here is the performance of each of the algorithms:

	DFS	BFS	UCS	A* Euclidean	A* Manhattan
execution time (ms)	6 199%	6,53 216%	6,96 230%	3,56 118%	**3,02** **100%**
nodes expanded	54 284%	54 284%	54 284%	**19** **100%**	20 105%
path length	**13**	**13**	**13**	**13**	**13**

From the table we can see that every algorithm has found the fastest path to find the food dot. This example however shows the dominance of the A* algorithms, due to their built in 'intuition' on where to look for the solution. The A* algorithms expanded only 35% nodes of the DFS, UCS and BFS algorithms and required only half of the time to execute.

Another interesting thing is that the DFS shared almost the same performance with BFS and UCS algorithms, even though the approach was completely different from what we can see on the Figures 11-13. The only difference was a little bit faster execution time, however the difference was not too significant.

Here are the charts for each of the categories:

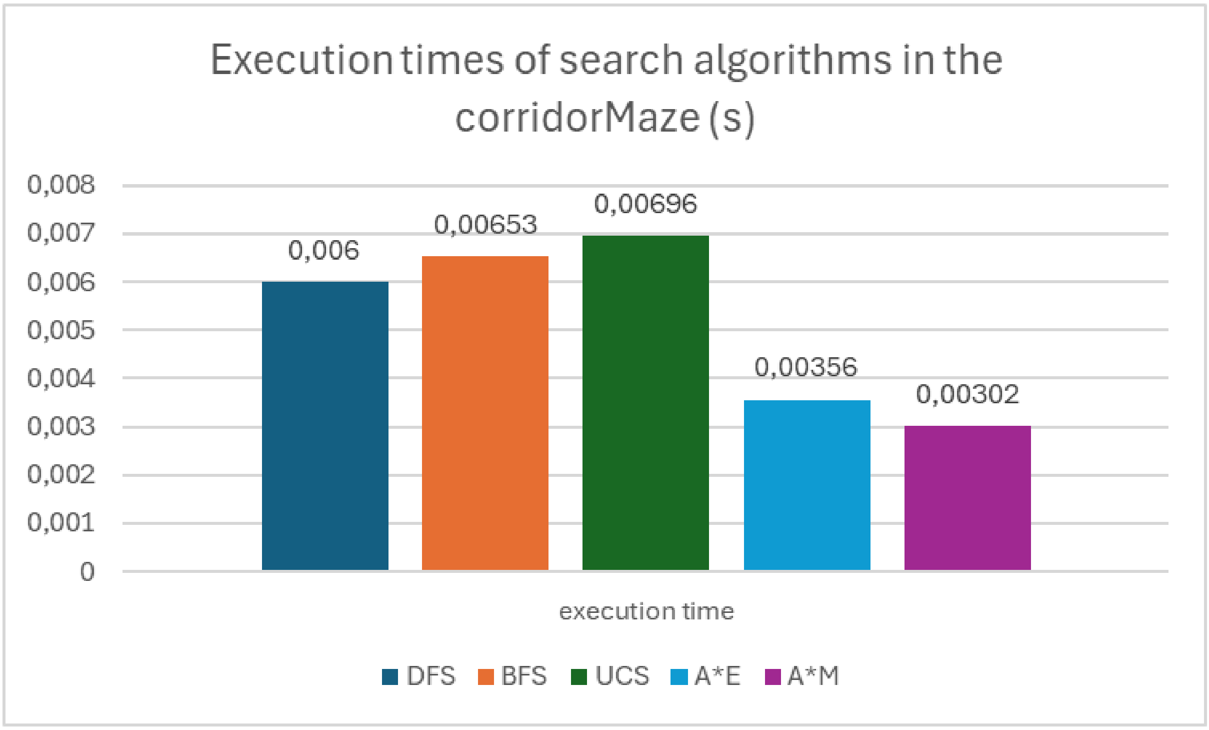


Figure 16: Execution times in the corridorMaze

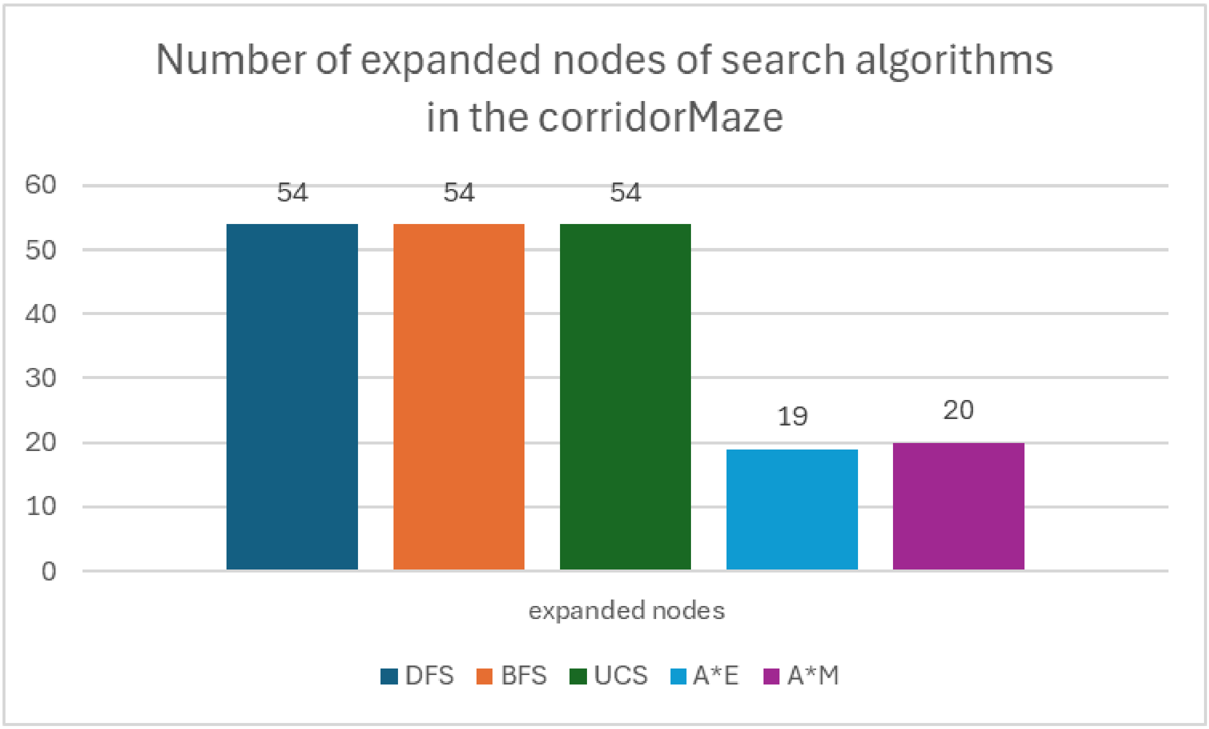


Figure 17: expanded nodes in the corridorMaze

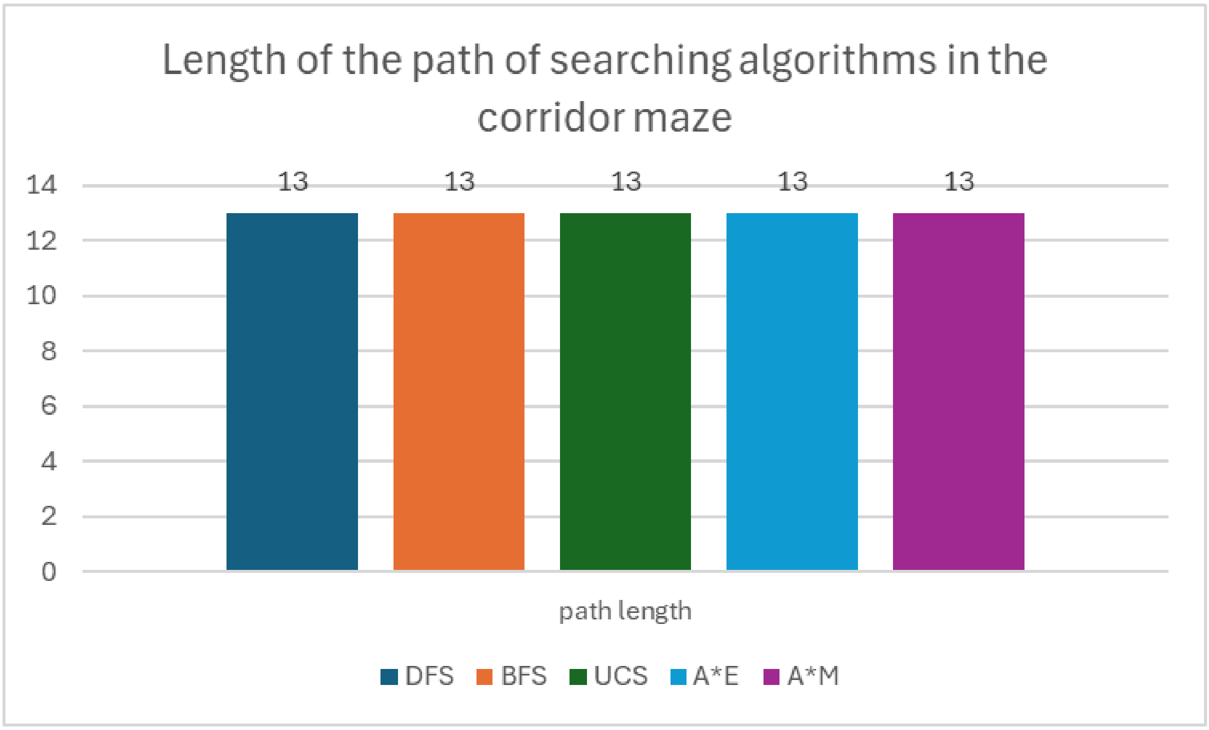


Figure 18: path length in the corridor maze

Level 3: The exploit. Taking advantage of the disadvantages.

This maze was designed to exploit the weaknesses of DFS and BFS algorithms depending on the placement of the goal state. It will have 3 variations:

- ICanSeeIt
- AroundTheCorner

ICanSeeIt:

Let's start with ICanSeeIt. To avoid redundant Images we will only look at 2 different heat maps, one for the DFS and A* algorithms and the other one for BFS and Dijkstra

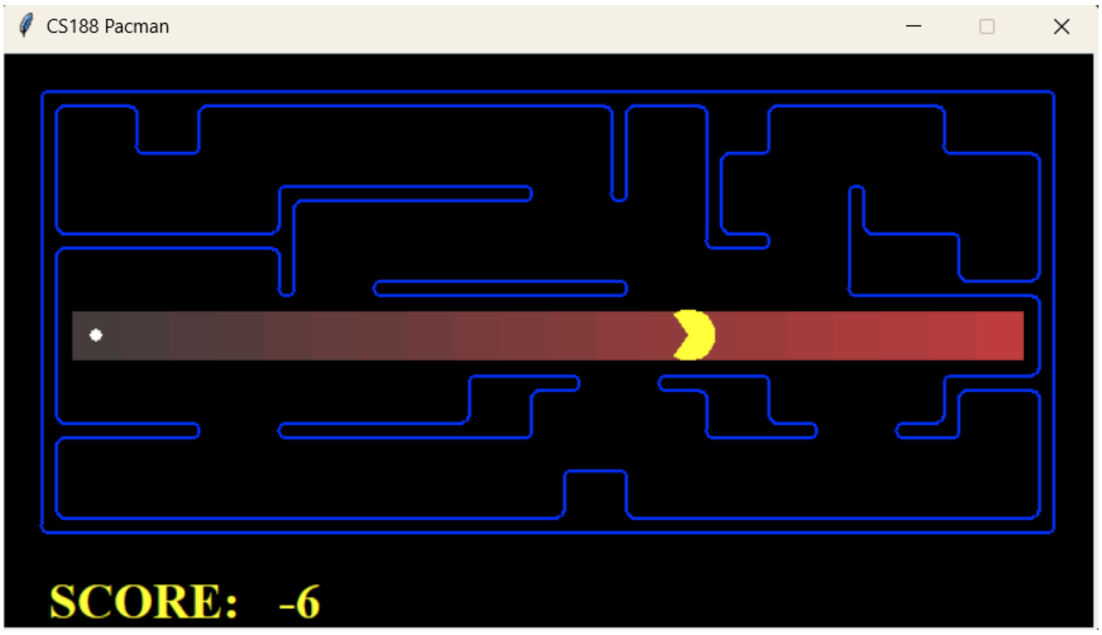


Figure 19: ICanSeeIt: DFS, A* Euclidean, A* Manhattan

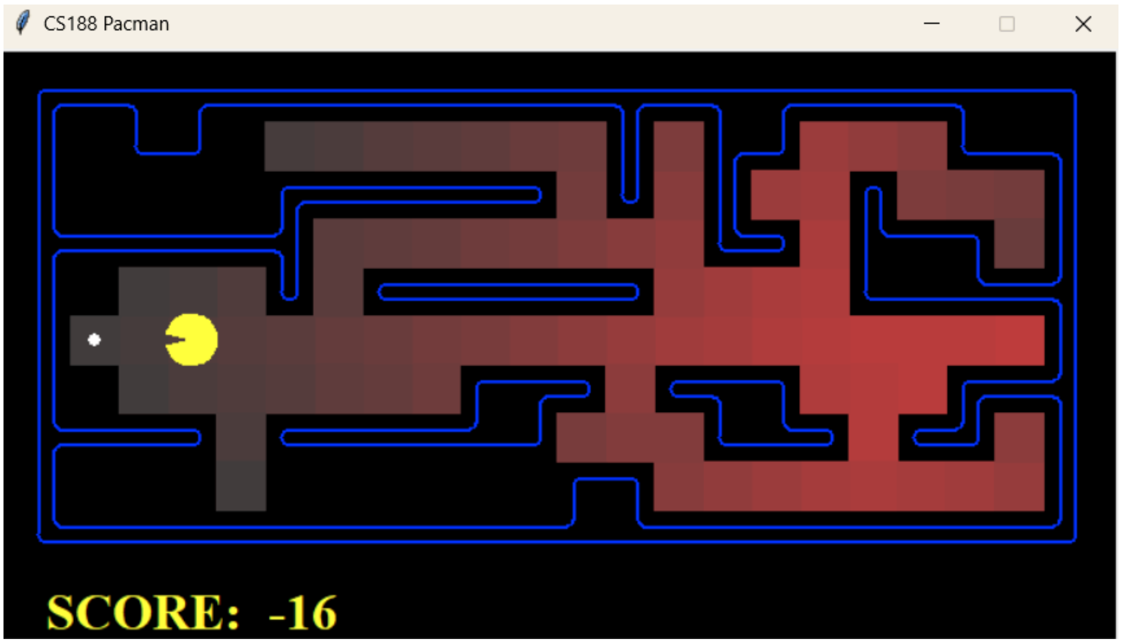


Figure 20: ICanSeeIt: BFS, UCS

This example clearly exploits the weakness of BFS and Dijkstra algorithms. Even though the path is trivial if we have either the perfect knowledge about the problem, basic intuition (A*) or we are lucky enough with the DFS algorithm. The BFS and Dijkstra explore almost the whole maze before finding the straight path to the food dot.

Here are the performances of each algorithm:

	DFS	BFS	UCS	A* Euclidean	A* Manhattan
execution time (ms)	**1,99 100%**	9,88 496%	9,54 479%	2,00 100,5%	2,99 150%
nodes expanded	**19**	81 426%	81 426%	**19**	**19**
path length	**19**	**19**	**19**	**19**	**19**

Again, every algorithm has found the fastest path, however BFS and Dijkstra took almost 5 times as much time to execute and expanded over 4 times more nodes to find the solution. In this level the charts will be uploaded at the end.

AroundTheCorner:

Let's continue with the AroundTheCorner maze. This time we need more examples of heat maps.

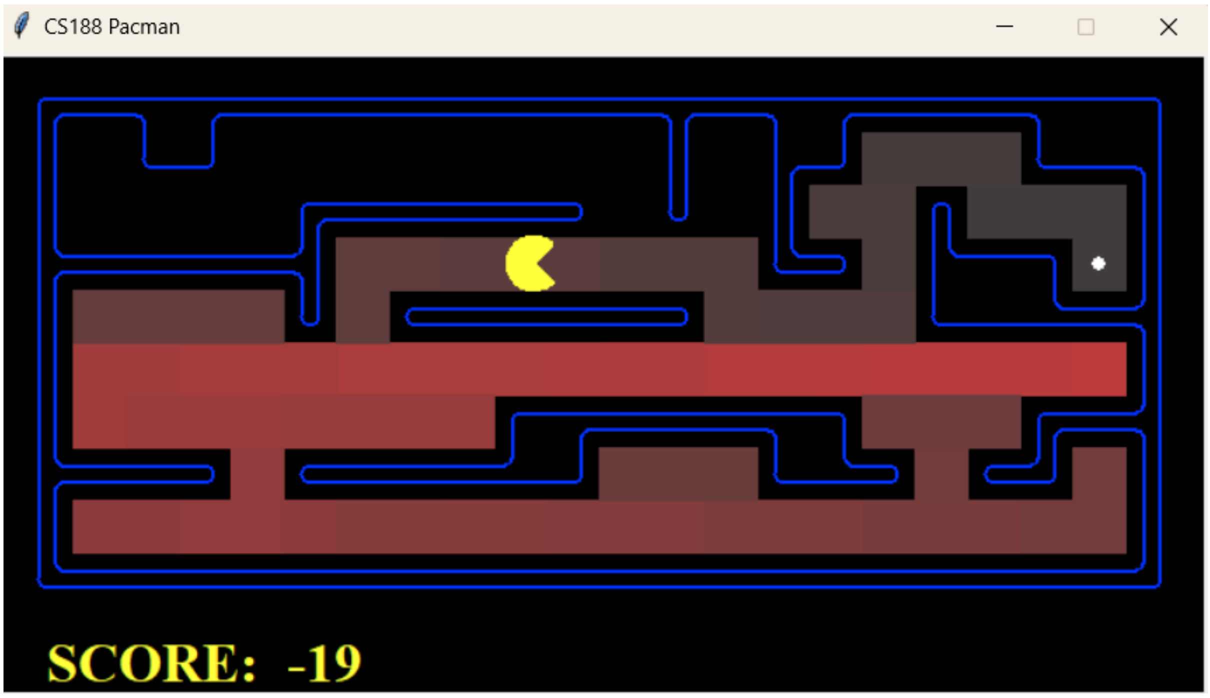


Figure 21: AroundTheCorner, DFS

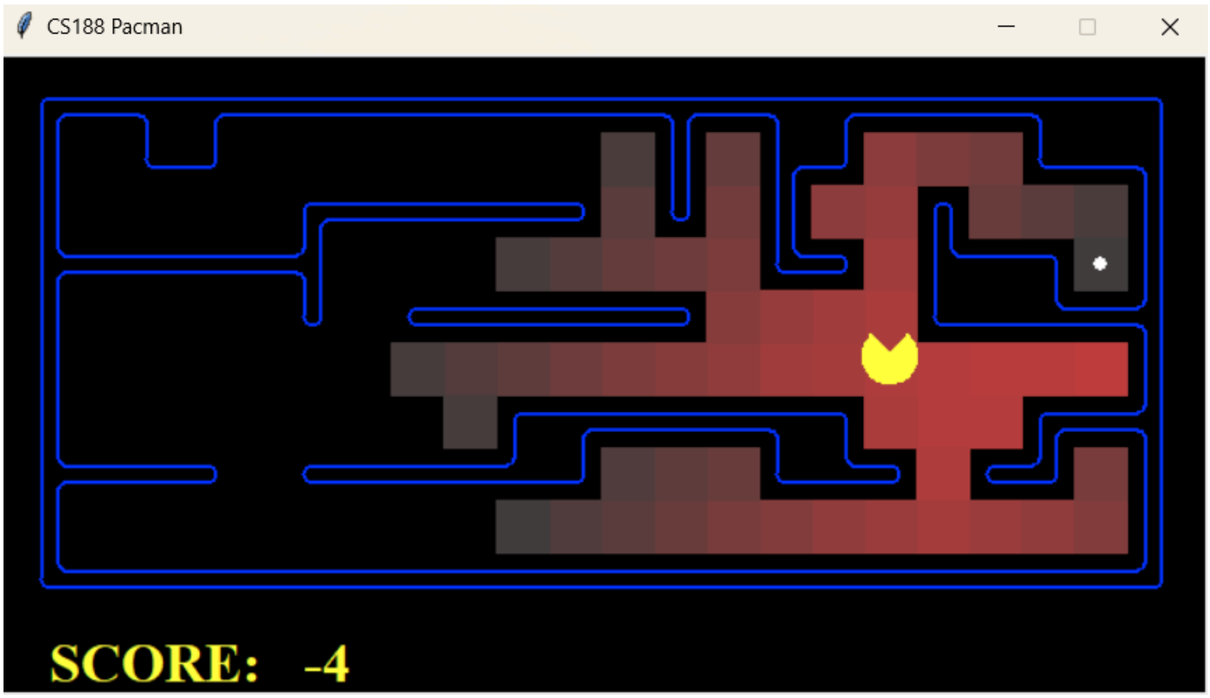


Figure 22: AroundTheCorner, BFS and Dijkstra

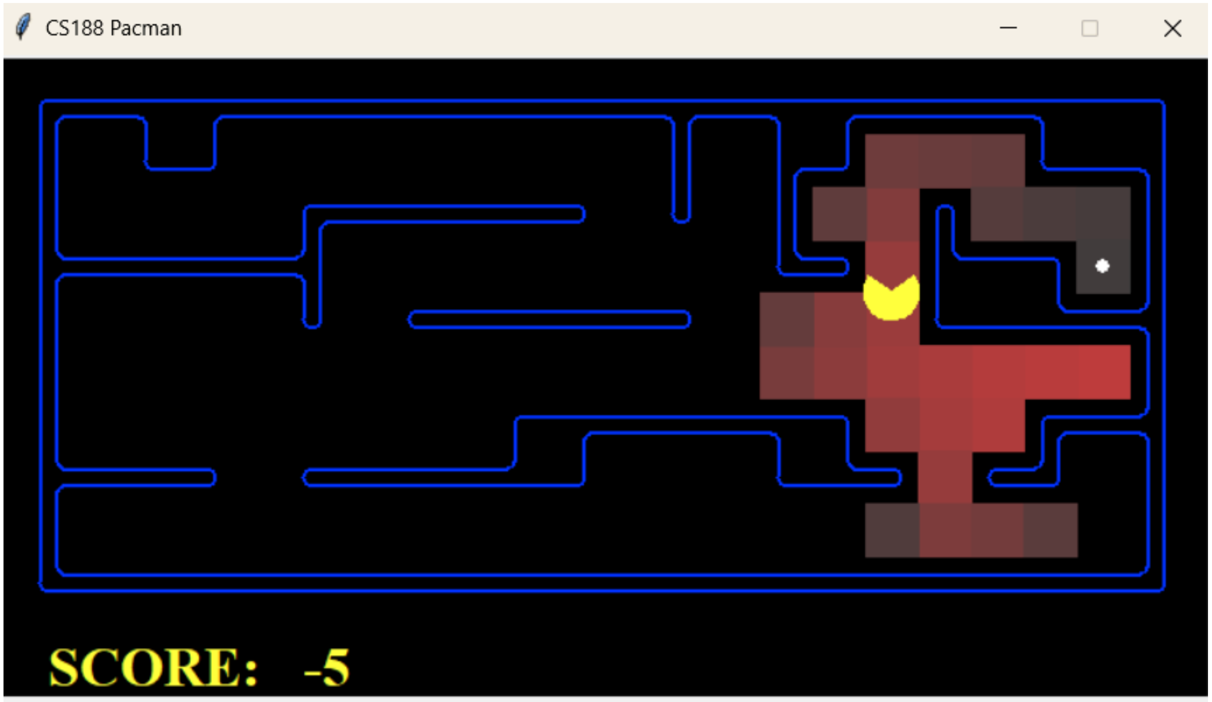


Figure 23: AroundTheCorner, A* with Euclidean Distance heuristic

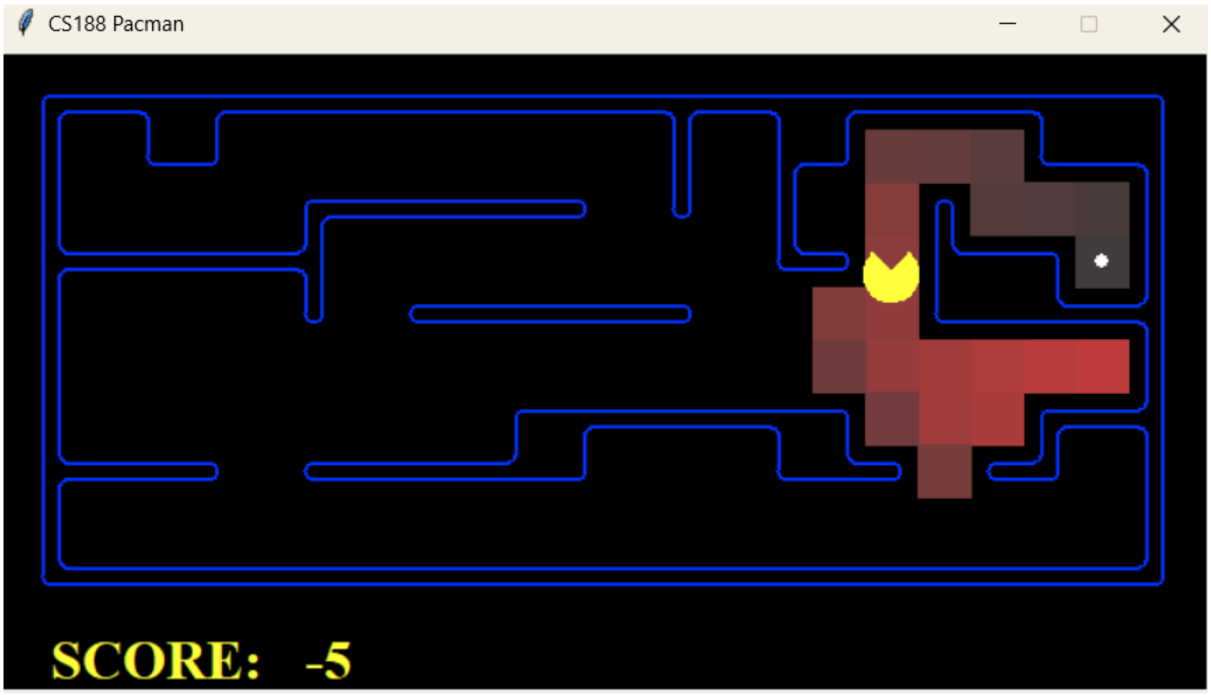


Figure 24: AroundTheCorner, A* with Manhattan Distance heuristic

As we can see from the figure 21, this is the first example where DFS didn't find the optimal way through the maze. Additionally it expanded by far the biggest number of nodes. The problem favors BFS and Dijkstra, however they still have to expand a lot of nodes to find the solution compared to A* algorithms which stay in the more or less correct area due to the heuristics guidance.

The performance of all algorithms:

	DFS	BFS	UCS	A* Euclidean	A* Manhattan
execution time (ms)	8,58 340%	5,20 206%	6,46 256%	3,99 158%	**2,52** **100%**
nodes expanded	83 307%	57 211%	57 211%	**27**	**27**
path length	36 257%	**14**	**14**	**14**	**14**

The path found by the DFS was 2,5 times longer than the optimal path, additionally it expanded 3 times more nodes than A* algorithms and required 3,4 times more time. This example really showcases the vulnerabilities of DFS. The algorithm had to backtrack 9 times to find a branch that leads to the goal state and even then the path was way longer than the proposed optimal solutions.

