INFO8006: Project 1 - Report

Théo Stassen - s150804
Hubar Julien - s152485

October 12, 2019

1 Problem Statement

a. Formalization of the game as a search problem

States	A state of the game is defined by the position of pacman in the grid (the orientation doesn't change anything) and the presence or not (a boolean value : 1 if present, 0 if not) of each foods in the grid. $s = (x, y, f_1, f_2, ..., f_{N-1}, f_N)$ if N is the initial number of food. The set of all possible states is so $M * (2^N)$ where M is the number of legal position (all except the walls) in the grid and N is the initial number of foods. The initial state is (5,5,1) for the small grid, (9,3,1,1,1,1,1,1,1,1,1,1,1,1,1,1) for the medium and (5,2,1,1,1,1) for the large
Legal actions	The set of legal actions is in general the direction the pacman can takes which depends on which are legal. If there are no walls next to pacman : $actions(s) = (left, right, up, down)$
Transition model	The transition model is "the new position and new situation of food = result of the current position and the taken direction for example $(1, 1, 1, 1) = result((2, 1, 1, 1), left)$
Goal test	The goal test is $if n = 0$ where n is the actual number of food in the grid.
Step cost	At each step the goal is to decrease the number of foods in order to increase the number of eaten food dots in the score and we want to do a path with the less step possible to minimize the number of time steps which inversely proportional to score. The step cost is so the distance between the previous and the next state, a constant 1 in the grid.

2 Implementation

a. DFS

The dfs.py implementation apply correctly the DFS algorithm (at each time the first node expanded is the leftest node of the graph). The problem is that the key of the state, which identify uniquely a state, is only based on the position of pacman. An each position visited is put in the closed nodes which cannot be visited again. So it cannot go two times in the same place. The configuration of the grids (medium and large) makes that it is impossible to eat all the foods without going two times in the same place. The code cannot find a solution. The idea to fix it is to allow the agent more actions. An idea is to suppress all the nodes in the closed nodes list each time the pacman is on a food and continue. It is equivalent to do a simple dfs research until we find a dot, and to begin a new dfs from this position. In fact, when a branch of the tree find a dot, there is no reason to go back up of this branch, so this node become conceptually the new root. And with this method, we will find a solution, because all the successive dfs will find a (probably sub optimal) solution. An equivalent method is to consider the key as not only the position but the position plus the presence (or not of the foods), as the search problem as been formalized. Each time a dot is eaten, the agent can go on all position already visited (because there are not anymore

already visited state) , and so it is also equivalent to begin a new dfs each time you eat a dot. It is this last version which was implemented.

b. Astar implementation

The implementation is done in the astar.py file by using a priority queue (implemented in util.py) as fringe. The key of the state is the same as in DFS. The algorithm is very similar to our modified version of DFS, with a priority queue instead of a queue.

c. Astar justification

We want that the path cost take account of the length of the path and the decreasing number of foods. So the path cost is the sum (from the initial state to the current state) of the number of food remaining at each state. The function g takes the previous value of the backward cost and the number of foods in the state we compute and send the new backward cost, simply the addition of the two arguments.

The admissible heuristic h(h) is defined as mean of the Manhattan distance between the current position and the position of the remaining foods. This h(n) guarantees the optimality because the goal is to eat food. For each position, it is at first glance more advantageous to go in a next position closer to the remaining dots. The walls arrangement can make the best path counter intuitive and that go to closest position will be a bad choice but in general (in mean) going in this direction is better.

In the beginning of the project, we implemented a different version of A*. In this version, the key of the state was only the position (so we must to remove some states from the closed list in the algorithm) and our algorithm was based on the graph-search algorithm seen in the practical lesson, which allow to pop nodes from the closed list and update nodes of the fringe if the new priority is better. The priority was not calculated in the same way, g(n) was the length of the path + a constant * the number of remaining dots. With the g used in the final A* the pacman was stuck in the dead-end (the priority was bigger in the dead-end than out of the dead-end). This method gives pretty good results with a really lower number of expanded nodes and computation time compared to the other A* but we didn't succeed to obtain a perfect score. The numerical results are presented at the end of the report. Another problem was that the solution was not general. The better result in the medium grid was not using the same coefficient than the better result in the large, for example. It is so we decide to change of key and algorithm.

