

Artificial Intelligence

CANIN Raphaël & KOCKAN Melisa

TP1	3
4 - Depth-First Search (DFS)	3
5 - Breadth-First Search (BFS)	7
6 - Uniform Cost Search (UCS)	10
7 - A* Search	14
8 - Finding all the Corners	19
9 - Corners Problem : Heuristic	21
10 - Eating All the Dots	23
11 - Suboptimal Search	25
TP2	27
1- Tests	27
2 - Python Files	27
3 - Starting	28
4 - Reflex Agents	29
Improve the ReflexAgent in multiAgents.py	29
2) Test your code	29
5 - Minimax	32
Implementation	32
2. Test your code	33
6 - AlphaBeta algorithm	34
7 - Expectimax	34
8- Evaluation function	35
Run away from ghosts	35
Reaching for food	35
Scared times of ghosts	36
4) Try bold moves	36
5) Testing Expectimax	37
Conclusion	37

TP1

4 - Depth-First Search (DFS)

For games 4 to 7, the goal is to eat a food point at a given location on the game map. To do such a thing, we will first use the Depth First Search algorithm. This algorithm allows exploration of a graph in depth and will help to establish the path between Pacman and the food. Here is a simple example of how this algorithm works.

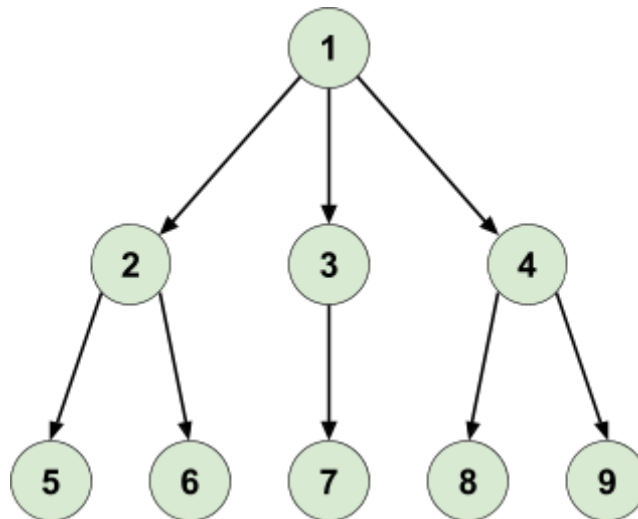


Figure 1 - Simple graph

The DFS algorithm uses two arrays to store the nodes. The first one is a list of already explored vertices, it allows not to visit twice the same vertex. The second one is a stack, which is a LIFO (Last In First Out) data structure, which allows to store the successors of each visited vertex. This algorithm will explore a branch as deeply as possible (until it reaches a leaf) and repeat for each branch.

N° of loop	Explored	Frontier
1	1	4, 3, 2
2	1, 4	9, 8, 3, 2
3	1, 4, 9	8, 3, 2
4	1, 4, 9, 8	3, 2
5	1, 4, 9, 8, 3	7, 2
6	1, 4, 9, 8, 3, 7	2
7	1, 4, 9, 8, 3, 7, 2	6, 5
8	1, 4, 9, 8, 3, 7, 2, 6	5

9	1, 4, 9, 8, 3, 7, 6, 5	Ø
---	------------------------	---

1 -

We are going to implement the DFS algorithm in Python in order to find a path to the food from the Pacman's location. Our function will return a list of actions (West, North...) that he will have to perform in order to eat the food point.

Once the food point was found using this algorithm, we had to reconstruct the path. A first functional idea was to store the actions in the action variable of each node. Thus, each node contained the path taken to reach it from the origin. This worked, but was problematic for the later parts (Corners Problems in particular). So we have chosen to follow the advice and the help of Mrs. Abdeddaim in order to modify our algorithm. Thus, we find the path using a dictionary of the predecessors of each node, the keys used being the nodes themselves.

For more details on the implementation, the code provided in the annex is commented in high detail.

2 - Now that we have implemented the DFS algorithm, let's test it on several examples.

Tiny Maze

```
$ python3 pacman_AIC.py -l tinyMaze -p SearchAgent

[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 10 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 500
Average Score: 500.0
Scores:          500.0
Win Rate:        1/1 (1.00)
Record:          Win
```

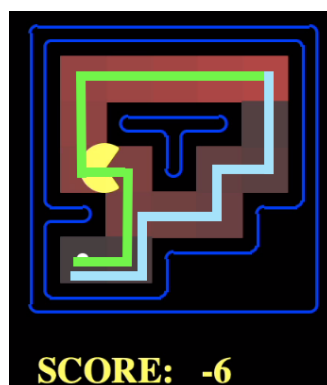


Figure 2 - Path to food using DFS on TinyMaze

Here is a GIF of the result : https://perso.esiee.fr/~caninr/E4S2_IA/DFS_TinyMaze.gif.

As can be seen, the algorithm was able to quickly find one of the paths to reach the food. The path found is represented in green on the figure above. However, we notice that it is not the shortest path, it has a cost of 11, while the path in light blue has a cost of 8. The DFS algorithm does not aim to find the shortest path but returns the first path it finds, however deep it may be.

Medium Maze

```
$ python3 pacman_AIC.py -l mediumMaze -p SearchAgent
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 130 in 0.0 seconds
Search nodes expanded: 144
Pacman emerges victorious! Score: 380
Average Score: 380.0
Scores:      380.0
Win Rate:    1/1 (1.00)
Record:      Win
```

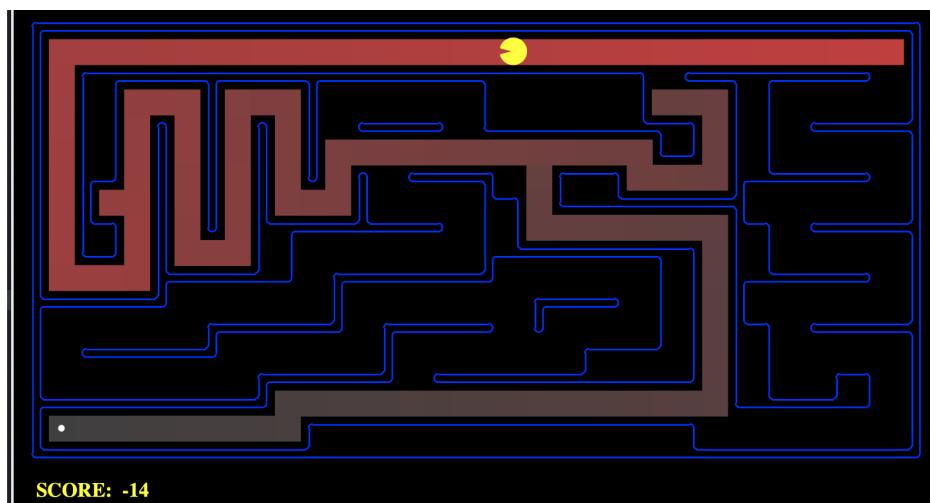


Figure 3 - Path to food using DFS on MediumMaze

Here is a GIF of the result : https://perso.esiee.fr/~caninr/E4S2_IA/DFS_MediumMaze.gif.

As we can see once again, the algorithm allows us to find a solution to our problem but does not give the shortest possible path.

Big Maze

```
$ python3 pacman_AIC.py -l bigMaze --frameTime 0 -z .5 -p SearchAgent

[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 391
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:      300.0
Win Rate:    1/1 (1.00)
Record:      Win
```

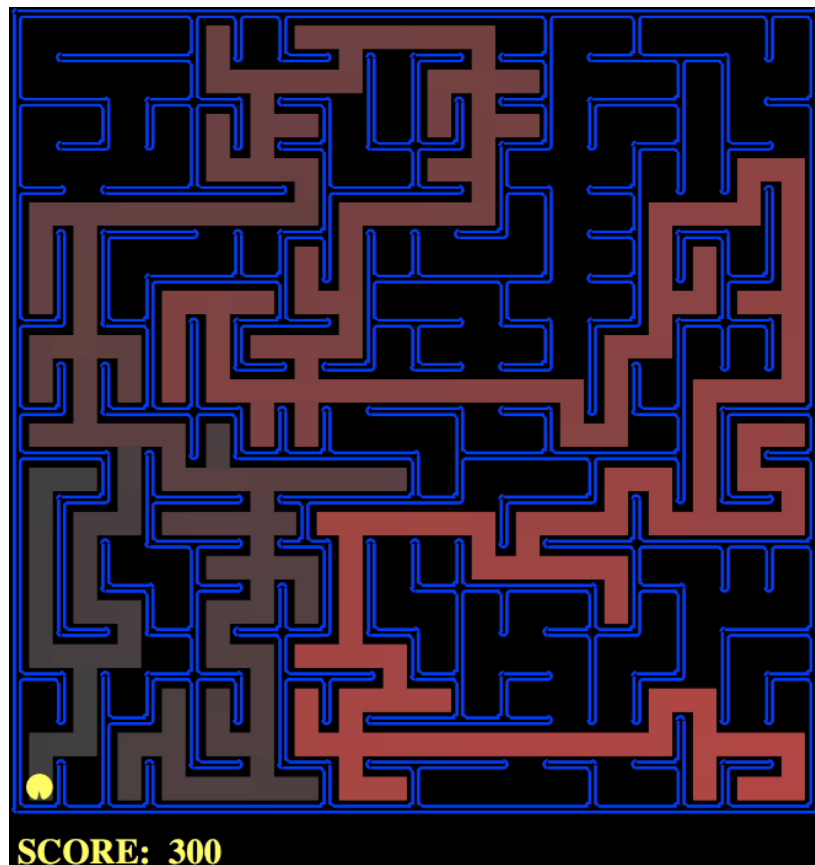


Figure 4 - Path to food using DFS on BigMaze

Here is a GIF of the result : https://perso.esiee.fr/~caninr/E4S2_IA/DFS_BigMaze.gif.

Once again, the DFS algorithm was able to quickly find a path to the goal. And once again, we see that it does not maximize the score, it does not return the shortest possible path

5 - Breadth-First Search (BFS)

The second exploration algorithm we will use is the Breadth-First Search (BFS).