Now let's compare both of the problems and see the charts:

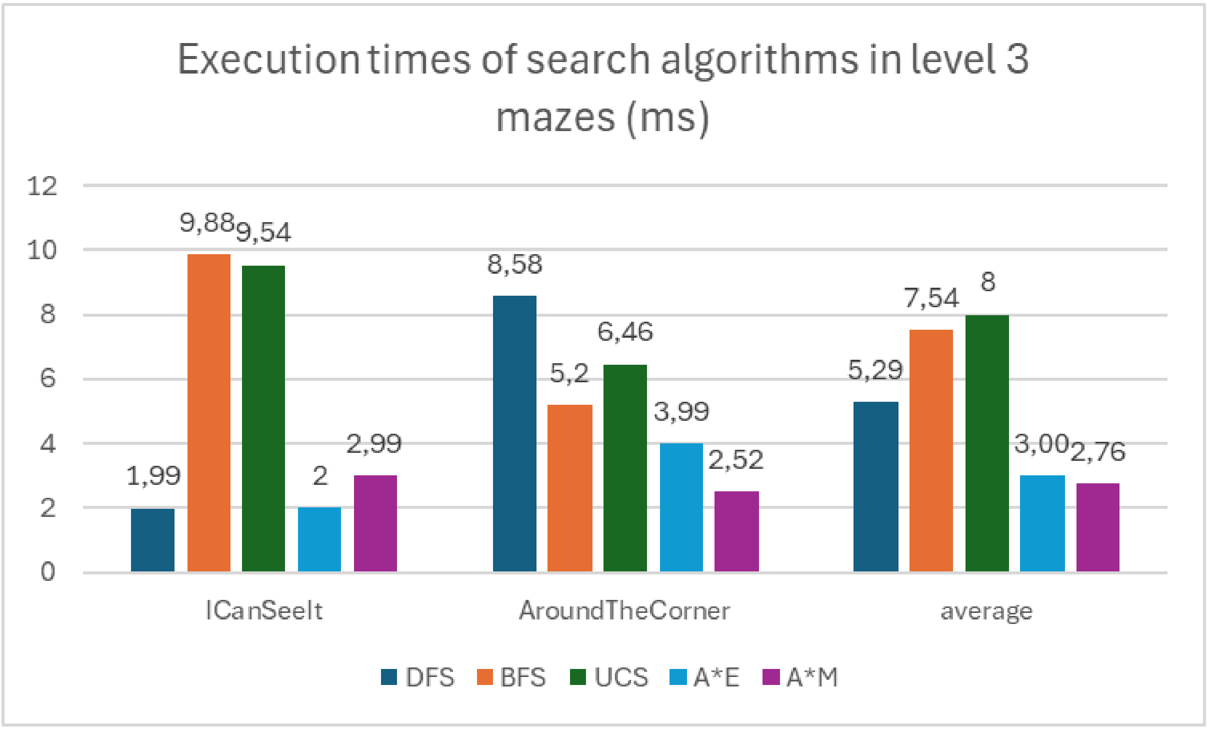


Figure 25: Execution times in Level 3

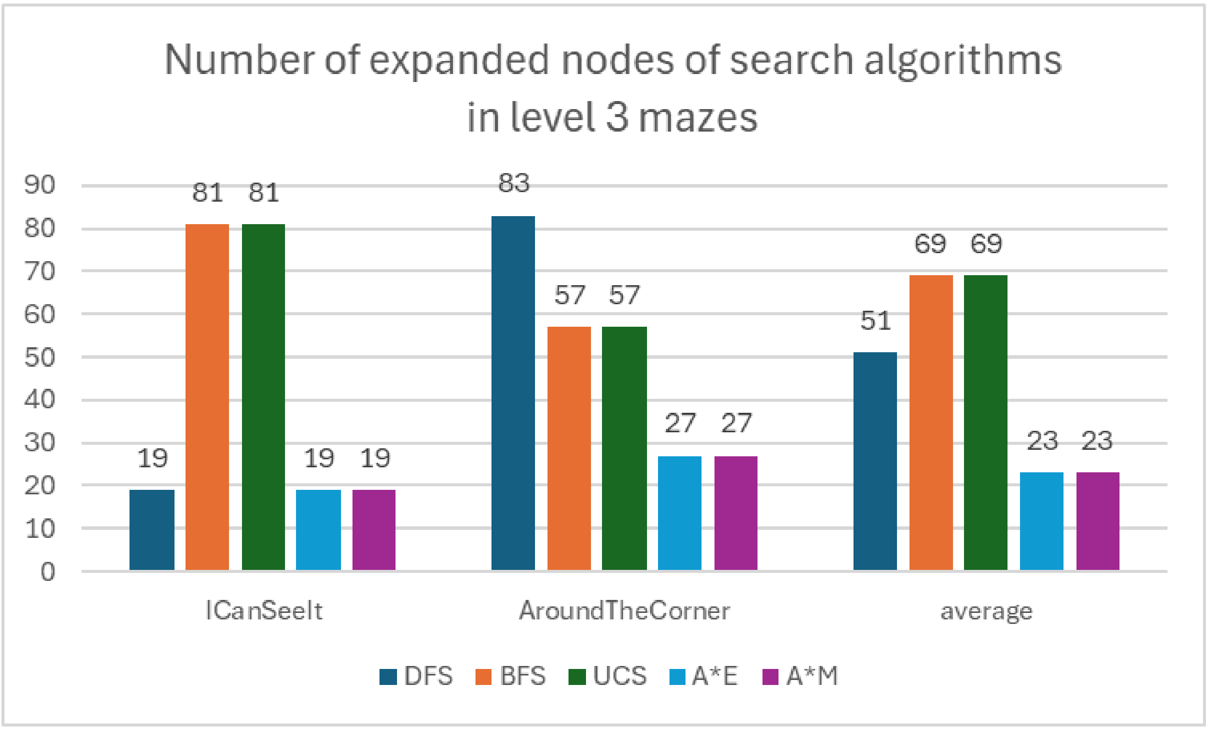


Figure 26: Expanded nodes in Level 3

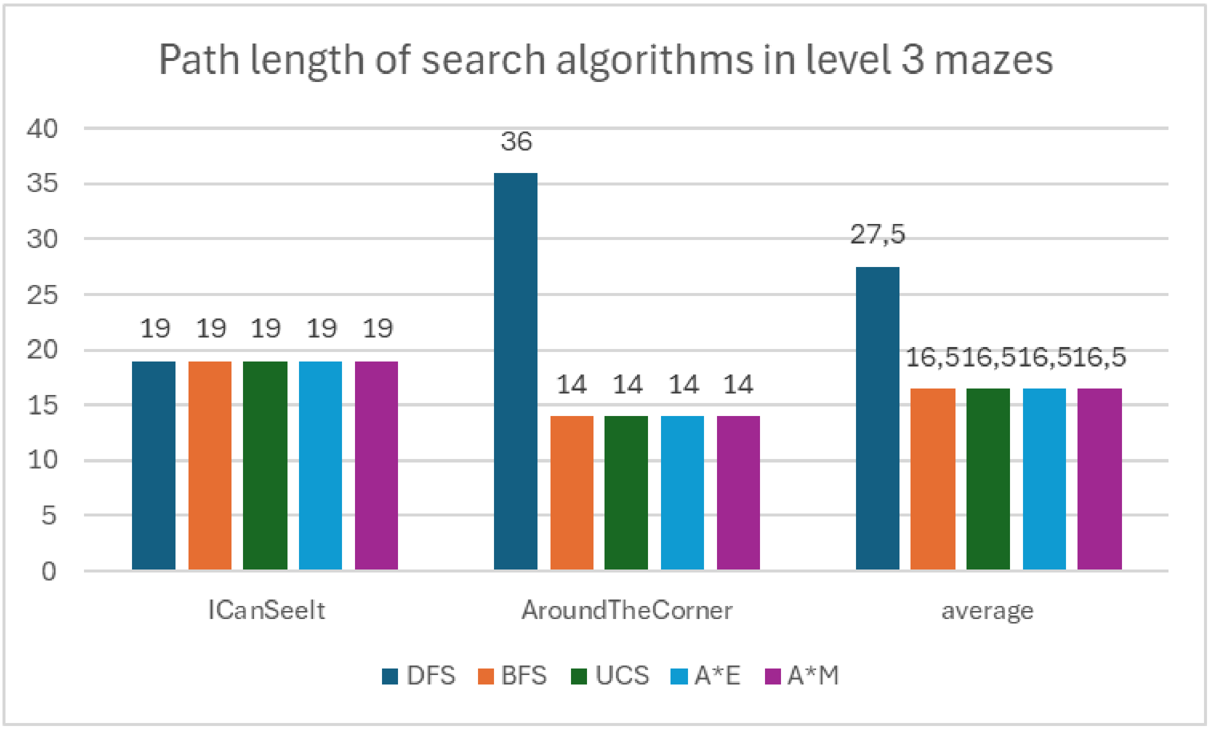


Figure 27: Path length in Level 3

From the figures 25-27 we can clearly see that the A* algorithms continue their dominance over the other search algorithms.They are performing the best in every category inspected by us.

Level 4: The Caves. Free for all.

This is the biggest maze. The design aimed to test the actual application of the algorithms. The design involves a lot of different corridors and a food dot is placed in one of them. Pacman is placed in the middle to avoid the predictability of the DFS algorithm and take advantage of the BFS nature to explore all directions at once. Additionally, the way the corridors are placed is meant to trick the A* algorithms into exploring the longer paths at first. Here are the heat maps.

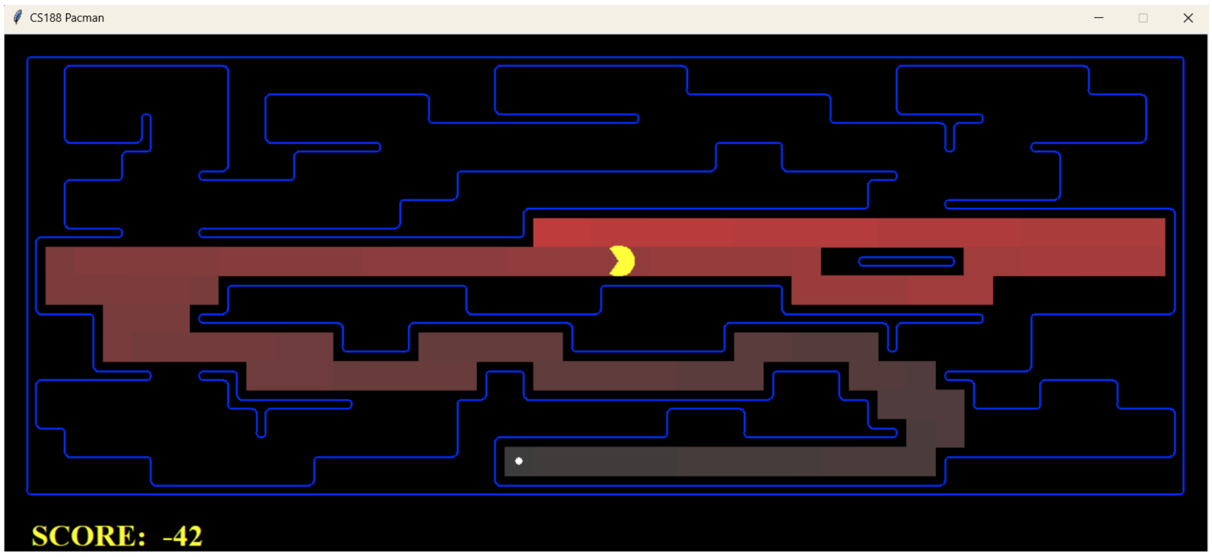


Figure 28: DFS in the Caves maze

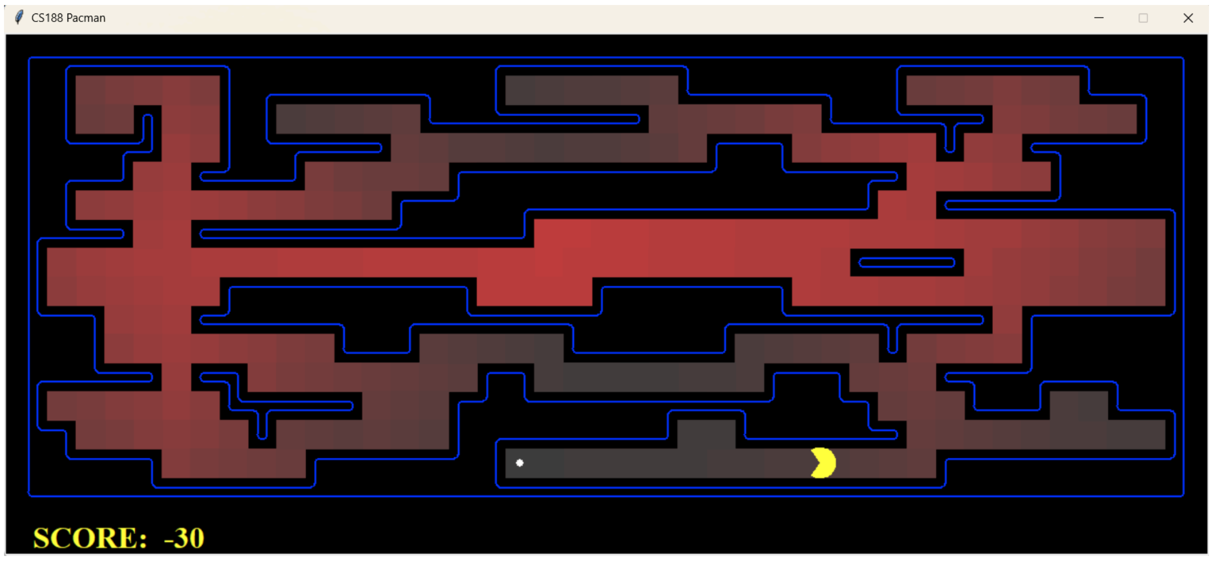


Figure 29: BFS and UCS in the Caves maze

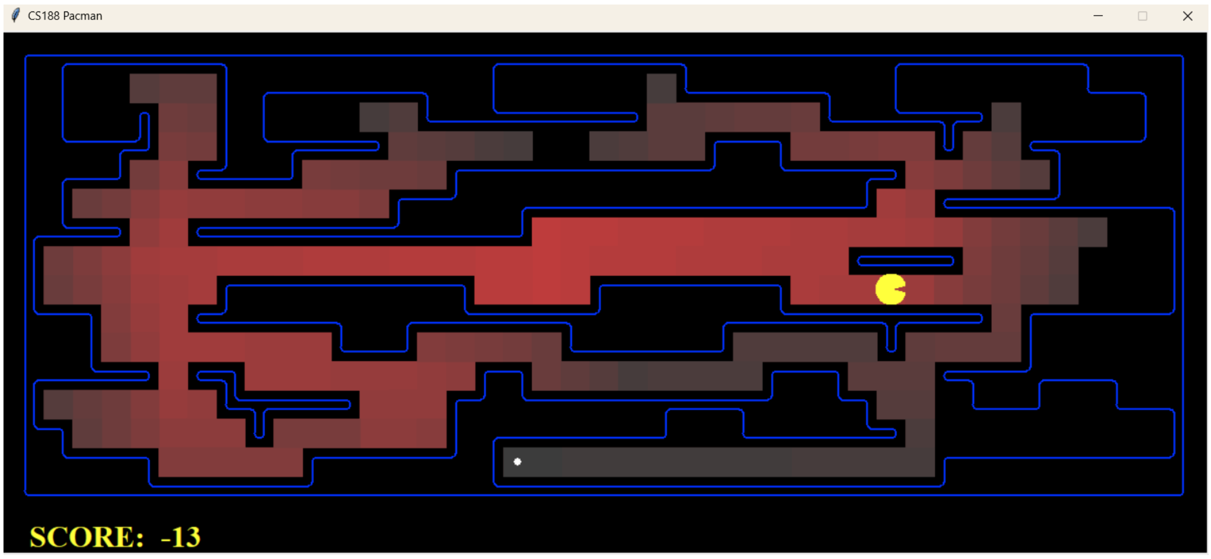


Figure 30: A* with Euclidean Heuristic in the Caves maze

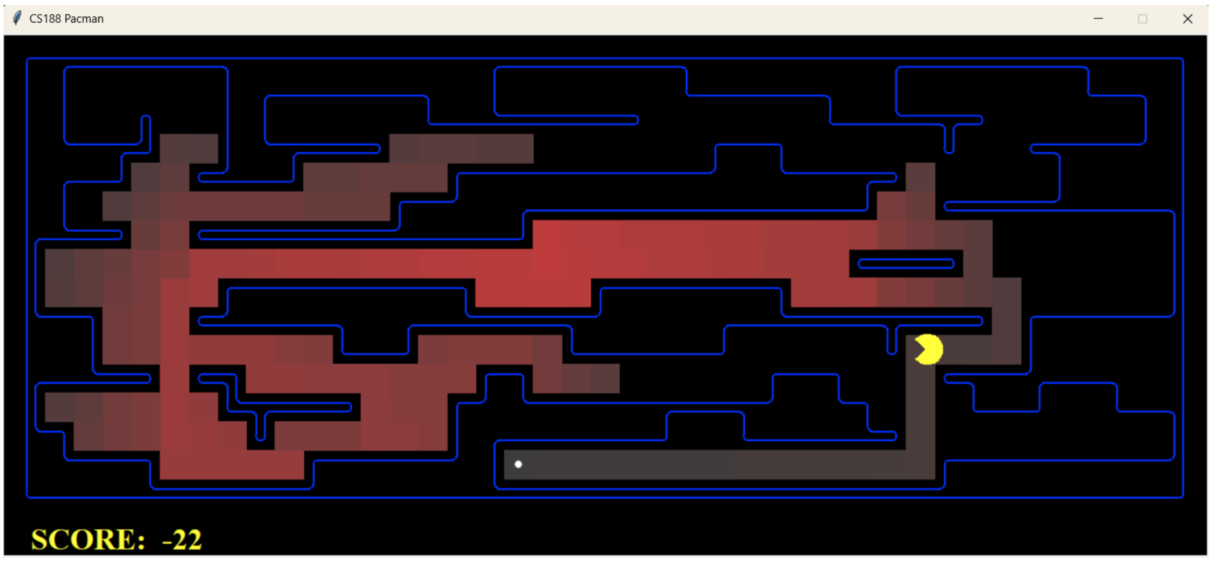


Figure 31: A* with Manhattan Heuristic in the Caves maze

The Caves maze is the perfect maze to explain the difference between Manhattan and Euclidean distance heuristics. The Euclidean A* algorithm is more likely to explore the areas further from the position due to the moving away being less penalized than in the case of manhattan distance. Additionally, the DFS algorithm has found a way the fastest, with the least nodes expanded, however the way was far from the optimal. BFS had to explore every node to get to the solution, however the Euclidean A* algorithm almost did the same.

Here are the statistics of every algorithm:

	DFS	BFS	UCS	A* Euclidean	A* Manhattan
execution time (ms)	**15,0 100%**	29,6 197%	33,1 220%	33,3 222%	32,2 214%
nodes expanded	**128**	266 207%	266 207%	221 173%	168 131%
path length	123 300%	**41**	**41**	**41**	**41**

This is the first time, when the heuristics calculation time really mattered. Even though the A* algorithms expanded significantly less nodes (especially the Manhattan heuristic one), the execution time was higher than the BFS. The DFS algorithm has scored by far the best in the first two categories, although the path found was exactly 3 times longer than the optimal one. We have to remember though that the accuracy of the DFS algorithm really depends on the maze and the placements of the initial and goal state. In this case DFS has only expanded 5 nodes which weren't on its final path to the goal state.

Let's take a quick look at the charts from level four and conclude our analysis afterwards.

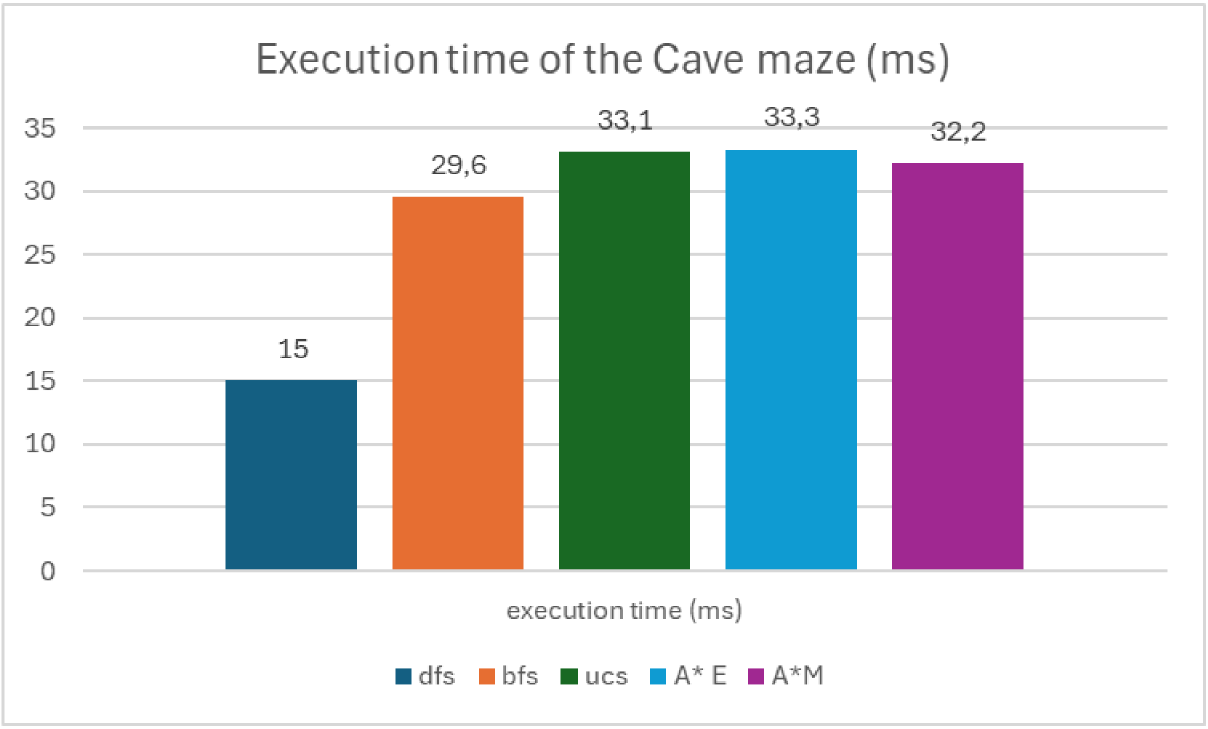


Figure 32: Execution times in the Cave maze

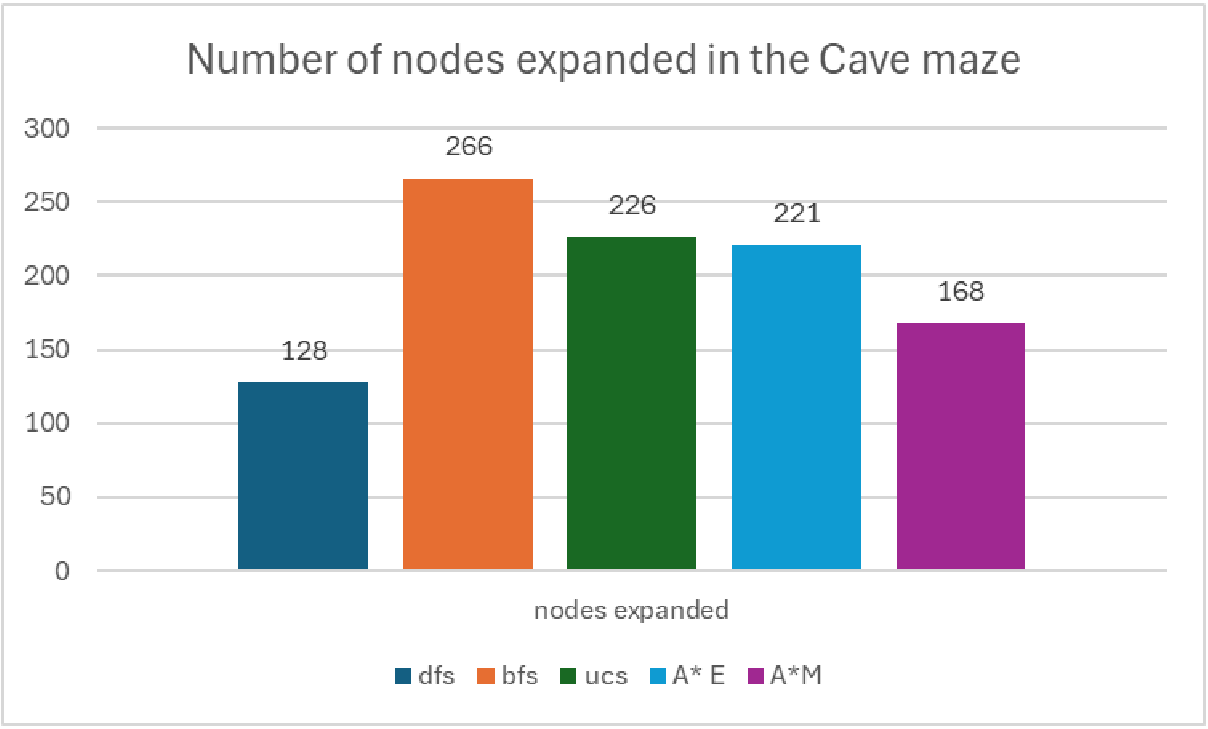


Figure 33: Nodes expanded in the Cave maze

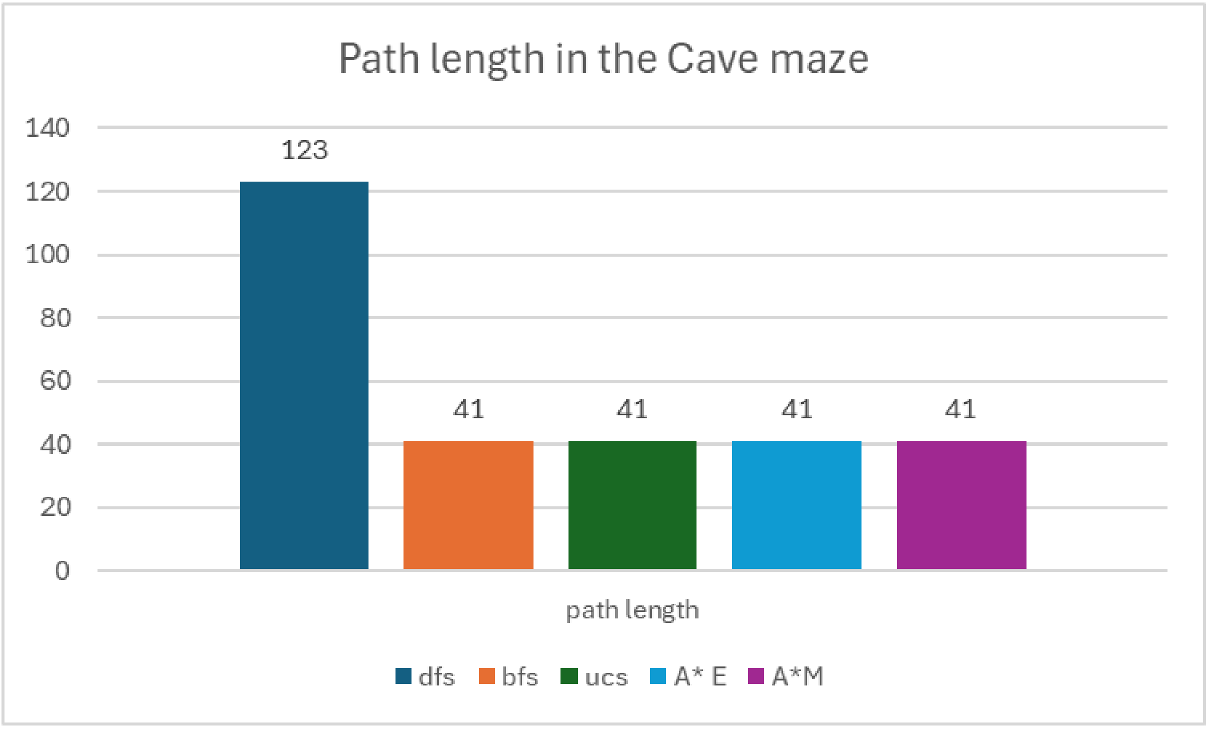


Figure 34: Path length in the Cave maze

Final comparison of the algorithms:

Across all of the proposed mazes, there usually were contenders to be the most suitable and unsuitable algorithms. If we have the basic knowledge about the maze we will be facing we can choose the most promising algorithm, however the true AI should be able to properly adapt to a given scenario without the need of our inference. Therefore we should look for the most consistent algorithm across all the mazes.

Let’s compare all the statistics we measured across all the mazes:

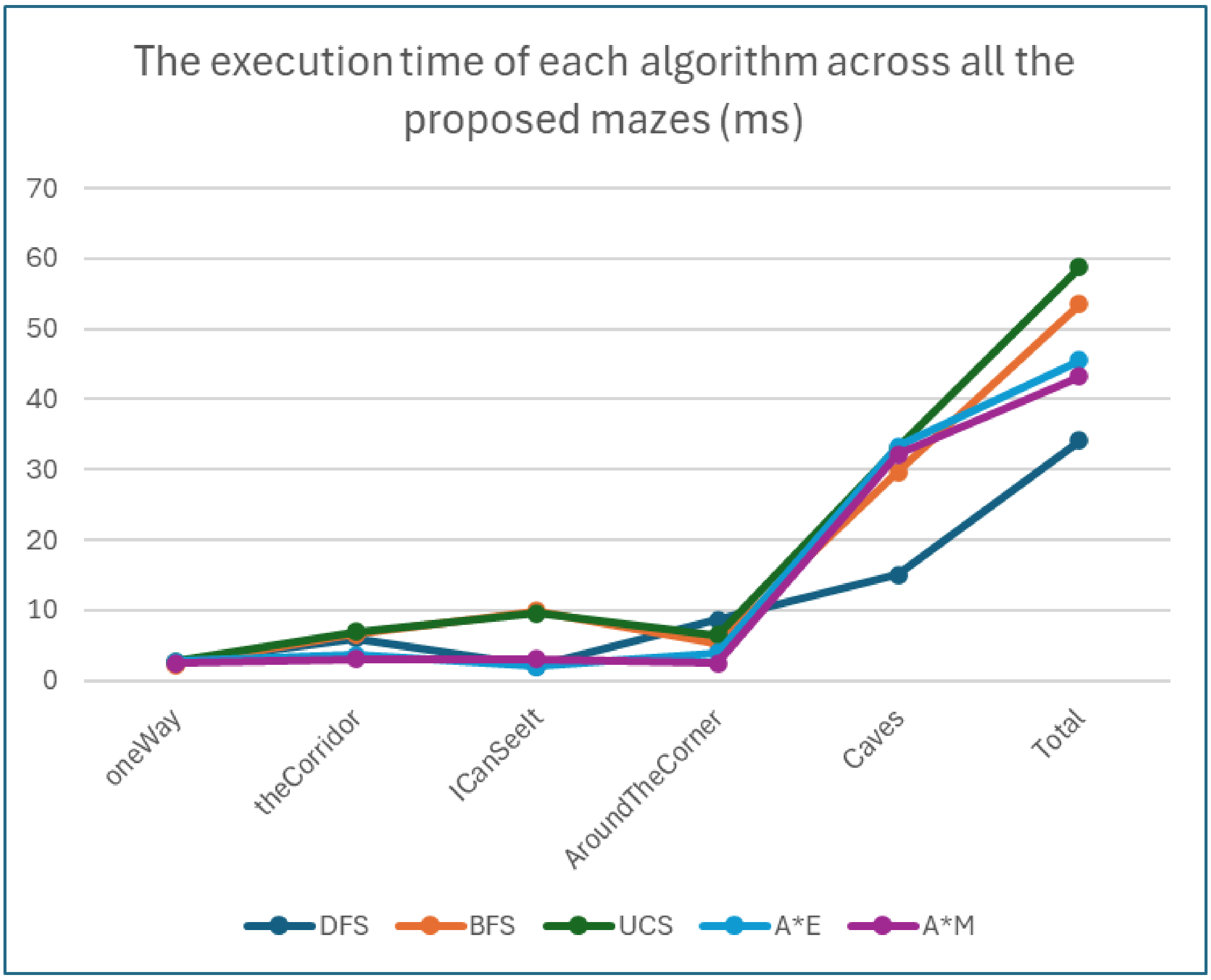


Figure 35: Execution times across all of the proposed mazes

From figure 35 we can see that the DFS algorithm has won the execution time war due to its performance in the most complex maze - Caves.
A* algorithms proved to be the most consistent, getting the best results in all of the remaining mazes, however got tricked in the caves which resulted in the worst total result.
BFS and Dijkstra algorithms were overall the slowest, with the exception of being faster than the A* algorithms in the caves maze.