d. Astar g(n) justification

If g(n) works alone, the completeness and optimality is preserved because it will guide us to the solution with the lowest backward cost. In any solution all the dots are eaten, so the solution with the lowest backward cost (and so the shortest path) gives us the better score, nowadays h(n). The admissible heuristic is used only to increase the research speed by going in priority in the most interesting zones of the grid. Using g(n) only is equivalent to a Uniform Cost algorithm which is theoretically optimal and complete, but slower in speed than a well tuned astar.

e. BFS implementation

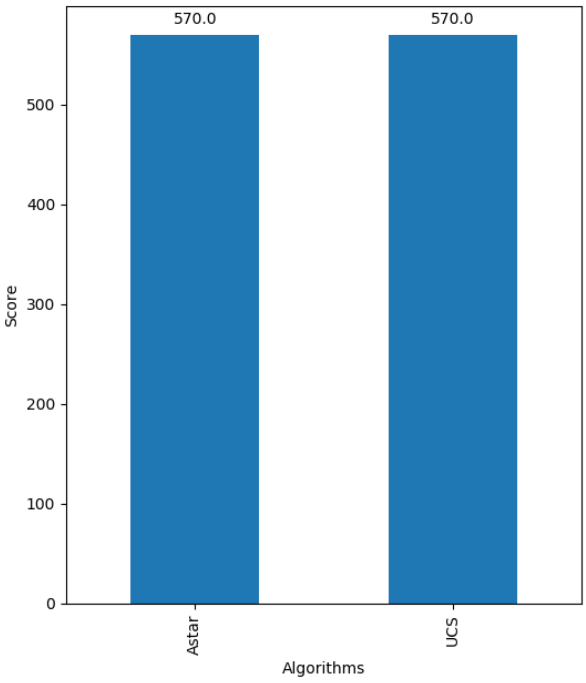
The implementation is done in bfs.py. The code is exactly the same as astar.py, except from the h(n) and g(n) static functions.

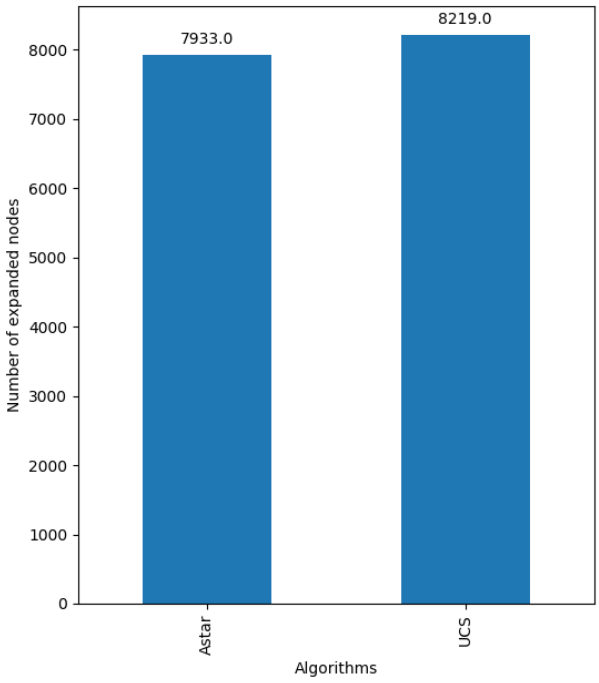
f. BFS justification

h(n) is the greedy part of astar, which gives a knowledge about were to go, the BFS algorithm doesn't know that, so h(n) is equal to 0 for all n. The g(n) is equal to the depth of the node

in the search tree (computed by add 1 to the previous backward cost at each step). With this cost and a priority queue, at each step the node expanded is the one with the lowest depth, which is equivalent to a FIFO queue and so the algorithm is equivalent of a BFS search.

3 Experiment 1





a. Differences between the results

The score obtains by A* is 570, which is after visual verification the optimal. The algorithm is correctly giving the optimal solution. The score of A* and A* with h(n) = 0 for all n are the same. The number of expanded nodes is a little lower for A*.

