Universidad Carlos III de Madrid

Bachelor's Degree in Applied Mathematics and Computer Science

Artificial Intelligence Project

Carlos Suárez - 100429792

Jorge Parreño - 100429982

Academic year 2020-2021

TABLE OF CONTENTS:

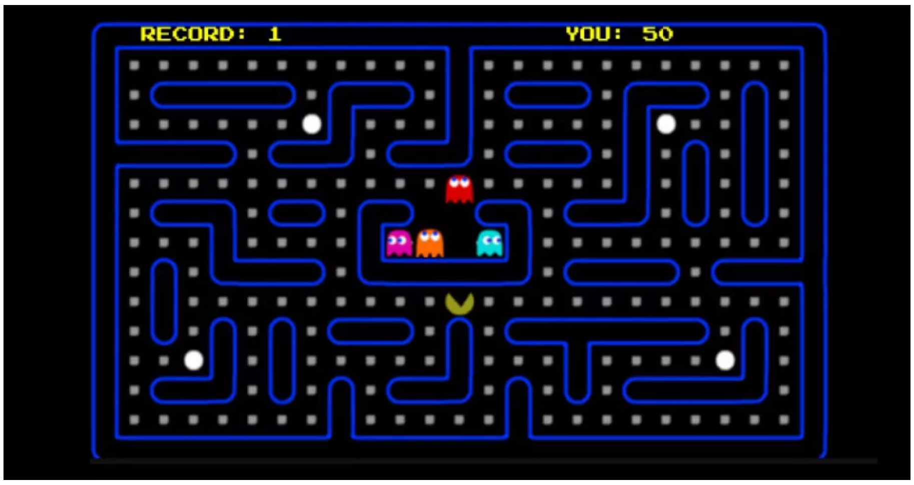
INTRODUCTION

For this project, we are asked to develop an analytical study over the implementation of the well-known Pacman game in Python. We are given access to the different files and pieces of code that conform the application, which we must use in order to experiment and answer questions about the different algorithms and aspects of the implementation, obviously with a main focus on those Artificial Intelligence algorithms and concepts incorporated in it.

Pac-Man, which was created in Japan back in 1980, is a worldwide staple for Arcade games. It can be described as a maze action game in which the player must achieve eating all the food dots in the maze while avoiding several colored ghost figures, which are also in constant movement inside the maze, in order to win.

Through this project, we hope to see the 'magic' behind the functioning of this game and, in particular, to understand the ideas related to AI in a maze-like setting: the structure of a maze type state space, the different ways to search within the maze, the intelligent agents that can perform movements along the maze and how they do so… amongst others. We aim to be able to translate these into our own implementations in the future.

In this paper, we will solve each of the proposed problems, while carefully explaining the process followed for each of them. Along with his paper, we will also provide a *layouts-student* file, necessary for problem 1.



UNDERSTANDING THE SEARCH ALGORITHMS

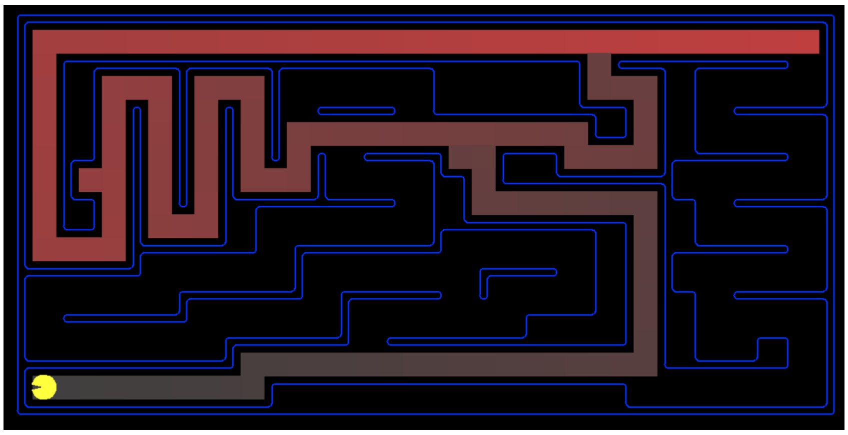
In this section, we take a look at both the different methods defined in *search.py,* that is, the different search algorithms that a pacman agent is capable of following when moving inside the maze; as well as the different game outcomes when executing commands where we specify the algorithm to be followed. This is done by writing *-a fn = [searchAlgorithm]* at the end of the command. After dissecting the code and running tests with these different algorithms in order to help us visualize the differences, we are ready to write an analytical description for each of the algorithms.

For each of the algorithms, we have opted for something similar to a list of steps that integrates both explanations and pseudocode ideas in order to develop our analytical description. We will also include a diagram of the search and the parameters returned from the search. In order to obtain fair comparisons, we will run all algorithms in a medium maze.

DEPTH FIRST SEARCH (DFS):

SEARCH RESULTS:

```
Path found with total cost of 130 in 0.00901 seconds
Search nodes expanded: 146
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:        380.0
Win Rate:      1/1 (1.00)
Record:        Win
```



DESCRIPTION:

Side note: steps 2-4 will be reused for the rest of the algorithm (reason explained later)

1. Initialize an **empty stack**
   Stacks are important for BFS because they follow a **LIFO** structure: the last element introduced is the first to be extracted. We need to extract a node in order to look at its successors and keep searching, so **using a stack we look at the successors of the last expanded node, which is exactly DFS**!

Elements of the stack will be paths. **Paths are arrays of tuples** with the structure (State, action (movement), cost).

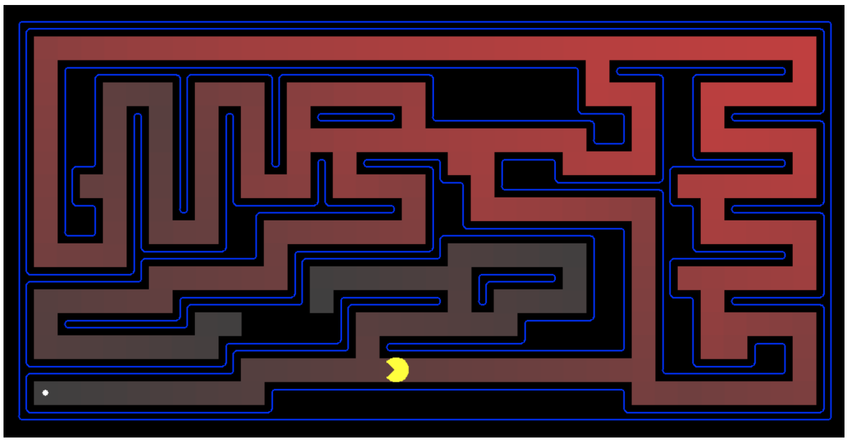
2. Push a tuple (Initial state, action = Stop, Cost = 0) into the stack
3. Create an **array of visited states**
4. Iterate while the stack is not empty:
    a. **Get the last path** (top element), which is the list of tuples stored up until that point.
    b. From the path **get the current state**, that is, the first element of the last tuple
    c. If the **current state is a goal** state, return a list with the second element for all tuples in the path except the first one, that is, all the preceding actions that take us from the initial state to the goal, other than the one corresponding to the initial, which is trivial (stop).
    d. If the **current state has not been visited,** add it to the array of visited states and iterate through its successors:
        - **If the successor state has not been visited either, add it to the path but NOT the visited array**. This is a very important distinction for DFS to work: in this specific implementation, we check a state as visited if it has been expanded. However, because of the nature of DFS **not all successors will be expanded (therefore not visited)** just because of being added to the path. Only the last one will, as it will be the one popped from the top of the stack in the next iteration

If the stack is empty, we exit the previous loop without any return, then the **search was not successful.**

BREADTH FIRST SEARCH (BFS):

SEARCH RESULTS:

```
Path found with total cost of 68 in 0.01474 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```



DESCRIPTION:

The code that implements this algorithm is the same as the one for DFS, as both of them are general search algorithms, which means that **steps from 2 until the end can be replicated from the previous description**. However, the difference comes in **step 1**, where instead of initializing a stack, now we **use a queue**.

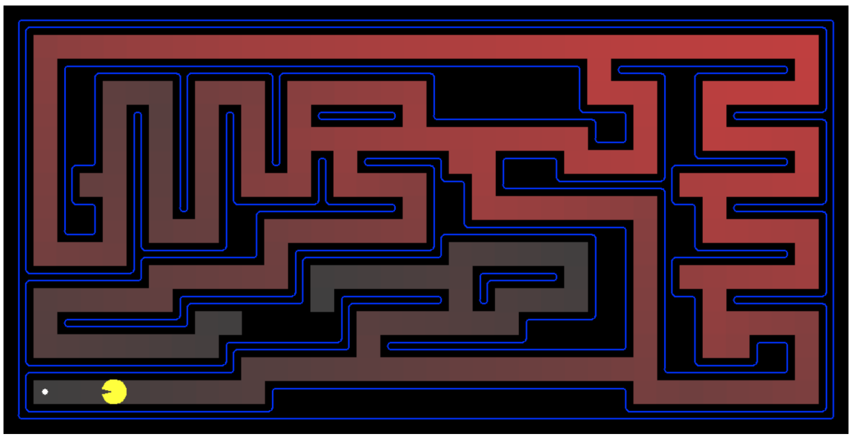
While the LIFO structure of the stack works for DFS (get the **last** element ~= keep expanding the **last** generated node), the workflow of BFS needs instead a queue to be programmed properly. The best way to understand this is to **think of the open list as a queue**. In fact, each of the paths (elements in the queue) is pretty much an open list with the associated actions and costs for the nodes. **Expanded nodes are put at the end of the open list**, but we select the node at the beginning of the open list to start expanding. That is exactly how a queue (**FIFO**, first in - first out) works!

This fact leads to an implementation in *search.py* where there is a **common method**, *generalGraphSearch(problem, structure, greedy=False),* able to **implement both algorithms**. Now one can understand that, if the structure passed is a **queue**, it will implement BFS, while it will implement DFS in case it is a **stack**.

DIJKSTRA (UCS):

SEARCH RESULTS:

```
Path found with total cost of 68 in 0.02942 seconds
Search nodes expanded: 269
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```



DESCRIPTION:

Just like with BFS and DFS, we can **reuse steps 2-end** as long as the **data structures** initialized previously and passed onto the second step make sense for the **purpose of this algorithm**. Recall that the **main difference** between this algorithm and the previous ones is that, instead of just inserting expanded nodes either at the end or the beginning of the open list and then keep expanding from the beginning of the list, **it actually analyzes in a way the list at each step, choosing the 'best' node to expand**. The best way to translate this to Python is through the use of a **priority queue**.