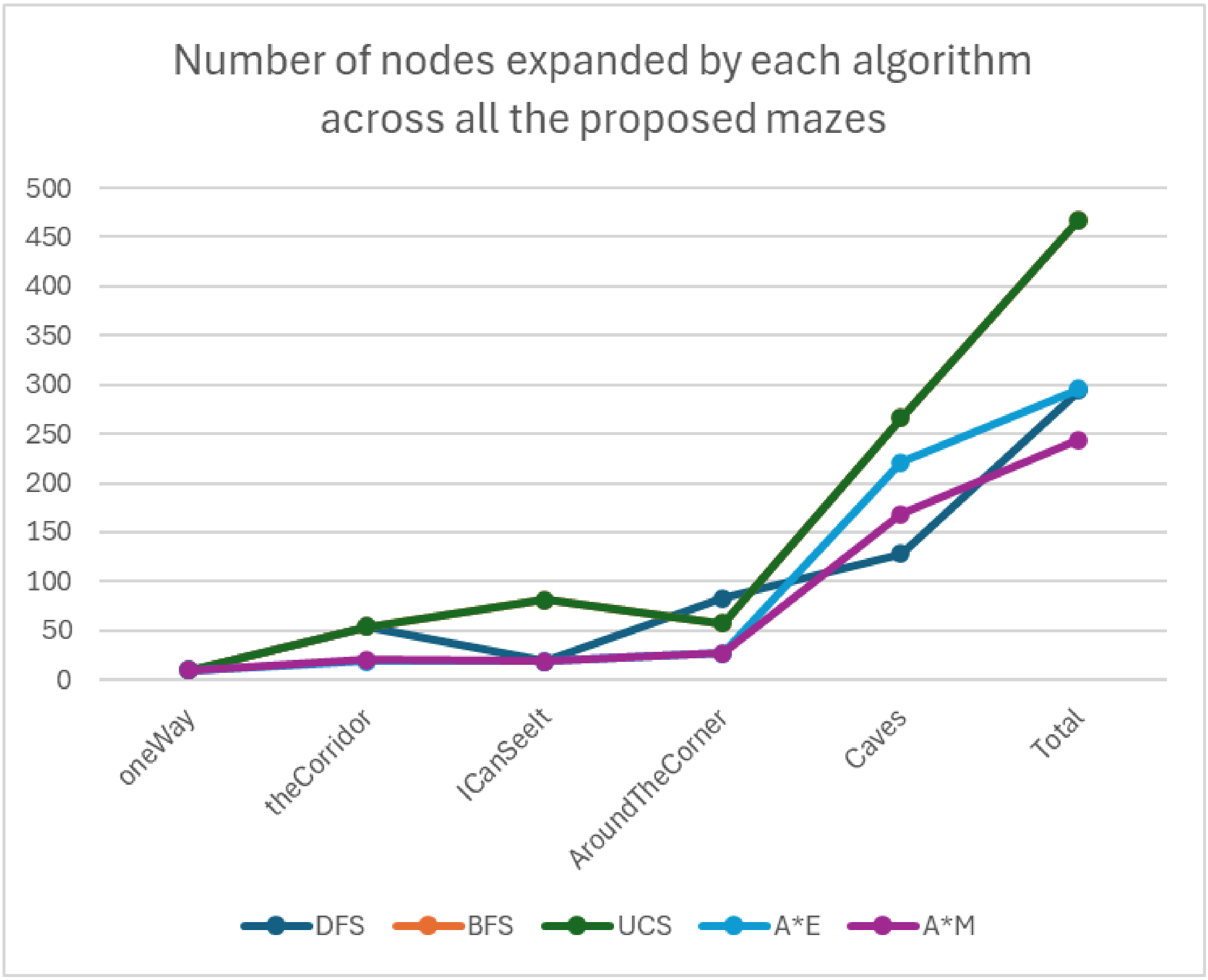


Figure 36: Expanded nodes across all of the proposed mazes

Figure 36 confirms our theoretical disadvantage of BFS and UCS algorithms, that is they need to explore a big amount of states in order to find the optimal solution. In comparison A* algorithm with Manhattan Distance as the heuristic function expanded almost 2 times less nodes, while still finding the optimal solution due to the admissibility of the heuristics function.

DFS performance confirms the quality of performance in deep mazes (for example the Caves) and the poor performance if the goal state is close to the starting point, but in a non prioritized path branch (AroundTheCorner).

The A* algorithms were performing alike until the Caves maze where the Euclidean Distance heuristic function lead the A*E algorithm to explore the opposite part of the maze. In this specific scenario it was a disadvantage, however this might prove very helpful if the only path leading to the goal state requires us to explore areas that aren't close to the goal state

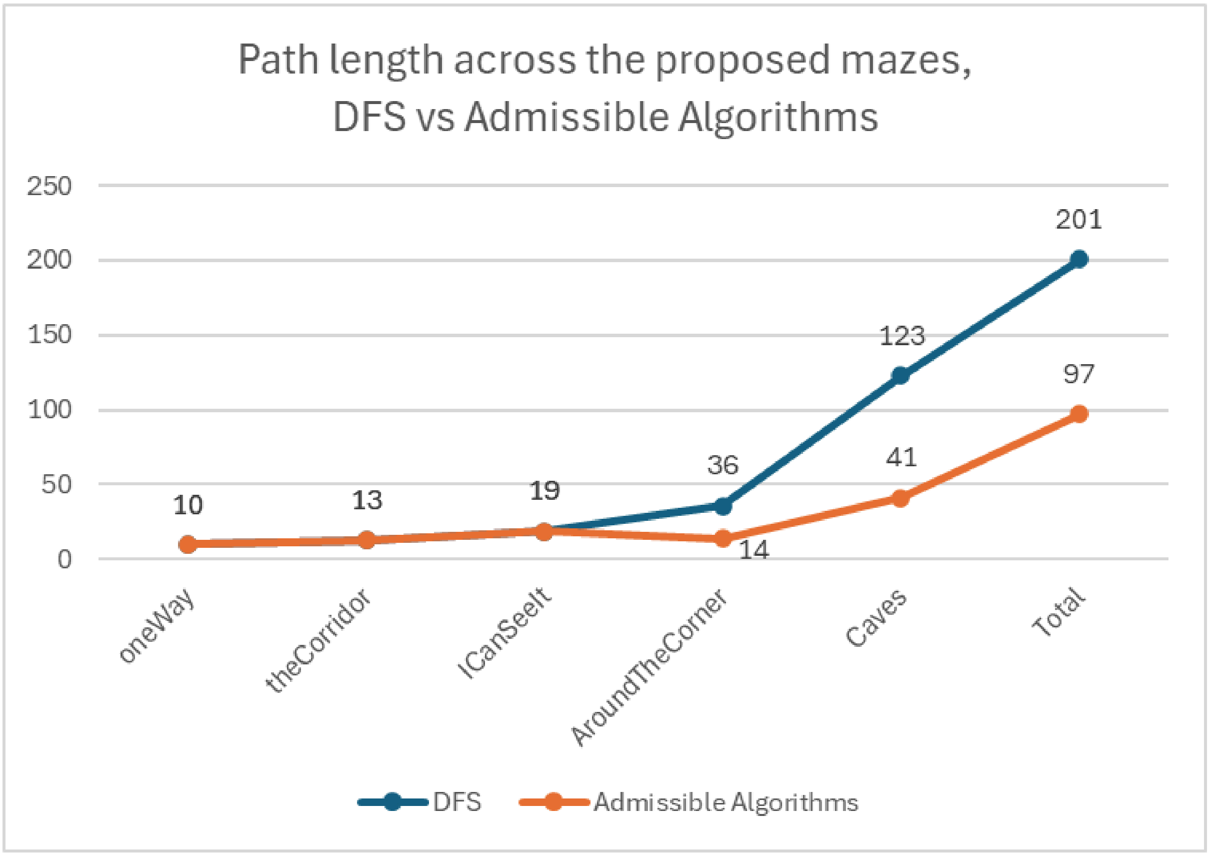


Figure 37: Path length across the proposed mazes.

Since all of the other algorithms proved to be admissible (always found the optimal solution) confirming our theoretical assumptions, we can reduce the amount of lines of the chart to only 2.

The performance of the DFS algorithm is also very expected. It found the optimal solutions in the first 3 mazes (although it wasn't too hard to do in the first maze since it was the only possible way), but struggled with optimality in the more complex problems.

For example in the caves it proposed a solution requiring exactly 3 times more steps to achieve the goal.

The overall path length was more than 2 times longer than the optimal. Even though the DFS performed exceptionally well in the previous two categories, the length of the paths proposed by it discards it from being an suitable algorithm in the problems that require the optimality of the solution.

Conclusions:

A* with Manhattan distance has proven to be the most consistent of the admissible algorithms proposed to us. It was performing the best in almost all the mazes (except for the execution time in the Caves) and had by far the least nodes explored in total.

Is it the perfect algorithm for our problem then? Let's try to trick it into working slower than the DFS and BFS algorithms.

Trying to dethrone the Manhattan Distance A* Algorithm

Case 1: The misleading intuition. DFS finding an optimal solution faster.

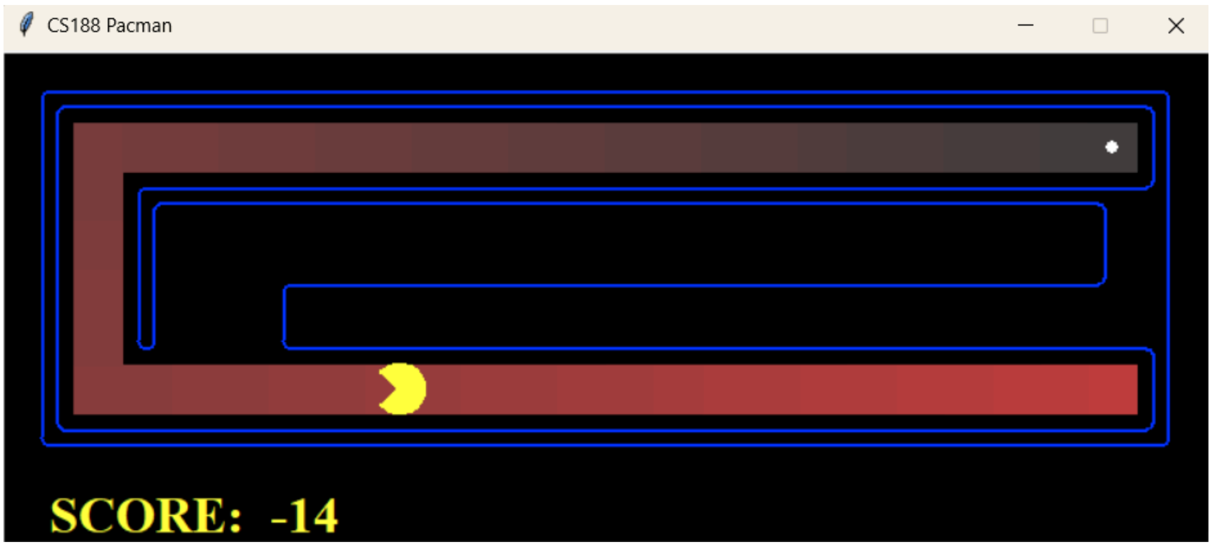


Figure 38: DFS finding the optimal solution

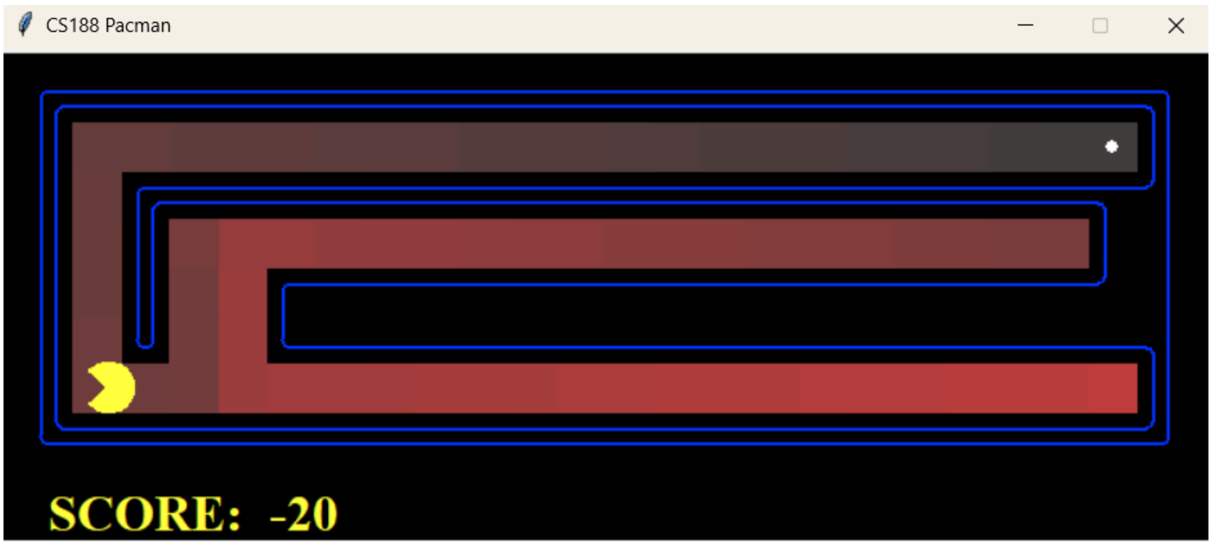


Figure 39: A* with Manhattan Heuristic getting mislead

Design of the maze from the figures 38 and 39 exposes one of the problems of the A*M algorithm. The heuristics function leads pacman to turn into a path that indeed goes into the direction of the goal state, however it does not have a direct connection to the food dot. Therefore DFS, which just continued the exploration of the first path, found the optimal solution faster.

Case 2: Can BFS be faster too?

Let's take a look at the following maze and its solution using the BFS.

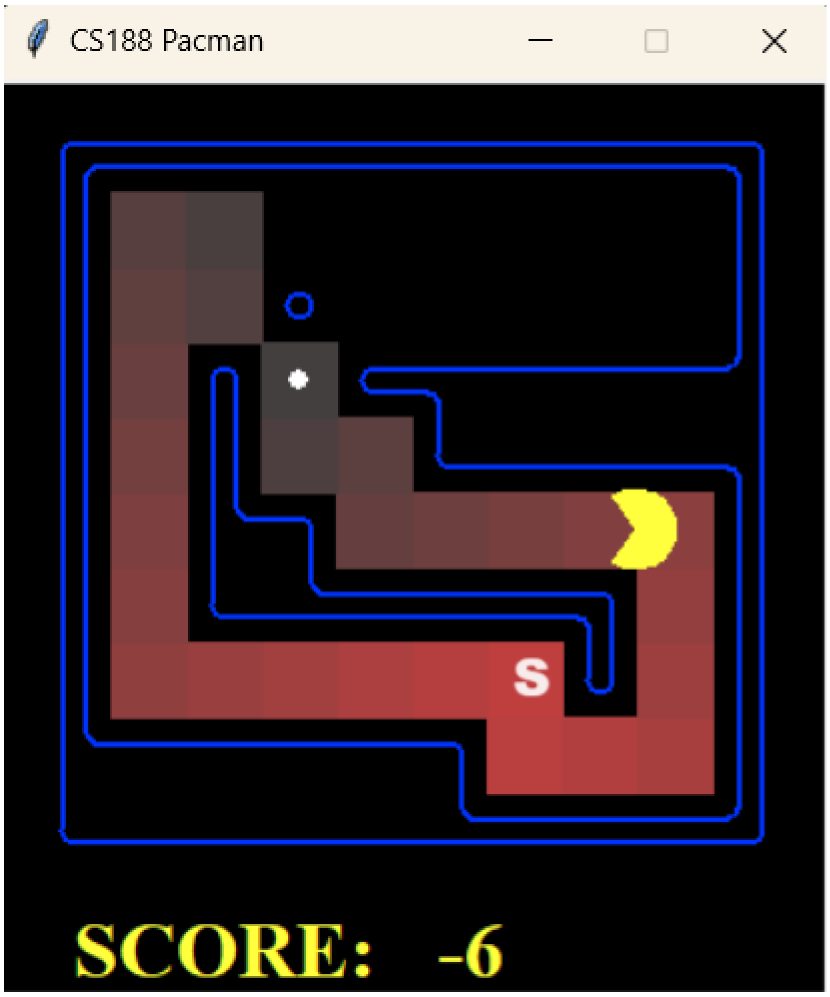


Figure 40: BFS solving a tricky problem

The solution looks promising. BFS didn't get tricked into exploring the places north of the goal state. They all have low Manhattan distance to the goal state so they should be easily prioritized by the A* algorithm, no?

Unfortunately that is not the case, let’s look at the A* solution:

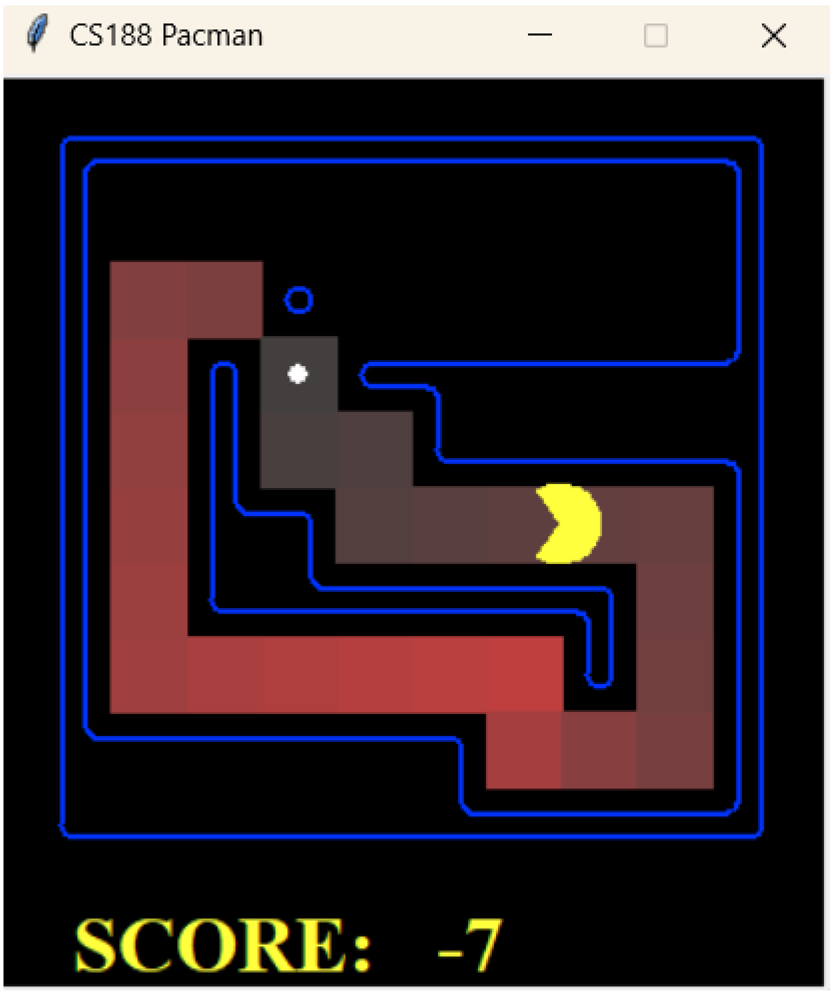


Figure 41: A* solving a tricky problem

What happened? Shouldn’t the A* explore the low heuristic area instead of going in the opposite direction?

Unfortunately, even though the other area had a low cost of the heuristic function, we can’t always go in the direction of the goal state without reaching it and the cost of getting to the northern part of the maze was already high.

The A* algorithm stumbled upon a decision to make. The two paths with the highest priority were the exploration of the northern area and going east in the southern part of the maze. The algorithm was exploring two paths simultaneously, until the southern path started to turn in the direction of the goal state, getting higher priority and eventually reaching it.

Would it ever be possible for BFS to achieve the goal faster? Let's consider the following restrictions:

- Misleading path must be longer than the path to get to the goal state, otherwise the BFS explores the whole path before reaching the food dot anyways
    - If that is the case, the cost of exploring the path gets too high and A* traces back to explore the directions opposite to the goal state (as it did in the figure 41)
- There are no multiple goals to mislead the A* algorithm further
    - the fixed food dot problem doesn't allow multiple goal states

Taking that into the consideration it is not possible to design a maze in which BFS outperforms A* with Manhattan Distance in the number of extended nodes without increasing the wage of the heuristics function to encourage the algorithm to further explore misleading paths. Therefore A* with Manhattan Distance as a heuristic function will always find the optimal solution to the problem, exploring less or equal amount of nodes as BFS.

.

Part 2: Slight modification of the problem

In this part of the Problem of finding a food dot we will do a slight modification, that is we will introduce the possibility of diagonal movement. To do that we have to modify some code.

The relevant code parts that have been modified:

- **class Directions (file game.py):**
    - introduction of the new directions:
        - NORTHWEST = ‘NORTHWEST’
        - NORTHEAST = ‘NORTHEAST’
        - SOUTHEAST = ‘SOUTHEAST’
        - SOUTHWEST = ‘SOUTHWEST’
- **class Actions (file game.py):**
    - introduction of the new directions in the _directions dictionary for the directionToVector method to work
        - `Directions.NORTHWEST: (-1, 1),`
        - `Directions.NORTHEAST: (1, 1),`
        - `Directions.SOUTHEAST: (1, -1),`
        - `Directions.SOUTHWEST: (-1, -1)`

- **class PositionSearchProblem (file searchAgents.py)**
    - **getSuccessors(self, state)**
        - introduction of the new directions in the for loop

    -
      ```
      for action in [Directions.NORTH, Directions.SOUTH,
                     Directions.EAST, Directions.WEST,
                     Directions.NORTHWEST,         Directions.NORTHEAST,
                     Directions.SOUTHEAST, Directions.SOUTHWEST]:
      ```

- New heuristic function (file searchAgents.py)

Since we have introduced the diagonal movement, our manhattanHeuristic and euclideanHeuristic have stopped being admissible:

- For example if the goal state is 1 diagonal step from us, the manhattanHeuristic will have a value of 2 and euclideanHeuristic will have a value of sqrt(2), therefore in both cases the h(x) > h*(x), so the functions aren’t admissible anymore

To achieve admissibility of the A* algorithm we have to introduce a new heuristic function. My proposition is a maximumAxisDiffHeuristic, that returns the biggest of the differences between the coordinates at the X axis and Y axis of goal and position. The pseudocode of this heuristic function might look like this:

```
def maximumAxisDiffHeuristic(curr_state, goal_state):
      curr_x, curr_y = curr_state
      goal_X, goal_Y = goal_state

      return (max(abs(curr_x - goal_x), abs(curr_y - goal_y))
```

The python implementation of the heuristic:

```
def maximumAxisDistance(point1, point2):
   x1, y1 = point1
   x2, y2 = point2
   return (max(abs(x1-x2), abs(y1-y2)))


def maximumAxisDiffHeuristic(position, problem, info={}):

   return minimunAxisDistance(position, problem.goal)
```

The new heuristic is admissible, because the amount of moves needed to achieve a goal state from the current state using the diagonal, horizontal and vertical movement won't be smaller than the biggest of the differences between the values of X and Y coordinates of the goal state and current state (we can't move more than 1 point in any of the axis with a single movement and the cost of each action is 1).

Now that we have introduced the possibility of diagonal movement, let's see how A* with different heuristics manage to solve the previous mazes.

Level 1: Let’s cut the corner.

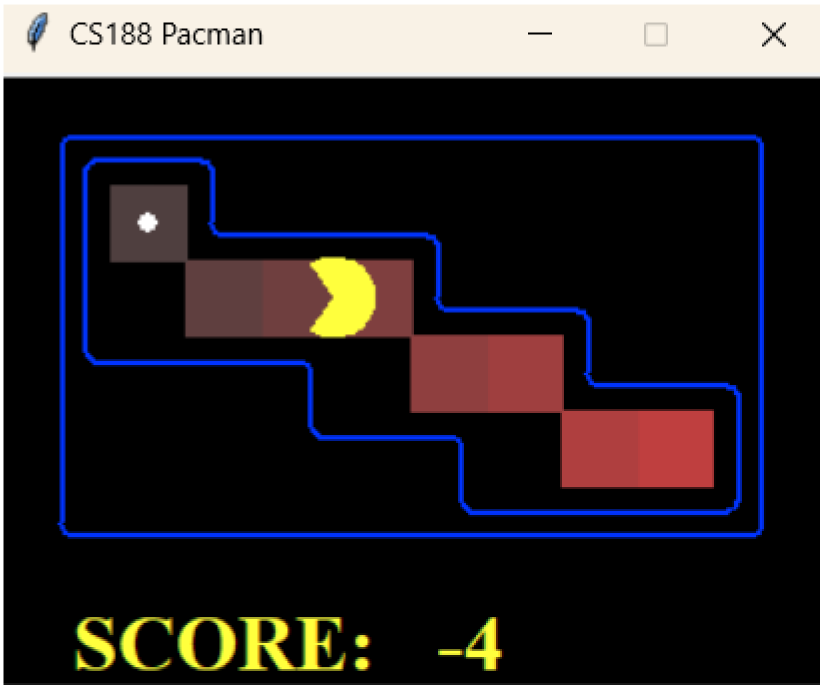


Figure 42: Diagonal movement, Manhattan and Euclidean heuristic in oneWay maze

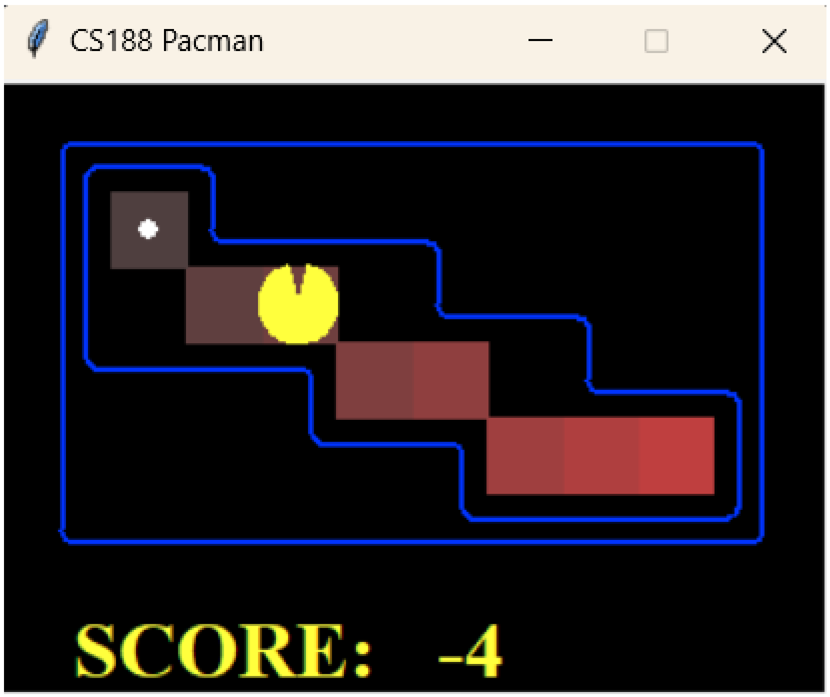


Figure 43: Diagonal movement, maximumAxisDiffHeuristic

	Euclidean	Manhattan	maximumAxisDiff
execution time (ms) (avg of 10 iterations)	2,03	1,93	2,05
nodes expanded	7	7	7
path length	7	7	7

Even though the results of every algorithm are very similar, we can already see the small difference between the old heuristics and the new. Manhattan and Euclidean distance heuristics prioritized going diagonally towards the goal state, when the maximumAxisDifference heuristic saw that there is no difference in the longest distance between coordinates of goal state and explored state between going diagonally and straight at the first three corners and preferred to go straight, because it was the first option it has explored.

All the results were either equal or the differences were insignificant, therefore we will skip charting this problem.

Level 2: The Corridor again. Remember about the walls!