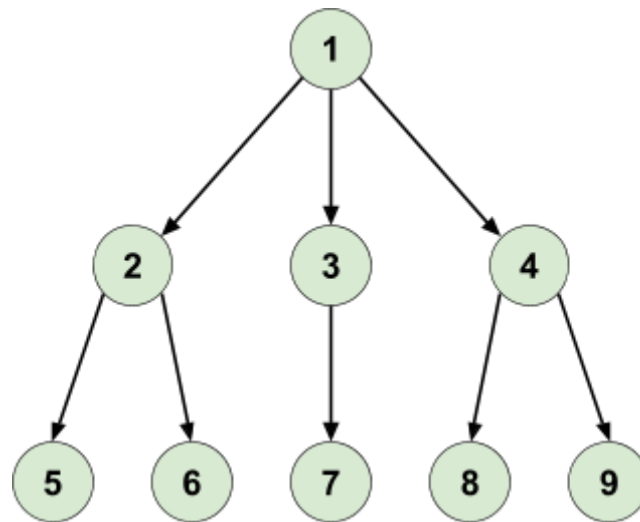


Figure 5 - Simple graph

BFS is rather similar to the DFS in operation with the difference that it is a width-based path and not a depth-based one. In particular, it allows us to find the shortest paths in a non-weighted graph. In fact, it browses the nodes from layer to layer and thus explores all the paths at the same rhythm. This implies that when a path is found, it will be a shortest path.

Like the DFS algorithm, BFS uses two arrays to store nodes. The only difference is the type of the second array. Indeed, we will use a queue, which is a FIFO (First In First Out) data structure. This algorithm will completely explore one layer of the graph before moving on to the next.

N° of loop	Explored	Frontier
1	1	2, 3, 4
2	1, 2	3, 4, 5, 6
3	1, 2, 3	4, 5, 6, 7
4	1, 2, 3, 4	5, 6, 7, 8, 9
5	1, 2, 3, 4, 5	6, 7, 8, 9
6	1, 2, 3, 4, 5, 6	7, 8, 9
7	1, 2, 3, 4, 5, 6, 7	8, 9
8	1, 2, 3, 4, 5, 6, 7, 8	9
9	1, 2, 3, 4, 5, 6, 7, 8, 9	Ø

1 -

The implementation of BFS is very straightforward now that DFS is already implemented. The only thing to change, as explained above, is the data structure used for the frontier. We will use a Queue which is a FIFO structure.

For more details on the implementation, the code provided in the annex is commented in high detail.

2 -

Medium Maze

```
$ python3 pacman_AIC.py -l mediumMaze -p SearchAgent -a fn=bfs

[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 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
```

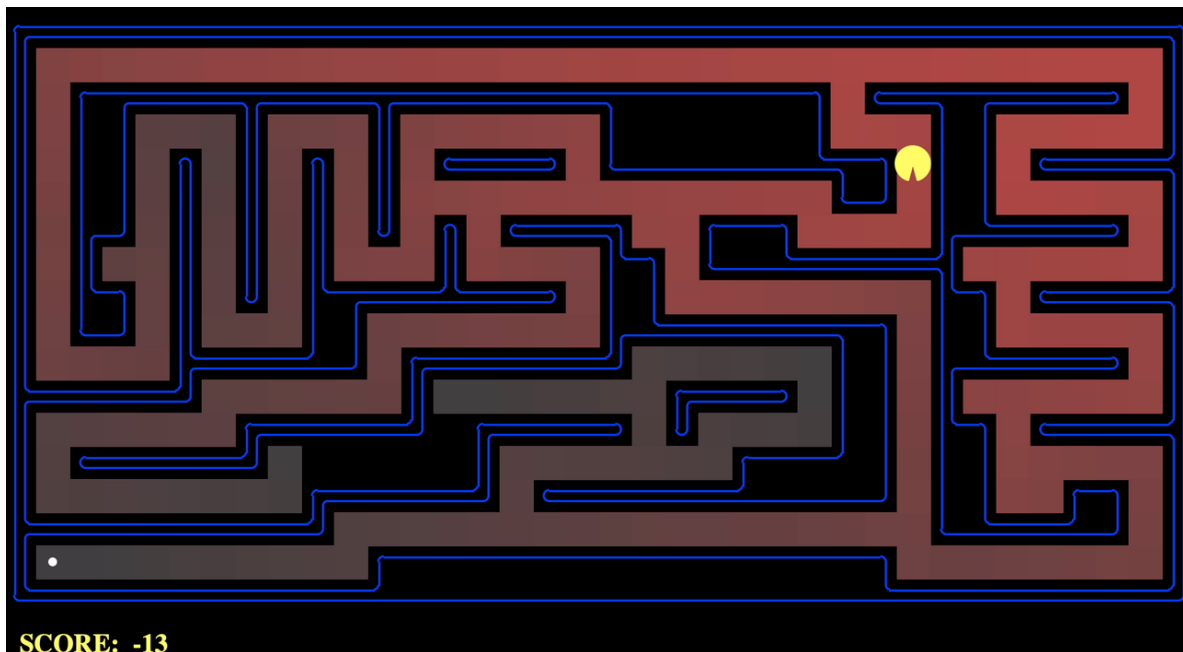


Figure 6 - Path to food using BFS on MediumMaze

Here is a GIF of the result : https://perso.esiee.fr/~caninr/E4S2_IA/BFS_MediumMaze.gif.

As we can see, BFS allows us to find the shortest path to the solution. Thus, we go from 130 using DFS to barely 68 using BFS. Of course, the score is also higher. For this particular problem, BFS is more relevant to use than DFS because finding a path that is the shortest is what we are looking for to maximize our score.

Big Maze

```
$ python3 pacman_AIC.py -l bigMaze -p SearchAgent -a fn=bfs -z .5  
--frameTime 0
```

```
[SearchAgent] using function bfs  
[SearchAgent] using problem type PositionSearchProblem  
Path found with total cost of 210 in 0.0 seconds  
Search nodes expanded: 619  
Pacman emerges victorious! Score: 300  
Average Score: 300.0  
Scores:      300.0  
Win Rate:    1/1 (1.00)  
Record:      Win
```

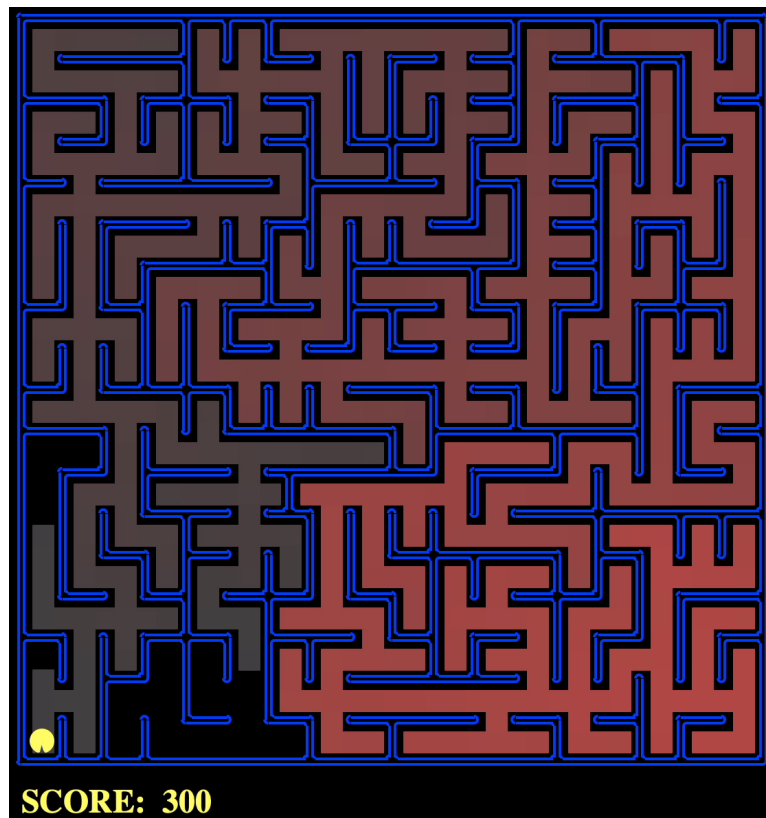


Figure 7 - Path to food using BFS on BigMaze

Here is a GIF of the result : https://perso.esiee.fr/~caninr/E4S2_IA/BFS_BigMaze.gif.

Here there is no improvement over the DFS algorithm. It seems that the DFS luckily found one of the shortest paths to the food. Note that the BFS algorithm explored twice as many nodes to achieve the same result. This might seem greedy, but we should not forget that BFS guarantees to find a shortest path.

6 - Uniform Cost Search (UCS)

The problem with the BFS algorithm is that when the cost is not unitary on each edge, the path will not necessarily be the shortest in the cost sense. The Uniform Cost Search algorithm looks at the lowest cost nodes first. Thus, it finds a shortest path based on the costs on each edge.

To implement such an algorithm, we will use a priority queue and update the cost of a node (i.e. the cost of the path to reach it) at each loop turn.

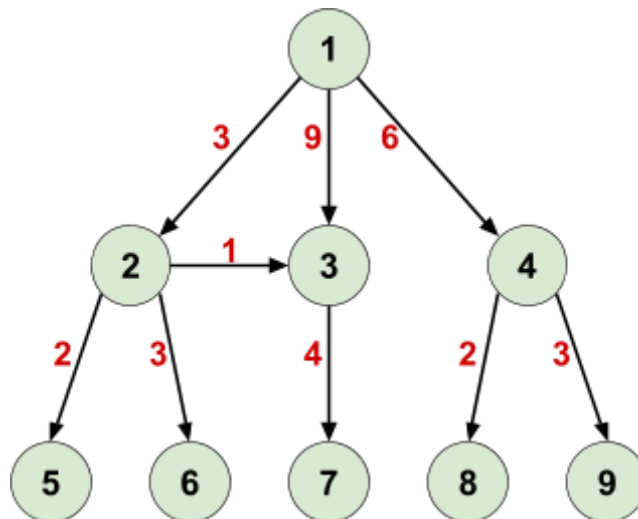


Figure 8 - Simple graph with non-unitary costs

N° of loop	Explored	Frontier
1	1	2 (cost: 3), 4(6) , 3(9)
2	1, 2	3(4), 5(5), 6(6), 4(6)
3	1, 2, 3	5(5), 6(6), 4(6), 7(8)
4	1, 2, 3, 5	6(6), 4(6), 7(8)
5	1, 2, 3, 5, 6	4(6), 7(8)
6	1, 2, 3, 5, 6, 4	7(8), 8(8), 9(9)

7	1, 2, 3, 5, 6, 4, 7	8(8), 9(9)
8	1, 2, 3, 5, 6, 4, 7, 8	9(9)
9	1, 2, 3, 5, 6, 4, 7, 8, 9	Ø

1 -

Implementing the Uniform Cost Search (UCS) exploration algorithm is simple, we just take the BFS code and change two things. First, we replace the queue with a priority queue. Second, we update the cost of each node on each loop.

