Report LINFO1361: Assignment 1

Group N°137

Student1: SANCLEMENTE TELLEZ Mateo - 54192000

Student2: HOURMAN DE TOBEL Tristan - 59492100

February 29, 2024

1 Python AIMA (3 pts)

1. In order to perform a search, what are the classes that you must define or extend? Explain precisely why and where they are used inside a *tree_search*. Be concise! (e.g. do not discuss unchanged classes). **(1 pt)**

We need to extend the Problem class to create a Pacman class that defines game-specific logic like actions, result of these actions on the states and goal states. It is used inside tree_searches when they expand nodes to update the state and to select the next node to expand. It is also used to check if a state is a goal state according to the rules of the game.

2. Both *breadth_first_graph_search* and *depth_first_graph_search* are making a call to the same function. How is their fundamental difference implemented (be explicit)? **(0.5 pt)**

DFSg uses a stack (frontier = [Node(problem.initial)]), as its frontier where nodes are added to the end of the list and the last node added is the next to be explored. It means that this search goes as deep as possible before going back.
BFSg uses a queue (frontier = deque([node])) to explore first all nodes at the current depth. The difference between the two searches is the data structure used to represent the frontier.

3. What is the difference between the implementation of the *graph_search* and the *tree_search* methods and how does it impact the search methods? **(0.5 pt)**

Graph search methods keep track of visited states with an 'explored' set. Once a state is explored it is added to the set and before expanding again a node the algorithm checks if its state has already been reached before. Tree search methodes does not do that so they can explore the same state multiple times if it can be reached by different paths. Therefore it can cause these methods to explore possibly infinite loops.

4. What kind of structure is used to implement the *reached nodes minus the frontier list*? What properties must thus have the elements that you can put inside the reached nodes minus the frontier list? **(0.5 pt)**

In the graph search methods, there is an 'explored' set to keep track of the reached nodes and their states. Each element (state) must be unique and we must be able to compare it with other elements to decide if it should be added or not to the set.

5. How technically can you use the implementation of the reached nodes minus the frontier list to deal with symmetrical states? (hint: if two symmetrical states are considered by the algorithm to be the same, they will not be visited twice) **(0.5 pt)**

We could by default rotate each state so that, for example, Pacman always appear in the top left part of the maze. The algorithm would then consider two symmetrical states to be the same.

2 The PacMan Problem (17 pts)

(a) **Describe** the set of possible actions your agent will consider at each state. Evaluate the branching factor**(1 pt)**

Our agent considers four possible directions (UP, DOWN, LEFT and RIGHT) combined with all the numbers of steps it can move without being stopped by a wall or by the boundaries of the grid. For example, if it can only move to the left and there is a wall in 3 moves, the possible actions are (LEFT,1) and (LEFT,2). The branching factor would then be 2. In general, the branching factor depends on the maze configuration. For a state, it is the number of tiles until reaching a wall or a boundary in all four directions.

(b) How would you build the action to avoid the walls? **(1 pt)**

There is a loop in the code that increments steps in a direction and checks each cell starting from the agent's current position until there is a wall, denoted by '#', (or if it reaches the boundaries). If an action leads the agent through a wall then it is not added to the set of possible actions.

2. **Problem analysis.**

   (a) Explain the advantages and weaknesses of the following search strategies **on this problem** (not in general): depth first, breadth first. Which approach would you choose? **(2 pts)**

Breadth-first strategy : Finds the shortest path (with the minimal moves) but consumes more memory because it stores all the nodes at the current depth. If the goal state (collecting all fruits) is far from the initial position and the maze is large it can also take longer to reach the solution.
Depth-first strategy : More memory efficient since it only stores a single path but can be stuck in loops (DFSt). Also does not guarantee the shortest path which is what we're interested in in this problem so we have to choose the BFS approach.

   (b) What are the advantages and disadvantages of using the tree and graph search **for this problem**. Which approach would you choose? **(2 pts)**

The tree searches does not have to keep track of visited states so it is more memory-efficient than graph searches but they might explore the same states multiple time because of that. Because the graph search keeps track of visited states it will not explore redundant paths and has a higher chance of finding the shortest path which is the goal here so we have to choose this approach. If a solution exists the graph search will find it.

3. **Implement** a PacMan solver in Python 3. You shall extend the *Problem* class and implement the necessary methods -and other class(es) if necessary- allowing you to test the following four different approaches:

   - *depth-first tree-search (DFSt)*;
   - *breadth-first tree-search (BFSt)*;
   - *depth-first graph-search (DFSg)*;
   - *breadth-first graph-search (BFSg)*.

   Experiments must be realized (*not yet on INGInious!* use your own computer or one from the computer rooms) with the provided 10 instances. Report in a table the results on the 10 instances for depth-first and breadth-first strategies on both tree and graph search (4 settings above). Run each experiment for a maximum of 1 minute. You must report the time, the number of explored nodes as well as the number of remaining nodes in the queue to get a solution. **(4 pts)**

Inst.	BFS						DFS					
	Tree			Graph			Tree			Graph		
	T(s)	EN	RNQ	T(s)	EN	RNQ	T(s)	EN	RNQ	T(s)	EN	RNQ
i_01	0.005	80	796	0	7	51	>60	N/A	N/A	0.004	54	26
i_02	0.006	132	1096	0	15	67	>60	N/A	N/A	0	3	18
i_03	29.489	695394	5194043	0.004	154	184	>60	N/A	N/A	0.004	63	51
i_04	21.625	293009	3242559	0.009	149	483	>60	N/A	N/A	0.015	181	54
i_05	0.391	6093	58253	0.004	49	212	>60	N/A	N/A	0.006	54	65
i_06	0.007	235	1896	0	14	67	>60	N/A	N/A	0.002	38	27
i_07	0.073	2184	16235	0.001	34	50	>60	N/A	N/A	0.002	54	21
i_08	0.002	57	288	0	8	25	>60	N/A	N/A	0	21	10
i_09	0.003	68	526	0	8	31	>60	N/A	N/A	0.001	8	30
i_10	0.004	132	1096	0.001	15	67	>60	N/A	N/A	0	3	18

T: Time — **EN**: Explored nodes — **RNQ**: Remaining nodes in the queue

4. **Submit** your program (encoded in **utf-8**) on INGInious. According to your experimentations, it must use the algorithm that leads to the best results. Your program must take as inputs the four numbers previously described separated by space character, and print to the standard output a solution to the problem satisfying the format described in Figure 3. Under INGInious (only 1 minute timeout per instance!), we expect you to solve at least 12 out of the 15 ones. **(6 pts)**

5. **Conclusion.**

   (a) How would you handle the case of some fruit that is poisonous and makes you lose? **(0.5 pt)**

If we know that it is poisonous then we can handle it as a simple wall, instead of only checking for "#" and boundaries we would also check for "X" (if poisonous fruits are represented like that) before adding an action to the set of possible actions.

   (e) Do you see any improvement directions for the best algorithm you chose? (Note that since we're still in uninformed search, *we're not talking about informed heuristics*). **(0.5 pt)**

We could explore multiple nodes in parallel or use hashing to compare two states. These would only speed up the search process not reduce the number of moves though.