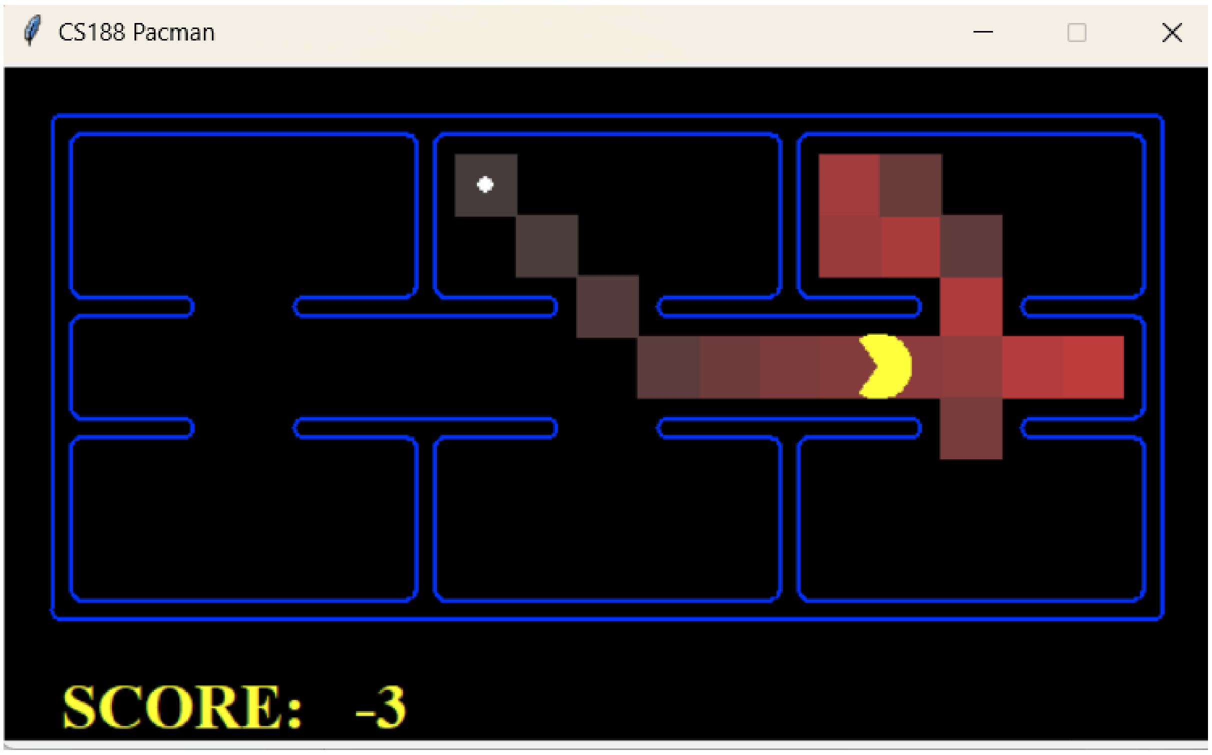


Figure 44: Diagonal movement, Euclidean Heuristic in the Corridor

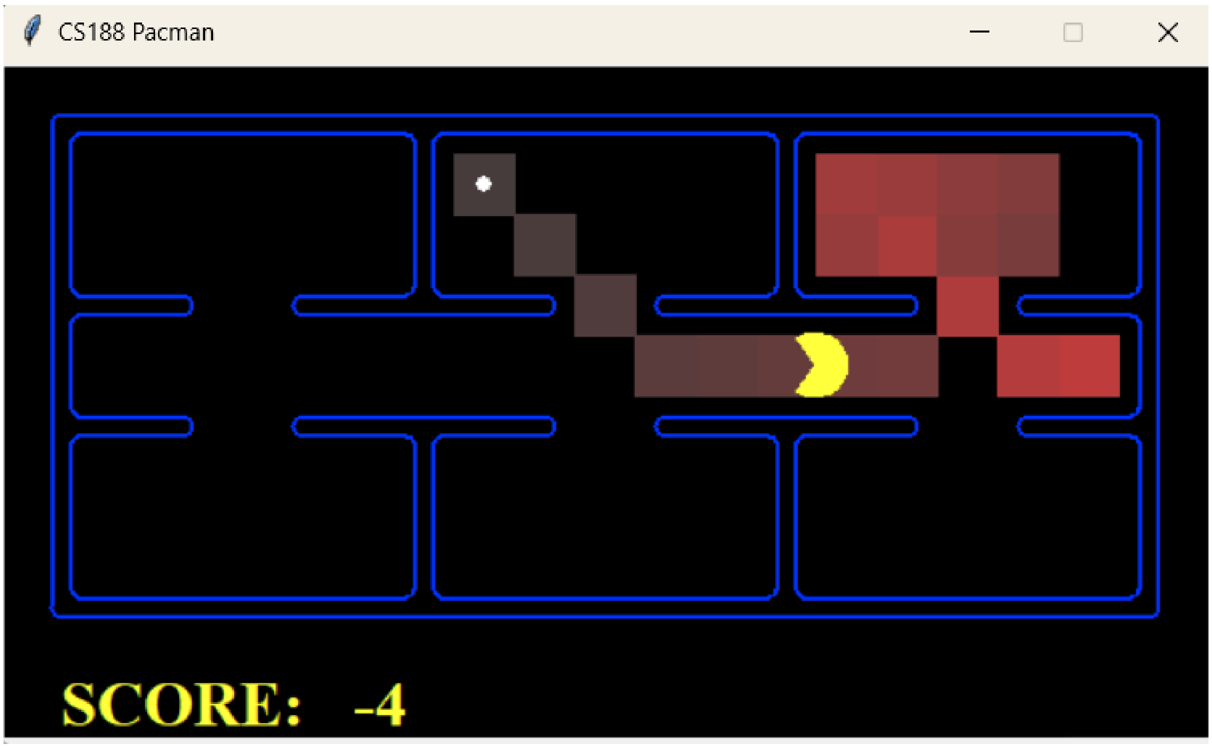


Figure 45: Diagonal movement, Manhattan heuristic in the Corridor

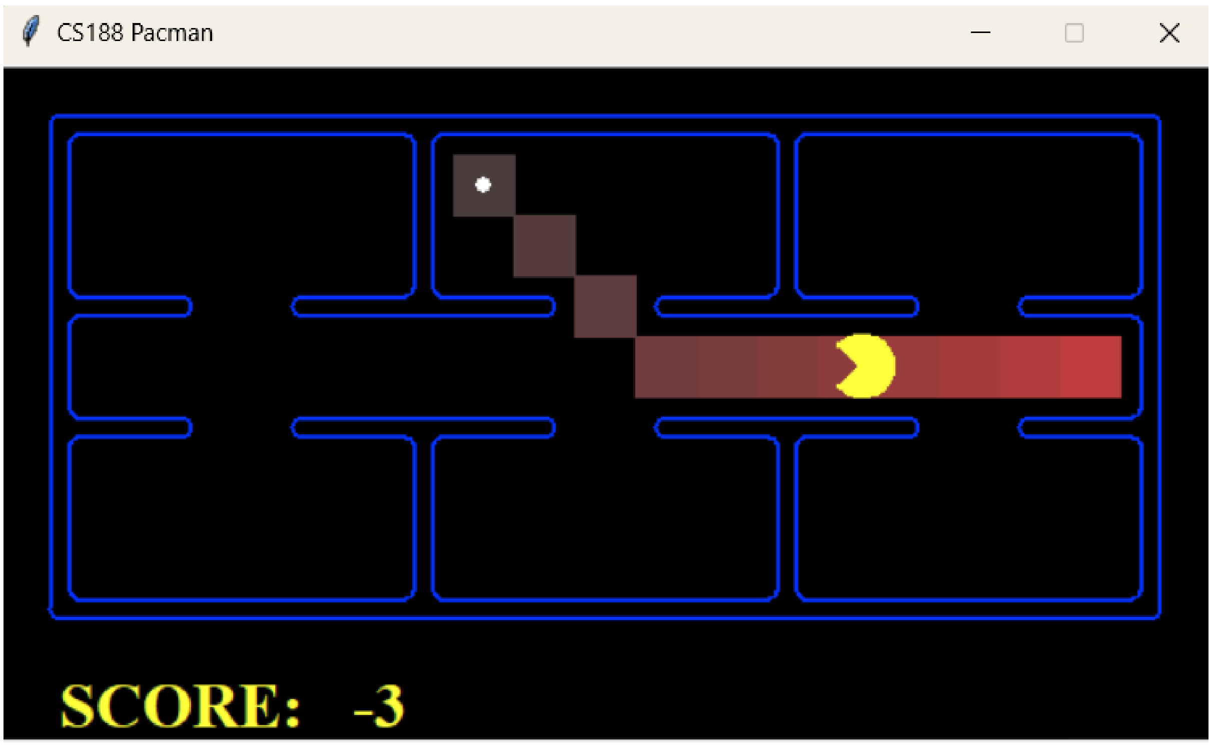


Figure 46: Diagonal movement, maximumAxisDiff heuristic in the Corridor

	Euclidean	Manhattan	maximumAxisDiff
execution time (ms)	3,75 188%	3,39 170%	**2,00** **100%**
nodes expanded	17 170%	18 180%	**10** **100%**
path length	**10**	**10**	**10**

Again every algorithm has found the optimal path. This time though the maximumAxisDiff is the clear winner. The other algorithms took almost 2 times as much time to compile and also got baited into exploring the top right room which resulted in almost two times as many nodes expanded.

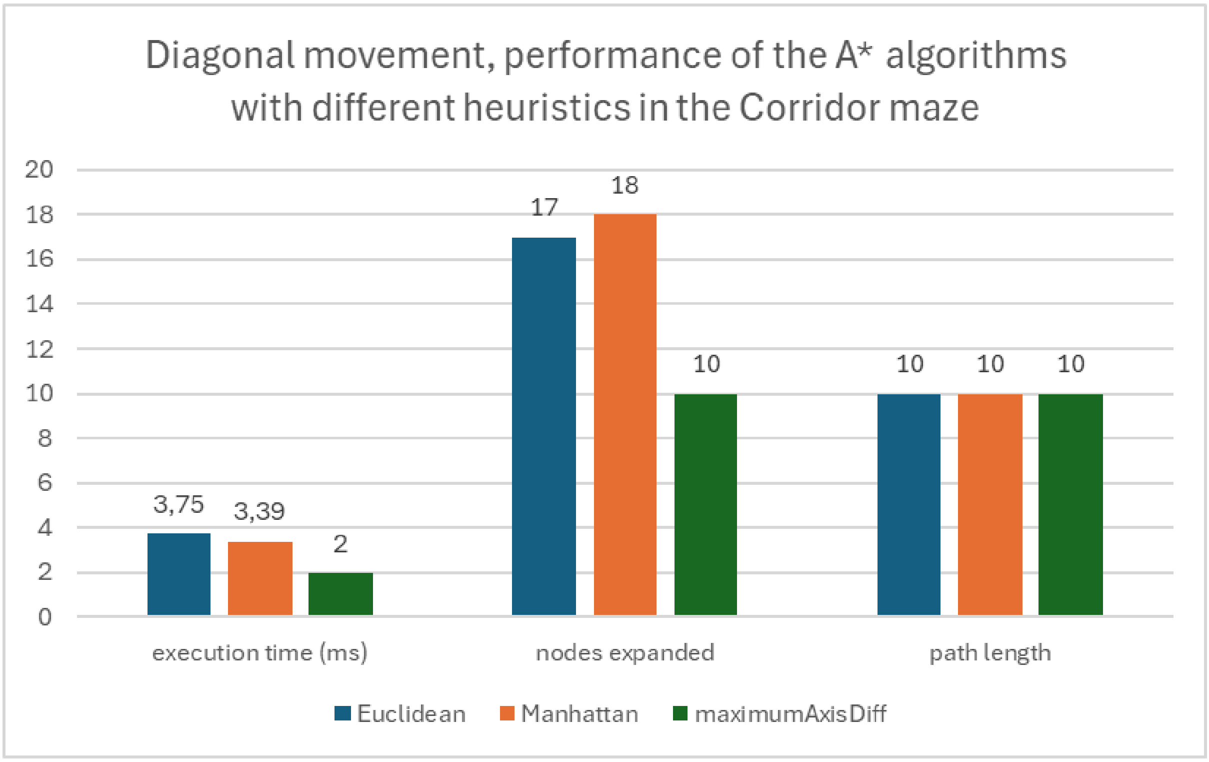


Figure 47: Diagonal movement, performance of A* algorithms in the Corridor

Level 3: They all can see it. What about the corner?

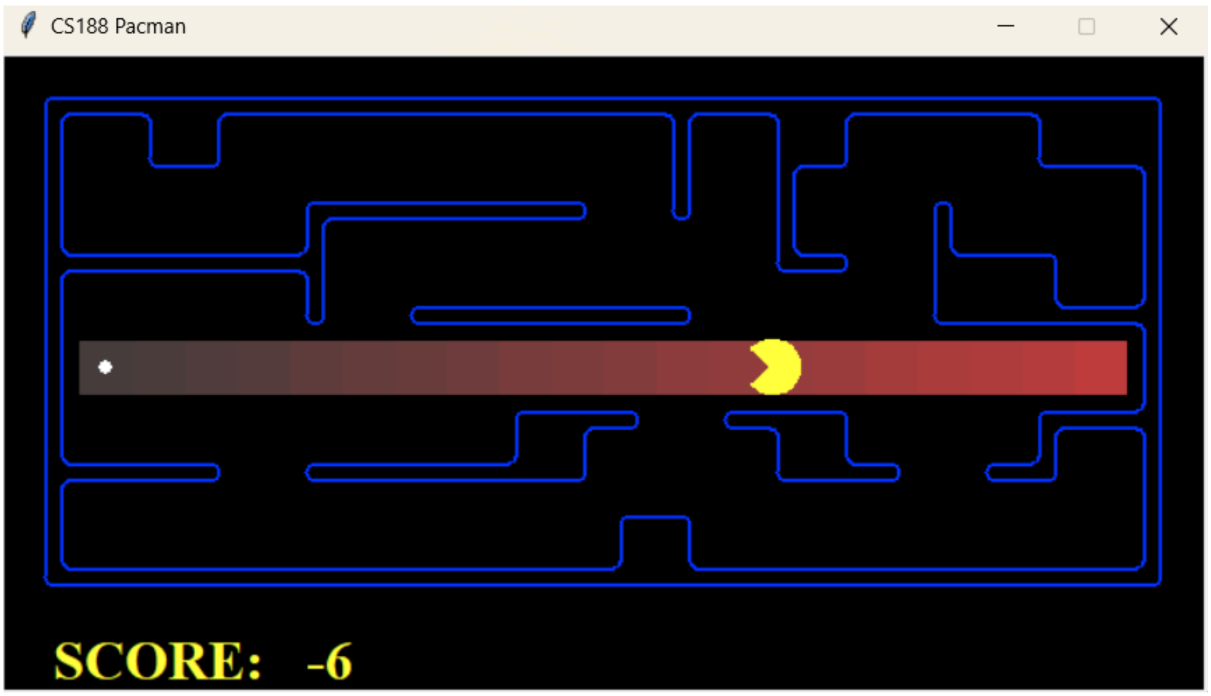


Figure 48: Diagonal movement. A* algorithms in the ICanSeeIt maze

	Euclidean	Manhattan	maximumAxisDiff
execution time (ms)	3,54	3,25	**3,00**
nodes expanded	19	19	19
path length	19	19	19

Another trivial example. Euclidean heuristic requires a little bit more time to compile in this one, however that is the only distance. Let's have a look into the Around The Corner maze.

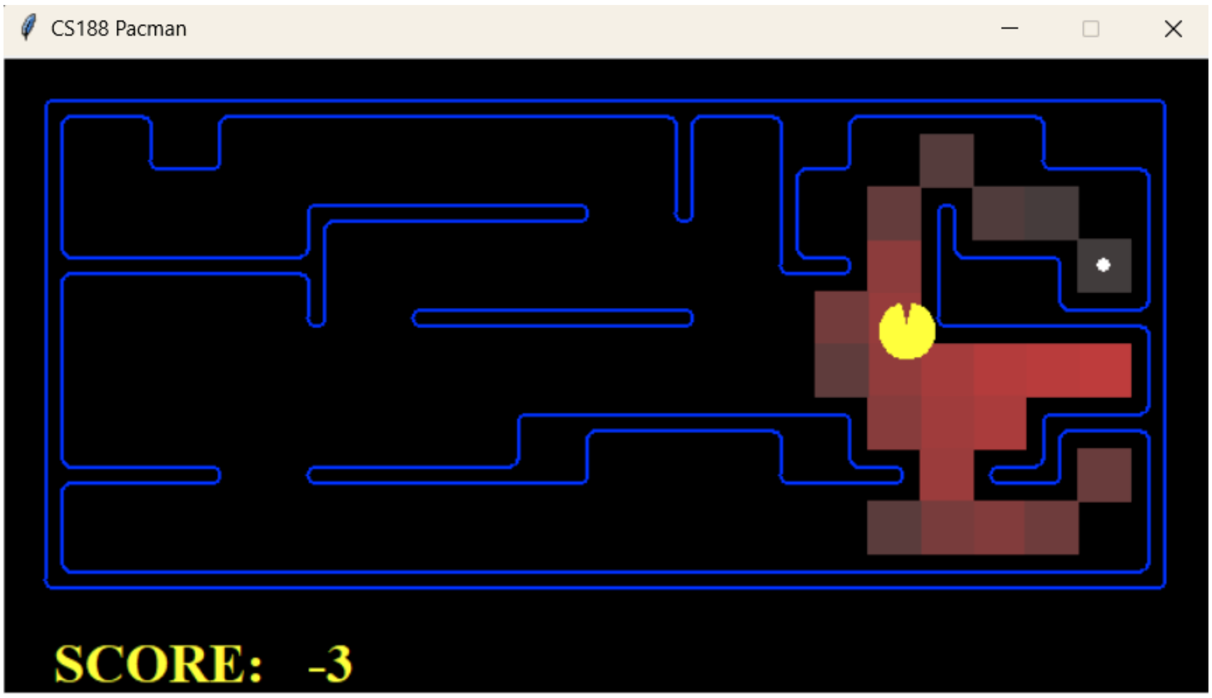


Figure 49: Diagonal movement, Euclidean heuristic in the AroundTheCorner maze

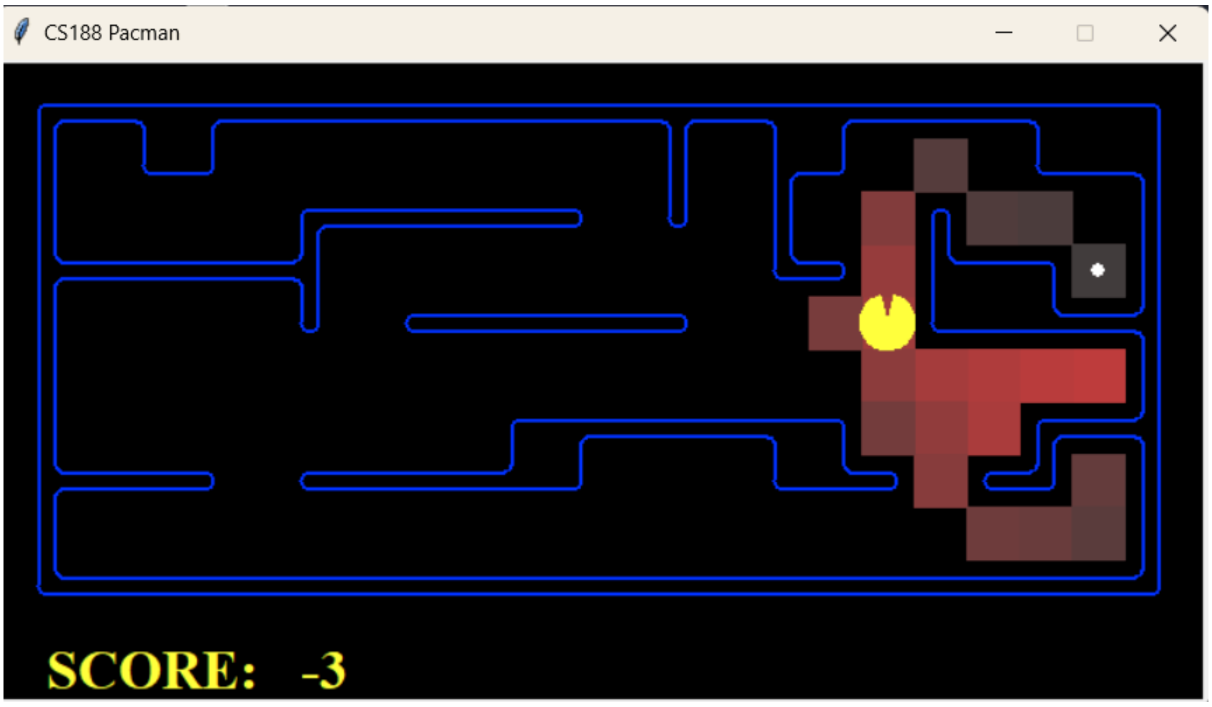


Figure 50: Diagonal movement, Manhattan Heuristic in the AroundTheCorner maze

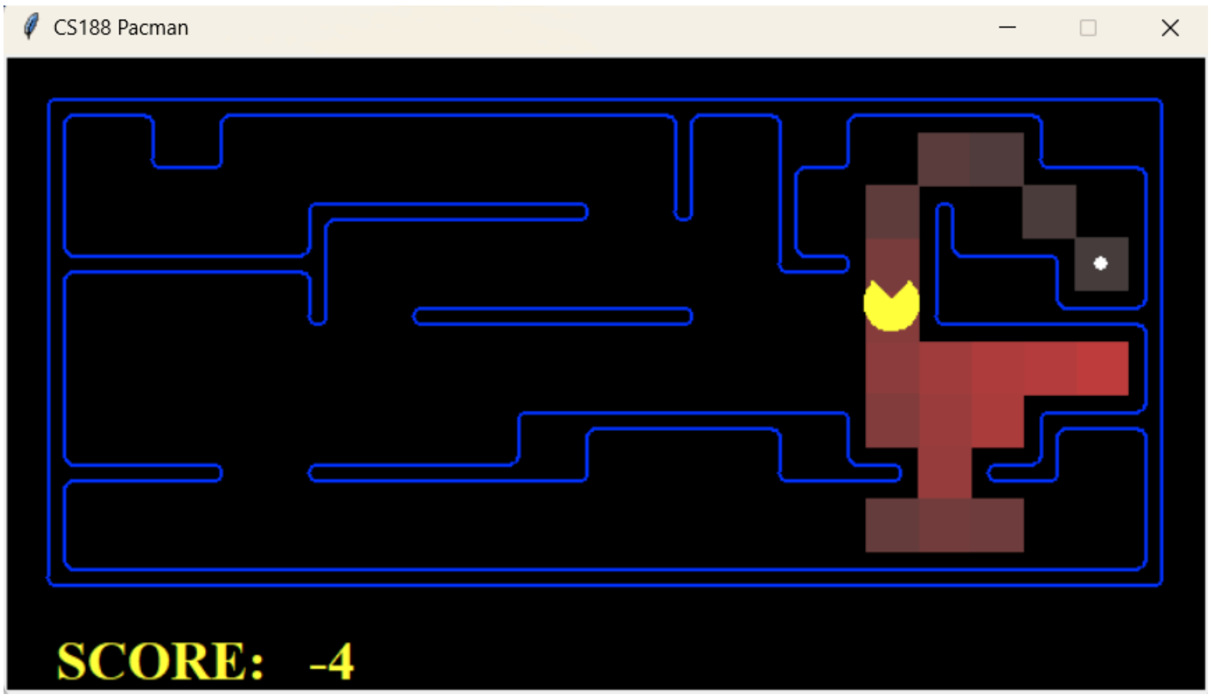


Figure 51: Diagonal movement, maximumAxisDiff in the Around the corner

	Euclidean	Manhattan	**maximumAxisDiff**
execution time (ms)	4,48 115%	4,00 102%	**3,89** **100%**
nodes expanded	22 122%	20 111%	**18** **100%**
path lenght	**10**	**10**	**10**

Yet another example where maximumAxisDiff is proving to work the best in the diagonal movement problems. The differences however are again not that significant. We have already checked out 4 of 5 of the proposed mazes and we have not yet found a solution that proves the Euclidean and Manhattan heuristics to not be admissible.

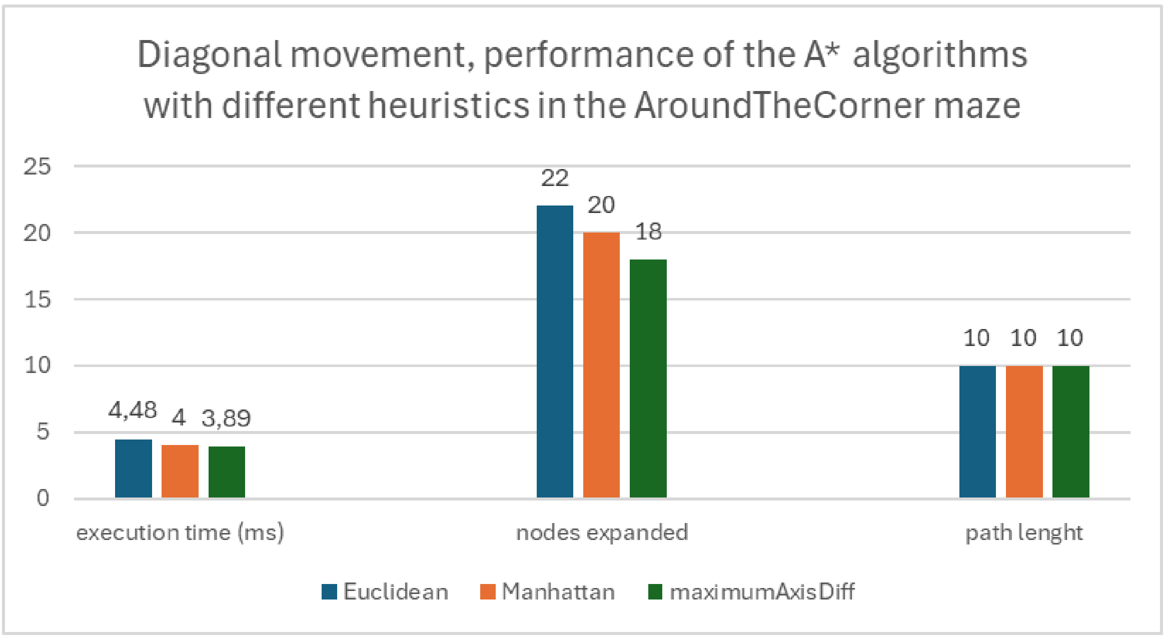


Figure 52: Diagonal movement, comparison of A* algorithms in the AroundTheCorner maze

Level 4: The Caves. Will they all find the optimal way?

This was the level that has exposed the disadvantages of each algorithm without the introduction of diagonal movement. How will the A* Euclidean and Manhattan heuristics do this time? Will the maximumDiffAxis heuristic prove the dominance once again?