For more details on the implementation, the code provided in the annex is commented in high detail.

2 -

Medium Maze

```
$ python3 pacman_AIC.py -l mediumMaze -p SearchAgent -a fn=ucs

[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 68 in 0.0 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
```

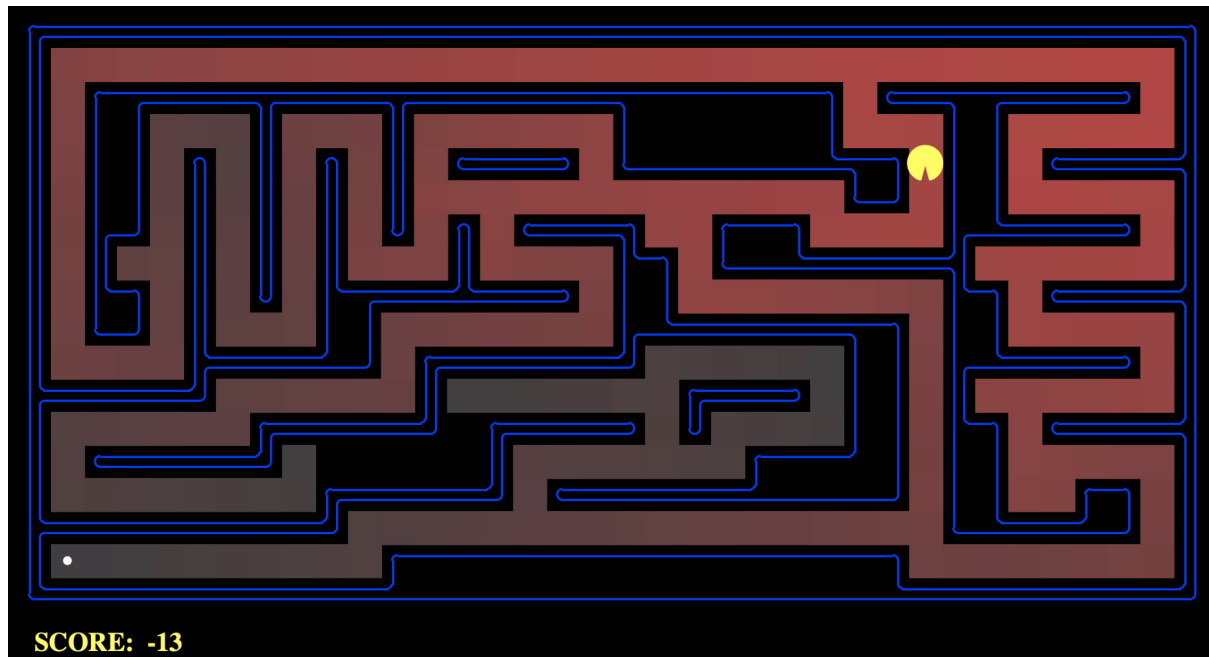


Figure 9 - Path to food using UCS on MediumMaze

Here is a GIF of the result : https://perso.esiee.fr/~caninr/E4S2_IA/UCS_MediumMaze.gif.

Each edge of this graph has a unit cost. Thus, the UCS algorithm performs here exactly the same way as BFS. It is in the following examples that the interest of UCS will appear.

Medium Dotted Maze

```
$ python3 pacman_AIC.py -l mediumDottedMaze -p StayEastSearchAgent

Path found with total cost of 1 in 0.0 seconds
Search nodes expanded: 186
Pacman emerges victorious! Score: 646
Average Score: 646.0
Scores:        646.0
Win Rate:      1/1 (1.00)
Record:        Win
```

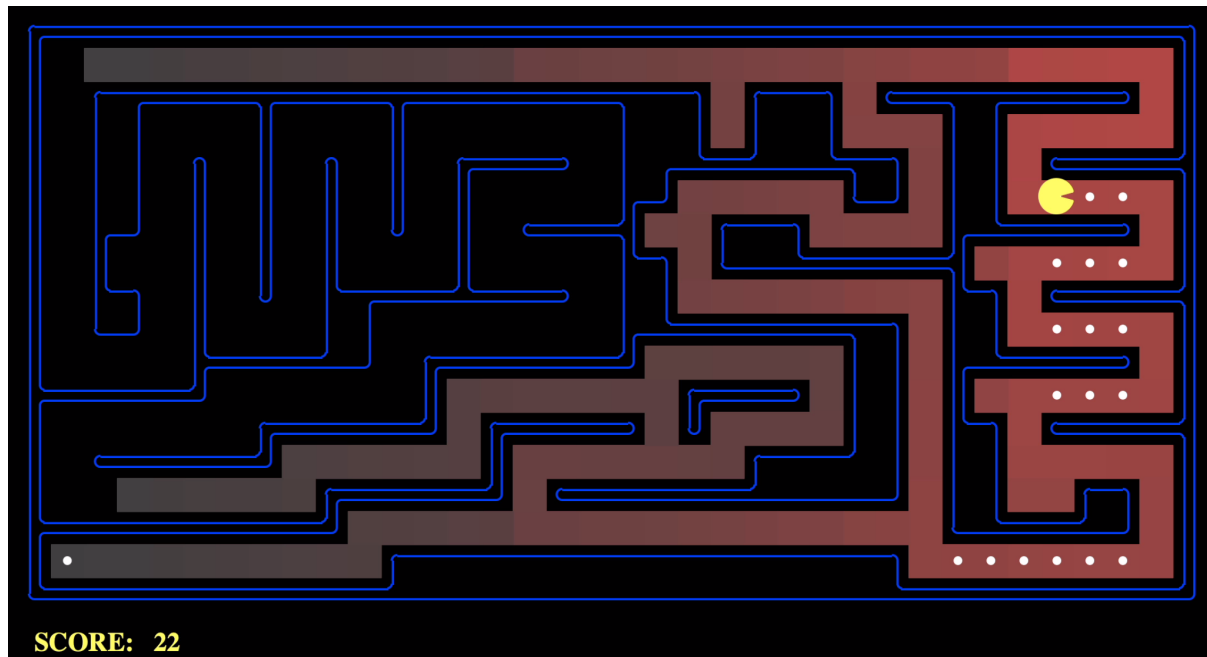


Figure 10 - Path to food using UCS on MediumDottedMaze

Here is a GIF of the result :

https://perso.esiee.fr/~caninr/E4S2_IA/UCS_MediumDottedMaze.gif

This time the result is interesting. As we can see, the UCS algorithm allows to choose a path where a maximum of food is present. In order to maximize the score, it is important to go through the cells where food is present. This is how we obtain a final cost of 1 and such a high score.

Medium Scary Maze

```
$ python3 pacman_AIC.py -l mediumScaryMaze -p StayWestSearchAgent
--frameTime 0
```

```
Path found with total cost of 68719479864 in 0.0 seconds
Search nodes expanded: 97
Pacman emerges victorious! Score: 418
Average Score: 418.0
Scores:         418.0
Win Rate:       1/1 (1.00)
Record:         Win
```

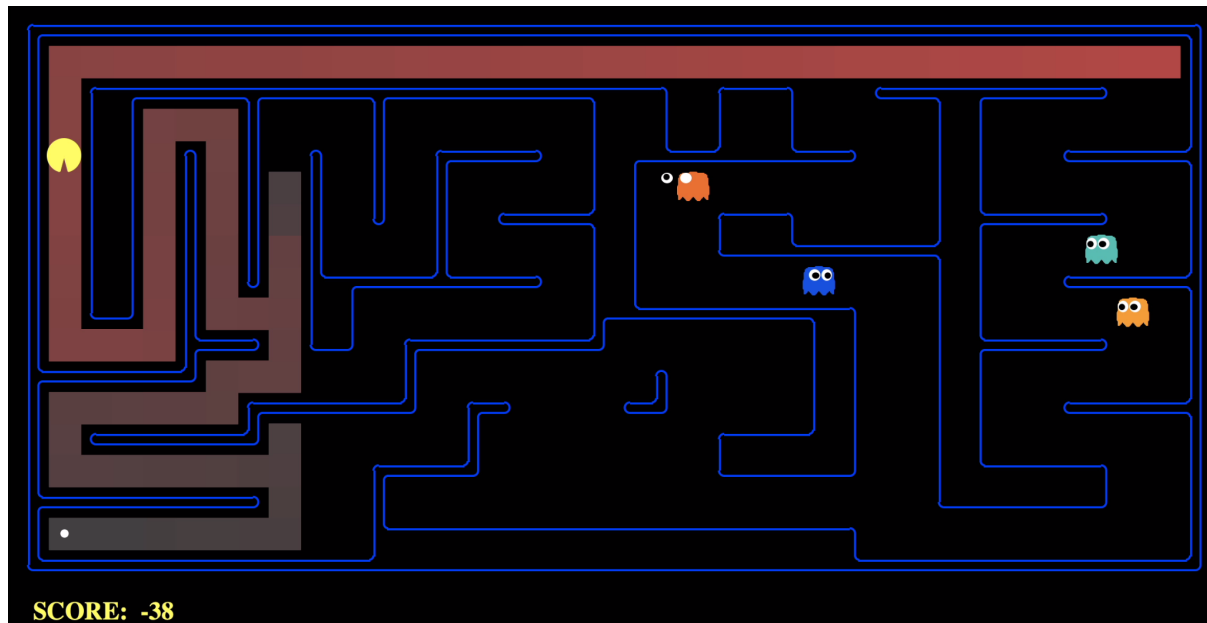


Figure 11 - Path to food using UCS on MediumScaryMaze

Here is a GIF of the result :

https://perso.esiee.fr/~caninr/E4S2_IA/UCS_MediumScaryMaze.gif.