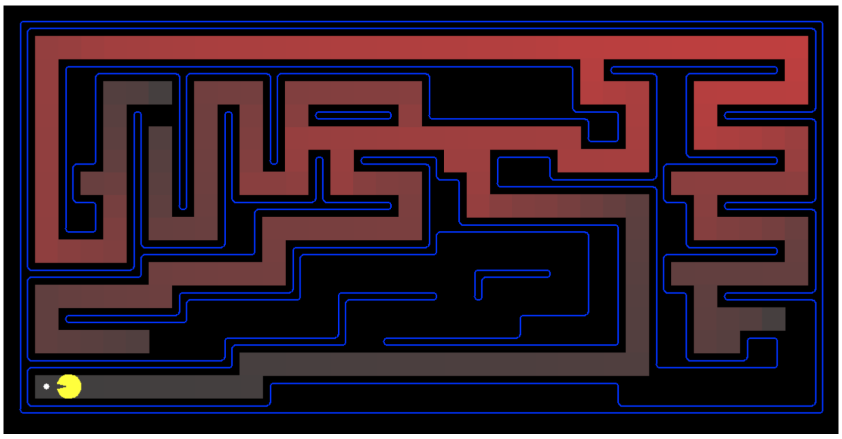
In order to obtain a proper priority queue:

1. We rely on the backwards cost, so we **get the actions necessary to move along a path** (just like in the queue and stack, elements in the priority queue are paths of tuples).
2. We calculate the **cost for the obtained actions** together and get a total cost.
3. We have obtained a number (total cost) associated with each path, which will be used to **establish priority.**
4. For each path pushed into the priority queue, it is **placed in the right position** depending on the priority parameter (in this case, the backwards to the initial state total cost)
5. Now we have the **'best' paths** (those with priority: the lower backwards cost) **in the beginning of the open list** in order to keep expanding.

A*:

SEARCH RESULTS:

```
Path found with total cost of 68 in 0.02846 seconds
Search nodes expanded: 224
Pacman emerges victorious! Score: 442
Average Score: 442.0
Scores:        442.0
Win Rate:      1/1 (1.00)
Record:        Win
```



DESCRIPTION:

Just like with BFS and DFS, we can **reuse steps 2-end** as long as the **data structures** initialized previously and passed onto the second step make sense for the **purpose of this algorithm**. Recall that the **main difference** between this algorithm and the previous ones is that it uses a **heuristic function** to 'sort' the open list (choose the **next node to be expanded**) in a more reliable way. In this implementation, the chosen default heuristic is the **null heuristic**, which is trivial.

After analyzing Dijkstra and the concept of the priority queue, understanding the next steps becomes rather simple. The key idea is that the cost (priority) for A* is determined by the sum of two parameters: g(path) is the accumulated backwards cost just like in Dijkstra and h(path) is the null heuristic for the last state in the given path:

1. Calculate **f(path) = g(path) + h(path)**
2. Create a **priority queue**
3. Pass **(f,h) as priority parameters** for the queue. That means that in case of a tie of values for the A* function f, the heuristic is the tie breaker.

It is worth noting that on top of the correct parameters that define each of the previous algorithms, the *generalGraphSearch* method also needs to be passed, at least, a *SearchProblem* type object. This is about the definition of a class more than search algorithms, so we will not explain it in depth, but in the *search.py* file one can find the different methods of this *SearchProblem* class, which are used by the algorithms themselves. These methods include functionalities such as getting the start and goal states, getting the successors of a state given as an input argument, or getting the cost of the actions. The definitions of these methods include references to other files of the pacman project

PROBLEM 1: FINDING A FIXED FOOD DOT

In this section we focus on the behaviour of the PositionSearchProblem, given in *searchAgents.py*, and how this class relates to the graph search algorithms previously analyzed in this paper. We will carry out an in depth interpretation of the code, and explain one by one, how each of the necessary formal elements of the search problem is implemented. We will also perform several tests, in mazes we will design for this purpose, using different agents to compare, plot and analyze how using different search algorithms may affect the performance in this problem.

RELATIONSHIP BETWEEN SEARCH PROBLEM AND THE AGENTS:

The class PositionSearchProblem receives the following as input parameters: an instance of **gameState** (a class whose definition we can find inside *game.py*), a **cost function** *costFn*, which if it's not specified by the user has a default value of 1 (this is defined as a lambda function for notation purposes), a **goal** *goal*, which has a default value of (1,1), a start parameter (we will refer to it as *start*), which has a default value of None, a **warn** parameter, which is is set as None by default and a **visualize** parameter set to True.

Inside the constructor we find the definition of several attributes, first the attribute ***self.walls*** in which we save the walls of the game, then the ***self.start*** parameter, which is set to the initial position of the pacman only if *start* is None, if it is not, we set as starting position the one the user passed as argument. Afterwards we define the parameter ***self.goal***, to do so we check two cases, if there's only one food on the maze, then it is set as the coordinates of said food, and in any other case it is set as the *goal* input parameter. The PositionSearchProblem also has a parameter ***self.costFn***, which is set to *costFn*, the input cost function.

For display purposes we keep track of the visited states, and count the number of expanded nodes, the necessary variables (***self._visited***, ***self._visitedList*** and ***self._expanded***) are initialized to the adequate values,(which are {}, [] and 0 respectively).

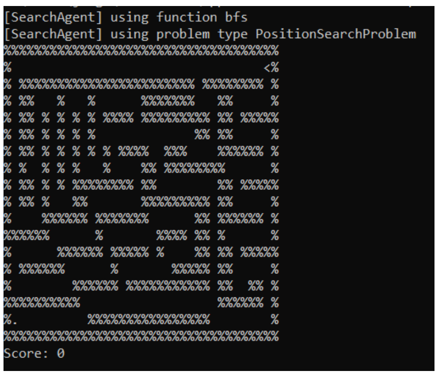
In all the search algorithms defined in search.py, we find that an input parameter is problem, and they all call the function *generalGraphSearch*, which to complete is task, must know all the information stored in the problem instance (which can be PositionSearchProblem or others), so in this manner the class PositionSearchProblem defines the problem space, which is information that every search algorithm needs to find the optimal path.

The class PositionSearchProblem, as well as storing key information, also provides us with some useful methods. The two we found more interesting are *getSuccesors*, which returns all of the successor states, the action the agent must perform and the cost of getting to each of them (in the following format [(successor_1, action_1, cost_1), … ,(succesor_k, action_k, cost_k)]), where k is the number of successors, and *getCostofActions*, which can be used to determine whether a particular sequence of actions is legal or not.

IMPLEMENTATION ANALYSIS - PROBLEM SPACE:

The PositionSearchProblem is provided to us in the form of a class inside the file *searchAgents.py*. We have explained the python implementation in detail in the last section, and now we will go deeper into each element explaining in detail the complete problem space.

- **STATE SPACE**: it is a two dimensional grid, this can be viewed using the command print(str(gameState)), where each of the tiles can be characterized as follows:
    - As we can see in the string representation of the gameState, there are three main items, wall tiles, represented as "%", then we have food tiles which are represented as ".", and the pacman tile which is represented as "<". No tile can be simultaneously a pacman tile and food tile, and neither the pacman nor the food can ever be in a wall tile.
    - Since no diagonal moves are allowed, if two tiles are **adjacent**, they are either in the same row or same column and they **differ in only one** unit in **only** one coordinate, x if they are in the same row, and y if they are in the same column
    - Each of the tiles can be uniquely characterized by the following:
        - **(x,y)** coordinates, where x and y, must be integer values within the bounds of the particular maze, e.g, given a 34 x 11 maze, $0 < x <= 34$ & $0 < y <= 11$.
        - A boolean operator called **Food**(t), where t is a tile, which given a tile returns True, if it's a food tile and False otherwise. Assuming the **map** in which we are playing **is valid** for this problem (in layman's terms, if it only has one food tile), Food(t) = True, would imply that for any other tile Food(t') = False (1), and at the end of the problem, all tiles must satisfy (1) . If it's not valid, then a warning message will be shown on the screen, and this operator will not have that implication.
        - A boolean operator called **Pacman**(t), which given a tile returns True if and only if pacman is in t, this implies that for all other tiles it returns False.
        - A boolean operator called **Wall**(t), which returns true if the tile t is in fact a wall, and false otherwise, if Wall(t) == True for a certain tile t, then that implies directly Food(t) = Pacman(t) = False.



```
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                 <%
% %%%%%%%%%%%%%%%%%%%%%%% %%%%%%%% %
% %%   %   %      %%%%%%%   %%     %
% %% % % % % %%%% %%%%%%%%% %% %%%%%
% %% % % % %             %% %%     %
% %% % % % % % %%%%  %%%    %%%%%% %
% %  % % %   %    %% %%%%%%%%      %
% %% % % %%%%%%%% %%        %% %%%%%
% %% %   %%       %%%%%%%%% %%     %
%    %%%%%% %%%%%%%      %% %%%%%% %
%%%%%%      %       %%%% %% %      %
%      %%%%%% %%%%% %    %% %% %%%%%
% %%%%%%      %       %%%%% %%     %
%        %%%%%% %%%%%%%%%%% %%  %% %
%%%%%%%%%%                  %%%%%% %
%.         %%%%%%%%%%%%%%%%        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Score: 0
```

Example of the of print(str(gameState)) execution for MediumMaze, and bfs

- **INITIAL STATE**: the set of initial state/s of the pacman in a **(M x N) maze** is composed by tiles *t* that must all satisfy the following conditions:
    - **Wall**(*t*) must be **False** as the pacman can never be in a wall tile.
    - The (x,y) coordinates must be integers within the bounds of the map, in other words, **$0 < x <= M$ & $0 < y < N$.**
    - If the map is valid, i.e, there is only one food, we restrict the initial state even more, so that it has to satisfy that Food(*t*) = False.