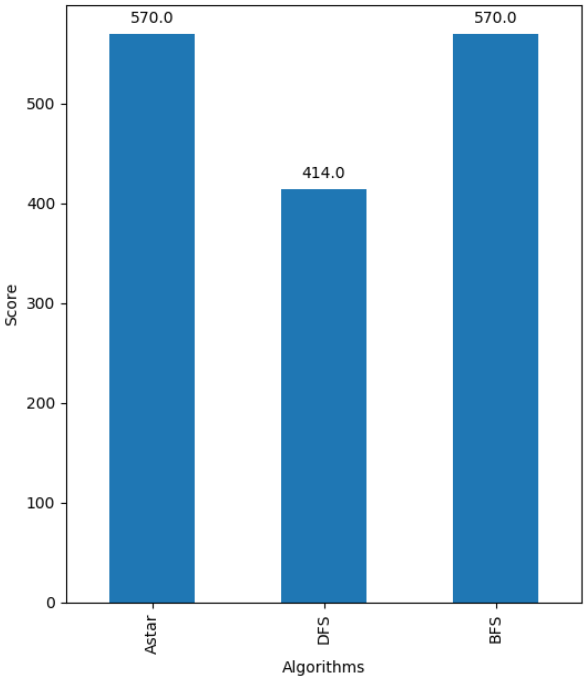
b. Reference to the course

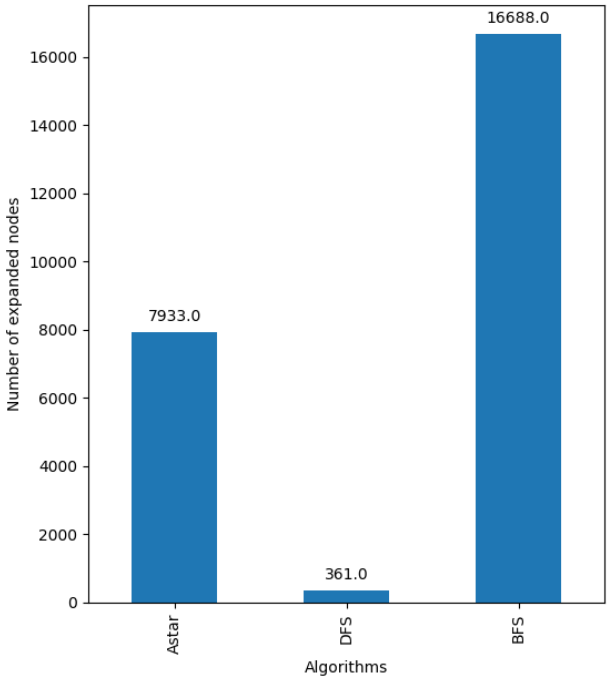
As explained in section 2.d the A* without h is equivalent to an UCS algorithm which is optimal. The score results are coherent. Nowadays, UCS need to expand more nodes to find the optimal solution, as it doesn't have information about where to go, which direction is more promising. It explains why there are a few more expanded nodes with the UCS. The difference between the two is more visible with the large grid, where there are less foods and the distances are bigger, so the indication given by the h(n) is more significant.

c. What is A* without n ?

It is the UCS (Uniform-Cost Search) algorithm.

4 Experiment 2





a. Differences between the results

The DFS algorithm gives a lower score than A*. The BFS algorithm gives the same score. Compared to A* the number of expanded nodes is really lower for DFS and is about twice for BFS.

b. Reference to the course

The DFS algorithm gives a lower score than A*. It is logical because as explains in the theoretical course the DFS algorithm gives always the leftmost solution in the research tree, regardless of the cost or depth, which has no special reason to be optimal.

The BFS algorithm gives the same score. As explains in the theoretical course, BFS is optimal only if the path cost is a non-decreasing function of the depth of the node, which is obviously the case here (the path cost is an increasing function of the depth of the node).

In terms of expanded nodes, DFS is very efficient because it will not need to cover a lot of paths, because each time it reach a dot it conceptually begins a new tree. The algorithm is so a succession of simple tasks (find a food, not all) which needs not a lot of exploration. In the case of this problem, DFS is so clearly more faster.

In terms of expanded nodes, BFS is approximately two times less efficient than A*. The BFS algorithm has no information about the food position (like UCS) but it has also a less significant g(n), which only takes account of the path length and not the number of remaining foods, than the UCS one. Using only the depth of the tree is less efficient than using a significant path cost to explore this tree, as seen in the course. Therefore, it is normal than BFS needs more expanded nodes to find the optimal solution.

	score	Computation time	Expanded Nodes
A* v1	570	2.70s	7933
A* v2	568	1,09s	168
UCS	570	2.50S	8219
DFS	414	0,15s	361
BFS	570	5.05s	10668

Tab. 1: Results of the different algorithms. A* v2 is the one in astar.py, A* v1 is the first implemented, as explained in section 2.c