As we can see again, the shortest paths in terms of distance are blocked by ghosts. Here, taking the longest path in terms of distance maximizes the score. In conclusion, UCS allows to privilege the cells with the lowest costs and to avoid the cells with too high costs (as for example those of the ghosts). It is more interesting to use than BFS when adding extra features to the game.

7 - A* Search

A final exploration algorithm that we will look at is the A* Search. It consists in an "improvement" of the UCS by introducing a heuristic function of cost calculation. To be clearer, while UCS prioritizes the node with the lowest cost, A* Search allows to take into account the heuristic cost. A* Search is an informed search and makes sense of the exploration. It tries to get closer to the goal rather than simply jumping from one node to the next like UCS.

There are different heuristic functions that can be used: null heuristic (which is a simple UCS), manhattan heuristic (which uses the Manhattan distance)...

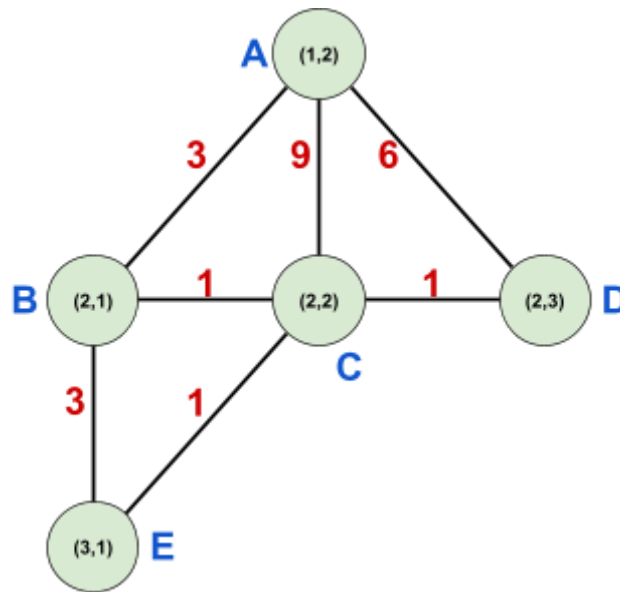


Figure 12 - Simple weighted graph

Let's apply the A* Search algorithm on the above graph, using heuristic function the Euclidean distance. We wish to find a path between A and E, as short as possible. In our example, as our goal is at (3, 1), this heuristic is :

$$H(x, y) = \sqrt{(3 - x)^2 + (1 - y)^2}$$

Node	h
A	2.24
B	1
C	1.41
D	2.24
E	0

N° of loop	Explored	Frontier
1	A	B(3+1), D(8.24), C(10.41)
2	A, B	C(5.41), E(6), D(8.24)
3	A, B, C	E(5), D(7.24)
4	A, B, C, E	D(7.24)

Thus, the shortest path found by A* Search is {A, B, C, E} with a cost of 5.

1 -

Now that we have seen an example of application of this algorithm, we will implement it in Python. To do so, we will add two heuristic functions : null heuristic (which corresponds to UCS) and manhattan heuristic. We update our Priority Queue using the sum of the cost of the node, the cost of its predecessor and the heuristic value of this state of the node. For more details, please refer to the code, which has been carefully commented.

Null heuristic : $(x, y) \rightarrow 0$

Manhattan heuristic : $(x, y) \rightarrow |a - x| + |b - y|$, with (a, b) the state of the goal

2 -

Null Heuristic

```
$ python3 pacman_AIC.py -l bigMaze -z .5 -p SearchAgent -a
fn=astar,heuristic=nullHeuristic

[SearchAgent] using function astar and heuristic nullHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 619
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:          300.0
Win Rate:        1/1 (1.00)
Record:          Win
```

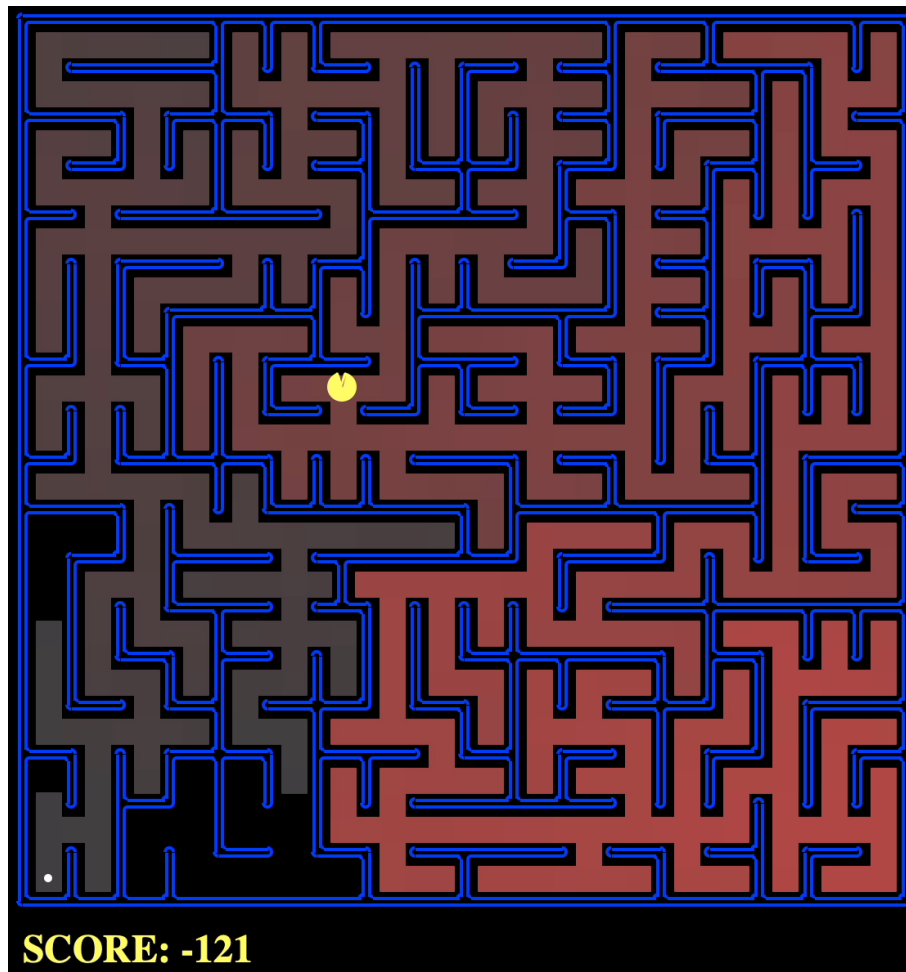



Figure 13 - Path to food using A* Search with Null heuristic on BigMaze

Here is a GIF of the result : https://perso.esiee.fr/~caninr/E4S2_IA/A_star_null_BigMaze.gif

Manhattan Heuristic

```
$ python3 pacman_AIC.py -l bigMaze -z .5 -p SearchAgent -a
fn=astar,heuristic=manhattanHeuristic --frameTime 0

[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.0 seconds
Search nodes expanded: 468 n619n
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:          300.0
Win Rate:        1/1 (1.00)
Record:          Win
```

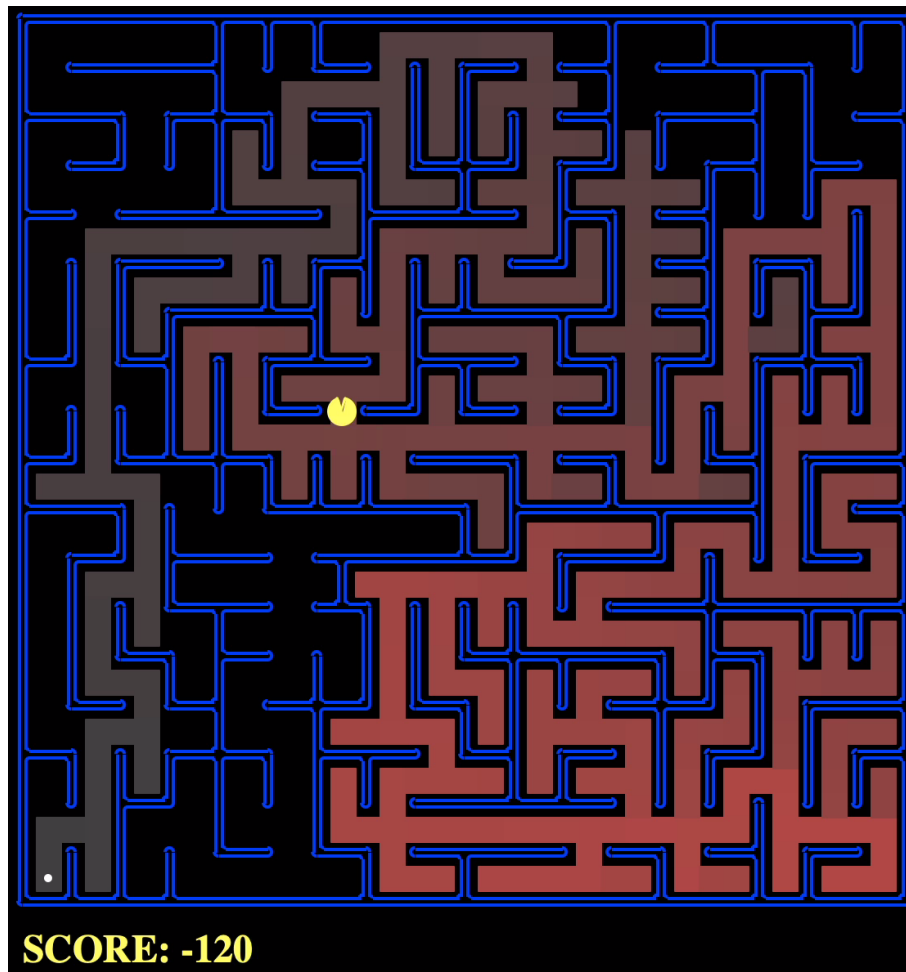


Figure 14 - Path to food using A Search with Manhattan heuristic on BigMaze*

Here is a GIF of the result :

https://perso.esiee.fr/~caninr/E4S2_IA/A_star_manhattan_BigMaze.gif.