- **GOAL STATE:** the goal state can be the only food in the maze if said maze is valid for this problem or the one specified by the user, in a **(M x N)** maze is a tile *t* that must satisfy the following conditions:
    - **Wall**(*t*) must be **False** as the pacman can never be in a wall tile.
    - The (x,y) coordinates must be integers within the bounds of the map, in other words, **$0 < x <= M$ & $0 < y < N$.**
    - **Food**(*t*) = True (2) since the goal is to find the food, if the map is valid t is the only tile for which (2) holds.

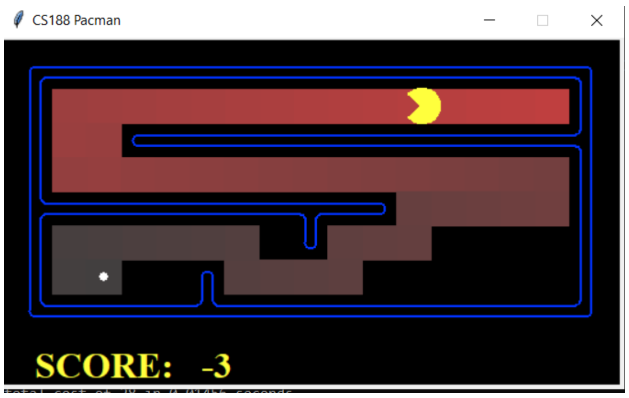
IMPLEMENTATION ANALYSIS -TEST CASES:

For this section we have designed a total of 5 mazes, in which we will analyze the performance of each of the following search algorithms: breadth - first search, depth - first search, A* with euclidean distance as heuristic and A* with manhattan distance as heuristic.

For each of the maps, we will analyze and explain the theoretical reasons that explain each of the results we got for each algorithm, and how those results were expected beforehand as the design of the maps was done on purpose to obtain a certain set of results.

We will follow an order of increasing complexity, i.e, we will cover first the maps which need less nodes to be expanded in order to find the solution, on average.

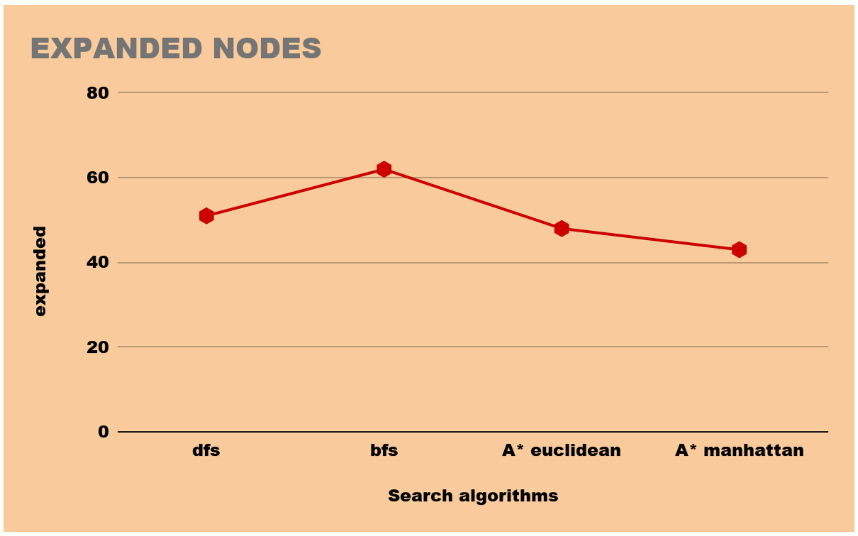
MAP 1



Expanded nodes using dfs

The results we obtained in this map were as follows.

- Number of expanded nodes: (1)



Number of expanded nodes per Search algorithm

The algorithm with the highest number of expanded nodes is breadth- first search with 62, then we find depth-first search with 51, then A* with the euclidean distance as heuristic with 48 and at last A* with the manhattan distance as heuristic with 43.

In depth-first search we know that it is neither complete nor admissible, in other words, we cannot ensure that it will find a solution, and if it does, we cannot know if it is the optimal one, which further along the analysis of the results of this map, we will explain why it does not find it in this case.

We know breadth-first search is admissible and complete whenever the branching factor is finite, i.e, when the maximum number of successors a given state has is less than infinity, in our case, that number is always two, as no diagonal movements are allowed, and thus it is both complete and admissible.

In most cases uninformed search will expand more nodes than informed search, as the heuristic function's purpose is to refine the searching process, we find such a circumstance here, as the map is big enough for the pacman to enter in unnecessary cycles (by cycles we mean redundant back in fourths, since it will always continue down a graph branch until it cannot go any more) when using depth first search, and it also expands all the nodes on the bottom right part when using breadth first, which are not needed for the optimal path.

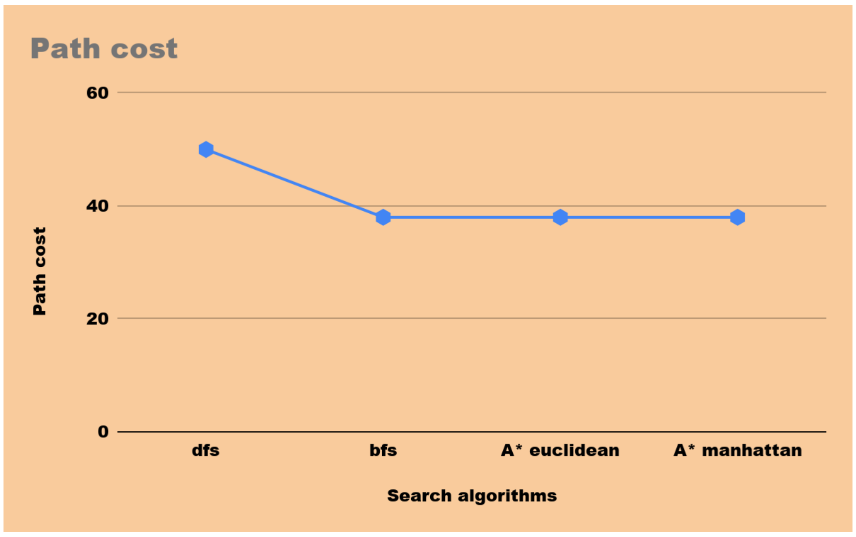
We also find a difference in expanded nodes in between A* with the euclidean distance and heuristic and the manhattan distance. To explain this difference we must take into account the formal definition of each of these distance notions, since we are working in a two dimensional plane we will state the definition for points in the plane, but it could be generalized to any dimension.

Given a point *a* = (x,y) and another point *b* = (x', y'):

- The manhattan distance between a and b is defined as **Manhattan**(a,b) = |x - x'| + |y- y'|
- The euclidean distance between a and b is defined as **Euclidean**(a,b) = √((x - x')^2 + (y - y')^2)

In the case of the Pacman game, the Manhattan notion of distance seems to be more suitable, since if we consider for example two points P = (x,y) and Q = (x+1, y+1), the euclidean distance Euclidean(P, Q) = √2, whereas Manhattan(P, Q) = 2, and 2 is the number of movements it would take the Pacman to go from P, to Q , assuming of course that the path is not blocked, for this reason the algorithm works better in general with the manhattan distance as heuristic function.

- Path cost: (2)



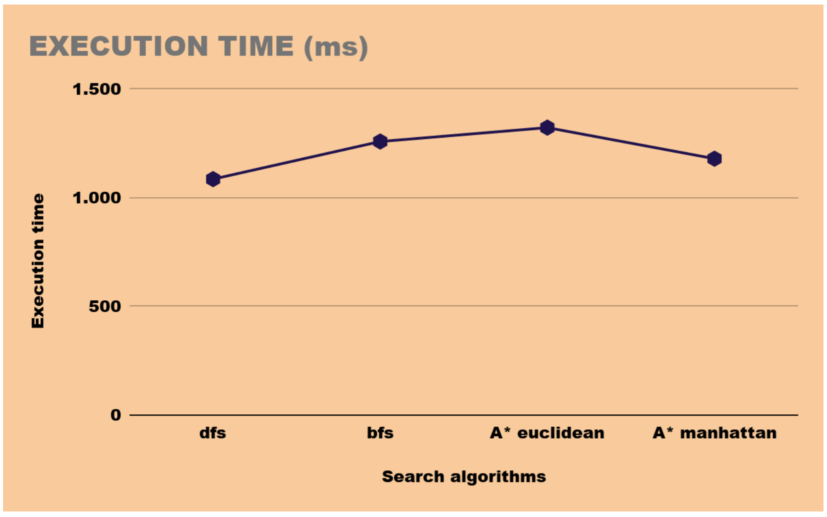
Path cost of each search algorithm

The highest path cost is obtained by depth-first search with a value of 50, and the all of the other algorithms, i.e, breadth-first search and A* with euclidean distance and manhattan distance as heuristic, obtain a cost of 38.

As explained in the part regarding the number of expanded nodes, in the case of the Pacman, breadth first search will always return the optimal path since its branching factor will always be finite. The manhattan distance and the euclidean distance as heuristic functions are both admissible since it never overestimates the distance from a point to the goal, since they are both admissible we can guarantee that the algorithm returns the optimal path.

On the other hand, depth-first search is not admissible and clearly returns a worse path than the other three, as going all the way in certain branches of the map causes the path to be suboptimal.

- Execution time: (3)



Execution time of each search algorithm (ms)

We can see that the fastest algorithm in this case is depth first search with a time of 0.01086 s, this is due to a variety of reasons. Firstly, as we have seen before it expands less nodes than breadth first search, for a detailed explanation refer to part (1) of the analysis, it also takes up considerably less space in memory.