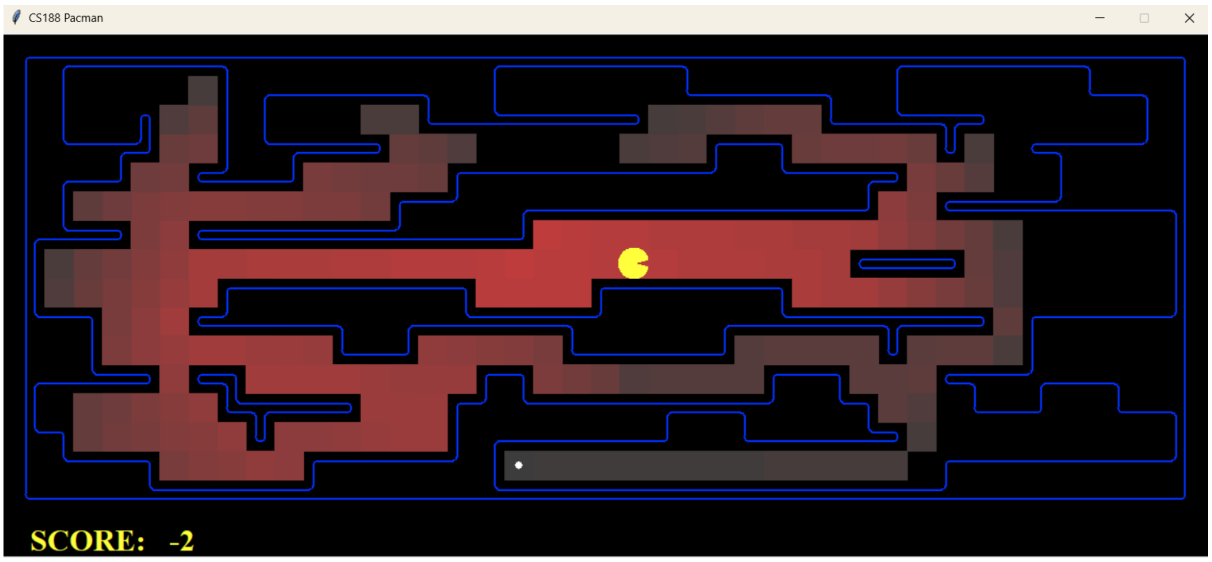


Figure 53: Diagonal movement, Euclidean distance heuristic A* in the Caves

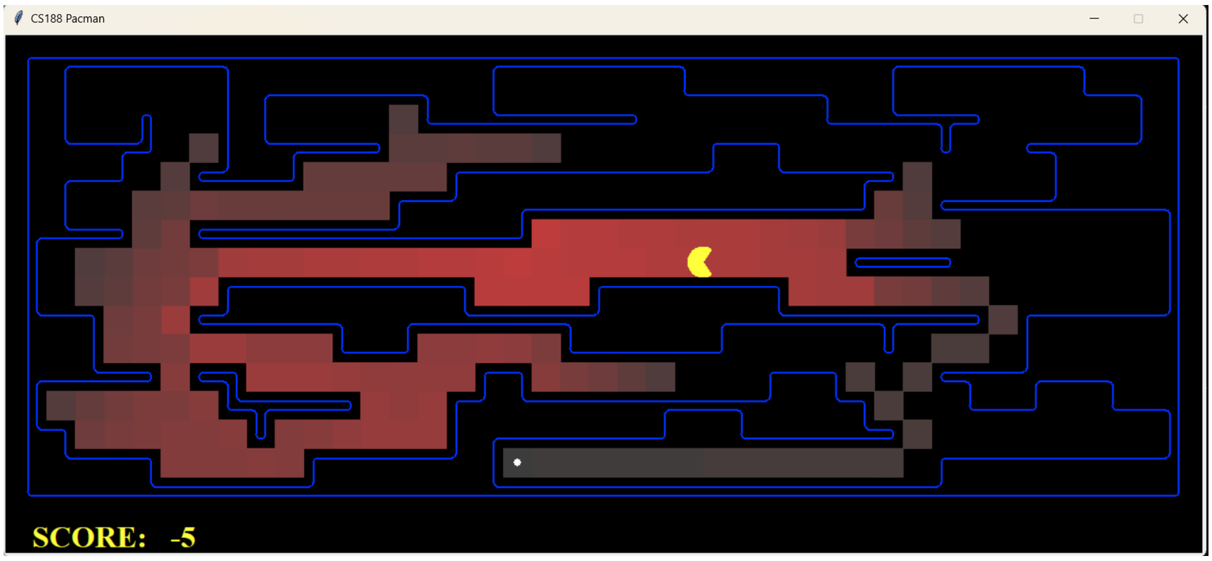


Figure 54: Diagonal movement, Manhattan distance heuristic A* in the Caves

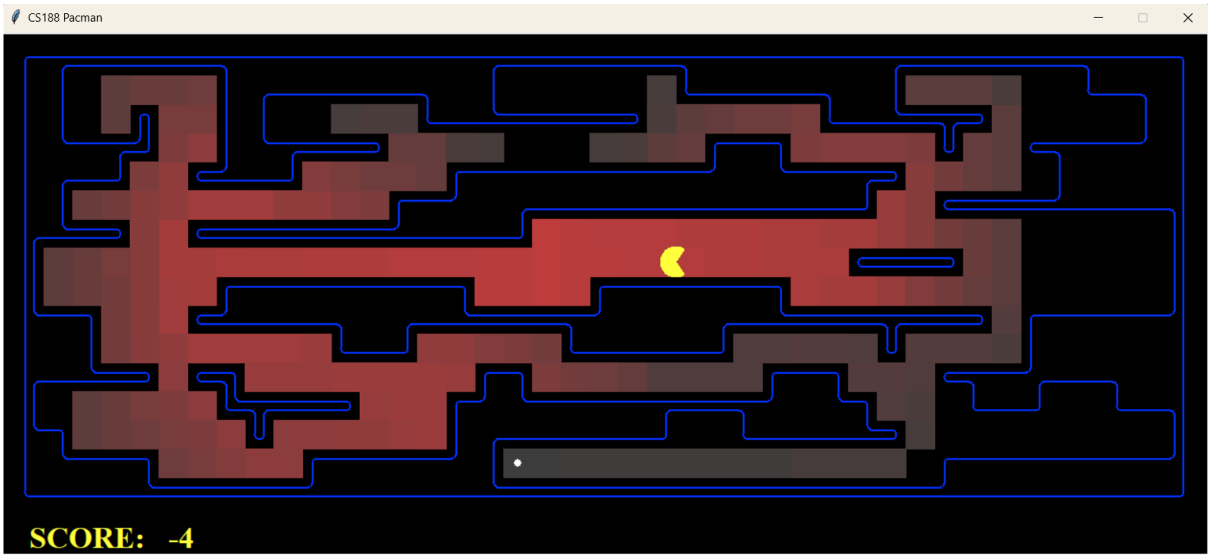


Figure 55: Diagonal movement, maximumAxisDiff heuristic A* in the Caves

	Euclidean	Manhattan	maximumAxisDiff
execution time (ms)	36,2	**25,3**	38,2
nodes expanded	202	**162**	217
path length	35	**35**	35

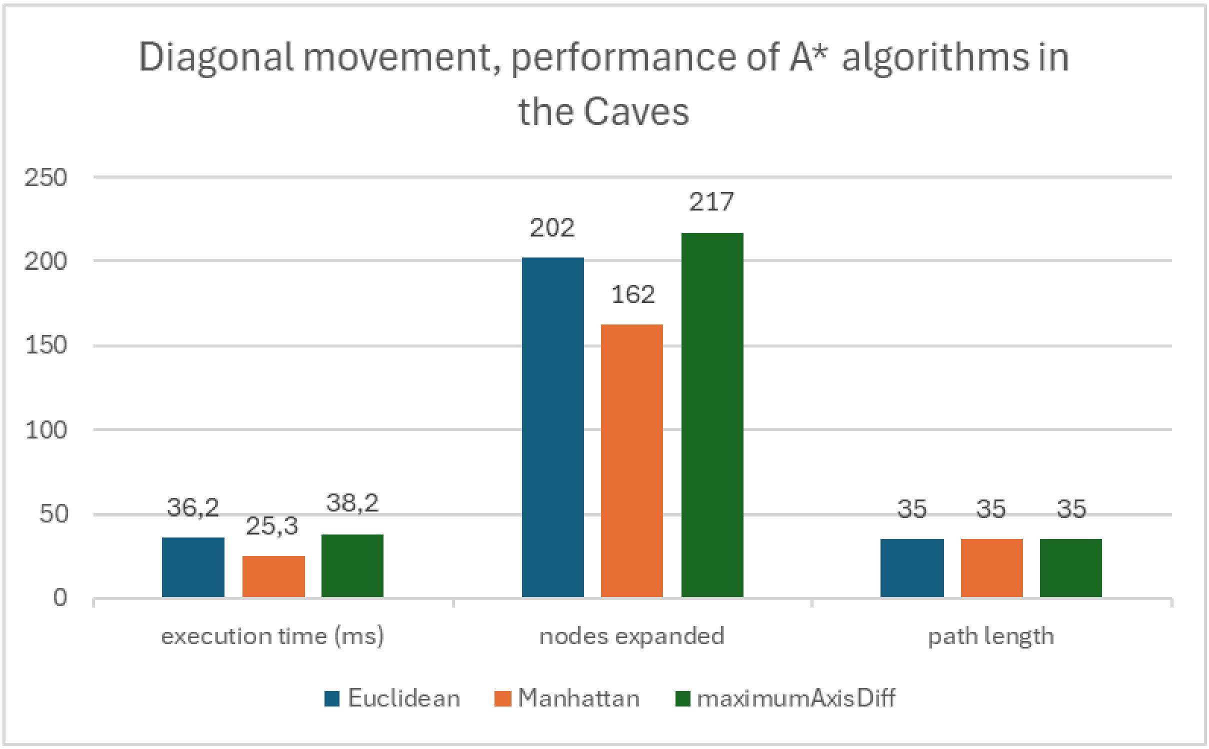


Figure 56: Diagonal movement, comparison of A* algorithms in the Caves

Not only the maximumAxisDiff heuristic has not proved to be the best, it has performed the worst in every category (aside from path length, where each of the algorithms found the optimal path).

This exposes the main weakness of the new heuristic. The maximumAxisDifference doesn't perform the best if the axis taken into consideration in the heuristics is blocked by a lot of walls (for example in this case). The algorithm has to work its way around all the tiles of most of the corridors close to the goal state, because their maximum axis distance is equal to the distance of the difference of the axis Y (in this case) while the other heuristics look mostly for the paths that lead us the closest to the goal state geometrically.

Comparison of the A* heuristics for diagonal movement problem.

Let’s take a look at the comparison of the single categories in each of the mazes.

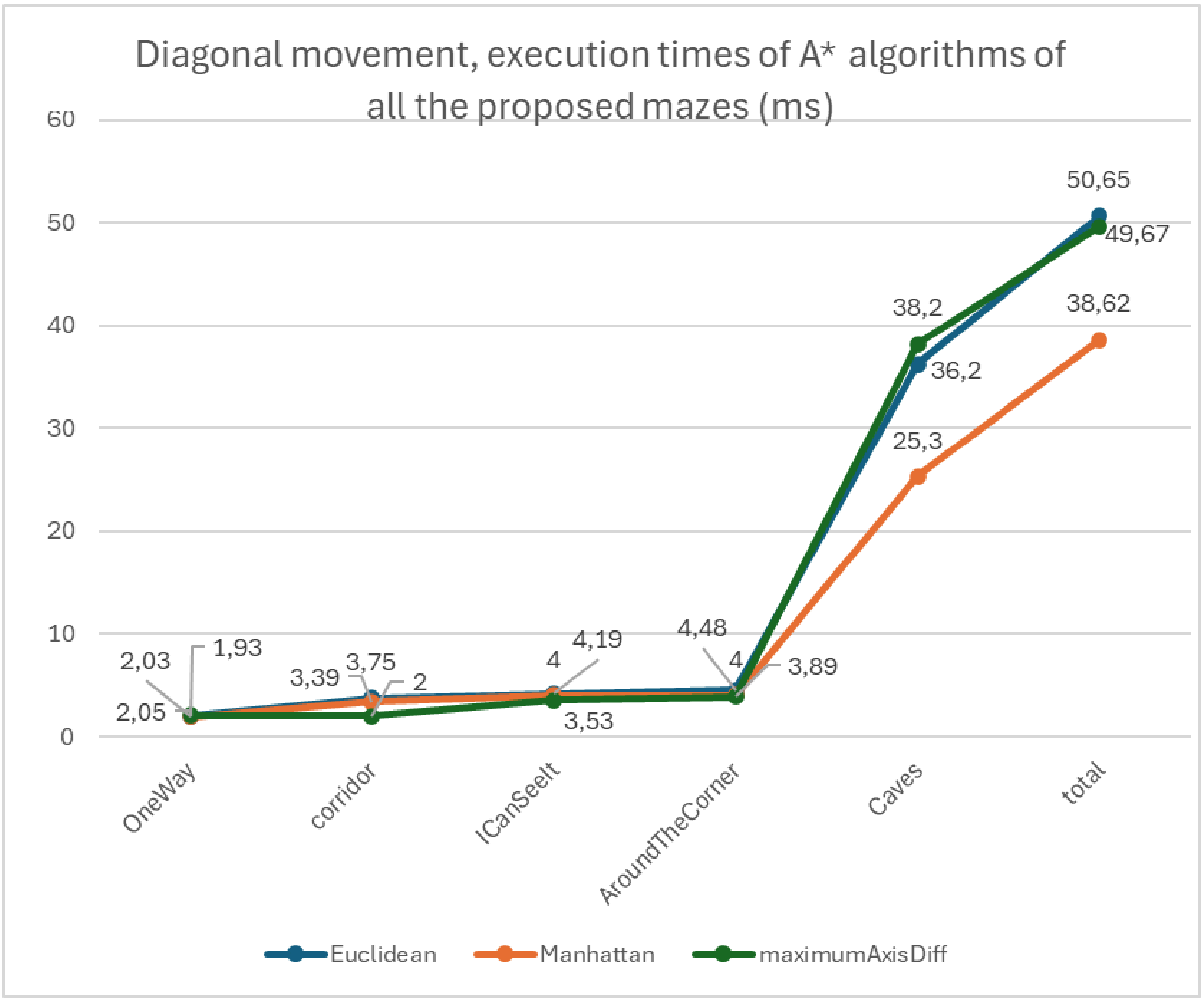


Figure 57: Diagonal movement, comparison of execution times of A* algorithms

Until the Caves maze all the algorithms performed more or less the same (except for the exceptional performance by maximumAxisDiff in the Corridor). In the Caves maze Manhattan Distance heuristic A* has proven its domination once again. Requiring less than 70% of execution time of the other algorithms in the Caves it took the lead in the total execution time, even though it was the slowest one before it. Maybe the situation would have looked differently if we compared more big mazes.

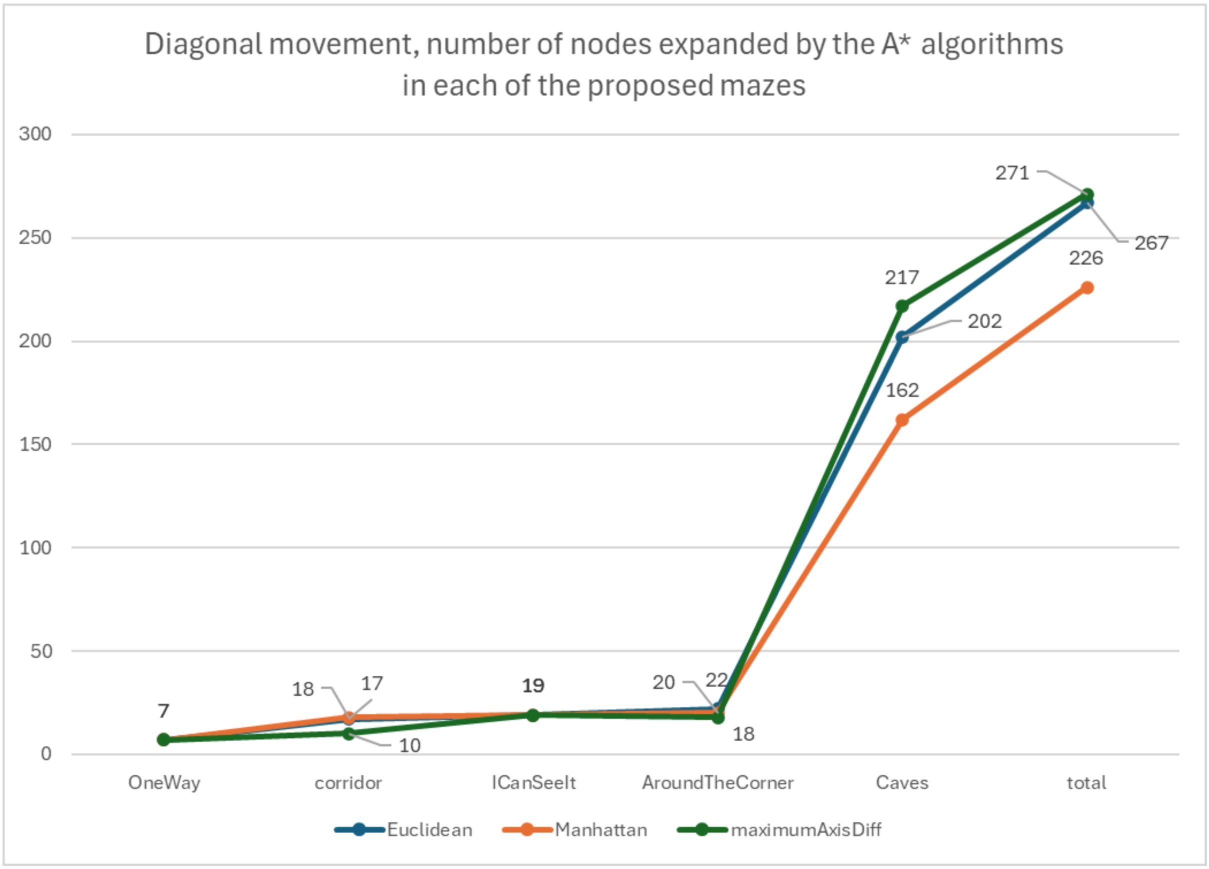


Figure 58: Diagonal movement, comparison of number of nodes expanded by A* algorithms

This chart resembles the execution times chart a lot. The main difference is that we can see the dominance of maximumAxisDiff in the corridor and the margin by which the Manhattan Distance heuristics won is (it has 84% of the nodes expanded by the Euclidean heuristics, while it only had 78% of its execution time).

The chart for the path length of each algorithm is redundant, since each of the algorithms has found the optimal solution in all of the mazes even though the Manhattan and Euclidean distance heuristics aren't admissible anymore.

What we can learn from that is that the non admissibility of used heuristics doesn't mean that it is never going to find the optimal solution, just that it is not guaranteed to find it.

.

Is the Manhattan distance always the way to go then? Will it always find the best solution? The optimality problem.

Let’s take a look at the following layout of the maze prepared for this problem and how the admissible heuristic algorithm finds the optimal path.

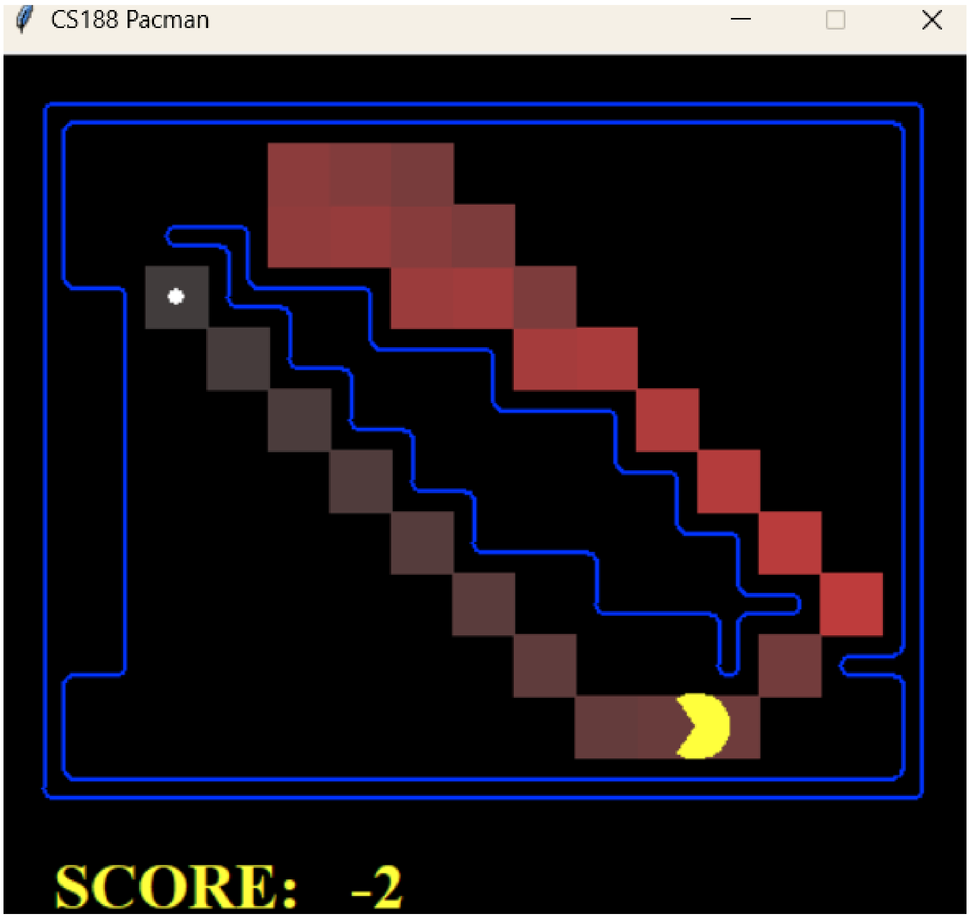


Figure 59: Diagonal movement, maximumAxisDiff finding the optimal solution

As we can see, when the maximumAxisDiff A* got into the top corner of the ‘staircase’ it found out that There is not a move that makes the heuristic output lower. Obviously going left would get the pacman closer to the goal, however since the heuristics is admissible, the algorithm went for a search of another path that has a lesser cost. This way Pacman explored the bottom left corner from the start and found a path that doesn’t require going around the top of the staircase.

Will the Manhattan Distance A* also find that path or will it get greedy, because of the nature of its heuristic function? Let’s find out.

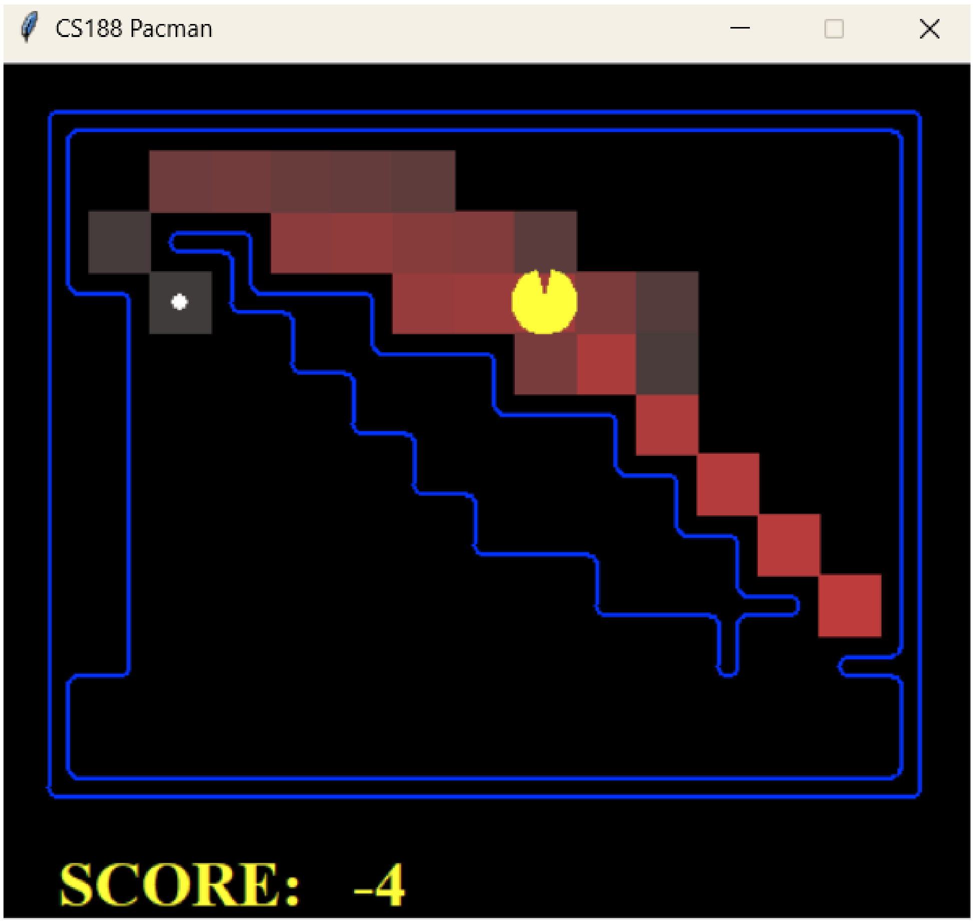


Figure 60: Diagonal movement, manhattanDistance heuristic getting too greedy.

As we can see from the figure 60, the manhattanDistance fell for a trap, since the ties between priorities are resolved by comparing the value of the heuristic function, the algorithm has chosen to further explore the top side of the map, instead of going back to start and finding the optimal path therefore we have proven that the manhattanDistance heuristic is indeed non admissible..

Here is the comparison of the performance of those two algorithms in this maze.

	maximumAxisDiff	Manhattan Distance
execution time (ms)	5,54	4,99
nodes expanded	26	23
path length	11	13

The path taken by the manhattanDistance A* was longer by 2 steps and the maximum Axis Diff has only expanded 3 more nodes. This is a clear example why it's not always best to use the Manhattan Distance as a heuristic in this problem even though it has performed the best in the previous section. That concludes our comparison of A* algorithms.

2.3 Problem 2: Eating all the food dots.

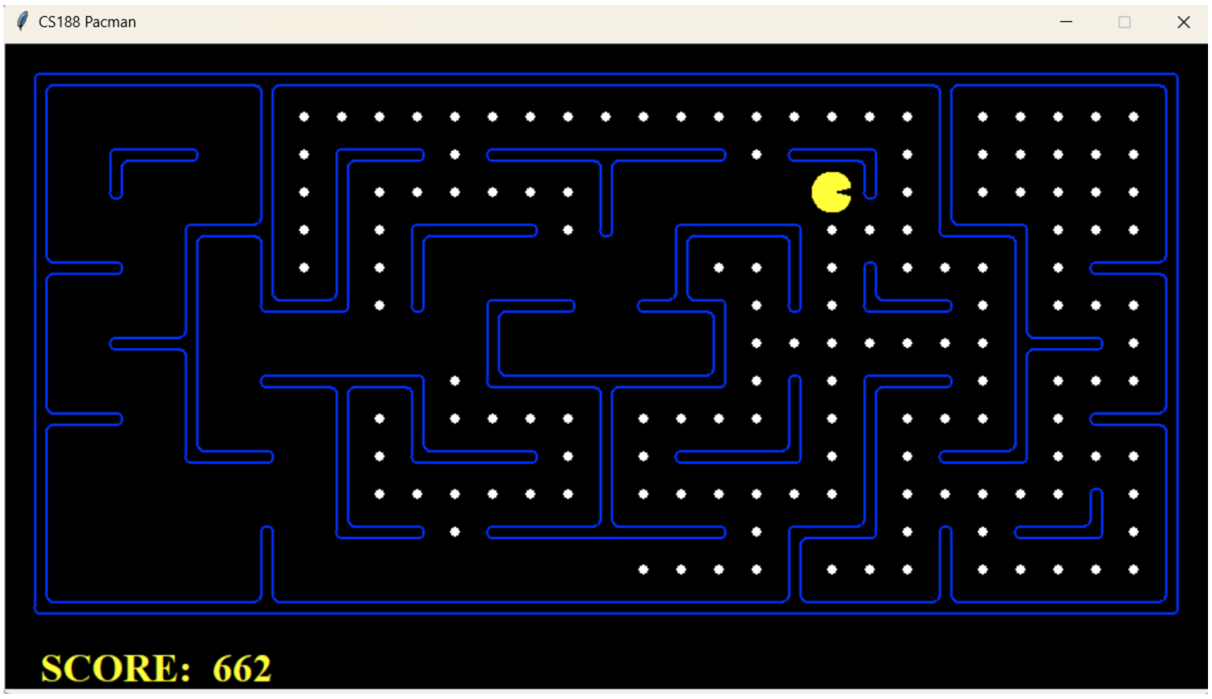


Figure 61: Problem 2: BigSearch layout

In this section, we shift our focus to a more complex problem where the maze has more than one food dot and Pacman has to collect all of them. This problem is defined by the FoodSearchProblem class in the searchAgents.py file.

This section will consist of the following parts:

1. Understanding of the code that implements the problem to solve
    a. Analysis of the FoodSearchProblem class and breakdown of its core mechanisms
2. Analytical explanation of the problem
    a. Description of the state space and operators
    b. Description of the initial state and the goal state
3. Understanding of the code responsible for implementing three search agents:
    a. Agent 1: A* with the foodHeuristicManhattan heuristic function
    b. Agent 2: A* with the foodHeuristicMaze heuristic function
    c. Agent 3: DotSearch sequences, finding a path to the closest dot
4. Comparison of the behavior of the search agents
    a. testing the behavior in 6 mazes
    b. comparison of the performance
    c. analysis and explanation of the performance

2.3.1 Understanding of the implementation of the problem:

The main difference in implementation compared to the first problem is the introduction of the food grid to the GameState.

Food grid was defined as an array of arrays of dimensions x and y representing the positions of the maze. To get the state of a position with coordinates x_cord and y_cord we use *grid[x_cord][y_cord]*.

An example grid for a 3x3 maze might look like this:

```
[ [False, False, True],
  [True, False, False],
  [True,  True,  True ] ]
```

Where True indicates the existence of a food node in the position represented by its x and y values.

With the introduction of the grid of nodes, the generateSuccesors function has been adjusted in the following way:

Whenever a successor of a state is created we copy the FoodGrid of its parent and mark the position of the successor as False (since the successor state represents a state in which Pacman has explored the successor node, therefore if there was a food dot, it was eaten by Pacman).

Search of the solutions is still made through the generalGraphSearch algorithm, however the heuristic functions have to use a grid of points.

2.3.2 Analytical explanation of the FoodSearchProblem:

a) State space

The search state is characterized by:
- Pacman position - the current position of Pacman
- Food grid - List of all the food dots and their position
- Walls grid - Coordinates of all the walls
- Pacmans orientation (last action done) - the direction Pacman is facing
- Score - the score of the game

b) Operators

Operators are the set of operations the Pacman can make, they stay the same as in the first part of the problem that is:

Directions:
- North
- South
- West
- East

Each of the operators is applicable if its application won't lead pacman to a position occupied by a wall.

The cost of each action is 1.

c) Initial state:

Initial state of the problem is characterized by:
- Starting position of Pacman
- Starting food grid

The relevant part of code is:

```
self.start = (startingGameState.getPacmanPosition(),
startingGameState.getFood())
```

d) Goal state

Goal state is the state in which no more food dots are remaining on the map.

The mechanism that checks if a state is a goal state is the isGoalState(state) function, which takes the foodGrid of a state and counts all the occurrences of 'True' - all the remaining food dots. If the count is equal to 0, that means the state is a goal state.

2.3.3 Understanding the algorithms used by agents to solve the problem:

We will take into the consideration 3 types of searchAgents:

a) Agent 1: A* with the foodHeuristicManhattan heuristic function
b) Agent 2: A* with the foodHeuristicMaze heuristic function
c) Agent 3: DotSearch sequences, finding a path to the closest dot

The main mechanism used by Agents 1 and 2 to solve the problem is still the generalGraphSearch function we have described at the very beginning of the record.

The agent 3 however has a unique approach to the problem, it is not exploring various possible states until exhaustion. Instead of that it is always looking for the closest food dot. We will go in depth into the implementation in a moment.

For the Agents 2 heuristic function to work properly, we have to undo the changes we introduced in the Diagonal Movement part of the first problem. From now on we will refer to the code implemented in the Part2 folder of the submission.

Let's start with analyzing the algorithms used by Agent 1:

a) Agent 1: A* with foodHeuristicManhattan heuristic function

As we have previously mentioned, the A* algorithm in the FoodSearchProblem is still using the same mechanism that we have described in the original analysis of algorithms part of the report. The only difference is the approach to calculating the heuristic.