As can be seen, the two explorations give a similar result. The path taken, the cost, and the score obtained are identical between the two trials. However, we notice that using the manhattan heuristic function allowed us to reduce the number of explored nodes by almost 24%.

For a maze of this size, we do not see any real difference in computation time, but the more nodes to explore, the more interesting it will be to use the Manhattan heuristic function. This is simply because the algorithm will favor the nodes "closest" to the goal rather than simply the nodes with a low cost. This avoids going in the opposite direction of the goal.

8 - Finding all the Corners

We will now change the type of exploration we are going to perform. The goal is no longer to get to a certain point on the map but to go through each of the four corners.

1 -

In order to make our algorithm work properly, we need to change the way the goal is validated. In fact, the goal should only be indicated when the four corners have been crossed. To do this, we will store the list of visited corners with the state of each node. We will then have to modify different functions.

For `getStartState`, we simply return a tuple containing the starting state and an empty tuple that will later be used to store the corners visited. A node previously had the format (state, action, cost). Now we have a structure ((state, Corners Visited), action, cost).

For the `isGoalState` function, we can no longer simply check that the current state is the location to be reached. In reality, the goal will be achieved when all four corners have been visited, i.e. the list of visited corners has the same length as the list of corners to be visited (`self.corners`).

For the `getSuccessors` function, we use the code of the `PositionSearchProblem` class. We have to add some modifications. Indeed, it is important to modify the list of visited corners in order to update it if the position on which we are is a corner. To do this, when the position in question is a corner and has not already been visited, it is added to the list of visited corners.

2 -

Tiny Maze

```
$ python3 pacman_AIC.py -l tinyCorners -p SearchAgent -a
fn=bfs,prob=CornersProblem

[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 28 in 0.0 seconds
Search nodes expanded: 412
Pacman emerges victorious! Score: 512
Average Score: 512.0
Scores:         512.0
Win Rate:       1/1 (1.00)
Record:         Win
```

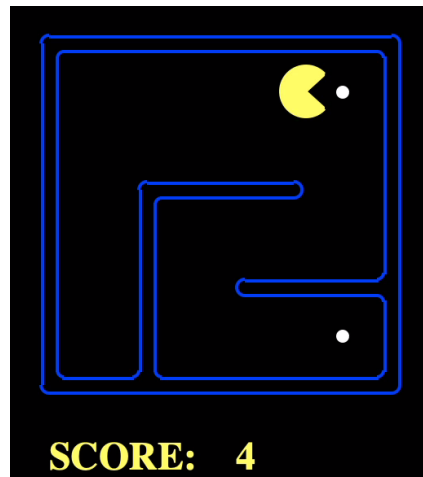


Figure 15 - Corners Problem using BFS on Tiny Maze

Here is a GIF of the result :

https://perso.esiee.fr/~caninr/E4S2_IA/CornersProblem_TinyMaze.gif.

As we can see, the BFS algorithm that we implemented for the previous problems works very well for our corner problem and this without having touched its code. We find a solution that seems to be a shorter path. Nevertheless, we had to expand 412 nodes in order to find a solution, which is relatively high for such a small maze.

Medium Maze

```
$ python3 pacman_AIC.py -l mediumCorners -p SearchAgent -a
fn=bfs,prob=CornersProblem
```

```
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.3 seconds
Search nodes expanded: 2381
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:         434.0
Win Rate:       1/1 (1.00)
Record:         Win
```

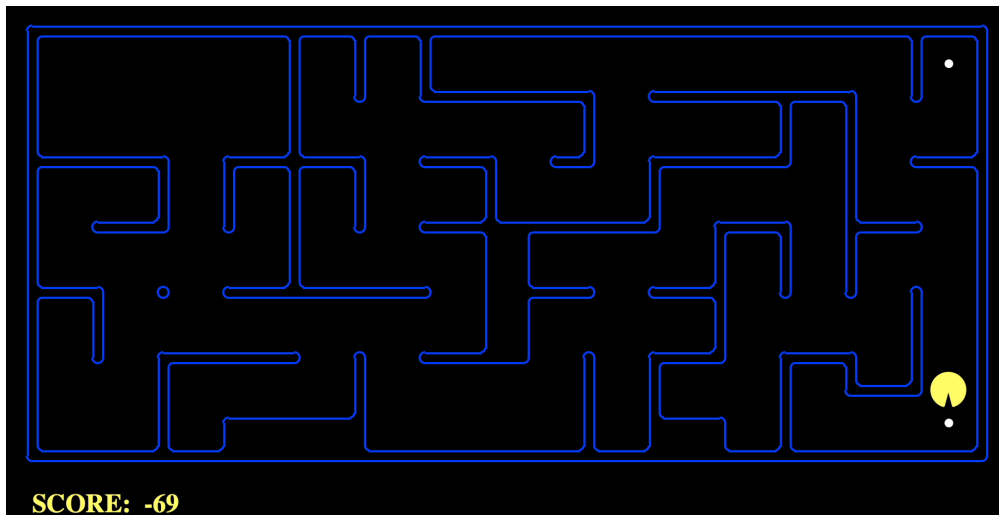


Figure 16 - Corners Problem using BFS on Medium Maze

Here is a GIF of the result :

https://perso.esiee.fr/~caninr/E4S2_IA/CornersProblem_MediumMaze.gif.

Once again, a path is found without problems. And we notice once again that the number of explored nodes is very consequent. Moreover, the computation time is for the first time since the beginning of this tp higher than 0.0s. Yet the map is not that big. The time complexity of this algorithm thus seems very questionable.

9 - Corners Problem : Heuristic

As explained just above, the code works but the complexity seems horrible. If we take a very large maze, the execution time would definitely be tremendous. So we will again use a heuristic function to eliminate the less interesting nodes. A heuristic function helps to determine whether a path is closer to the goal than another.

We can no longer simply calculate the distance from an observed node to the goal since we now have 4 points to reach in order to complete the goal. We tried different functions but they were mostly inconsistent, i.e. they gave paths of different length from a simple UCS. For example, we tried to return the sum of the distances to each of the unvisited corners or the maximum distance among these same corners.

We finally arrived at a heuristic function that works and is consistent. We compute the minimum distance to cover all the not yet visited corners. To do this, we look for the nearest unvisited corner using the Maze distance function. It is a distance calculation function that is present at the end of the Python file. It allows to calculate the distance from one point to another by taking the walls of the maze. To use it, you have to give the gameState as input to the function. To get it, we will simply use startingGameState, which we added as an attribute of the CornersProblem class.

Then, from this closest corner, we search for the next closest unvisited corner. And so on until we have visited all the unvisited corners.

2 -

AStarCornersAgent

```
$ python3 pacman_AIC.py -l mediumCorners -p AStarCornersAgent -z 0.5

Path found with total cost of 106 in 0.8 seconds
Search nodes expanded: 109
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:         434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

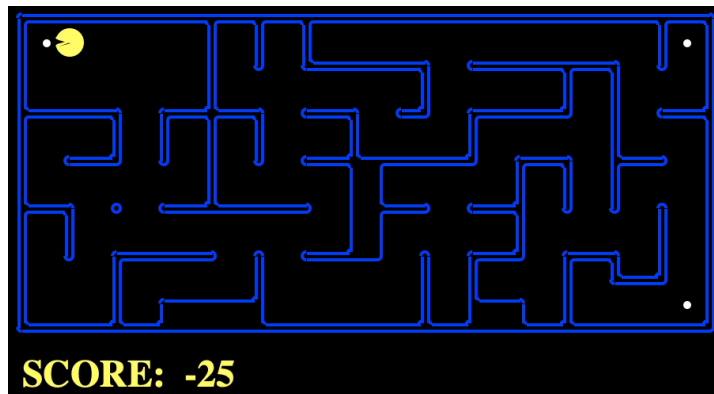


Figure 17 - Corners Problem using A* on Medium Maze

Here is a GIF of the result :

https://perso.esiee.fr/~caninr/E4S2_IA/AStarCornersAgent_MediumMaze.gif.

We can see that the number of expanded nodes has drastically decreased using our heuristic function. It is well consistent because we obtain the same score (434) and cost (106) as when we simply use UCS.

Nevertheless, we notice that the computation time is higher than 0.0s, it seems that the MazeDistance function is quite greedy. We could try to improve it to have a faster execution time. But for the moment, the goal is reached, we have a finite and consistent heuristic function that allows us to greatly reduce the number of explored nodes (95.4% decrease of the number of expanded nodes).

10 - Eating All the Dots

Now we want to make Pacman eat all the points on the map. We realize that our A star function does not work here. Indeed, we use a dictionary to find the predecessors and reconstruct the path. However, the FoodSearchProblem class uses food variables which are not easily serialized. Thus, the algorithm runs ad infinitum while trying to reconstruct the path.

We had to go back to our old algorithm (the one before using the dictionary). To use it, we have to specify `fn=FoodaStarSearch`. In this modified version of the algorithm, we store the path in each node directly. Indeed, action is this time the list of actions to be performed in order to reach the node in question.