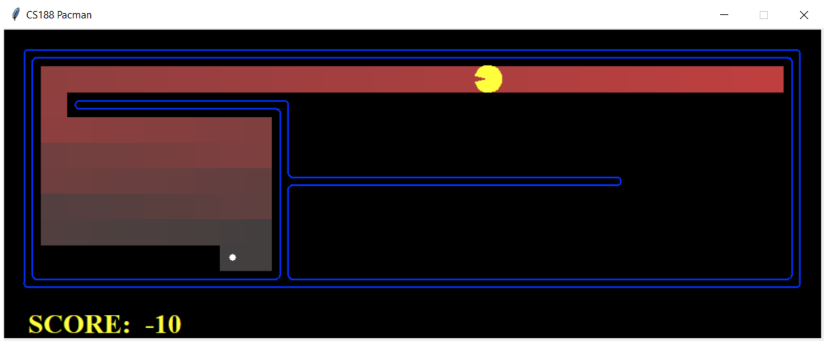
Breadth first search takes a heavy toll in memory as maps grow larger and larger since its space complexity is **O(|b|^d)**, where b is the branching factor and d is the number of edge traversals from start to goal, (since it is an uninformed search algorithm). On the other hand, A* although it expands significantly less nodes than breadth first, it takes longer to execute, since the memory space it will occupy is exponential to the number of vertices, and for each iteration it must perform an extra operation to decide how to continue building the optimal path, which is computing the sum of the accumulated cost **g(n)** and the heuristic function, **h(n)**.

It is interesting to see that A* with the manhattan distance outperforms breadth first search, as we saw in part (1), in this case A* expands less nodes, and it also takes up less space in memory, and makes faster and less computations to build the optimal path, since it does not have to square to terms and then take its root.

The exact execution times are the following:

Algorithm	Execution time (s)
Depth-first search	0.01086
Breadth-first search	0.01259
A* euclidean	0.01323
A* manhattan	0.01180

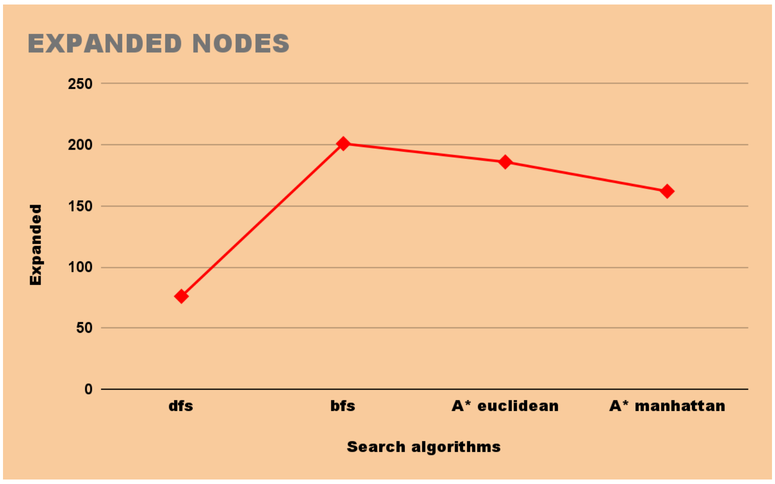
MAP 2



Expanded nodes using dfs

The results we obtained in this map were as follows.

- Number of expanded nodes: (1)



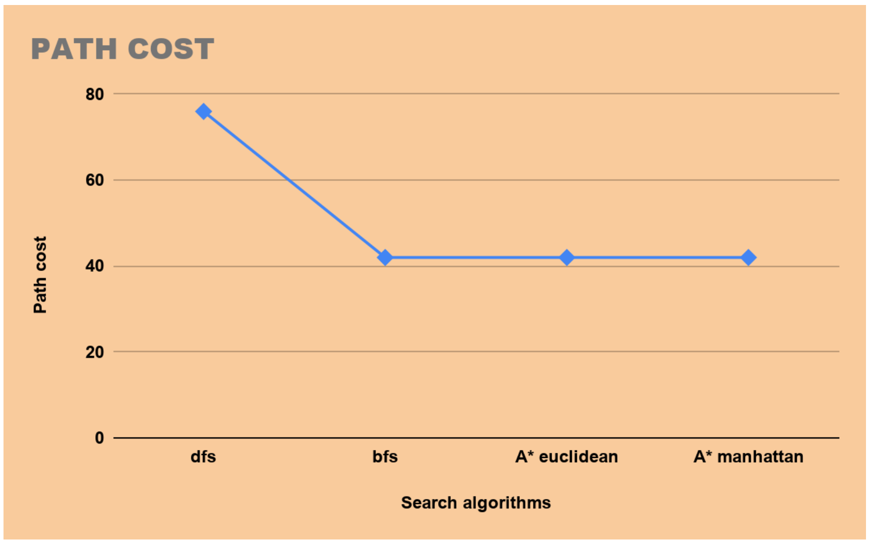
Number of expanded nodes per Search algorithm

The algorithm with the lowest number of expanded nodes in this case is depth-first search with 76, then A* with the manhattan distance with a total of 162, then A* with the euclidean distance which expands 186, and the highest is breadth-first search with a total of 201.

We know that depth-first search will choose a branch of the graph, and then traverse it, once it has run out of nodes, backtrack as little as possible and traverse the next available branch. As we can see in the photo of the map, it is designed in such a way that the first branch that depth-first chooses will get the pacman inside a “trap region” *r*, where the food (i.e our goal) is located at, in such a way that it will never need to backtrack out of said region, so it does not have to expand nodes that are not in it, that is the reason why dfs expands a number so low compared to the other algorithms, because breadth-first and A* will expand nodes out of the region *r*.

In this case breadth-first and A* with the euclidean distance expand around the same number of nodes, with the first one expanding 10% more nodes than the second. The difference is substantial when comparing breadth first and A* with the manhattan distance since the manhattan distance is the optimal heuristic, for the explanation refer to page 13 of this paper.

Path cost: (2)



Path cost of each search algorithm

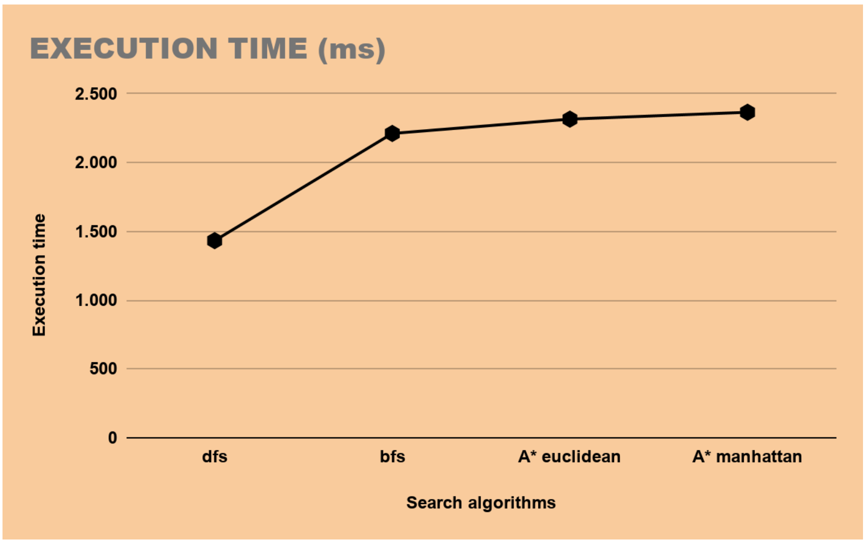
The highest path cost is obtained by depth-first search with a value of 76, and the all of the other algorithms, i.e, breadth-first search and A* with euclidean distance and manhattan distance as heuristic, obtain a cost of 42.

As explained in map1 (1), depth-first search is not admissible and therefore we cannot guarantee that the solution is optimal, if found. In the previously mentioned region *r*, the pacman will bounce on and on between the walls, and thus the solution will not be optimal, (by bounce we mean advance down a row until we find a wall, in that case it will need to backtrack).

Breadth - first search and A* with the manhattan heuristic always find the optimal solution, for a detailed explanation refer to page 12.

Since the euclidean distance is not a valid heuristic, we cannot assure that the path found was optimal, but in this case, since the cost is the same as in best-first and in A* with the manhattan distance, we conclude that it's also optimal.

- Execution time: (3)



Execution time of each search algorithm (ms)

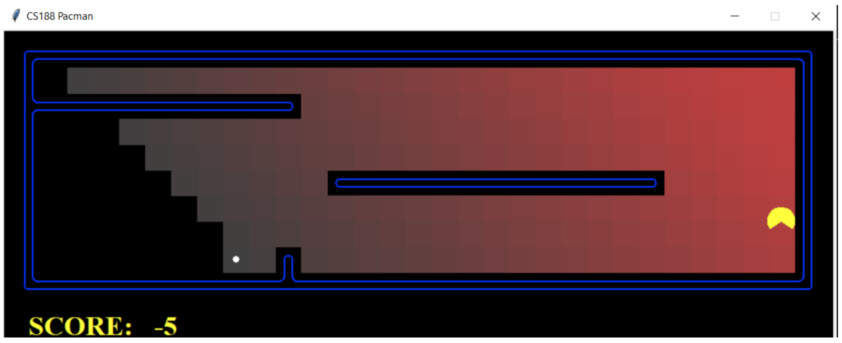
We can see that in the same manner as for Map1, the fastest algorithm is clearly depth-first search. In this case a very important reason for this difference is the fact that it expands substantially less nodes than the rest of the algorithms.

As explained on section (1) of map1, breadth-first search is extremely heavy on memory, especially as maps grow larger and larger, since its space complexity is **O(|b|^d),** where b is the branching factor and d is the number of edge traversals from start to goal,

Breadth-first search takes a heavy toll in memory as maps grow larger and larger since its space complexity is , where b is the branching factor and d is the number of edge traversals from start to goal, (since it is an uninformed search algorithm). On the other hand, A* although it expands significantly less nodes than breadth first, it takes longer to execute, since the memory space it will occupy is exponential to the number of vertices, and for each iteration it must perform an extra operation to decide how to continue building the optimal path, which is computing the sum of the accumulated cost **g(n)** and the heuristic function, **h(n)**.