The value returned by the heuristic function is the manhattanDistance to the furthest food dot.

The heuristic is admissible (never overestimates the cost of reaching the goal), because the distance from the state to the furthest food dot is always going to be smaller or equal to the cost of the actions needed to collect all the food dots (it includes traveling to the furthest food dot and the cost of a single action is 1).

Let's take a look at the pseudocode:

```
foodHeuristicManhattan(state):

      #retrieve the current position of Pacman and the foodGrid
      position, foodGrid = state

      heuristic = 0 #initialize the initial heuristic value as 0

      for each food in foodGrid: #iterate through all the food dots
            #calculate the manhattan distance to the food dot
            distance = manhattanDistance(position, food)

#if the calculated distance is bigger than the current heuristics value
            if (distance > heuristic):

#assign the new found maximum manhattan distance to the heuristic
                  heuristic = distance

return heuristic #return the manhattan distance to the furthest foodDot
```

As we can see, the algorithm is iterating through every food dot left in the grid, calculating the Manhattan distance to it and if the distance is larger than the previous biggest distance, it becomes the new heuristics value. That way the returned value is the Manhattan distance to the furthest food dot from the state.

b) Agent 2: A* with the foodHeuristicMazeDistance heuristic function

This heuristic function works in a similar way, however the distance used as the heuristic value is an actual cost of the optimal path from the current state to the furthest food dot.

The maze distance is calculated by simulating the PositionSearchProblem with the same layout, where the starting position is the current state position and the goal state is the position of the food node.

The originally implemented algorithm to find the solution is the Breadth-First Search algorithm, however if the compilation takes too much time we might replace it with the A* with the Manhattan distance as the heuristic function, since it has proven to be the most consistent admissible algorithm in the first problem.

```
foodHeuristicMazeDistance(state):

      #retrieve the current position of Pacman and the foodGrid
      position, foodGrid = state

      heuristic = 0 #initialize the initial heuristic value as 0

      for each food in foodGrid: #iterate through all the food dots
            #calculate the Maze distance to the food dot
            #the GameState is given as an argument so the mazeDistance
            #function can rebuild the layout of the map
            distance = mazeDistance(position, food, GameState)

#if the calculated distance is bigger than the current heuristics value
            if (distance > heuristic):

#assign the new found maximum manhattan distance to the heuristic
                  heuristic = distance

return heuristic #return the maze distance to the furthest foodDot
```

Since:
$mazeDistance(point1,\ point2,\ maze)\ \geq\ manhattanDistance(point1,\ point2)$

We know that the Agent 2's heuristic function is more informative, therefore Agent 2 will always expand fewer or equal number of nodes than Agent 1. The drawback is that finding the mazeDistance using BFS will take a lot of time and the calculation of the manhattanDistance is almost instantaneous.

c) Agent 3: DotSearchSequences

This agent takes a completely different approach to the problem. Instead of looking for the best possible path to solve the maze, it looks for the shortest path to the next food dot. After collecting one food it starts looking for the next closest one until all the food dots have been eaten.

It does not utilize the generalGraphSearch, instead it provides its own functionality to find the path to the closest dot utilizing a BFS type approach in a new problem: AnyFoodSearchProblem.

AnyFoodSearchProblem is a problem that inherits the methods of the PositionSearch problem. The only difference is the goalState calculation.

The state is a goal state whenever there is a food dot in its position, so there are many goal states in one maze.

The path to the closest foodDot is found and returned by the findPathToClosestDot(gameState) function. It implements a BFS approach based on a queue to explore the nodes closest to the current position and stops when it finds a food dot.

The agent uses the findPathToClosestDot function in a while loop with the condition of there being at least 1 food dot left in the foodGrid of the problem. The while loop has variables responsible for storing the following info:

a) the path pacman takes to eventually reach the goal state (the solution)
b) total number of expanded nodes
c) total cost of the path

Here are the breakdowns of the findPathToClosestDot function and the while loop that is responsible for searching the path to the goal state (a search algorithm).

```
def findPathToClosestDot(position, foodGrid, walls, problem):

      #initialize the variables
      BFS_queue = new Queue()
      visited = []
      action_list = []
      total_cost = 0

      BFS_queue.push(problem.startState)#push the startState to the queue

      while BFS_queue.notEmpty(): #while there are paths to explore
            node, actions = BFS_queue.pop() #retrieve the path
            if node not in visited: #if the position hasnt been expanded
                  visited.append(node) #mark it as visited

                  if problem.isGoalState(node): #if there is a food dot
                        #return the:
                        # actions required to reach the closest food dot
                        # amount of nodes expanded
                        # cost of actions to get the food dot
                        return actions, expanded, costOfActions(actions)


                  #if the node isn't a goal state
                  #generate the new branches of the problem
                  successors = generateSuccessors(node)

                  #add the new branches to the queue
                  for successor in successors:
                        BFS_queue.push(successor.position,
                                          actions + successor.direction)
                  #go back to exploring the next branches of the problem
```

```
search_Algorithm(state):

#initialize the variables responsible for measuring the performance
actions = []
startTime = currentTime()
totalExpanded = 0
totalCost = 0
currentState = state

while(currentState.countFood() > 0): #while there are still food dots
      nextPathSegment, expanded, cost = findPathToClosestDot(state)

      #update the variables
      totalExpanded += expanded
      totalCost += cost
      actions.append(nextPathSegment)

totalTime = currentTime() - startTime
```

Summary of the agents and the predictions of performance:

Agents 1 and 2 are using an A* algorithm with an admissible heuristic function, therefore they will always look for optimal solution (a path that requires pacman to do the least actions to achieve the final goal).

The heuristic function of Agent 2 is more informative than the heuristic function of Agent 1, therefore the agent 2 will always expand less or equal amount of states to find the optimal solution. The computational complexity of the mazeDistance heuristic function is however way bigger than the complexity of manhattanDistance heuristic, because it uses a BFS algorithm to find the value of the heuristic function n times, where n = amount of food left on the map, while the manhattanDistance just calculates an easy problem n times.

The approach of agent 3 focuses on a different goal: finding the solution as fast as possible. There are no branches in agent 3's approach, there is only one explored path. That means that it is not looking for the optimal solution and probably won't find it most of the times but the execution time and number of the expanded nodes should be significantly smaller than the ones required by A* algorithms.

Here's a quick summary of the predictions of performances of the agents in the 3 measured statistics.

	Agent 1	Agent 2	Agent 3
execution time	VERY HIGH	EXTREMELY HIGH	VERY LOW
expanded nodes	VERY HIGH	HIGH	VERY LOW
path length	OPTIMAL	OPTIMAL	NOT OPTIMAL

2.3.4 Comparison of the behavior of the searchAgents.

This section will be similar to Problem 1 comparison of the algorithms. We will start with the most trivial mazes and build up our way towards the most complex ones.

Let's start with the tinySearch maze.

Level 1: TinySearch maze. Outstanding performance.

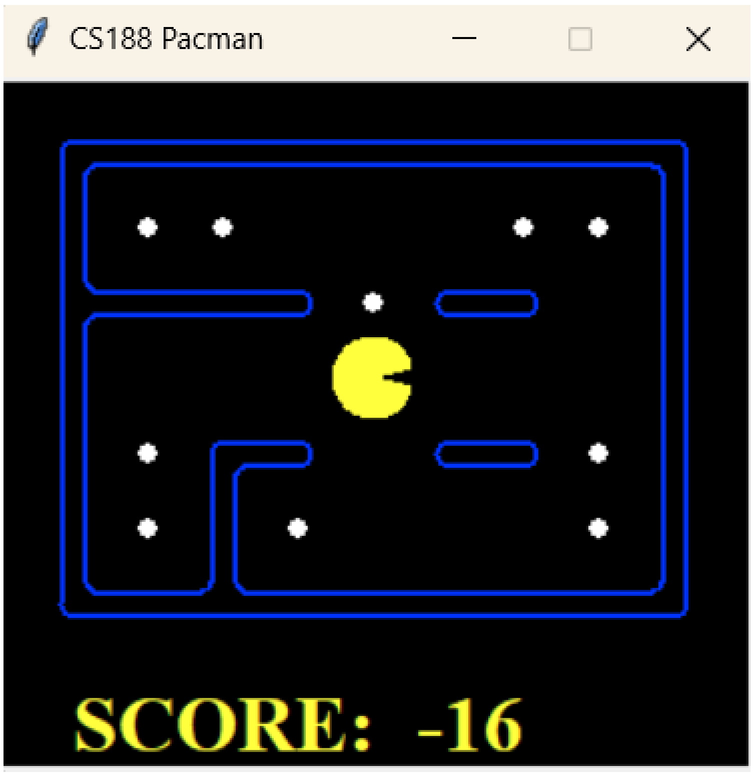


Figure 62: Problem 2: tinySearchMaze

This is a really small maze, therefore none of the algorithms should have big problems with solving that. Let's see how they did.

	Agent 1	Agent 2	Agent 3
execution time (s)	0,42	27,2	0,00199
nodes expanded	1812	1932	73
path length	27	27	31

It was the simplest problem, however it has already shown us a lot of things.

a) Agent 3 doesn't find the correct solution even in a simple maze

b) Our assumption of the amount of nodes expanded of agents 1 and agent 2 was incorrect. Even though Agent 2's heuristic is more informative, the rule doesn't seem to work in this case.
c) The execution time of agent 2 is unfeasible. 27 seconds in a simple maze in which the path to collect all food dots is incredible. We have to optimize the heuristics calculation.

Side quest: Optimizing the mazeDistance heuristic.

The problem of the current state of Agent 2 is that it is computing a BFS search n times. Problem 1 has shown us that A* with the Manhattan Distance heuristic was way better suited to search for the optimal path and maze distance to a single food dot.

Let's try swapping the searching algorithm then and see if it helps.

	Agent 1	Agent 2 with A* Manhattan used in the mazeDistance search	Agent 3
execution time (s)	0,42	12,6	0,00199
nodes expanded	1812	1932	73
path length	27	27	31

This optimization halved the execution time, however it is still 30 times longer than Agent 1 and over 6000 times longer than Agent 3.

Even though A* Manhattan is way better than BFS in terms of finding the optimal solution faster, we still have to compute it the number of times equal to the number of food dots left in the maze.

Let's try to find a solution that calculates the maze distance to each food dot at once, so the algorithm might contest its counterparts.

To do that we can do a single Breadth-First Search of the whole maze, and when we stumble upon a food dot, we compare the cost required to get to it with the previous maximum distance. Instead of calculating mazeDistance n times we can only call this function once.

This function will be called furthestMazeDistanceBFS. Here is the python code to implement that:

```
def furthestMazeDistanceBFS(position, foodList, gameState):
   """
   This function performs a Breadth-First Search (BFS) of the whole maze
to find the distance
   to the furthest food dot from the starting position.
   """
   walls = gameState.getWalls()

   if not foodList:
       return 0

   # Initialize the BFS queue
   BFS_queue = util.Queue()
   BFS_queue.push((position, 0))  # (current_position, current_distance)
   visited = set()
   visited.add(position)

   furthest_distance = 0

   while not BFS_queue.isEmpty():
       current_position, current_distance = BFS_queue.pop()
       x, y = current_position

       # If we reach a food dot, update the furthest distance
       if current_position in foodList:
           furthest_distance = max(furthest_distance, current_distance)

       # Explore the neighbors
       for direction in [Directions.NORTH, Directions.SOUTH,
Directions.EAST, Directions.WEST]:
           dx, dy = Actions.directionToVector(direction)
           nextx, nexty = int(x + dx), int(y + dy)
           next_position = (nextx, nexty)

           if not walls[nextx][nexty] and next_position not in visited:
               visited.add(next_position)
               BFS_queue.push((next_position, current_distance + 1))

   return furthest_distance
```

We also have to adjust the mazeDistance heuristic function to include our new function instead of calling the mazeDistance for every single food dot.

The new code looks like this:

```
def foodHeuristicMazeDistance(state, problem):

   position, foodGrid = state
   food = foodGrid.asList()

   return furthestMazeDistanceBFS(position, food,
problem.startingGameState)
```

And here is the result of Agent 2 with the optimized calculation of the mazeDistance heuristic.

	Agent 1	Agent 2 optimized	Agent 3
execution time (s)	0,420	0,985	0,00199
nodes expanded	1812	1932	73
path length	27	27	31

Now the competition looks somehow fair, the Agent1 and Agent 2 are competing in finding the optimal path the fastest, while Agent 3 is looking for the fastest solution but doesn't always find the optimal path.

With the optimized way to calculate Agent 2's heuristic function, we should update our theoretical beliefs of the performance of the algorithms.

	Agent 1	Agent 2 optimized	Agent 3
execution time	VERY HIGH	~~EXTREMELY~~ HIGH	LOW
expanded nodes	VERY HIGH	HIGH	LOW
path length	OPTIMAL	OPTIMAL	HIGH

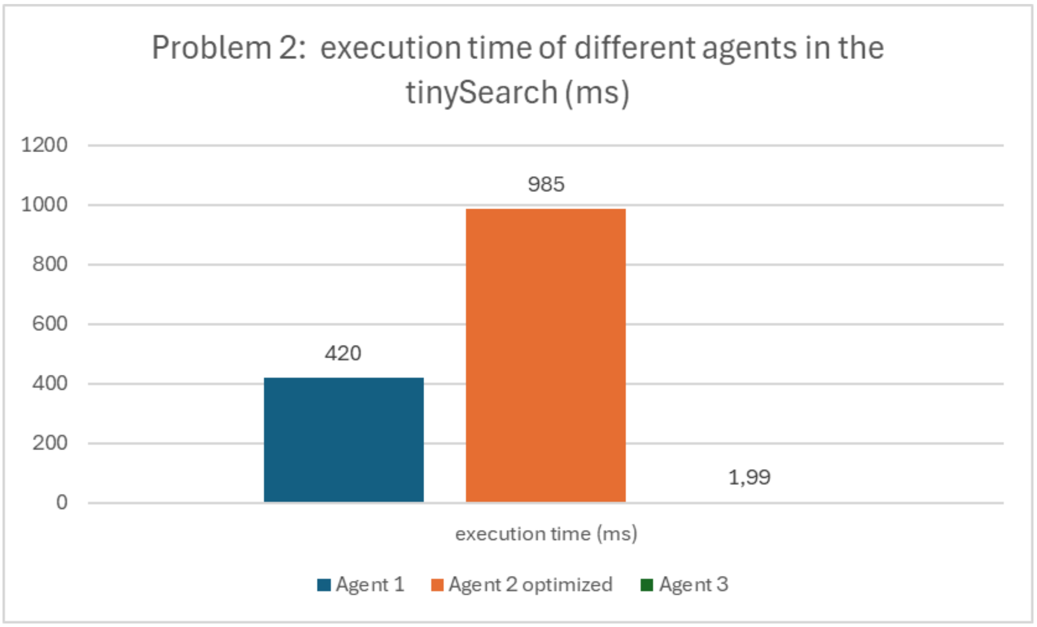


Figure 63: Problem 2: execution time of Agents 1-3 in tinySearch

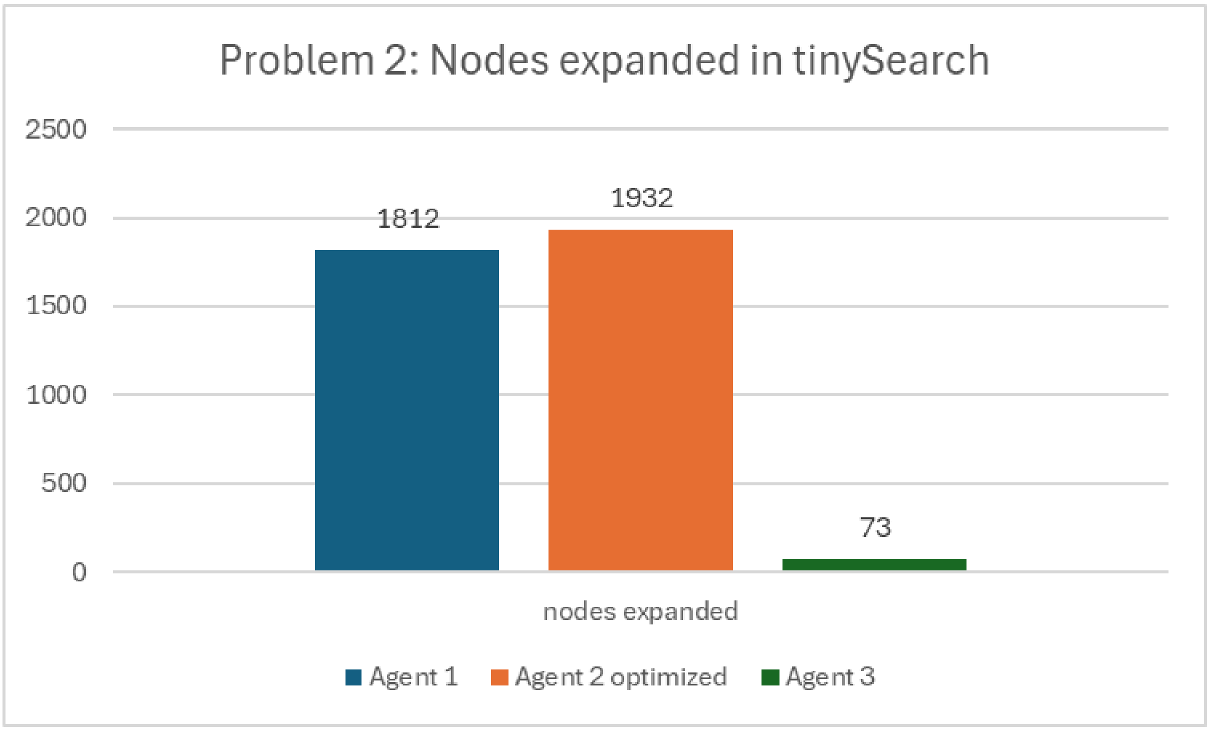


Figure 64: Problem 2: nodes expanded of Agents 1-3 in tinySearch

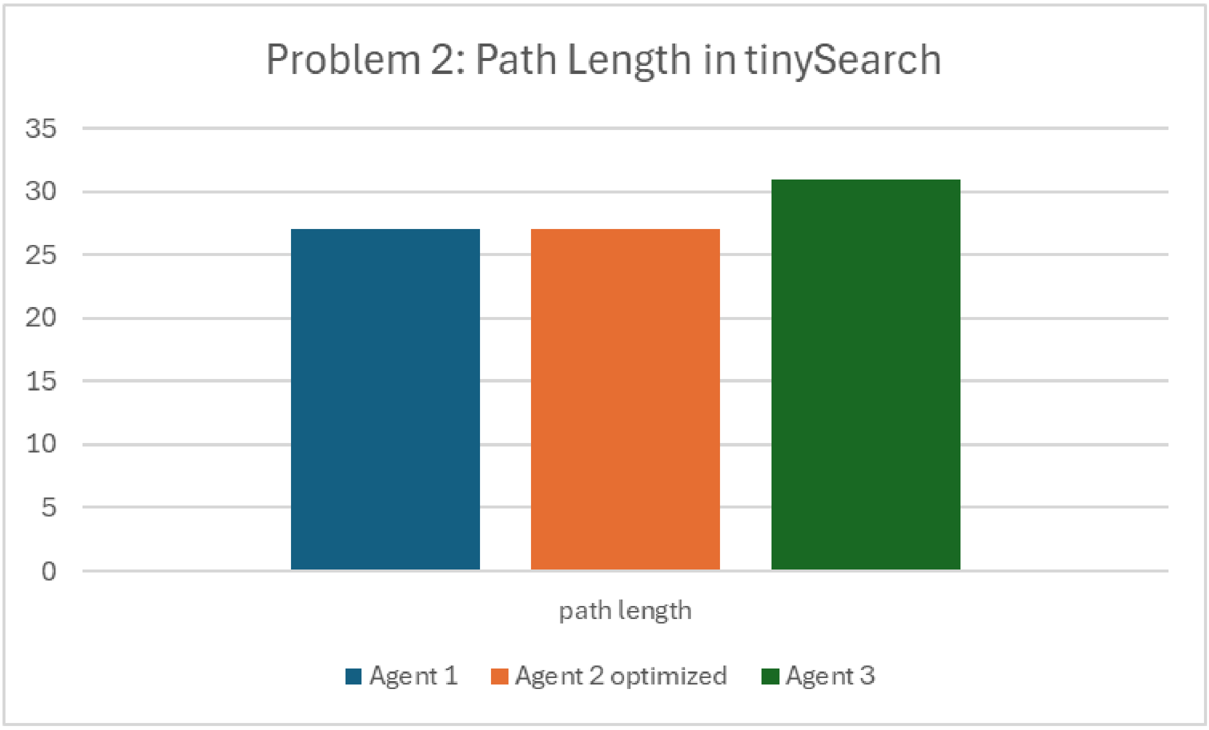


Figure 65: Problem 2: path length of Agents 1-3 in tinySearch

Level 2: Small search. Expanding the size of the maze.

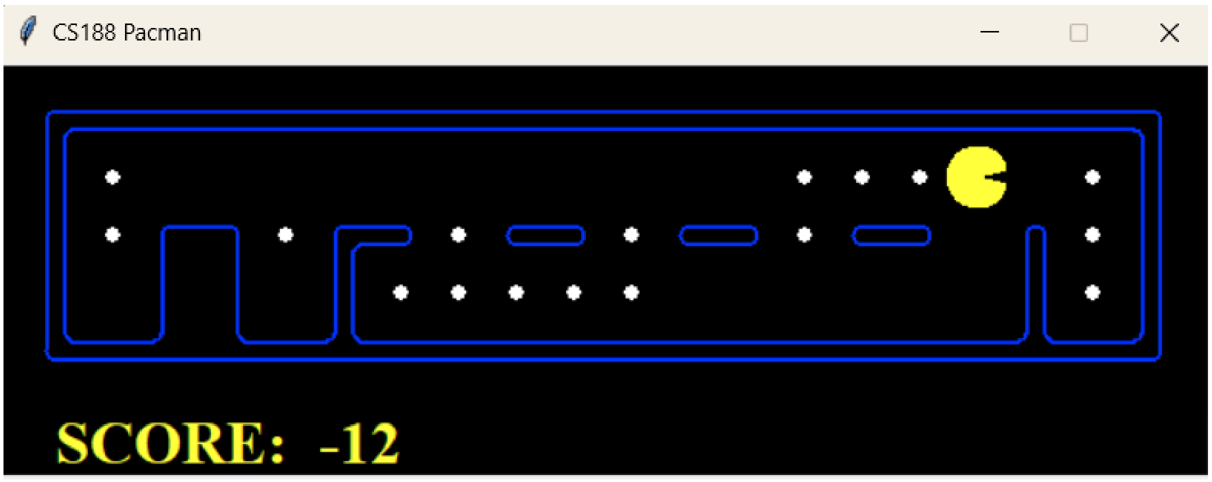


Figure 66: Problem 2: smallSearch maze

Another simple problem. There is a clear optimal path of getting all the food dots. How will the agents do in solving this maze?

	Agent 1	Agent 2 optimized	Agent 3
execution time (s)	4,63	4,06	0,00578
nodes expanded	5611	4175	105
path length	34	34	48

This time the more informative heuristic proved to expand less nodes. The execution time of Agent 2 was also better, however it didn't stand a chance against the execution time of Agent 3.

The DotSearchAgent found a solution that required 140% moves of the optimal one, however it expanded 40 and 50 times less nodes and was 800 times faster.

This time everything was in line with our theoretical assumptions.

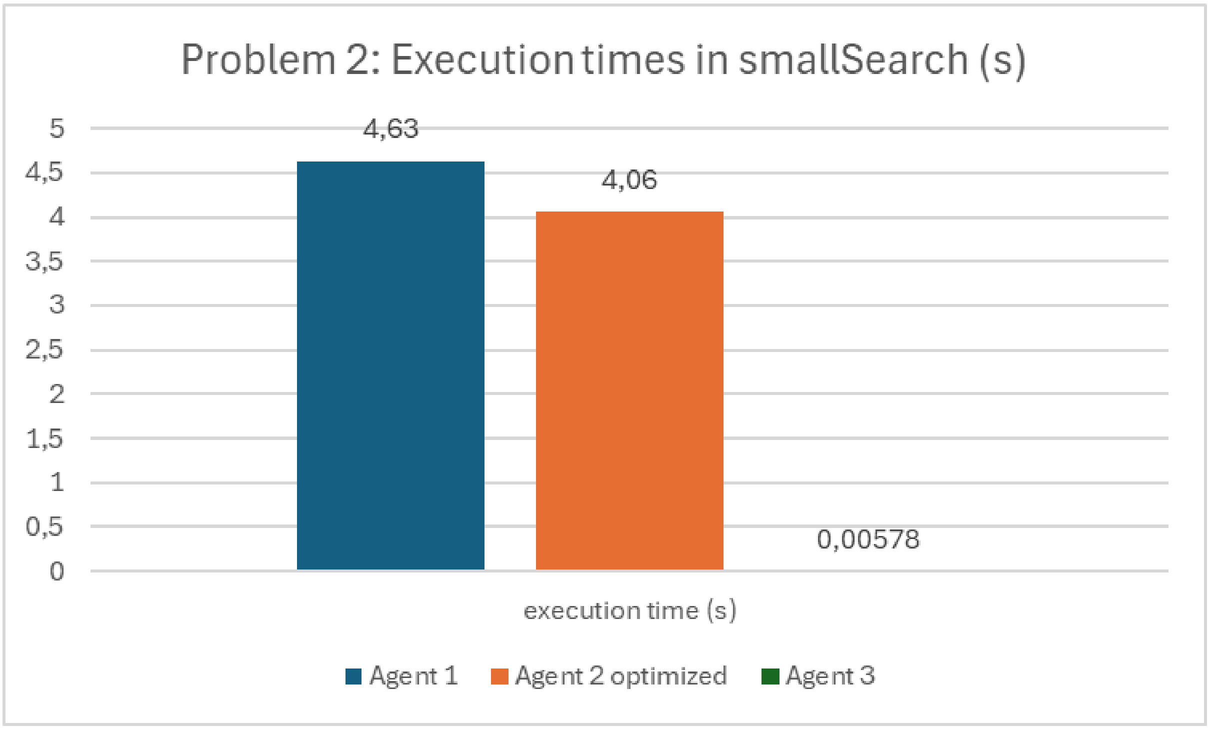


Figure 67: Problem 2: execution times in the smallSearch maze

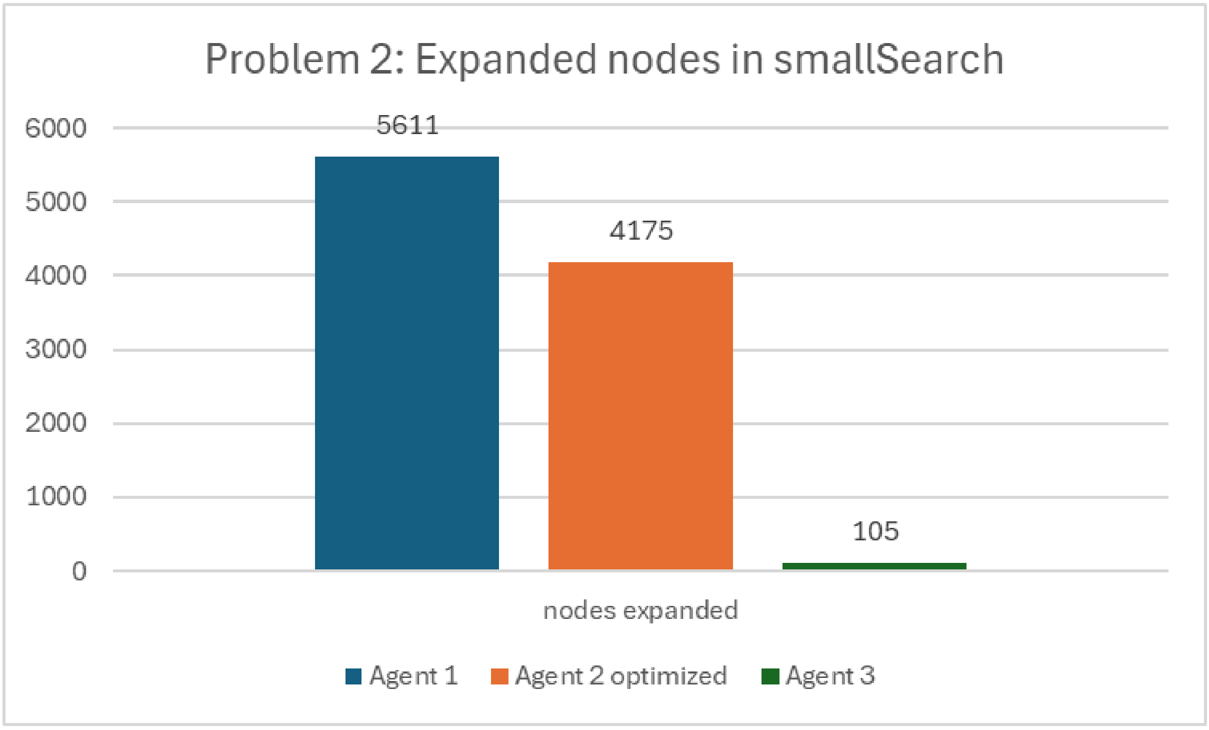


Figure 68: Problem 2: expanded nodes in the smallSearch maze

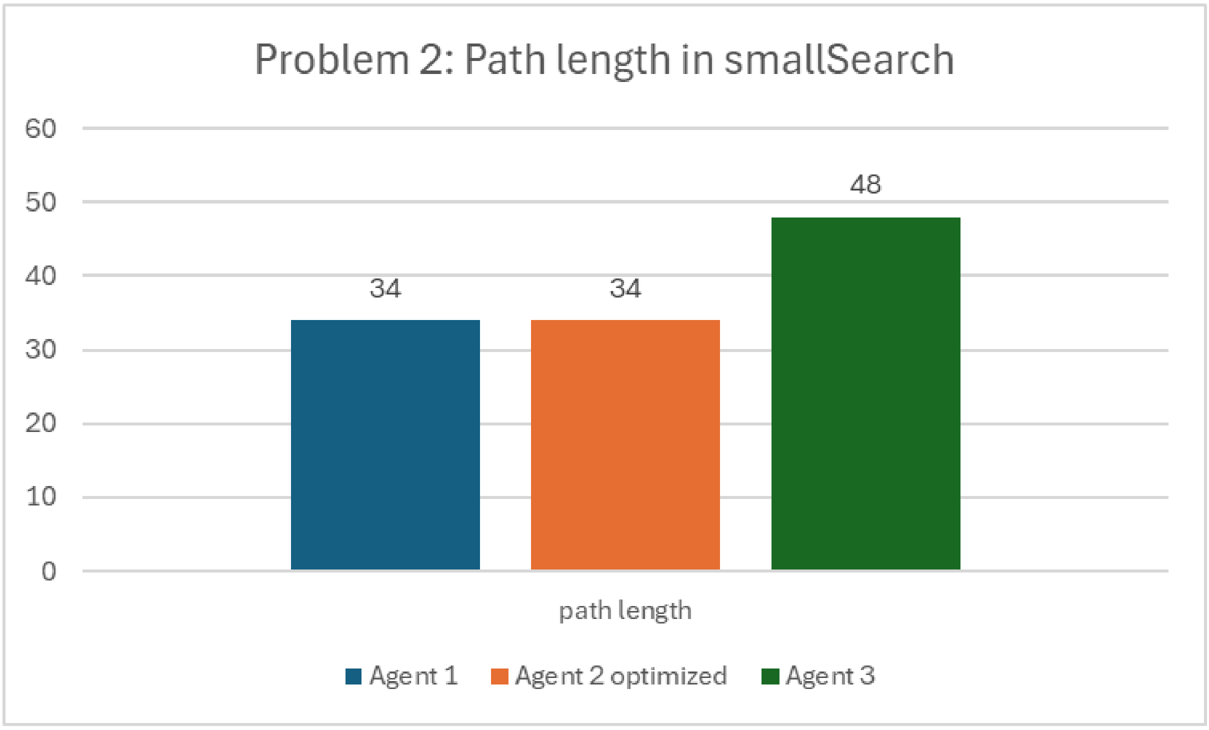


Figure 69: Problem 2: Path length in the smallSearch maze

Again, the time of execution of Agent 3 is invisible in the execution times chart. Agent 3 didn’t find the optimal path and the performance of Agent 1 and optimized Agent 2 is comparable, this time in favor of Agent 2.