```
$ python3 pacman_AIC.py -l testSearch -p SearchAgent -a
fn=FoodaStarSearch,prob=FoodSearchProblem,heuristic=foodHeuristic

[SearchAgent] using function FoodaStarSearch and heuristic foodHeuristic
[SearchAgent] using problem type FoodSearchProblem
Path found with total cost of 7 in 0.0 seconds
Search nodes expanded: 15
Pacman emerges victorious! Score: 513
Average Score: 513.0
Scores:          513.0
Win Rate:        1/1 (1.00)
Record:          Win
```

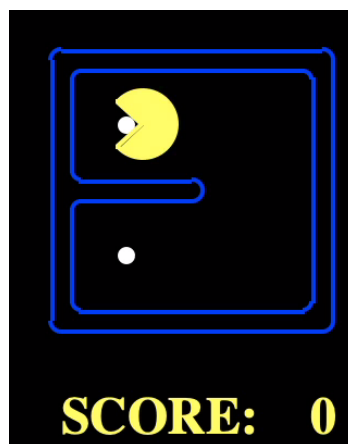


Figure 18 - Food Problem using A* on Mini Maze

Here is a GIF version of the result :

https://perso.esiee.fr/~caninr/E4S2_IA/FoodAstar_MiniMaze.gif.

As we can see, the FoodaStarSearch algorithm works perfectly and we get the requested cost of 7.

The problem posed seems quite similar to the previous one with the corners. We will proceed in exactly the same way to calculate the heuristic value for each node.

The function will return the minimum distance that must be traveled to take all the uneaten food.

2 -

```
$ python3 pacman_AIC.py -l trickySearch -p SearchAgent -a
fn=FoodaStarSearch,prob=FoodSearchProblem,heuristic=foodHeuristic

[SearchAgent] using function FoodaStarSearch and heuristic foodHeuristic
[SearchAgent] using problem type FoodSearchProblem
Path found with total cost of 62 in 1.1 seconds
Search nodes expanded: 65
Pacman emerges victorious! Score: 568
Average Score: 568.0
Scores:      568.0
Win Rate:    1/1 (1.00)
Record:      Win
```

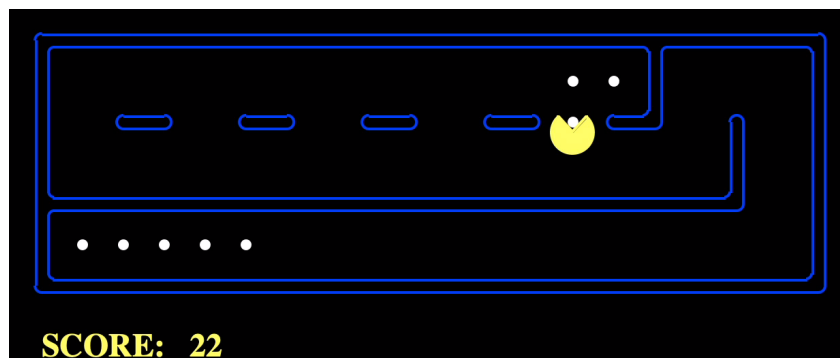


Figure 19 - Food Problem using A* on Tricky Maze

Here is a GIF version of the result :

https://perso.esiee.fr/~caninr/E4S2_IA/FoodAStar_TrickyMaze.gif.

As we can see, our function is very powerful. The path it finds seems to be the best one, it does not make unnecessary moves. We really expand few nodes for the size of the labyrinth. The function discriminates very well between good and bad paths. It seems to be efficient.

11 - Suboptimal Search

1 -

We will now look for a more local solution to our problem. Indeed, we simply want Pacman to eat the food as close to him as possible. To do this, we need to complete the `findPathToClosestDot` function.

We can simply reuse one of the search algorithms such as BFS on a problem with the objective of eating the closest food.

2 -

```
$ python3 pacman_AIC.py -l bigSearch -p ClosestDotSearchAgent -z .5  
--frameTime 0
```

```
[SearchAgent] using function depthFirstSearch  
[SearchAgent] using problem type PositionSearchProblem  
Path found with cost 414.  
Pacman emerges victorious! Score: 2296  
Average Score: 2296.0  
Scores:          2296.0  
Win Rate:       1/1 (1.00)  
Record:         Win
```

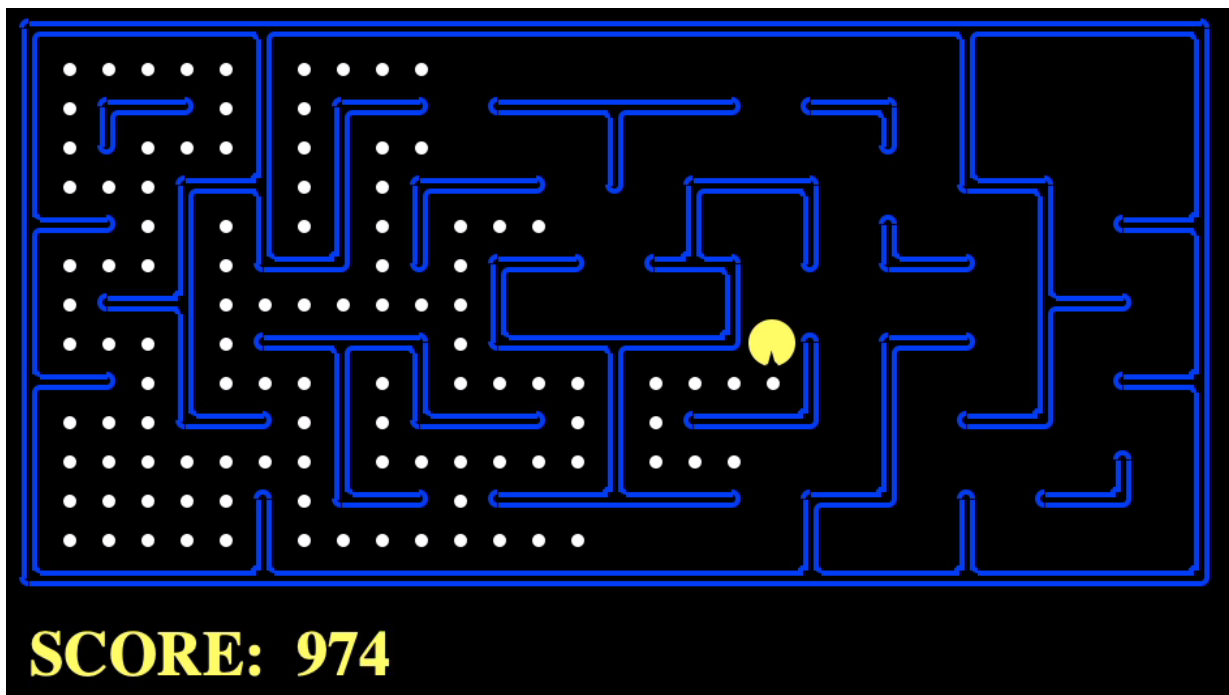


Figure 20 - Closest Dot Search

Here is a GIF version of the result :

https://perso.esiee.fr/~caninr/E4S2_IA/ClosestDotSearchAgent.gif.

Thus, this algorithm allows to eat all the food points. However, it is not completely optimal because it sometimes leaves behind food points that it comes back to get later. Nevertheless, locally, the algorithm seems to be consistent and efficient.

TP2

In the following second lab, there are ghosts trying to prevent Pacman from eating food. Hence, Pacman wouldn't only have to care about how to access food, but also to take into account the danger of encountering a ghost.

1- Tests

A zip file was provided onto Blackboard. By executing `python3_pacman AIC.py`, a game is launched as follow :

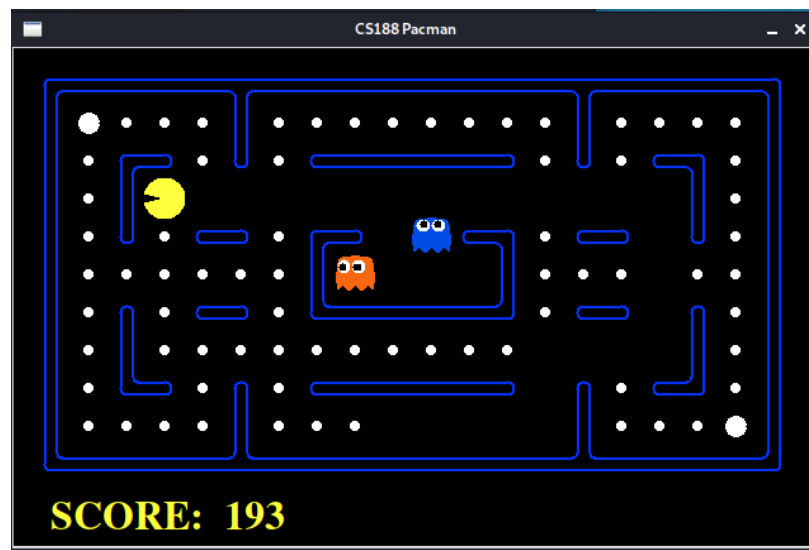


figure 21 - Pacman game without using any particular agent

During this game, Pacman didn't care about the ghosts around him and only tried to reach the food around him. The purpose of the following exercises will be to find a method which respects the trade-off between eating dots and escaping from ghosts.

2 - Python Files

The provided zip file contained 13 .py files and a folder.

The layout folder contains the 11 layouts available all of different sizes and configurations.

The 4 files that we will focus on are the following one :

multiAgents.py: Defines the class for each Multi-agent search

pacman AIC.py: The file that we will execute to launch a game

game.py: Defines all the elements necessary to build a game, such as AgentState, Agent, Direction, and Grid.

util.py: Defines all the useful data structures for implementing search algorithms. We

It was important to take a look at the `pacman_AIC.py` and `game.py` files in order to have an idea of all the methods available for our implementation.

3 - Starting

The `multiAgents.py` file contains 5 classes for each of the Agents methods that we will have to implement. The first method we will try to implement is the `ReflexAgent`.

To test how it was initially working we executed the two commands : `python3 pacman_AIC.py -p ReflexAgent -l testClassic` and `python3 pacman_AIC.py -p ReflexAgent`.

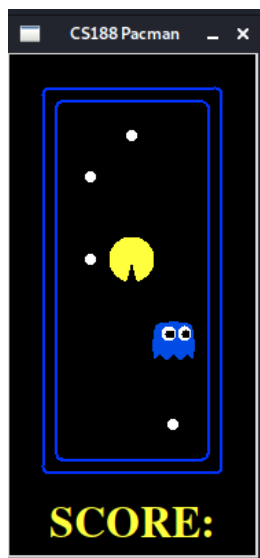


figure 22 - Reflex Agent on testClassic layout

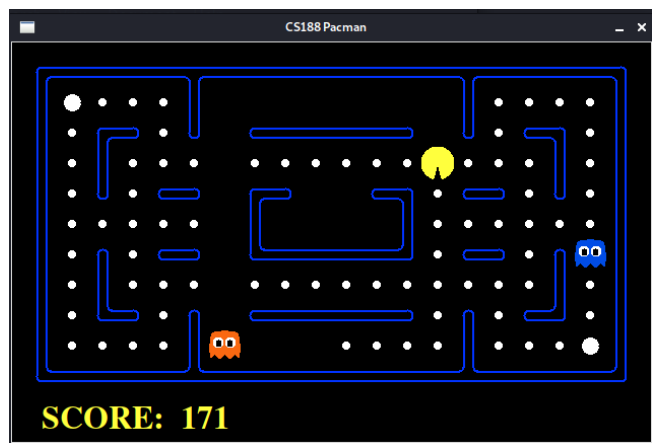


figure 23 -Reflex Agent on mediumSizelayout

Pacman seemed confused during this simulation, he was sometimes going back and forth between 2 same states.

To understand this behavior, we looked at the code of `ReflexAgent` in `multiAgents.py`, here is described how it works :

For a given state, `ReflexAgent` looks for all available actions. For each of these actions, it looks for the global score that Pacman would have if it decides to go there thanks to the function `getScore()`. This global score is only equivalent to the total number of eaten dots and doesn't take into consideration the position of ghosts. Then, `ReflexAgents` keep in mind the actions with the best score, and randomly choose one.

Hence, once Pacman is stuck in a place where there are no dots around, he happens to stagnate between 2 states or stops.

4 - Reflex Agents

1) Improve the ReflexAgent in multiAgents.py

The purpose of this part is to improve the evaluation function of the ReflexAgents. One of the things that has been previously blamed, was to not take into account the presence of Ghosts.

In the ReflexAgents class, it was suggested to use the following methods : `manhattanDistance`, `generatePacmanSuccessor(action)`, `getPacmanPosition()`, `getFood()`, `getGhostStates()`

Hence, we wanted to create a score using the distance to a Ghost and a the distance to a Food dot. The use of Manhattan Distance was necessary since only 4 directions are available in the Pacman game.

Since we want to keep Pacman away from ghost, the score should increase as the `distance_to_ghost` increases.

On the contrary, since Pacman is trying to get close to food, the score should decrease as the `distance_to_food` increases.

Hence, we started to use the following score calculation :

`score = distance_to_ghost - distance_to_food`

However, as we did our first tests, we realized that Pacman wasn't able to take decisions quickly ; hence, we decided to test some multiplication factors in order to prioritize the criteria "staying away from a ghost" or "eating food".

The best trade-off that we observed was for the following formula :

`score = distance_to_ghost2 - distance_to_food`**

This formula isn't optimized enough yet, but it will be improved in the last part 8 - Evaluation Function.

2) Test your code

We tested our code on the `testClassic`. It performed pretty well although it didn't always lead to a win.