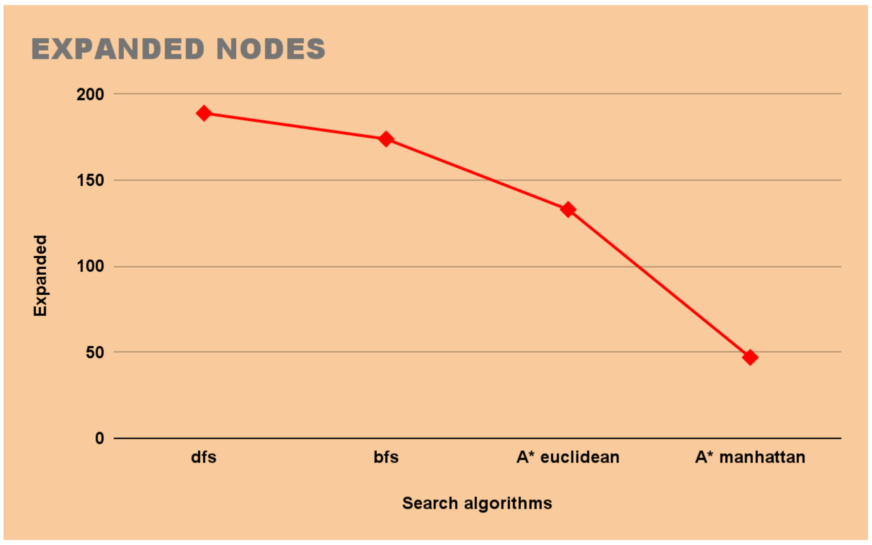
It is interesting to see that A* with the manhattan distance outperforms breadth first search, as we saw in part (1), in this case A* expands less nodes, and it also takes up less space in memory, and makes faster and less computations to build the optimal path, since it does not have to square to terms and then take its root.

MAP 3



The results obtained in this map were as follows:

- Number of expanded nodes: (1)



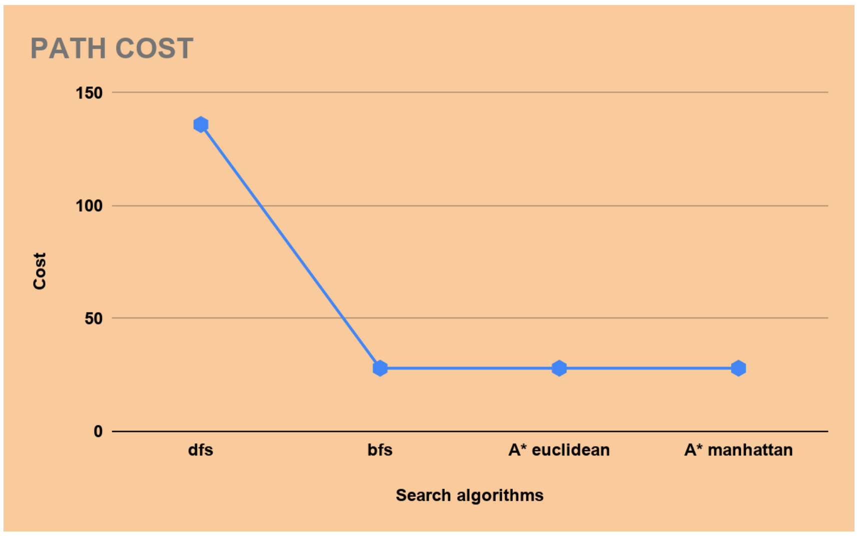
Number of expanded nodes per search algorithm

In this case depth-first search expands the highest number of nodes, followed by breadth-first, then we find A* with the euclidean distance as heuristic and at last A* with the manhattan distance as heuristic.

This map was designed on purpose as an example to illustrate how given a certain graph, depth-first search may expand a far greater amount of nodes than A*, as it has long and wide lanes, to create long back in fourths. It also expands less nodes than breadth-first search because in this case the goal is not found in the boundary of the map.

It is of interest to observe the massive difference in performance between both heuristics in A*, since one expands almost three times more than the other. We have explained before why the manhattan distance is the optimal heuristic for this problem, in this case the difference is clear, since the worse a heuristic is, the more the complexity rises.

- Path cost: (2)



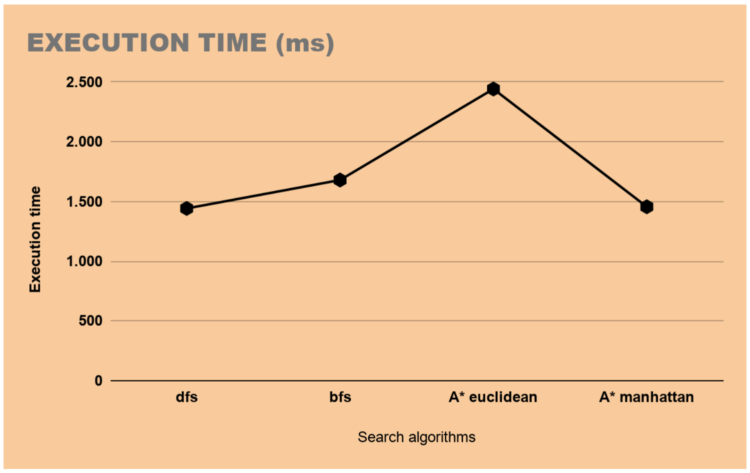
Path cost for each search algorithm

The most expensive path is the one found by depth first search, with a cost of 136, both breadth-first search and A* find a path with cost 28, which is optimal, (A* with both heuristics since both are admissible).

When using depth-first search, the pacman requires to backtrack a lot, since the number of walls is very small, which was designed on purpose, it must traverse a great number of unnecessary positions to get to the goal, which is why its cost is almost 5 times higher than that of the ideal path, this maze is a textbook example of a case in which depth first search performs poorly.

Breadth - first search and A* all return the same cost, as expected since the branching factor is finite, (which makes breadth-first admissible), and both the euclidean distance and the manhattan distance serve as admissible heuristics and thus A* is guaranteed to return the optimal path.

- Execution time: (3)



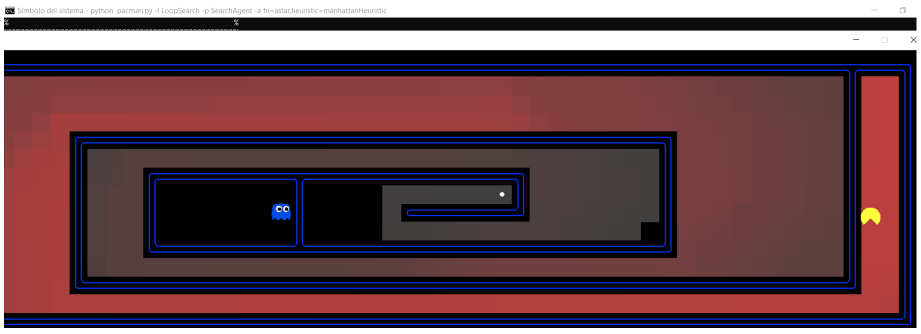
Execution time per search algorithm (ms)

The fastest algorithm is depth-first search, followed by A* with the manhattan heuristic, then breadth-first search, and the slowest in this case is A* with the euclidean distance as heuristic.

We expected A* with the manhattan distance to be one of the fastest, if not the fastest, in this case since it expands substantially less nodes than the others, but it is slightly outperformed in this regard by depth-first search (difference is of 0.00012 s), because this second one does not require nowhere near the amount of computations that A* does.

Breadth-first search expands less nodes than depth-first but it is heavier on memory, that's why it is less efficient in terms of time. The A* with the euclidean distance as heuristic takes the longest of them all, since it is the third in number of expanded nodes, but it has the most time consuming operation per iteration.

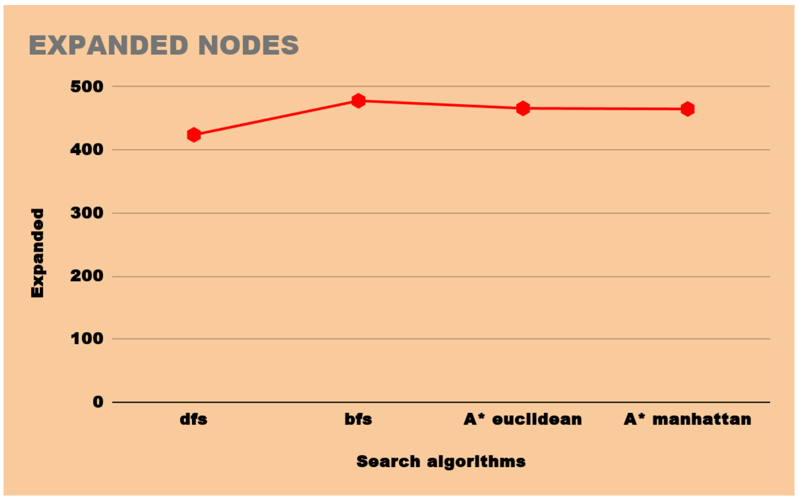
MAP 4


Símbolo del sistema - python pacman.py -l LoopSearch -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

Expanded nodes using A with the manhattan distance*

The results obtained in this map were as follows:

- Number of expanded nodes: (1)

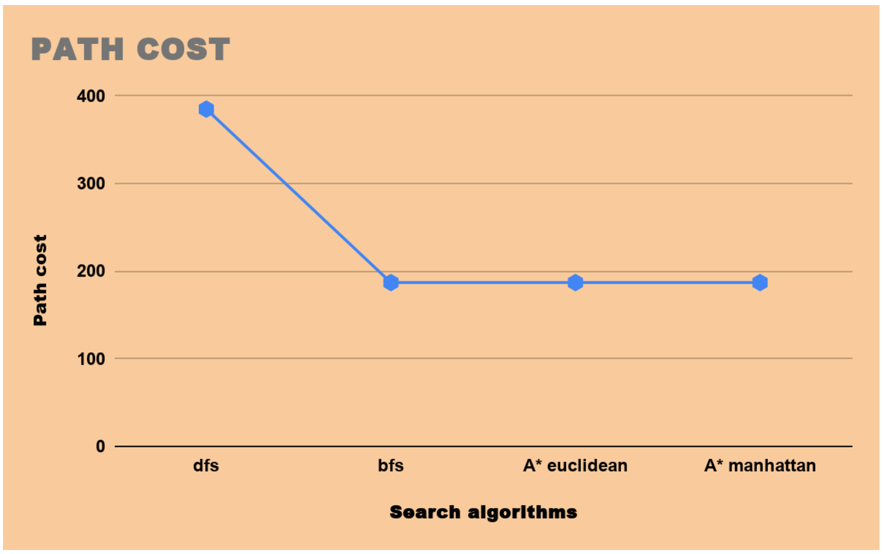


Number of expanded nodes per search algorithm

Breadth-first search expands the most nodes in this case, followed by A* with h(n) = Euclidean distance, then A* with the manhattan distance as heuristic and at last depth-first search. It is important to remark that the differences in this map are very small, the highest difference is of 54 nodes, i.e, breadth-first expands 112% of the nodes that depth-first does.