Level 3: Open space, open search.

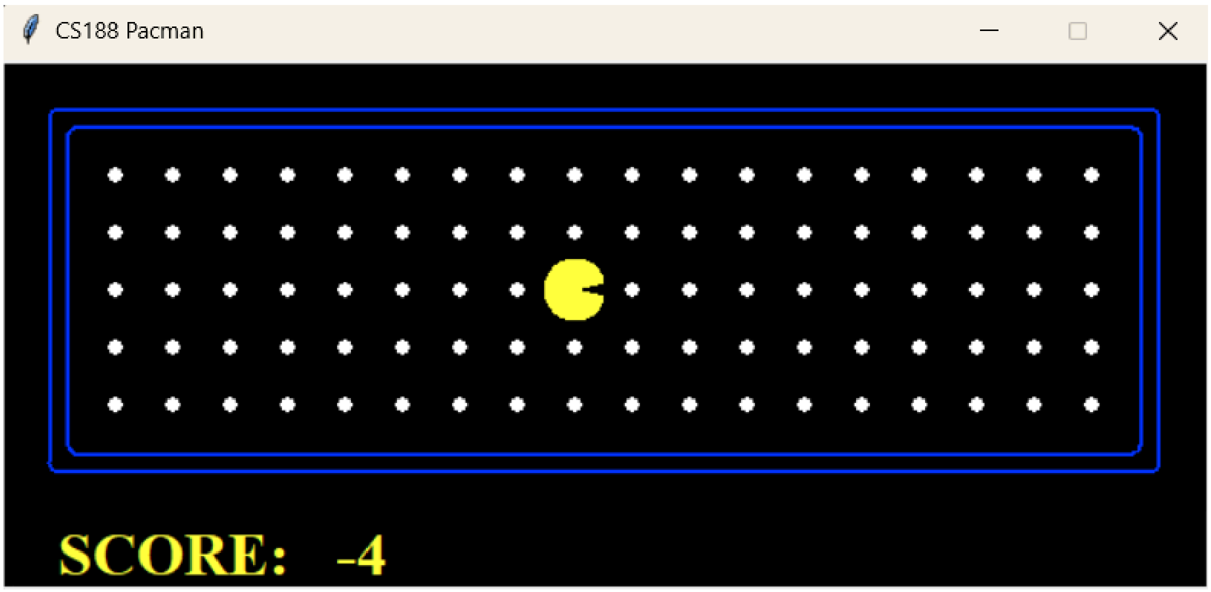


Figure 70: Problem 2: openSearch maze

In this level we will check the performance of Agents 1-3 in a foodSearchProblem in an environment without walls. Will Agent 3 prove that it can find an optimal solution or will it get lost again?

	Agent 1	Agent 2 optimized	Agent 3
execution time (s)	NOT COMPUTED	NOT COMPUTED	0,012
nodes expanded	-	-	186
path length	-	-	98

Well, maybe Agent 3 didn't find an optimal solution, but at least it found one (the optimal solution is 89 - eating a dot in every move). The other algorithms got overwhelmed by the amount of food dots. Both of the A* agents have been working for over 15 minutes before I stopped the process, because they couldn't compile.

Agent 3 is a clear winner of this problem space.

Level 4: TrickySearch. Ouch, there is a wall.

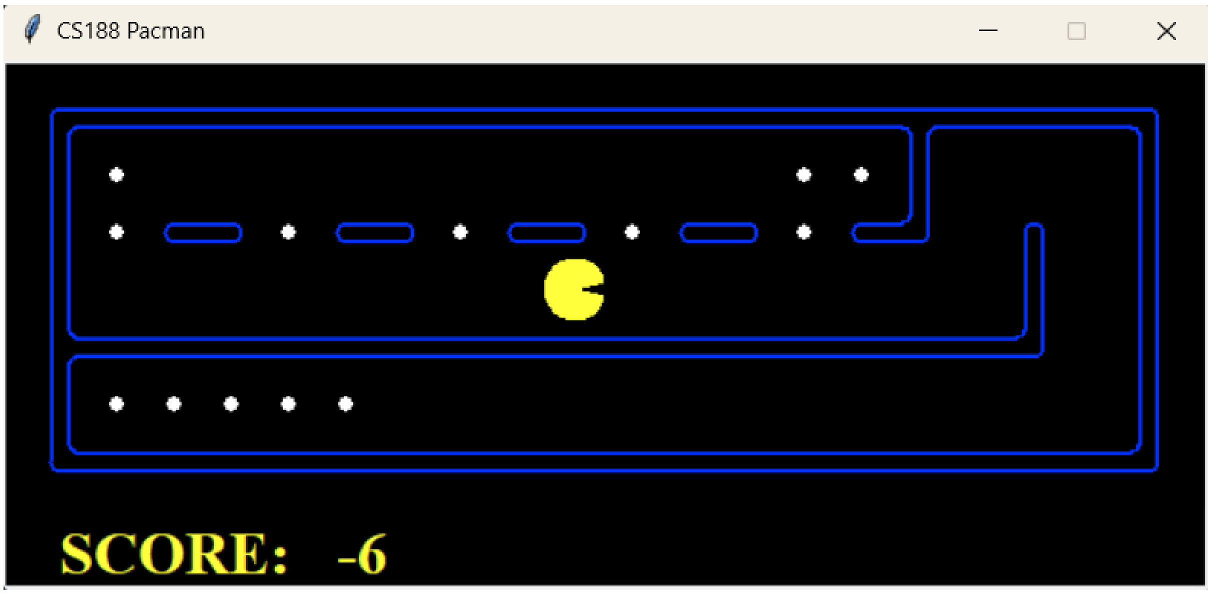


Figure 71: Problem 2: trickySearch maze

	Agent 1	Agent 2 optimized	Agent 3
execution time (s)	11,1	3,11	**0,00667**
nodes expanded	9402	3712	**132**
path length	**60**	**60**	68

This time the number of food dots allowed for the computation of every of the algorithms. This is a clear example of dominance of the mazeDistance heuristic over the manhattanDistance. Agent 1, due to the nature of its heuristic at first explored most of the possible ways to collect the dots in the top part of the maze, and every time tried to get to the bottom left corner through a dead end corner, before choosing the correct path through the tunnel on the east side.

Agent 2 however, knew that there is no direct passage from the western side of the map to the southern part, therefore it had to expand only 40% of the nodes of its counterpart.

Agent 3 again proved its superiority in finding the solution fast, again not being too far away in the case of optimality.

Here are the charts for this level.

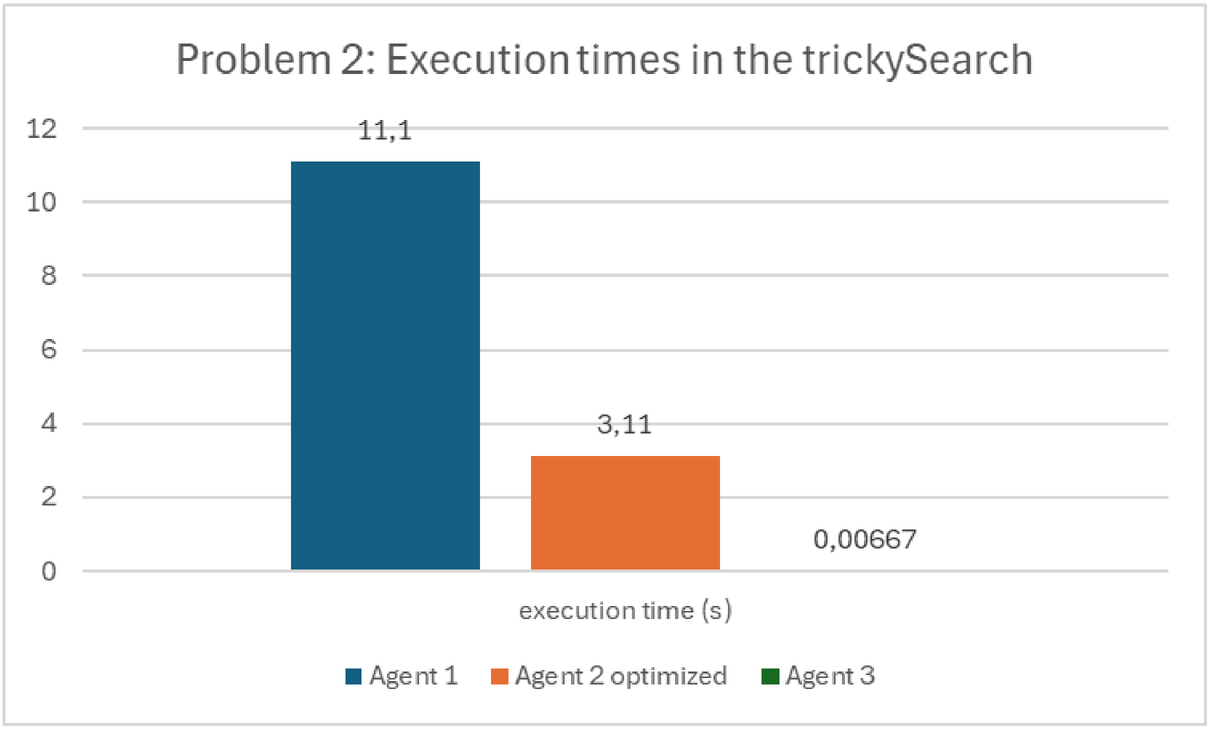


Figure 72: Problem 2: Execution times in the trickySearch maze

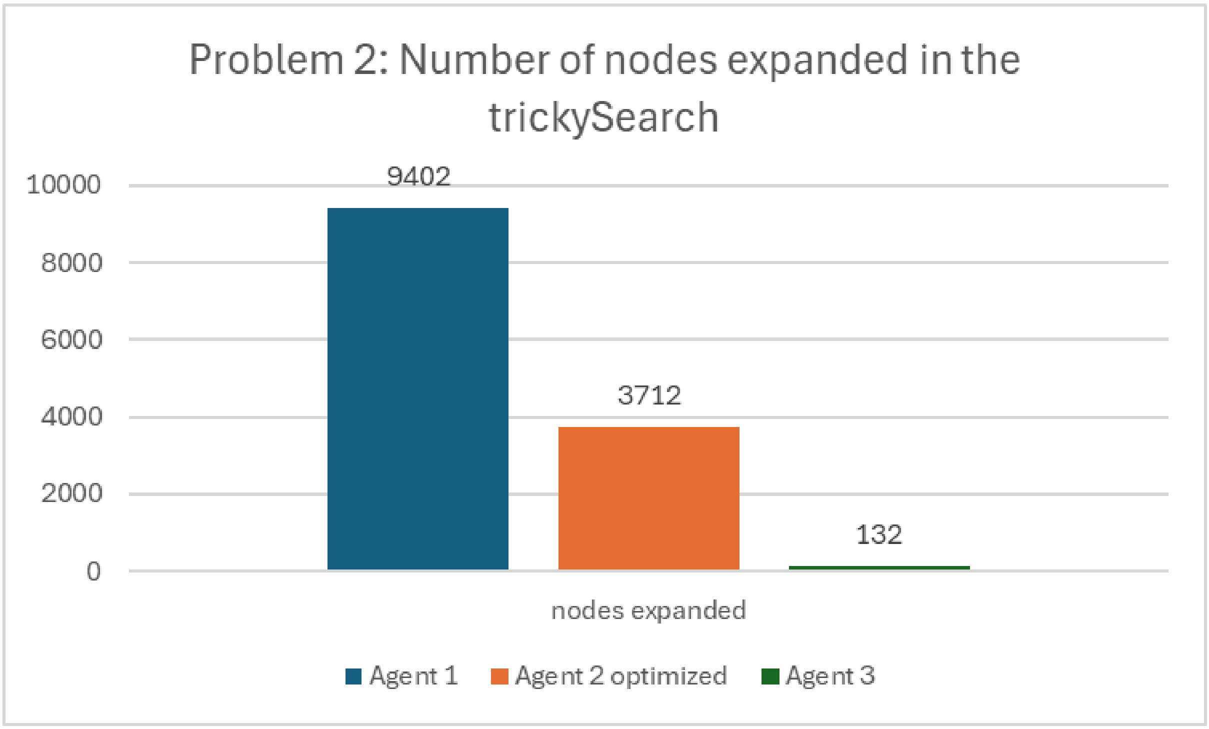


Figure 73: Problem 2: expanded nodes in the trickySearch maze

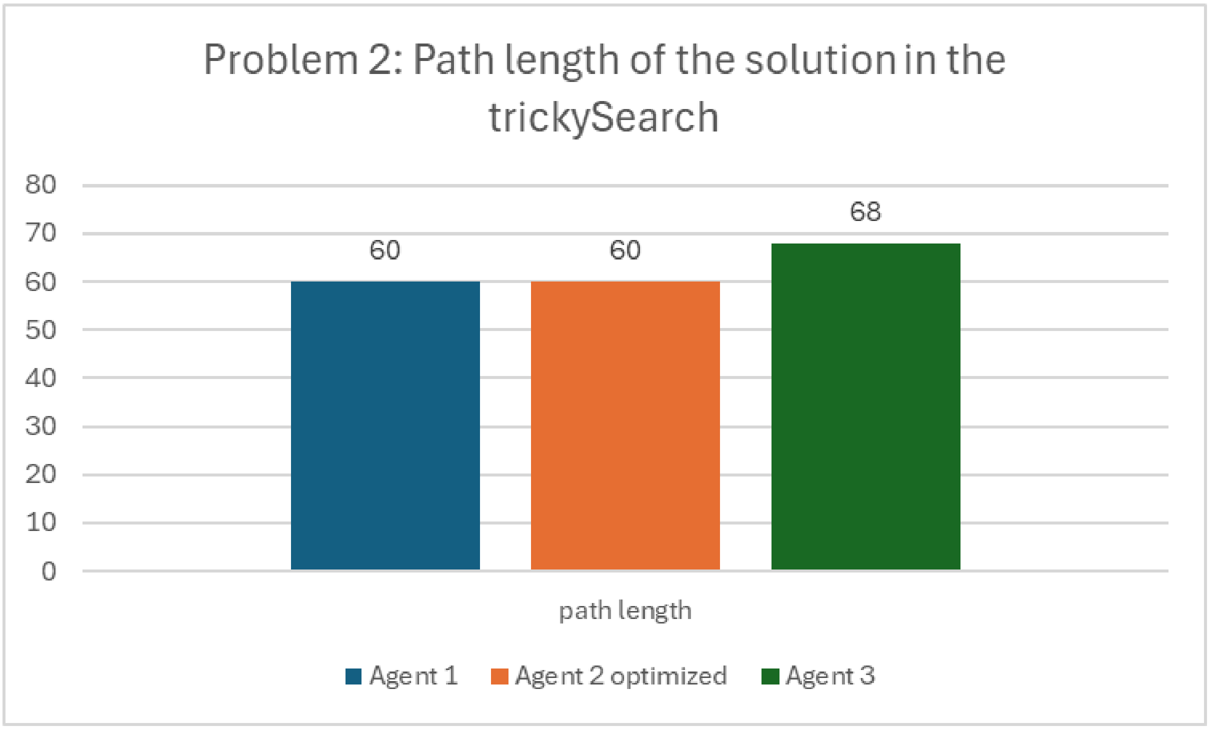


Figure 74: Problem 2: Path length in the trickySearch maze

Let’s take a look at the figures 72 and 73. The proportions between charts are almost the same. What would happen if we represented the execution time in milliseconds instead of seconds and showed both execution time and number of nodes expanded at the same chart?

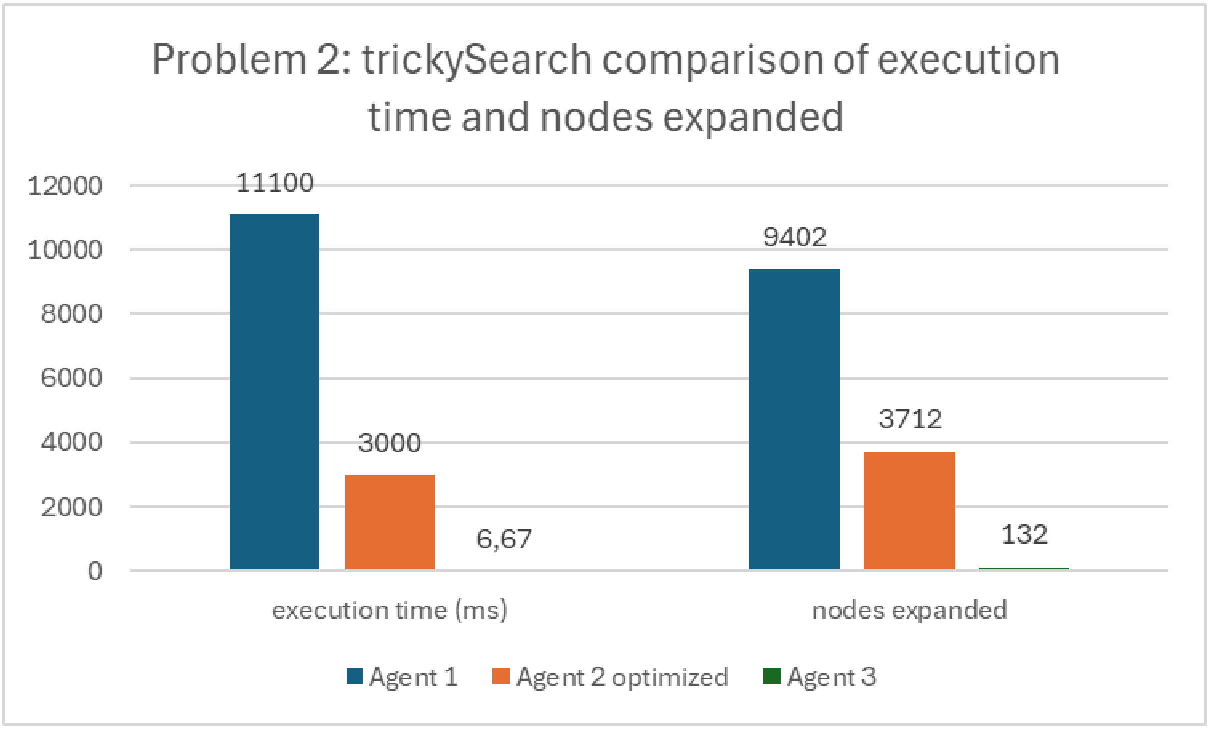


Figure 75: Problem 2: Comparison of execution times and nodes expanded in trickySearch

We can clearly see that there is correlation between those two statistics. But is there a causality or that's just a coincidence?

Obviously there is causality. Expanding a single state is followed by calculating its heuristic value. It's either searching for the Manhattan distance of all the remaining food nodes, traversing the whole maze with the BFS algorithm to find the furthest mazeDistance to a food dot or finding a way to the next closest food dot. All of the following operations require a solid amount of calculations done, therefore they require time to execute.

Level 5: Greediness. The problem of agent 3.

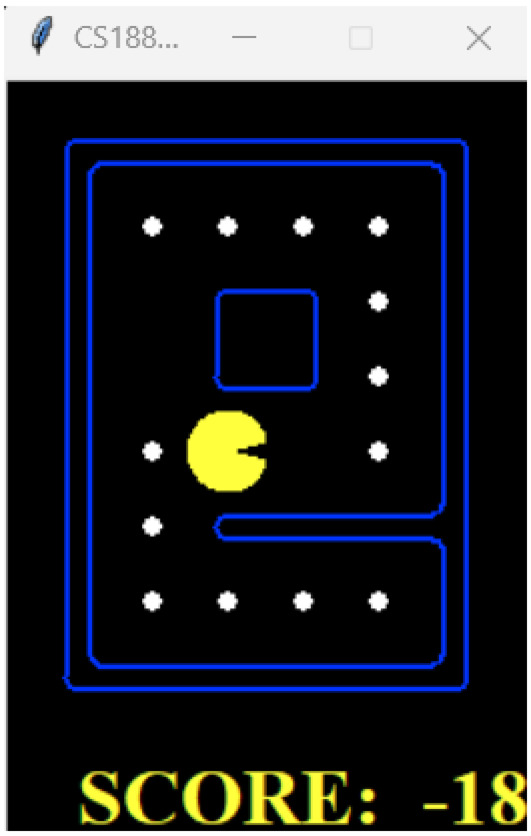


Figure 76: Problem 2: greedySearch maze

This is the doom scenario for Agent 3. We can clearly see that the optimal path is going the right from the initial state, however there is a food dot only 1 tile from us to the left, therefore the DotSearch will prioritize it and go into the dead end corridor at first, meaning it will have to walk it twice to collect the rest of the dots, while the optimal path crosses it only once.

Let's see how much longer is the path Agent 3 took compared to the optimal one.

	Agent 1	Agent 2 optimized	Agent 3
execution time (s)	0,021	0,034	**0,0011**
expanded nodes	185	138	**29**
path length	**16**	**16**	20

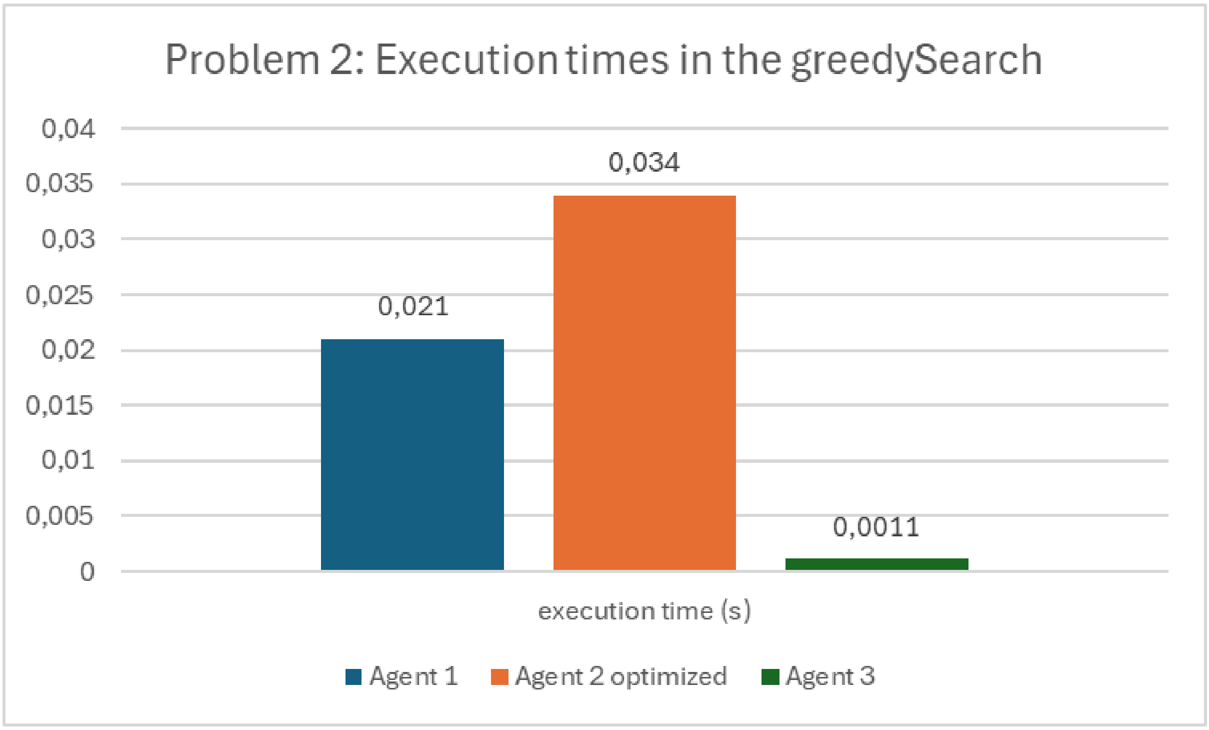


Figure 97: Problem 2: execution times in the greedySearch maze

This time agent 3 was “only” 20 times faster than the second fastest algorithm. The execution time of agent 3 is visible on the chart for the first time since the tinySearch problem.

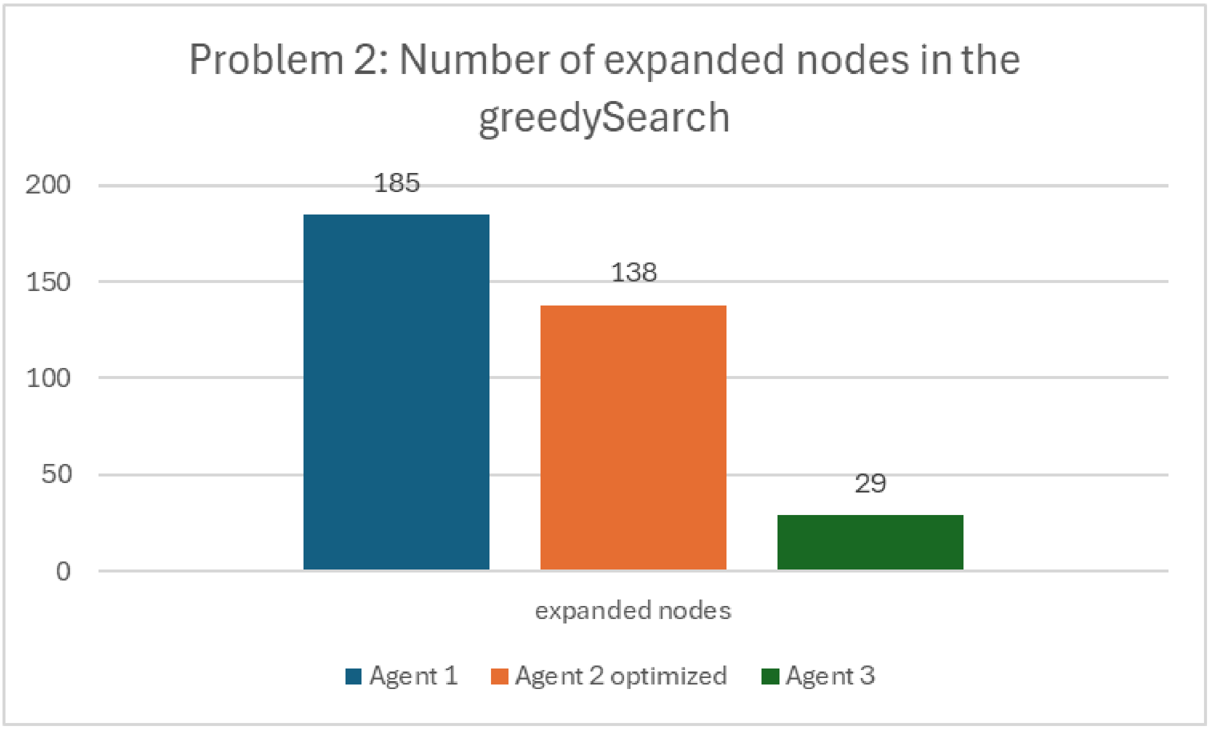


Figure 98: Problem 2: expanded nodes in the greedySearch

Since the maze was small, the amount of nodes expanded by the A* algorithms were moderate. That's the smallest difference we have so far seen, however Agent 3 has opened almost 5 times less nodes than the best of A* agents anyway.