```

(kali㉿kali)-[~/Documents/artificial_intelligence/tp2/multiagent_AIC]
$ python3 pacman_AIC.py -p ReflexAgent -l testClassic
Pacman emerges victorious! Score: 452
Average Score: 452.0
Scores:      452.0
Win Rate:    1/1 (1.00)
Record:      Win

```

Figure 24 - `python3 pacman_AIC.py -p ReflexAgent -l testClassic`

Then, we tested our code on the mediumSize layout. Pacman wasn't smart enough to win ; but it indeed takes into consideration the presence of food and ghosts as we expected him to do.

```

(kali㉿kali)-[~/Documents/artificial_intelligence/tp2/multiagent_AIC]
$ python3 pacman_AIC.py -p ReflexAgent -k 1
Pacman died! Score: -360
Average Score: -360.0
Scores:      -360.0
Win Rate:    0/1 (0.00)
Record:      Loss

```

Figure 25 - MediumSize layout with 1 ghost

```

(kali㉿kali)-[~/Documents/artificial_intelligence/tp2/multiagent_AIC]
$ python3 pacman_AIC.py -p ReflexAgent -k 2
Pacman died! Score: -163
Average Score: -163.0
Scores:      -163.0
Win Rate:    0/1 (0.00)
Record:      Loss

```

Figure 26 - MediumSize layout with 2 ghosts

As explained previously, it is not a big deal if tests aren't successful since the evaluation function will be improved in the last part of the lab. We noticed that it performed better in configurations with 2 ghosts than with only one.



Figure 27 - Pacman almost winning with 2 ghosts

To understand more the performance of our current evaluation system, we executed 6 different types of tests on each layout.

The 6 tests we performed corresponded to the following situations :

- one random ghost (*python3 pacman_AIC.py -p ReflexAgent -n 30 -q -k 1*)
- one random ghost with fixed seed (*python3 pacman_AIC.py -p ReflexAgent -n 30 -q -f -k 1*)
- one non random ghost (*python3 pacman_AIC.py -p ReflexAgent -n 30 -q -g DirectionalGhost -k 1*)
- two random ghost (*python3 pacman_AIC.py -p ReflexAgent -n 30 -q -k 2*)
- two random ghost with fixed seed *python3 pacman_AIC.py -p ReflexAgent -n 30 -q -f -k 2*
- two non random ghost (*python3 pacman_AIC.py -p ReflexAgent -n 30 -q -g DirectionalGhost -k 2*)

For each test, we performed it 30 times by disabling the graphic visualization.

There are 11 layouts which are : capsuleClassic, contestClassic, mediumClassic, minimaxClassic, openClassic, originalClassic, powerClassic, smallClassic, testClassic, trappedClassic, trickyClassic

However, we only tested 2 of these layouts namely mediumClassic and smallClassic.

Test on medium classic

<pre>Average Score: -373.7 Scores: -400.0, -439.0 -437.0, -368.0, -439.0, -439.0 Win Rate: 0/30 (0.00)</pre> <p>test 1</p>	<pre>Average Score: -464.6666666666667 Scores: -520.0, -468.0, -426.0, -355.0, -1586.0, -455.0, -426.0 Win Rate: 0/30 (0.00)</pre> <p>test 2</p>	<pre>Average Score: -459.6333333333333 Scores: -494.0, -563.0, -425.0, -617.0, -482.0, -549.0, -506.0, -6.0 Win Rate: 0/30 (0.00)</pre> <p>test 3</p>
<pre>Average Score: -242.56666666666666 Scores: -97.0, -280.0, -39.0, -127.0, 48.0, -174.0, -331.0, -274.0 Win Rate: 0/30 (0.00)</pre> <p>test 4</p>	<pre>Average Score: -252.2 Scores: -401.0, -111.0, 0.0, -157.0, -365.0, -356.0 Win Rate: 0/30 (0.00)</pre> <p>test 5</p>	<pre>Average Score: -219.66666666666666 Scores: -308.0, -293.0, -79.0, 0.0, -270.0, 53.0, -307.0, -424.0, -307.0 Win Rate: 0/30 (0.00)</pre> <p>test 6</p>

In medium Classic, Reflex agent performs better when there are two ghosts. The best average score is when there are two non random ghosts.

Test on smallClassic

<pre>Average Score: -298.5 Scores: -243.0, -428.0, -378.0, -396.0, -371.0, 506.0 Win Rate: 3/30 (0.10)</pre> <p>test 1</p>	<pre>Average Score: -309.7 Scores: -385.0, 773.0, -505.0, -374.0, -241.0, 70.0 Win Rate: 4/30 (0.13)</pre> <p>test 2</p>	<pre>Average Score: -426.6 Scores: -428.0, -437.0, -428.0, -437.0, -428.0, -437.0 Win Rate: 0/30 (0.00)</pre> <p>test 3</p>
--	--	---

Average Score: -346.8666666666667 Scores: -238.0, -224.0, -431.0, -448.0, -292.0, -449.0, -233.0, -2.0 Win Rate: 0/30 (0.00)	Average Score: -305.3 Scores: -535.0, -292.0, -244.0, -406.0, -416.0, -433.0 Win Rate: 1/30 (0.03)	Average Score: -321.3333333333333 Scores: -356.0, -356.0, -360.0, -320.0, -330.0, -365.0, -357.0, -0.0 Win Rate: 0/30 (0.00)
test 4	test 5	test 6

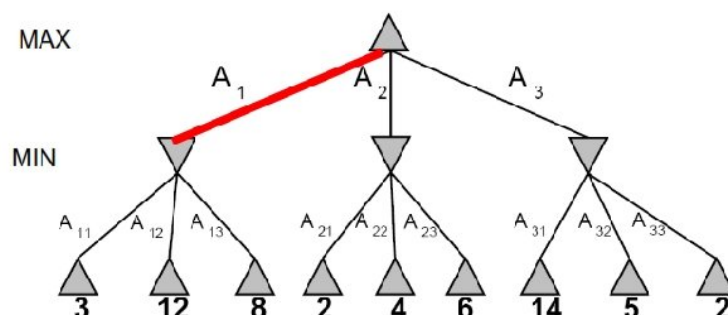
In smallClassic, ReflexAgents sometimes enable to win when there is 1 random ghost. Contrary to mediumClassic it performs worse when there are 2 ghosts ; however, this observation isn't surprising since it is harder to run away from ghosts in a small Maze.

5 - Minimax

1. Implementation

The minimax algorithm consists in using the depth-first search algorithm to select either the minimum, or the maximum of each stage of the tree. The purpose of this algorithm is to predict the score we would get for each different possibility available after a given number of rounds, where player minimizer and player maximizer each play one after the other.

Let's take the following example which was explained during the lectures :



<https://slidetodoc.com/adversarial-search-game-playing-chapter-6-outline-games/>

Calling minimax on A leads to taking the maximum between A1, A2 and A3. Thereafter, since there is a MIN layer, A1, A2 and A3 each look for the minimum among their children. Hence, A1 would take 3, A2 would take 2 and A3 would take 2. The maximum number between {3,2,2} is 3. That is why A will choose the option A1. To implement this algorithm, it was advised to implement 3 functions :

- MAXValue, to take the maximum value
- MINValue, to take the minimum value
- MinMax, to do the transition between calling MAXValue or MINValue

The implementation of this algorithm was done in a recursive way since just like we explained for the example below, to calculate the result of minimax in the state A we had to call MinMax on A1, A2, and A3.

The implementation of this algorithm was inspired by the slide 84 of the first lecture and the explanations given in class. In this implementation, the testing of terminal states

(winning, losing, final depth) was done directly in the MAXValue and MINValue functions. However, we decided to do these tests in our function MinMax since it takes the role of the leader by indicating which function we should call.

To get more details about the implementation of this code, please take a look at the comments in the code.

2. Test your code

According to the assignment, it is normal if Pacman loses the game in some tests.

We tested our implementation by using the command `python3 pacman_AIC.py -p MinimaxAgent -a depth=5 -l smallClassic`



Figure 28 - Pacman playing with MiniMaxAgent on smallClassic layout

Pacman was doing a good job ; he was sometimes almost winning but always ended up being eaten by a ghost. Sometimes, we saw him stopping ; we thought that it was because he was taking time to compute all the available possibilities for a depth of 5. Our implementation doesn't take advantage of the scared time of the ghosts ; we should adapt it in the last part of this lab.

We tried to use different depths in order to see how it affected the algorithm. The best values were 5, or 3. A depth of 5 takes better decisions but has more chance to die than a depth of 3 which takes decisions faster. The score still remains negative though because Pacman always get scared and stand still, without moving, and gets eaten by a ghost.

