Report LINFO1361: Assignment 1

Group N°13

Student1: Mathis Delsart - 31302100

Student2: Léopold Hélard - 81452000

February 20, 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)**

The first one is the Pacman Class, which extends the Problem class, defining the problem and providing methods like actions(), result(), goal_test(), and path_cost(). It enables tree search algorithms to navigate and evaluate states effectively. The second one is the Node Class, which is a structural unit within the search tree that tracks node information (such as parent element, current state, action taken, path cost, and depth within the tree, etc.). The last one is the State class representing the game state (grid, remaining fruit count, game dimensions, etc.). We need to implement the __eq__() and __hash__() methods to avoid redundant paths in graph methods (See Q3).

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)**

The fundamental difference between breadth_first_graph_search and depth_first_graph_search lies in their choice of frontier data structure.
breadth_first_graph_search uses a queue (FIFO order), exploring nodes layer by layer, ensuring that nodes at the same depth level are explored before deeper nodes. It's a graph exploration by breadth. In contrast, depth_first_graph_search employs a stack (LIFO order), exploring nodes by depth and choosing the deepest unexplored node for expansion. It's a graph exploration by depth.

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)**

The primary difference lies in handling repeated states: tree_search doesn't avoid revisiting states, risking infinite loops, while graph_search maintains an explored set to prevent revisiting. Graph search guarantees completeness and optimality, albeit slightly slower due to the overhead of managing the explored set. Tree search may be faster but lacks completeness and optimality.

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)**

The structure used for "reached nodes minus the frontier list" is typically a set data structure, which is implemented as a hashtable in languages like Python. Elements in this structure must possess two key properties: uniqueness and hashability or immutability. Uniqueness ensures that each element is distinct within the set, while hashability or immutability enables efficient storage and constant-time lookup in the hashtable. By maintaining these properties, the set efficiently manages explored nodes, excluding those present in the frontier list, facilitating effective state space exploration.

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)**

To use the implementation of reached nodes minus the frontier list to deal with symmetrical states, we can apply a transformation function to each state before adding it to the data structure. This transformation function should ensure that two symmetrical states produce the same transformed representation. Consequently, if two symmetrical states are considered equivalent after transformation, only one of them will be added to the data structure, avoiding redundant exploration during the search. This approach prevents multiple exploration of equivalent state.

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)**

In this game scenario, the agent evaluates all accessible tiles in four directions: right, left, below, and above its current position. It examines each direction until it encounters a wall, at which point further exploration in that direction stops. The branching factor, representing the "mean" number of potential actions at each state, is calculated as the sum between the horizontal and vertical possible movements of the agent. The maximum branching factor is therefore calculated as the sum between the horizontal and vertical dimensions of the game grid, minus two. However, in practice, the actual branching factor may be lower than this maximum value when walls limit the agent's movement.

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

The initial step involves determining the current position of Pacman on the game grid. Subsequently, we iterate through four loops, one for each possible direction of movement. Within each loop, starting from Pacman's current position, we check whether the adjacent tile in that direction is a wall. If the tile is not a wall, we add it to a list of valid movement options. However, if a wall is encountered, we terminate the loop using a break statement. This iterative process ensures that we gather all feasible movement choices for Pacman, excluding any directions obstructed by walls.

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)**

For this problem, depth-first search (DFS) conserves memory but may lead to longer execution times due to its tendency to explore deeply before backtracking, with time complexity $O(b^m)$. In contrast, breadth-first search (BFS) explores nodes at each depth level before progressing to the next, resulting in shorter execution times but higher memory requirements, with time complexity $O(b^d)$. The increased memory requirement in BFS arises from the need to store all explored nodes, unlike DFS. Given the problem's likely shallow goal depth compared to its maximum depth, BFS is preferred due to its shorter time complexity. Memory use is not significant as both b and d are relatively small.

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

For this problem, tree search offers the advantage of requiring less memory since it does not store previously explored states. However, it may have long execution times due to revisiting states. In contrast, graph search prevents repetition of states, which is common in grid-based games, significantly accelerating execution time. However, to prevent repetition of states, we must store explored states.
Despite potentially requiring more memory due to storing explored states, graph search is preferable for this problem because it offers faster execution.

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	7.52 ms	116	1421	651.75 μs	9	70	N/A	N/A	N/A	5.88 ms	58	22
i_02	4.61 ms	96	861	2.29 ms	9	50	N/A	N/A	N/A	718.92 μs	10	29
i_03	69.18 s	1321026	9885054	5.69 ms	150	170	N/A	N/A	N/A	2.86 ms	27	69
i_04	18.06 s	241486	2766189	9.18 ms	141	456	N/A	N/A	N/A	20.43 ms	107	125
i_05	0.4 s	6809	65874	4.39 ms	66	283	N/A	N/A	N/A	12.23 ms	69	86
i_06	10.65 ms	280	2259	597.71 μs	14	57	N/A	N/A	N/A	3.06 ms	41	24
i_07	116.08 ms	2939	22495	1.23 ms	32	46	N/A	N/A	N/A	3.47 ms	53	25
i_08	2.42 ms	86	433	326.54 μs	10	24	N/A	N/A	N/A	856.04 μs	23	9
i_09	3.36 ms	76	581	514.67 μs	11	37	N/A	N/A	N/A	3.73 ms	51	11
i_10	4.33 ms	96	861	508.17 μs	9	50	N/A	N/A	N/A	728.08	10	29

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)**

In the case of a poisonous fruit that leads to a loss upon stopping on it, I would treat it as a tile that can be passed through but not stopped upon. Therefore, I would ensure that the player's movement algorithm excludes the tile containing the poisonous fruit from potential stopping points while still allowing movement through it. In every direction, I specifically check if the tile corresponds to a poisonous fruit. If it does, I use the "continue" statement to skip that iteration of the loop (for this direction), allowing the player to move through the tile without halting their traversal.

   (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)**

Adding the Pacman's position as an attribute of the State class eliminates the need to search the grid for the Pacman's position for each state, thus improving efficiency (Already implemented).
To efficiently calculate the new possible movements of Pacman, we aim to remember the previous movement and the previous available moves. This allows for rapid computation of the new possible movements (Not implemented).