Since the goal is very far away from the starting position, it was expected that breadth-first search expanded the highest number of nodes. We find such a case in this map, as it was designed on purpose to create a very long path from start to goal, that's the reason why bfs expands the most nodes out of the 4. A* expands almost the exact number of nodes with both the euclidean and manhattan distance as heuristics (466 and 465 respectively). Depth - first is the lowest in this case because of the map purposefully having very long lanes with very little possible moves, as we have seen in class, it behaves better when the goals are far, because it can get us "close" "quicker".

- Path cost: (2)

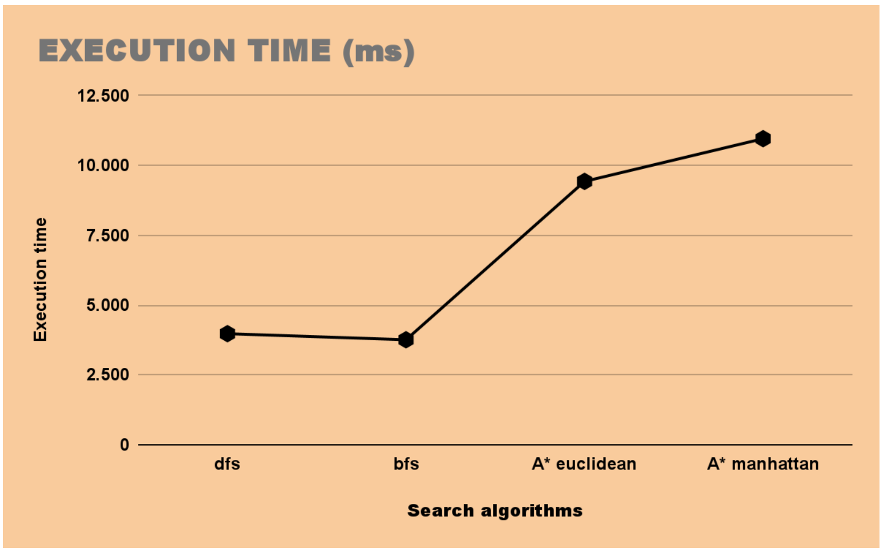


Path cost for each search algorithm

The most expensive path is the one found by depth first search, with a cost of 385, both breadth-first search and A*, with both the manhattan and the euclidean distance as heuristics find a path with cost 187, which is optimal, (A* with both heuristics since both are admissible).

Since the necessary conditions hold, for a detailed explanation refer to page 19 of this paper, breadth first and A* with both of the heuristics given will return the optimal path, and that is the reason as to why they return a path with the same cost.

- Execution time: (3)



Execution time per search algorithm (ms)

The fastest algorithm in this case is breadth-first search, followed by depth-first search, then A* with the euclidean distance as heuristic and the slowest in this case is A* with the manhattan distance as heuristic.

The uninformed search algorithms and the heuristic search algorithms appear clustered into the same orders of magnitude in terms of execution time, with the heuristic search being significantly slower, which could be expected since the map is rather big, and the computations that the heuristic algorithms must carry out are more time consuming than those of the uninformed search algorithms, which is particularly important for this maze since the number of expanded nodes for each algorithm are very close.

EXTRA MAZES

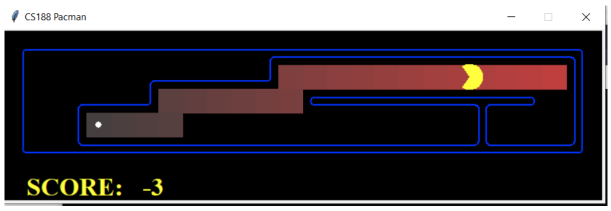
We were asked to design 2 mazes that satisfy the following conditions:

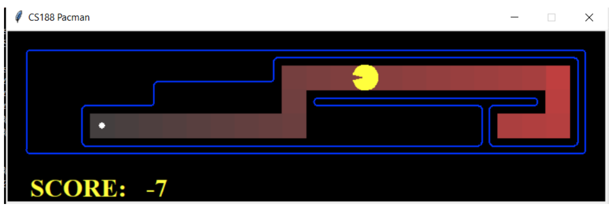
- Breadth-first search expands less nodes than A* with the manhattan distance as heuristic.

- Depth-first search expands less nodes than A* with the manhattan distance as heuristic, and find the optimal path.

We were not able to design a maze which satisfied the first condition since the purpose that the heuristic satisfies is expanding only the nodes that seem "more promising", and since for the pacman, with each action being of unitary cost, the manhattan distance is the optimal heuristic **h***(n), for the complete explanation as to why this heuristic estimates perfectly the cost of getting to a goal refer to page 13. Since the heuristic is optimal it will not expand nodes “in vain”, and thus it will expand less nodes than breadth-first since it will be properly informed in contrast to bfs which by definition is not.

The map we designed that satisfies the latter conditions is the following:





Nodes expanded in dfs vs nodes expanded in A with the manhattan distance as heuristic*

In the first image we see how depth first search expands directly to the left and since the solution is in that branch, in this case dfs also finds the optimal solution, with less nodes expanded than A* with the manhattan distance as heuristic.

On the right we see how A* expands nodes inside the trap region as the combination heuristic h(n) + cost g(n), forces it to carry on expanding through the branch expanded by the state right below the start state (the term ‘*branch*’ refers to the notion of branch inside a binary tree) that eventually is useless, as it will end.

COMMENTS ABOUT EXECUTION TIMES

When discussing the different execution times for each algorithm in all of the 4 previously mentioned algorithms, we have failed to mention a critical element that could cause substantial differences that may not be explainable otherwise, hardware capabilities.

All the tests were run consecutively so as to decrease the impact that hardware inefficiencies may have had in execution time, but there is still some error that must be taken into account in this regard, as concurrent processes that could have been running may have hindered some of the execution times and thus influence the numbers provided in the descriptions above, since the times are of the order of milliseconds.

PROBLEM 2: EATING ALL THE FOOD DOTS

For this section, we are concerned about understanding and testing the behaviour of pacman search agents (in particular, of 3 of them) on a FoodSearchProblem, which is nothing but a search problem where the pacman must eat all the food dots in order to finish. We will study and analyze the python implementation, then run some tests with different agents and problem spaces/layouts/mazes as parameters, compare, contrast and plot the different results and, finally, explain those differences in the obtained results.

The idea behind a search agent is to implement in the pacman the ability of automatic movement, instead of being told where to move by the player. The algorithms previously discussed inside this paper come in very handy at this point, as we need to provide the pacman agent with some sort of AI mechanism that allows it to make movement decisions on its own.

IMPLEMENTATION ANALYSIS - PROBLEM SPACE:

The file *searchAgents.py* provides as with the definition of a class **FoodSearchProblem** that defines the problem space, along with the implementations for 3 different agents to test this search problem. For now, we will focus on the **problem space**, before moving on to the agents with their own methods and algorithms.

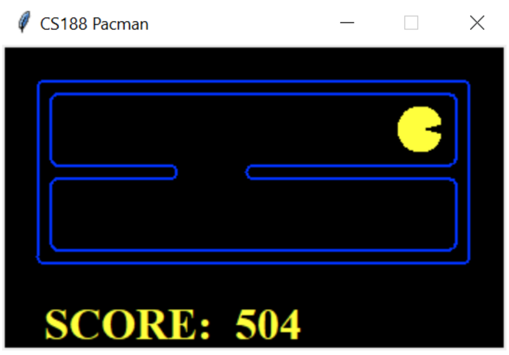
The class FoodSearchProblem defines a problem space, so in this portion we must analyze the **main characteristics that define a search problem**, just like we did in class

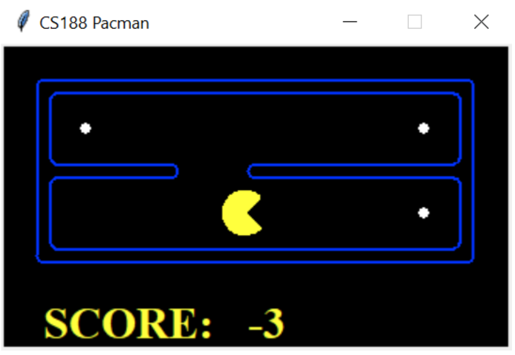
- **STATE SPACE:** it is two-dimensional space (**grid**) where each **position P** within the grid at a particular moment is characterized by:
    - Its **x-y coordinates**, such that either:
        - coord is a 2D array such that **coord(P) = (x,y)**
        - x(P) = x coordinate and y(P) = y coordinate, two separate values
    - A boolean parameter **Food** such that:
        - If there is still food in P, then **Food(P) = True**
        - If there is no food (there was no food to begin with or pacman ate it), then **Food(P) = False**
    - A boolean parameter **Pacman** such that:
        - If pacman is in P, then **Pacman(P) = True**. This implies that Pacman(P') = False for all P' different from P
        - If not, **Pacman(P) = False**
        - **Alternatively**, we could create a variable pacmanPosition = (x,y), just like P(coord). This is more efficient memory-wise, as pacman can only be in one position and creating this parameter just to define it as False for all positions but one is pretty redundant. This is what is done in the python implementation.