Let's check the path length chart and move on to the final, most complex problem.

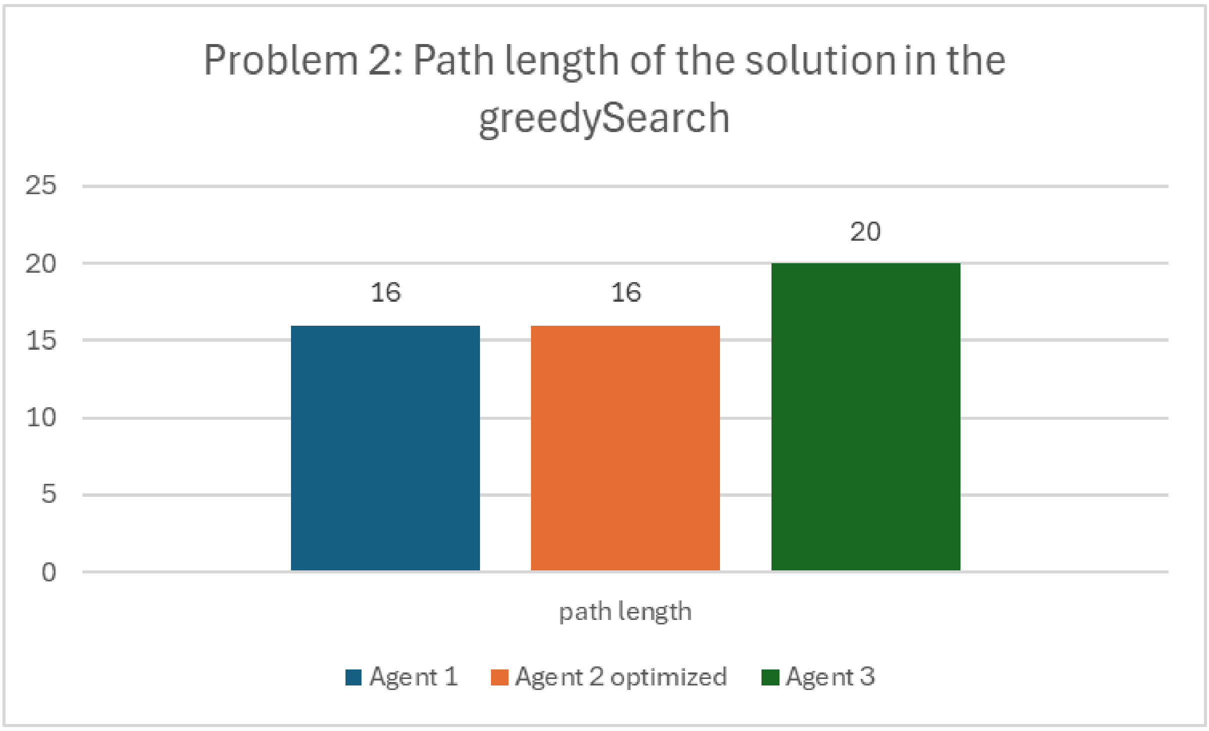


Figure 99: Problem 2: Path length in the greedySearch

Level 6: Final boss. The bigSearch maze.

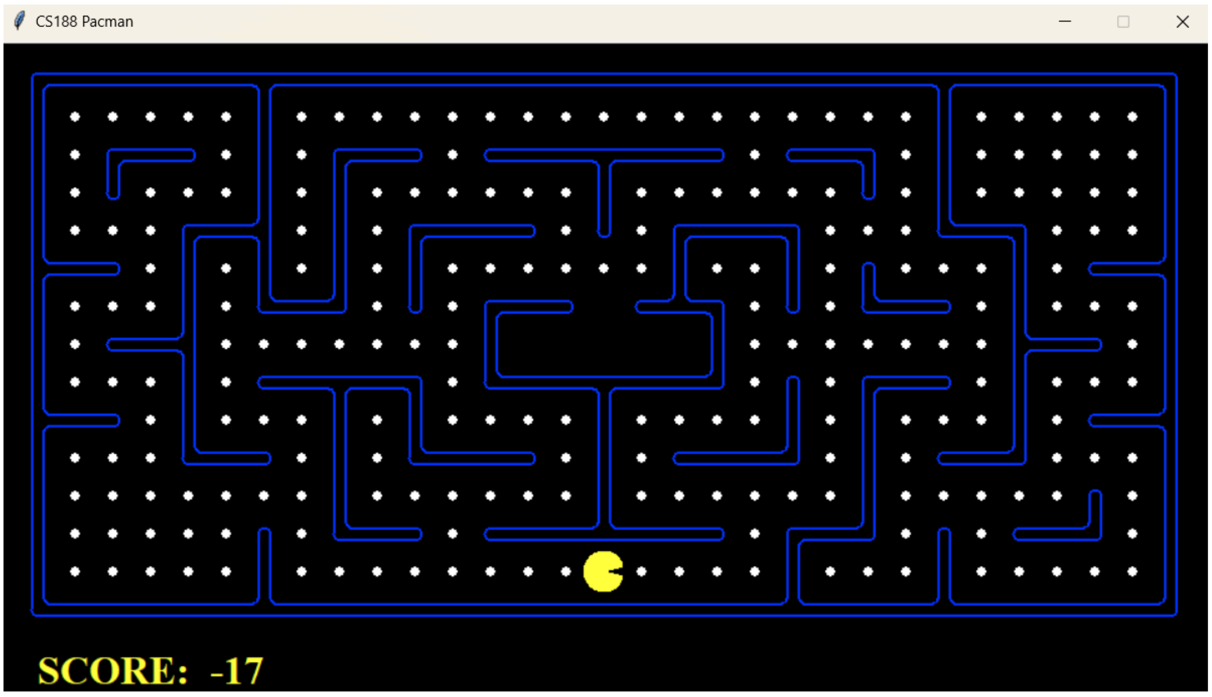


Figure 100: Problem 2: bigSearch maze

This is the final, most complex maze in the report. Will the A* algorithms be able to compute? How many detours will Agent 3 take? What's the optimal path? Let's test the agents for the last time.

Let's start with agent 3, we can safely assume that he will find a solution to the maze, however we can be almost sure that it won't be the optimal one.

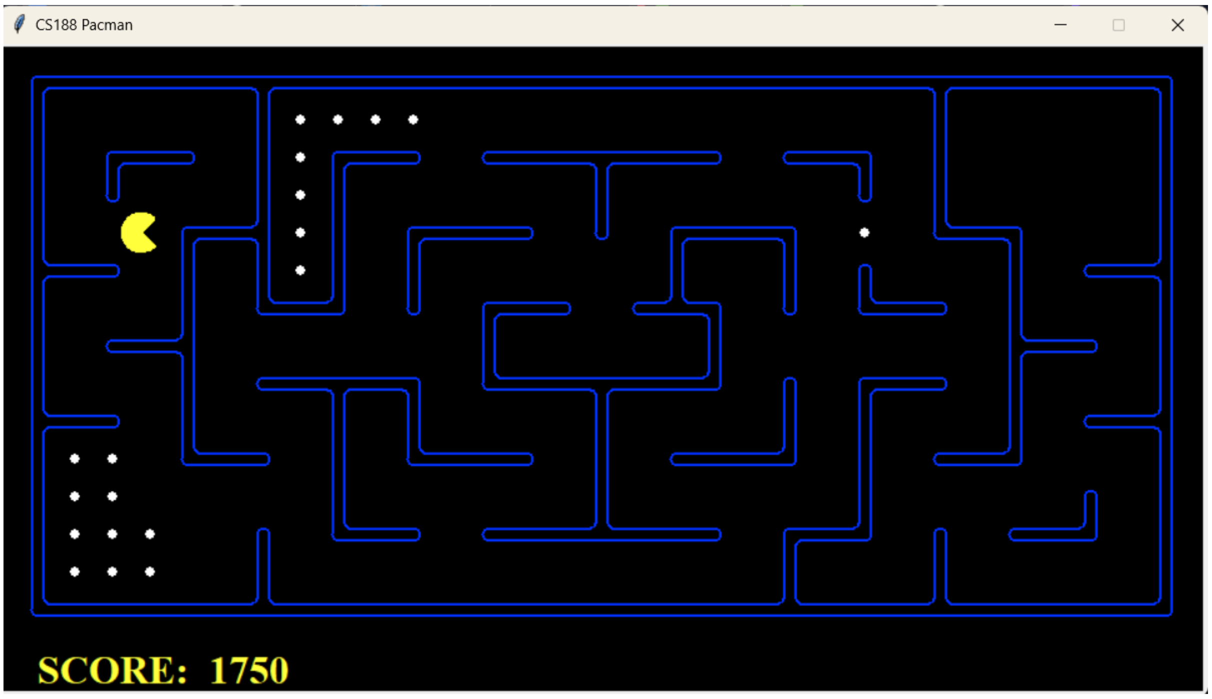
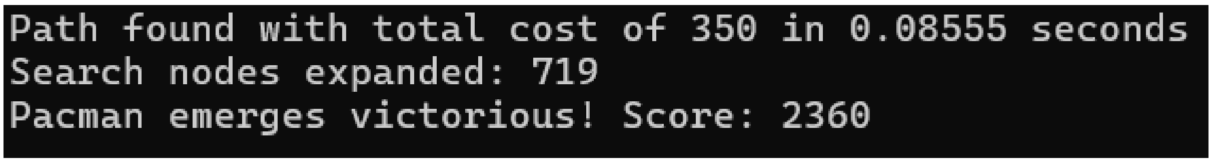


Figure 101: Problem 2: bigSearch maze being solved by Agent 3



```
Path found with total cost of 350 in 0.08555 seconds
Search nodes expanded: 719
Pacman emerges victorious! Score: 2360
```

Figure 102: Problem 2: performance of Agent 3 in the bigSearch maze

From figures 101 and 102 we can see that the path agent 3 took was nowhere close to being optimal. It left a few unsearched spaces that require a lot of backtracking to collect.

The execution speed however was incredible. For that complex of a maze it took less than $\frac{1}{10}$ th of a second to find a valid path.

Will the A* Agents find the solution?

The answer is no. This problem is way too big for them to solve. I gave each of the algorithms over an hour to compile, however neither of them managed to find the optimal solution.

Let's take a look at the final results of each level.

2.3.4.2 Final comparison of the problem 2 agents.

This is the last task of the project. In this section we will take a look at the results of each category in every of the problem 2 mazes.

We will start with the execution times of each of the Agents searches.

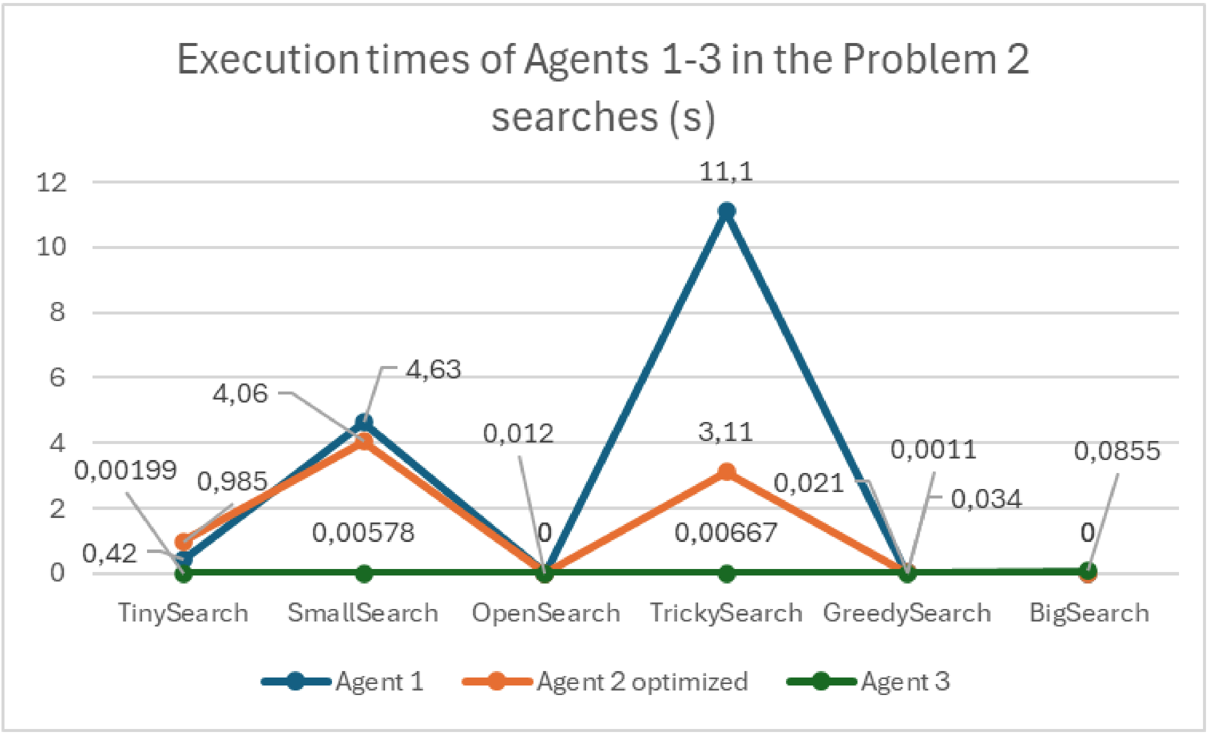


Figure 103: Problem 2: execution time of Agents 1-3 in each of the mazes

As we can see, the execution times were clearly dominated by Agent 3. It was the only algorithm that has found a solution in each of the mazes, even though the A* Agents were given a lot of time to compute (15 minutes in the OpenSearch problem and 1 hour in BigSearch).

The time required by Agent 3 to solve the most complex problem, the bigSearch would be the third fastest time of Agents 1 and 2 in all of the mazes (Both have gotten faster results in the GreedySearch (A1: 0,021 and A2: 0,034)).

There is no point in comparing the execution time of those three anymore, let's look at the chart of only Agent 1 and Agent 2.

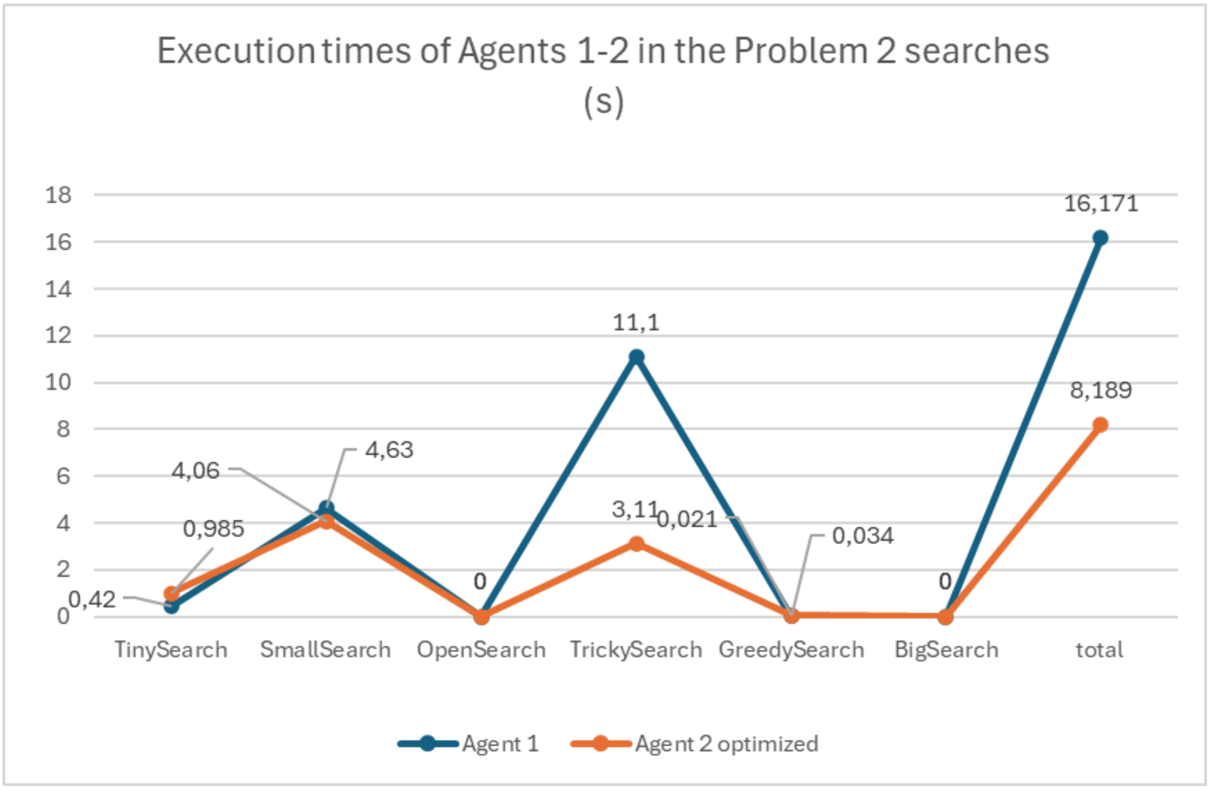


Figure 104: Problem 2: execution times of Agents 1-2 in each of the mazes

The performance of those two agents were really similar until the trickySearch, where Agent 2 required only 28% of Agent 1's time. That made a difference that resulted in a total cost of a half of Agent 1's execution time, therefore winning the competition.

The execution times of Agent 1 and Agent 2 were so big that in a real Pacman game, where we can lose by touching a ghost, they would stand no chance. Therefore in this category the only suitable algorithm is the DotSearchSequence.

The next category is the number of nodes expanded. Let's look at the chart.

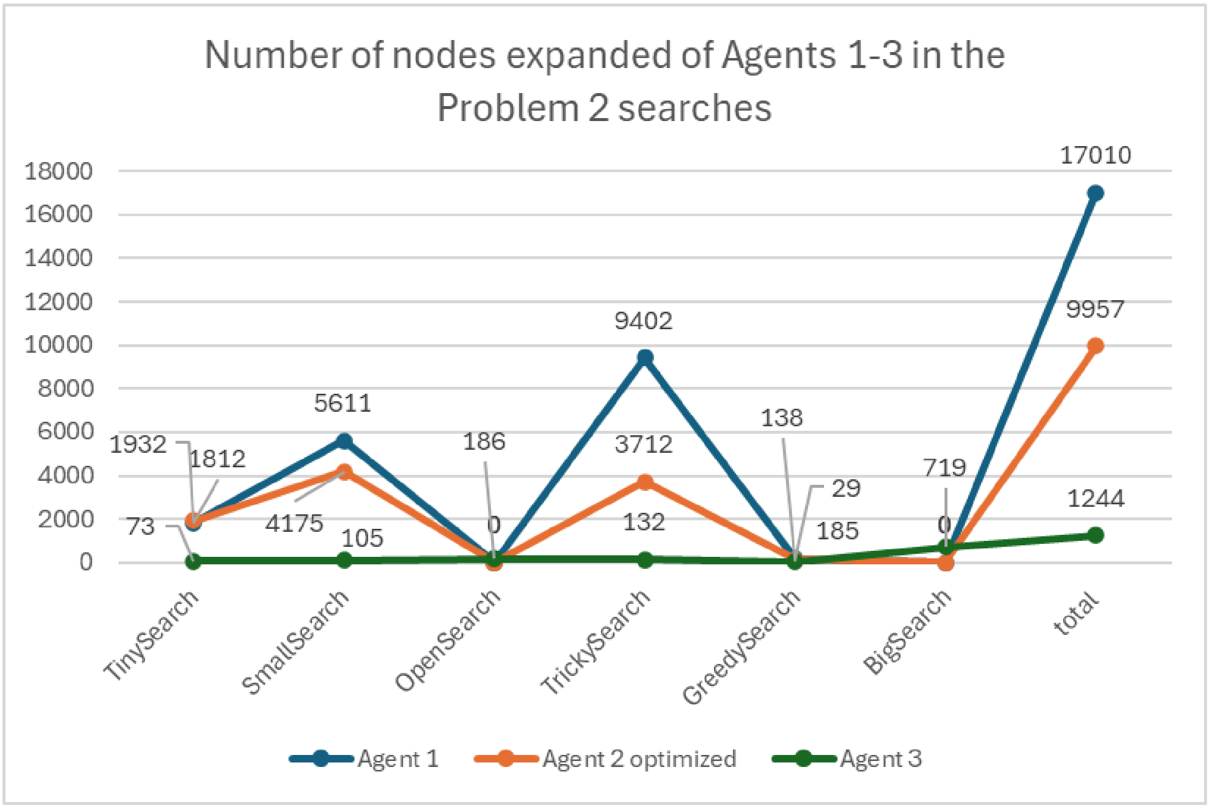


Figure 105: Problem 2: number of nodes expanded by Agents 1-3 in each of the mazes

Another category clearly won by the Agent 3. Even though the total number of nodes expanded was calculated with OpenSearch and BigSearch problems, the total of Agent 3 was lower than even just the amount of nodes expanded by Agents 1 and 2 in the tinySearch maze.

The heuristic of Agent 2 has proven itself to be more informative, even though the number of nodes expanded in the TinySearch problem was bigger in the search of Agent 2. Over the course of the 4 solved mazes. Agent 2 expanded only 59% of the nodes of Agent 1, the biggest difference being trickySearch again.

The final category is the path length of each of the solutions. We know that Agents 1 and 2 always look for the optimal solution, therefore we will consider their solutions together. This category is about checking how much the solutions of Agent 3 differ from the optimal one.

We will consider openSearchs optimal cost of 90, because it is easy to find a solution that eats a dot with every move. The optimal solution of bigSearch is not that trivial though, because there are many promising paths.

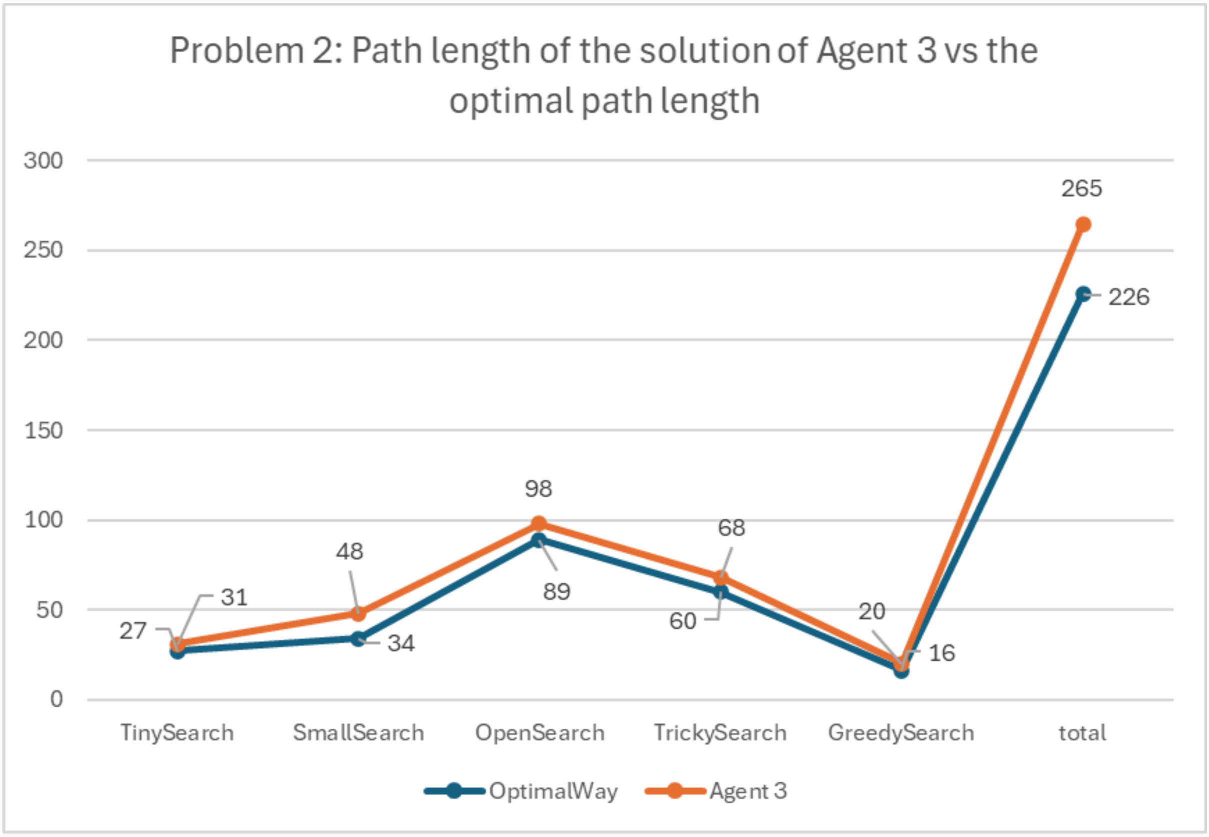


Figure 106: Problem 2: Comparison of the path length of Agent 3's solution vs the optimal path length

From Figure 106 we can see that Agent 3 hasn't found an optimal path in any of the mazes. However, the paths proposed by the DotSearchAgent weren't that far from optimality. Over the course of all 5 mazes in which we know the optimal solution, Agent 3 total cost was only 117% of the optimal paths.

This is a great performance, given how fast agent 3 came up with the solutions. Additionally, the algorithm used by agent 3 is great when it comes to computing the calculations while simultaneously using the operators, because it doesn't require backtracking at any point, it follows a single path until it finds a solution.

Concluding the performance of agent 3 in the FoodSearchProblem, it has proven to be an insanely good performing algorithm if we can miss the optimal path by a small margin, however it's not suitable if we always have to find an optimal path.

All of the categories match our theoretical assumptions. The only performance of an algorithm that didn’t match the theoretical prediction was the amount of nodes expanded in the tinySearch (the Agent 2, whose heuristic function was more informative, expanded more nodes than the Agent 1).

That concludes the analysis of the behavior of different search agents and at the same time concludes the 2nd part of the project report: *Understanding of the search algorithms.*

3. Conclusions

In this project, we implemented and analyzed various search algorithms within the classic Pacman environment, focusing on two main problems: finding a fixed food dot and collecting all the food dots. The key conclusions drawn from the project are summarized below:

Search Algorithms Performance Analysis

1. DFS vs. BFS vs. UCS vs. A*:
    - Depth-First Search (DFS):
        - Advantages: Low memory usage, efficient in deep mazes.
        - Disadvantages: Inadmissible, can lead to suboptimal paths, especially in complex mazes with many branches.
    - Breadth-First Search (BFS):
        - Advantages: Admissible, always finds the shortest path in unweighted graphs.
        - Disadvantages: High memory usage, not efficient in deep or large mazes.
    - Uniform Cost Search (UCS):
        - Advantages: Admissible, optimal in weighted graphs.
        - Disadvantages: High memory usage, similar performance to BFS in unweighted graphs.
    - A*:
        - Advantages: Combines UCS with heuristic for efficient pathfinding, optimal with admissible heuristics.
        - Disadvantages: Heuristic choice significantly impacts performance, can be computationally intensive.

Problem 1: Finding a Fixed Food Dot

1. Performance Comparison:
    - Simple Mazes: All algorithms performed similarly with slight differences in execution time due to data structure differences.
    - Complex Mazes: A* with Manhattan heuristic outperformed others in terms of nodes expanded and execution time due to better guidance from the heuristic.
    - DFS Limitations: DFS often found suboptimal paths, especially in mazes designed to exploit its weaknesses.

2. Heuristic Analysis:
    - Manhattan Distance: Proved to be effective and consistent, leading to optimal solutions in most scenarios.
    - Euclidean Distance: Slightly less effective than Manhattan due to less accurate distance estimation in grid environments.

Problem 2: Eating All the Food Dots

1. Agent Performance:
    - *Agent 1 (A with Manhattan Heuristic)**:
        - Consistently found optimal paths.
        - High computational cost in complex mazes due to heuristic calculations.
    - *Agent 2 (A with Maze Distance Heuristic)**:
        - More informative heuristic, expanded fewer nodes than Agent 1.
        - High execution time due to computational complexity of the heuristic.
    - Agent 3 (DotSearchAgent):
        - Fastest in terms of execution time and nodes expanded.
        - Did not always find the optimal path, especially in mazes designed to exploit its greedy approach.

4. Personal comments.

This project was a great way to get a better grasp on the disadvantages and advantages of different search algorithms. It was also the first time in the course where we saw an actual way, the AI related code is getting implemented.

The main difficulty for me was getting started with the project, because the first path of analyzing the algorithms was monotonous and repetitive, however when we started the comparison of the actual performance of different algorithms in various problems represented in the Pacman environment was really interesting and made me want to research more in the field of Artificial Intelligence and its usage in games optimization problem.

The most challenging part for me was trying to design a maze that makes BFS outperform A* with Manhattan Distance as the heuristic function in the PositionSearchProblem, however I think I managed to prove that it's impossible, without the modification of the heuristic wage or introduction of additional, misleading goals.

Overall, this project turned out to be way more difficult and interesting than I thought in the first place, but it made me more interested in the field of AI research.

5. Sources and frameworks used in the project

- Modified version of the Berkeley Open Source Pacman project (http://ai.berkeley.edu/project_overview.html) provided on the Aula global
- Geeksforgeeks portal - graphics for the data structures:
  - Figure 2 - Stack Data structure
  - Figure 4 - Queue Data structure
- netmatze.workspace.com - Figure 6 - Priority Queue data structure graphic
- Slides from the theoretical classes of the AI course on the UC3m university