```
(kali㉿kali)-[~/Documents/artificial_intelligence/tp2/multiagent_AIC]
$ python3 pacman AIC.py -p MinimaxAgent -a depth=3 -l smallClassic

Pacman died! Score: -147
Average Score: -147.0
Scores: -147.0
Win Rate: 0/1 (0.00)
Record: Loss
```

Figure 29 - Score obtained with Minimax and a depth of 3

6 - AlphaBeta algorithm

The alpha Beta algorithm is very similar to the MiniMax algorithm. The only difference between these two algorithms is that AlphaBeta algorithm uses two values alpha, and beta, in order to get rid of irrelevant subtrees and fasten the search.

According to the pseudo code provided in the first lecture, as we call MAXValue, the algorithm updates the value of alpha and stops if it finds a value greater than beta. On the contrary, the MINValue stops if it finds a value lower than alpha and updates the value of beta.

To implement this algorithm, we only had to take the code of the previous exercise MinMax and to adapt it according to the specifications detailed in the paragraph above.

To test our code, we use the following command :

```
python3 pacman AIC.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

```
(kali㉿kali)-[~/Documents/artificial_intelligence/tp2/multiagent_AIC]
$ python3 pacman AIC.py -p AlphaBetaAgent -a depth=5 -l mediumClassic

Pacman died! Score: 294
Average Score: 294.0
Scores: 294.0
Win Rate: 0/1 (0.00)
Record: Loss
```

Figure 30 - result of the AlphaBetaAgent on mediumClassic

Pacman is faster and more efficient than with the MiniMax algorithm ; however, it still end up being eaten by a ghost.

7 - Expectimax

The Expectimax algorithm is very similar to the MiniMax algorithm ; however, it considers the fact that ghosts won't always try to act as a minimizer. According to Expectimax, ghosts will sometimes behave like chance nodes.

According to the explanations provided on this website <https://www.geeksforgeeks.org/expectimax-algorithm-in-game-theory/>, this behavior is equivalent to computing the mean of the children and to take the minimum among these average means. This is the principle that we used in order to implement our Expectimax

algorithm.

The only modifications we had to do were on the MINValue function that we renamed “expected”. In this function, we computed the mean of each child in order to take the minimum among these.

```
(kali@kali)-[~/Documents/artificial_intelligence/tp2/multiagent_AIC]
$ python3 pacman_AIC.py -p ExpectimaxAgent -a depth=5 -l mediumClassic

Pacman died! Score: -175
Average Score: -175.0
Scores: -175.0
Win Rate: 0/1 (0.00)
Record: Loss
```

Figure 31 - result of the ExpectiMaxAgent on mediumClassic

Pacman seemed to play slightly better ; however, he was too slow to make decisions and always ended up being eaten by a ghost. This observation is the result of a bad Evaluation function ; that's why the following last part will try to improve it.

8- Evaluation function

The main purpose of this part was to force Pacman to take decisions ; during the previous part we saw that he was either stagnating because he was too scared of ghosts, or he was either hesitating between 2 options and got eaten by a Ghost.

As a smart player , we will try to :

- run away from ghosts, in extend, to take the opposite direction when they start getting close to us
- go into the direction where food is
- take into consideration when Ghost are scared/inoffensive
- take advantage of big distance from the ghost to try bold moves

1) Run away from ghosts

In order to run away from ghosts, we decided to put a very bad score on states where the `closest_distance_to_ghost` was too small. We started by trying a distance smaller than 5, decreased it, and saw that 2 was a good tradeoff.

2) Reaching for food

During the first part, we say that the score should decrease as the distance to the `closest_food` increases. Subtracting `closest_food_distance` to the score was a good idea, however, another option would have been to say that the score should be inversely proportional to the `closest_food_distance`.

We tried to use the scoring function $1/\text{closest_food_distance} + \text{closest_ghost_distance}$ and saw that Pacman was performing a lot better.

3) Scared times of ghosts

In order to take into consideration the fact that ghosts may be inoffensive, we used a boolean which equals **True** when ghosts are scared. When this boolean is true, we returned a score where the distance of the ghosts wasn't taken into consideration at all :

$$1000000 * 1/\text{closest_food_distance} + 1/\text{closest_food_distance}$$

However, it didn't have any impact on Pacman's behavior. We checked that the boolean effectively valued True when the ghosts were scared by using a print and observing the terminal : it was effectively the case.

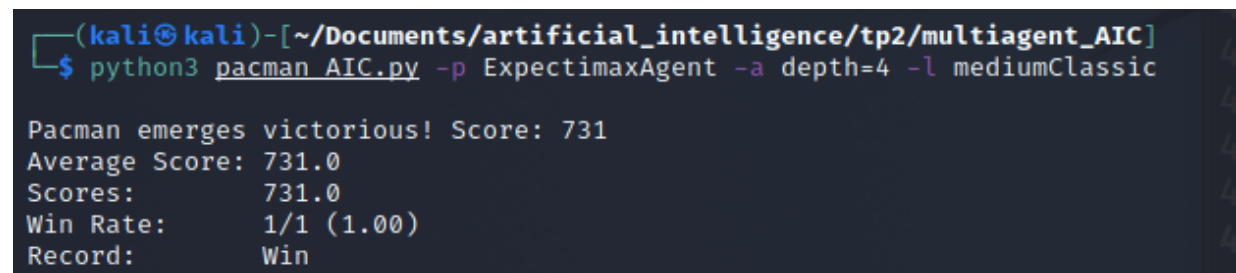
To find a solution to this issue, we remembered that the behavior we are trying to reach when ghosts are scared is the same as when there were no ghosts.

Back then, when there were no ghosts, we used `scoreEvaluationFunction(gameState)` to compute the score. Pacman was however stuck when ghosts were scared and that no food was near 2 moves. Hence, we decided to add $1/\text{closest_distance_food}$ to this score.

When ghosts are scared, score is computed by :

$$\text{scoreEvaluationFunction(gameState)} + 1/\text{closest_distance_food}$$

We tested this evaluation function on `ExpectimaxAgent`, and Pacman won with a very good score.



```
(kali㉿kali)-[~/Documents/artificial_intelligence/tp2/multiagent_AIC]
$ python3 pacman AIC.py -p ExpectimaxAgent -a depth=4 -l mediumClassic

Pacman emerges victorious! Score: 731
Average Score: 731.0
Scores:       731.0
Win Rate:     1/1 (1.00)
Record:       Win
```

Figure 32 - Score obtained with the modifications done so far

4) Try bold moves

However, Pacman is sometimes too afraid of the ghosts and doesn't dare to try bold moves in order to reach food. We wanted to encourage him to run until food when ghosts were far away from him, by using the following tests :

```
"if closest_ghost_distance>5 and closest_food_distance<=3:
    return 100*1/closest_food_distance+closest_ghost_distance "
```

It didn't work at first, but we remembered the trick of using `scoreEvaluationFunction(currentGameState)` as we did for the previous part, we tried it by returning the following score :

1/closest_food_distance+closest_ghost_distance+10*scoreEvaluationFunction(currentGameState)

And it performed way better than previously ; as depicted in the picture below, we obtained a score of 1163 on our first test.

```
(kali㉿kali)-[~/Documents/artificial_intelligence/tp2/multiagent_AIC]
$ python3 pacman AIC.py -p ExpectimaxAgent -a depth=4 -l mediumClassic

Pacman emerges victorious! Score: 1163
Average Score: 1163.0
Scores:      1163.0
Win Rate:    1/1 (1.00)
Record:      Win
```

Figure 33 - Best score obtained thanks to a better evaluation function

5) Testing Expectimax

We performed the same tests as we previously did for ReflexAgent on mediumClassic.

<pre>Average Score: 310.46666666666664 Scores: 979.0, 824.0, -1214.0, -6 , -273.0, 830.0, 698.0, 490.0, 535.0, -6 Win Rate: 26/30 (0.87)</pre>	<pre>Average Score: 268.06666666666666 Scores: 980.0, 879.0, 1060.0 , 824.0, 1049.0, 503.0, 236.0, 650. Win Rate: 26/30 (0.87)</pre>	<pre>Average Score: 1272.3666666666666 Scores: 301.0, 1450.0, 1360.0, 1415 1465.0, 1100.0, 1177.0, 1421.0, 1445.0, 1 .0 Win Rate: 29/30 (0.97)</pre>
test 1	test 2	test 3
<pre>Average Score: 222.73333333333332 Scores: 198.0, 140.0, -30.0, 1 -215.0, 1335.0, -276.0, 105.0, 893.0 Win Rate: 8/30 (0.27)</pre>	<pre>Average Score: 244.26666666666668 Scores: -104.0, 818.0, -68.0 126.0, 860.0, 113.0, -108.0, 819.0 Win Rate: 7/30 (0.23)</pre>	<pre>Average Score: 253.1 Scores: -285.0, 58.0 305.0, -430.0, 1431.0, 434. Win Rate: 5/30 (0.17)</pre>
test 4	test 5	test 6

The results have been considerably improved for one ghost, and this, no matter if it is random or not. It performs even better on non random ghosts.

However, the majority of the games for two ghosts led to loss ; improvement is still possible.

Conclusion

Expectimax seems to be the best algorithm to play Pacman. The evaluation function has been well improved but could be enhanced even more by taking into account other parameters such as considering the capsules - which didn't seem to be considered as food by Pacman -, or putting a bad score for trapped isolated places in which Pacman often got eaten by the ghosts.

The Evaluation Function has been improved by looking at the results on the mediumClassic layout ; however, by testing it on other layouts, other relevant parameters could be suggested in order to improve the Evaluation Function.