- **INITIAL STATE**: a grid **InitialGrid** defined in a way such that:
    - **For all P** contained in InitialGrid, that is, all possible positions for the Pacman to be in, their coordinates **coord(P) = (x,y) verify that:**

        - x > 0
        - y > 0
        - If coord(P)=(x,y), coord(P')=(x',y') and P and P' are adjacent, then either | x - x' | = 1 or | y - y' | = 1
    - **Food(P) = True** if and only if in the the layout of the maze, a **food dot is placed in (x,y)** and coord(P)=(x,y)
    - **Pacman(P)** = True only for P such that coord(P)=(x,y) and pacman is placed at (x,y) in the layout of the maze. Alternatively, **pacmanPosition** = (a,b) with (a,b) = (x,y) where the pacman is placed in the layout diagram. It **is represented by a P**

- **GOAL STATE:**
    - **For all P** contained in InitialGrid, their coordinates **coord(P) = (x,y) verify that:**
        - x > 0
        - y > 0
        - If coord(P)=(x,y), coord(P')=(x',y') and P and P' are adjacent, then either | x - x' | = 1 or | y - y' | = 1
    - **Food(P) = FALSE for all P**
    - **Pacman(P)** = True only for P such that coord(P)=(x,y) and pacman ends up placed at (x,y) after eating all dots. Or, what is the same, if P was the only position such that **Food(P) = True one action before pacman moved into it**. Alternatively, **pacmanPosition** = (a,b) with (a,b) = (x,y) where the pacman ends up placed in the layout diagram after eating all dots.





Examples of a goal state (1) and a state space during execution (2)

IMPLEMENTATION ANALYSIS - AGENTS:

We have 3 agents, where each of them is an 'autonomous' version of the pacman that gets its ability to perform actions by itself from a particular algorithm. When we pass one of these agents along with a particular layout (maze) as input parameters, our program can automatically perform a search problem. We can test each of the 3 in a set of layouts and compare their performance, but first let us analyze the difference in their algorithms:

AGENT 1

The algorithm implemented in this agent is **A* with the Manhattan distance** as heuristic. The A* algorithm is nothing without a properly defined heuristic, so:

1. **Define Manhattan between two particular** points. If p1 = (x1,y2) and p2 = (x2,y2), then Man(p1,p2) = | x1 - x2 | + | y1 - y2 |
2. Define Manhattan **for a particular state in the problem space**. This is done in *foodHeuristicManhattan()*. The idea is to:
    a. **Get the current state**, which is coded as a variable containing the current position of the pacman and the grid of food dots.
    b. Show the grid as a **list of food dots**.
    c. Iterate through **all the dots** with food calculating their **Manhattan distance** with respect to the pacman current position.
    d. **Return the largest** one. In order words, the heuristic is the Manhattan distance between the current pacman position and the furthest food dot.

With the heuristic defined, there is not a whole more to talk about. Recall that the technical analysis/explanation of **A*** is available in pages 7-8 of this paper. Recall that it is under ***aStarSearch()*** in the *search.py* file. This method needs to **input parameters**, so we pass:

1. A ***FoodSearchProblem*** **type object**. Refer to the first part of this section (IMPLEMENTATION ANALYSIS - PROBLEM SPACE) to understand the class corresponding to such objects.
2. The **heuristic** we just obtained.

Now, everything is ready for the definition of **AGENT 1**, which in the implementation is the class *AStarFoodSearchAgent_FoodManhattanDistance.* This is **defined from the more general class *SearchAgent***, which is defined from the even more general class *Agent,* and includes the **necessary methods to be performed by the agent** (get action, register initial state, etc). All we need to specify for our super-version of this class (Agent 1) are **2 parameters in the constructor:**

1. The associated search function is the ***aStarSearch()*** **method** with the previously discussed arguments (problem state object and Manhattan heuristic function).
2. The search type is a **FoodSearchProblem**. Recall that by default this is instead a PositionSearchProblem (discussed in section problem 1 of this paper).

AGENT 2

The algorithm implemented in this agent is **A* with the *mazeDistance*** function as heuristic. The idea is the **same as with the previous agen**t but with a **different heuristic**, so we will limit ourselves to explain the heuristic to avoid repetition. In order to get the heuristic function that we want to pass to *aStarSearch()*:

1. **Define the maze distance between two particular** points. This is quite trickier than the Manhattan case:

   a. Defined **p1 and p2** by its (x,y) **coordinates**.
   b. Get the points of the search/problem space where there are **walls**. A function of the gameState class assists us with this.
   c. Check that **p1 and p2 are not walls**.
   d. Perform a **position search problem** with p1 as starting point and p2 as end point. Refer to the previous section of this paper for an in depth explanation of the position search problem.
   e. The **length of the result** of that search (number of actions taken or positions visited) **will be the maze distance**.

2. Define the food heuristic maze distance **for a particular state in the problem space**. This is done in *foodHeuristicManhattan()*. The idea is to:
   e. **Get the current state**, which is coded as a variable containing the current position of the pacman and the grid of food dots.
   f. Show the grid as a **list of food dots**.
   g. Iterate through **all the dots** with food calculating their **Maze distance** with respect to the pacman current position.
   h. **Return the largest** one. In order words, the heuristic is the maze distance between the current pacman position and the furthest food dot, according to this distance.

AGENT 3

This agent is **very different** from the previous two in the fact that, rather than relying on a pre-defined algorithm (and heuristic function), it **follows its own internal scheme**. The code of such scheme is rather complex and makes use of auxiliary methods, so what follows is a more instruction-like **explanation of it**:

1. Adapt the **position search problem to a food search problem**:
   a. The position search problem focuses on the agent getting from an initial position to a goal.
   b. The idea is to **divide the food search problem into as many position search problems as food dots** exist in the maze.
   c. That way, each partial move from a food dot to the next is a position search problem between the positions of such food dots.
   d. This is repeated until all food dots are 'eaten' (visited)
2. Create a method to **perform this partial moves in the most efficient** possible way:
   a. The idea is to find the **path to the closest food dot** given the current position, starting from the initial state and repeating the process until all food dots are eaten.
   b. Create an array to keep track of the visited dots and an array to store the actions. In each position of this array there is a partial move.
   c. Follow a **BFS strategy to obtain the path**. Refer to page 5 of this paper for an in - depth description of the BFS algorithm.
3. Create a **standard search agent.**

4. Create variables to store **meaningful information:** time, current state, accumulated cost of actions, number of expanded nodes…
5. While there are **still food dots** in the maze, iterate:
    a. Use our method defined in (2) from the current state.
    b. Check that the actions returned are indeed legal.
    c. Update the values of the meaningful variables.

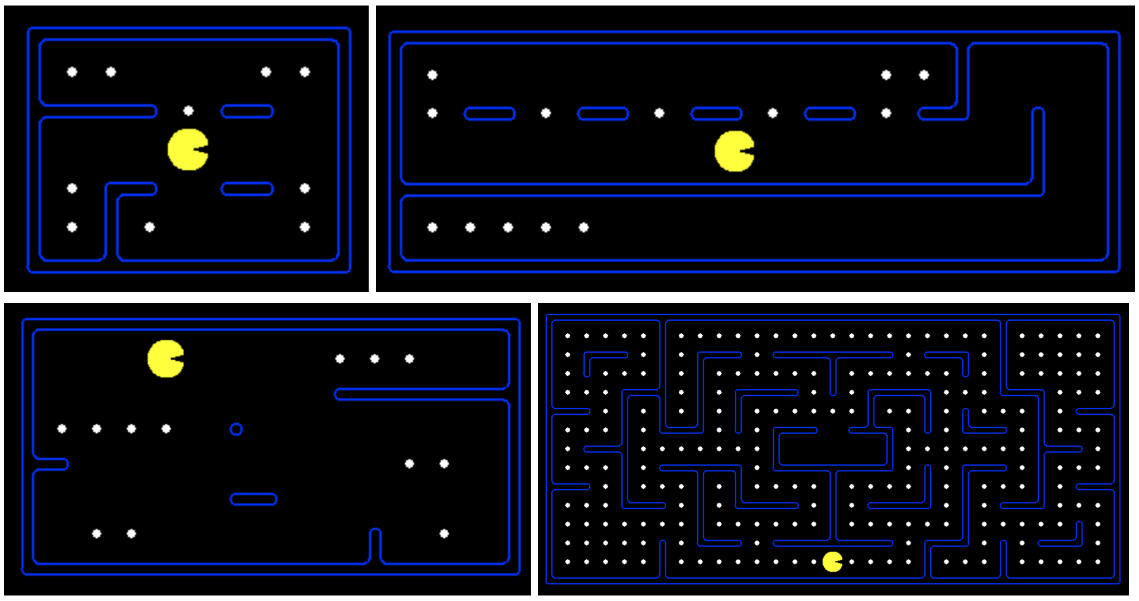
AGENT COMPARISON:

Inside this section. a series of tests are shown in order to contrast the behaviour and performance of each of the previous agents.

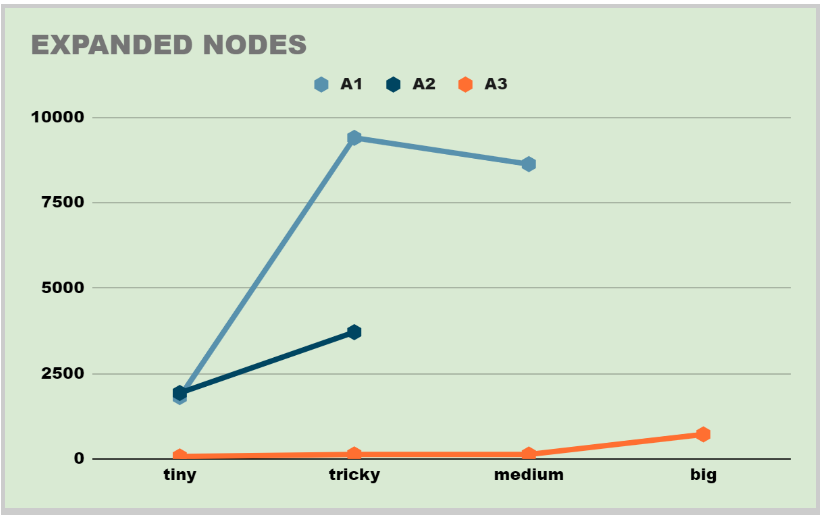
- Search performed: Food Search Problem.
- Search agents used: Agent 1, Agent 2, Agent 3.
- Problem spaces used: tinySearch, trickySearch, bigSearch, mediumFoodSearch
    - mediumFoodSearch.lay was specifically designed by us in order to evaluate the different agents somewhere in the 'middle' of tinySearch and bigSearch, which are tiny and massive. It gave us a lot of insight and interesting results, and it can be found in the *layout-students* delivered file.

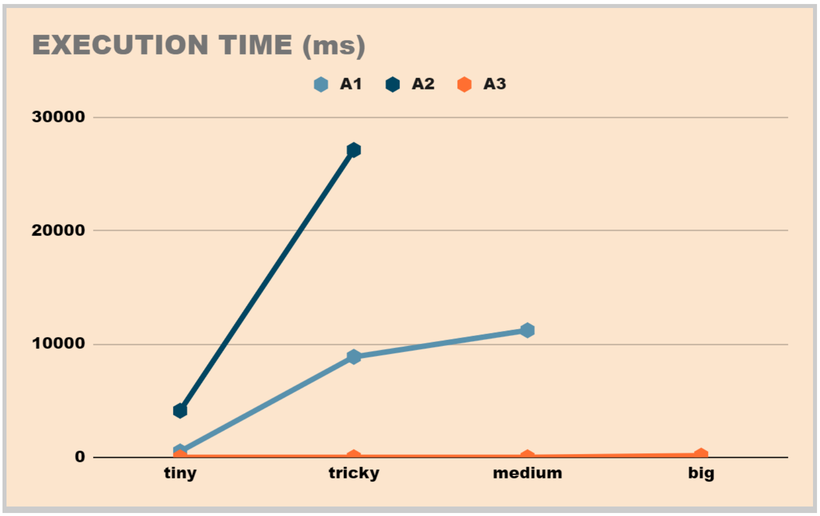
We will provide plot representations of the results obtained from each agent-layout combination. In the next section, we will synthesize these results and relate them to all the previous study of the agents and their corresponding algorithms.

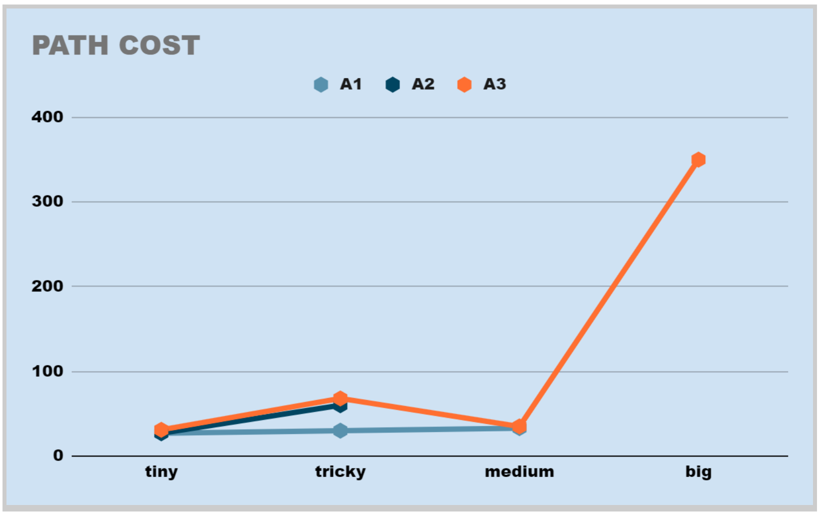
OUR MAZES [in order: *tinySearch, trickySearch, mediumFoodSearch, bigSearch*]



TEST GRAPHS


EXPANDED NODES
A1
A2
A3
10000
7500
5000
2500
0
tiny
tricky
medium
big


EXECUTION TIME (ms)
A1
A2
A3
30000
20000
10000
0
tiny
tricky
medium
big


PATH COST
A1
A2
A3
400
300
200
100
0
tiny
tricky
medium
big

ANALYSIS OF RESULTS:

Values missing for a particular agent-layout combination in any of the graphs mean that the search agent was not able to be run in that particular. This will happen due to the algorithm that is followed by the agent being too costly for layouts bigger than a concrete size. As much as we try it, the exponential growth of the search in these cases will cause the system to crash.

This happen specifically for:

- Agent 2 and medium search
- Agent 2 and big search
- Agent 1 and big search

Which means that:

- Agent 3 was able to perform in all 4 mazes
- Agent 1 was able to perform in 3 mazes
- Agent 2 was able to perform in 2 mazes

These are pretty good insights, but the question is, why does this happen? Think of the algorithm followed by Agent 3: from each state, it gets the nearest food dot and the necessary action(s) to move to it. It will do this as many times as necessary to eat all food dots, which is as many times as food dots. Therefore, the growth is linear, and increases linearly as we increase the amount of food dots.

On the other hand, we said that the other 2 agents follow algorithms whose cost increases exponentially. Let us see why:

When it comes to Agent 1, notice how after each action the agent finds itself in a new state with a new set of characteristics. And for each new state, that is, every time it moves, it needs to calculate the heuristic for that particular state, and for that, it must calculate the heuristic of all remaining food points… this process repeatedly becomes extremely costly. On top of that, this heuristic is passed to the A* algorithm which is, in itself, costly: everytime an action is taken it must assign priorities to the nodes in the open list, compare (sort) them and choose the best one to expand.

All the previous can be said for Agent 2, with the slight difference that it does not even reach the capacity to perform the medium search. There is indeed a reason for this: although most of its search strategy is a copy of that of Agent 1, the heuristic is even more costly: refer to the implementation analysis in this section, where the manhattan distance (h for Agent 1) and the food maze distance (h for Agent 2) are discussed.

CONCLUSIONS

In the introduction of this project, we stated how we were excited about finding the magic behind pacman and getting our hands on a practical application of Artificial Intelligence. We can now confirm that we have indeed gotten very comfortable with this type of code and the application of AI in programming, and that the developed ideas will surely be helpful for us whenever we want to apply AI within a real project.

In our opinion, one of the main benefits extracted from this activity, while at the same time one of the biggest challenges, is the ability of abstraction when it comes to interpreting code. It is one thing to use a library with predefined algorithms and use them in the most efficient way to achieve a purpose, and it is a whole different thing to go through the code of such algorithms and relate all the implementation decisions to the theoretical aspects. There is no doubt that our knowledge of search problems and search algorithms acquired during the beginning of this semester was key for a successful completion of this project.

What made the code itself quite challenging was the high level of distribution along the many files that conform the pacman python project. The constant references to objects, importation of methods and data structures, etc really made us pay attention to every small detail in order to properly dissect the code.

What we also think was highly beneficial was the balance between the analysis of code and the actual practical part in which, through the commands provided or the option to create our own mazes, we were able to visualize the different problems, the performance of the different agents' algorithms, etc. Sometimes there is nothing better than a game to create an immersive experience, even if it is developing or analyzing rather than playing.

All in all, we firmly believe that our knowledge of the practical side and applications of the world of search in Artificial Intelligence has escalated a lot through the development of this project, and we hope the quality of the paper will be able to reflect it.

PERSONAL COMMENTS

One of the main questions that arose during the completion of the project was, how else can we design an intelligent agent that solved the pacman, as in the files we were given it is modeled as a search problem, but how could we design it so that an agent that is capable of reasoning under uncertainty? At first we thought that the answer to that question was too far-fetched for our current knowledge, as it seemed rather complicated, but after some thought we came up with some ideas on how the Pacman could be solved using Markov decision processes (MDP).

We know that an MDP, *M,* is defined as a tuple, with 4 elements $M := <S,A,T,R>$, where:

- S : set of all possible states.
- A: set of all possible actions.
- T: transition function.
- R: reward function

In our case, i.e the Pacman problem, S = all possible states in which the pacman may be, A = {North, South, East, West}, the transition function is deterministic, in other words, given a state *s*, and a legal action *a*, we know with 100% certainty the state to which the pacman arrives at. The reward function offers more options, as in this case we could use as reward, the differences in score between the current state or the previous, or simply only give reward whenever the pacman eats a ghost or food. We will choose the second one.

One of the issues we saw with this definition is that the set of all possible states may be too large, we could reduce its number by reducing the number of attributes that characterize each state, instead of saving for each state, for example, the position of each ghost, the number of living ghosts, the position of the pacman, and the distance to each ghost, and the set of legal actions, we could reduce the set of all possible states, by saving only the position of the pacman, the distance to the the nearest living ghost and the set of legal actions.

In this manner the set of all possible states decreases considerably, but another issue arises, the set of all possible distances may be very large, so we could fuzzify, in *n* different categories, and assign a category of distance based on the degree of membership to each of this *n* fuzzy sets, this would reduce the set of states by more than enough but it would also hinder the performance of the agent, as the information loss is great. We must be careful as this information loss may limit the ability of the model to learn, as the attributes chosen may lead to loops.

Assuming that the MDP was properly defined, by applying Bellman's optimality equations, and value iteration algorithm, we should be able to find the optimal policy to complete the game, but at the beginning we do not have the information to build the MDP.

In this stage we thought that the best option was to execute random actions until we have enough information, but how can we determine what having enough information means?

As we could not figure out a reasonable way in which we can define enough information, we thought that the best way was, to set a value $0 < x <= 1$, and execute a random action with probability x, but also this parameter decreases over time gradually following an input monotonous function, we could try different ones such as 1/x, or e^-x, and tune that parameter in that way. In that manner x would be some sort of learning rate, which we could only decrease if we assume that the difficulty of the games does not increase over time, because otherwise we would need to learn information for every game we play.

We also considered it useful to keep a history of states visited by the pacman on a stack, as for some states, we can gain information and prevent a random action from taking us there. For example if the pacman was in a state *s* in which the only legal action was to go back to its previous state, we can keep track of that to further bound the set of useful states that the pacman can visit.

This history of movements can be useful for a particular set of maps, but not have a lot of effect on others, and it would take a considerable amount of memory, especially if the mazes are huge, and so to decide whether we implement it or not we must take into account some information regarding hardware capabilities.

This model would be able to determine the ideal action, based on the maximization of a reward, and thus it would fall under the realm of machine learning, and it would be strongly related to the paradigm of reinforcement learning.