Report LINFO1361: Assignment 1

Group N°13

Student1: Mathis Delsart - 31302100

Student2: /

April 23, 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)**

1) **Defines the Pacman Class**, which **extends the Problem class**, defines the specific problem of the Pacman game and provides methods such as **actions(), result(), goal_test()**. These methods are **used in tree_search to generate possible moves, apply actions, and test if the problem's goal is achieved**. 2) **Defines the Node Class** which represents a structural unit in the search tree, containing **information about the node** such as **its parent, current state, action, etc**. **Used in tree_search, it allows for creating and expanding nodes**. 3) **Defines the State class** which represents the game state, including the **grid, remaining fruit count, game dimensions, etc**. This class is **used in the Node class because one of its attributes is a state**.

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** = ***First In First Out***), 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** = ***Last In First Out***), 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**, while **graph_search maintains an explored set** (more explanations in the next question) **to prevent revisiting**. **graph_search guarantees completeness and optimality**, but **requires more memory to keep the explored set updated**. **tree_search can lack completeness and optimality** because it **risks infinite loops**. In our game, m is infinite (back-and-forth movements), and therefore, a DFS_tree will **almost never terminate** (except by luck if one happens to be on the correct path directly). A **BFS_tree ensures completeness** but **its time complexity can be substantial**.

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 the explored set is a **dictionnary**. **The keys must be unique, immutable and hashable. Immutability** means that the **keys can't be changed after they are created**. **Uniqueness** ensures that **each element is distinct within the set**. **Hashability** enables **efficient storage and constant-time lookup => keys must be able to be converted into a hash value (= be hashable)**.

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 handle symmetrical states effectively, we need to **implement the __eq__() and __hash__() methods within the State class**. The **__hash__()** method **generates a hash value** from the state's attributes. On the other hand, the **__eq__()** method defines the **behavior of the equality operator (==)** for instances of the State class. The hash value generated by **__hash__()** must be **the same for two symmetrical states**, and the **equality operator between two symmetrical states must return True**.

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**. **The minimum branching factor is 1** (*or* ***0 if the initial state is the Pacman surrounded by 4 walls***).

(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**. We **store this position as a state attribute** to avoid searching the entire grid repeatedly. Subsequently, we **iterate through four loops, one for each possible direction of movement (Up, Down, Left, Right)**. 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 movements**. 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, we chose breadth_first strategy because **it guarantees finding the shortest path to the goal** (not like depth_first strategy !) **because the cost is uniform (= 1)**. It explores all possible paths level by level, ensuring that the first solution found is the optimal one in terms of path length. It is also **complete, meaning it will find a solution if one exists**. However, breadth_first **requires more memory compared to depth_first because it needs to store information about all nodes at each level of the search tree**.
In the best case, the **time complexity of depth_first is good** (but **terrible in the worst case**, or even infinite !) ($O(b^m)$) but **for breadth-first, it's bad** (but **same as the worst case**) ($O(b^d)$). The **space complexity of depth_first is good (kind of bilinear)** ($O(b * m)$) but **for breadth-first, it's very bad** ($O(b^d)$).

   (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, graph_search avoids revisiting the same state multiple times (see Q1.3), significantly **reducing the number of explored nodes** and thus the **execution time**. Although there **may be an overhead due to the method __hash__() if there are few equivalent states**. **In our problem, there are many identical states** (such as back-and-forth moves), which is common in grid-based games. The **main disadvantage of graph_search** is the **extra memory required to keep track of explored nodes**. On the other hand, **tree_search doesn't use extra memory** but **can revisit the same state multiple times**, leading to a significant **increase in execution time**, especially when there are **many identical states, as is the case in this problem**. Ultimately, we **chose the graph_search approach** because there are **many identical states in this problem, and the extra memory required to avoid revisiting them seems acceptable compared to the time saved in 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)**

We would treat the poisonous fruit as a tile that can be passed through but not stopped upon. Therefore, We 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 (*in the four loop*) , We would specifically **check if the tile corresponds to a poisonous fruit**. **If it does, We would use the "continue" statement to skip that iteration of the loop**, allowing the player to move beyond the poisonous fruit. Alternatively, we can store all positions of poisonous fruits in the state attributes. Then, at the end of the actions() function, we exclude these positions if any poisonous fruits are present. This approach enhances modularity as it doesn't remove any current code.

   (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)**

1) **Check at each depth if the number of fruits has decreased**. If this is the case, the search should **focus only on the states where the number of fruits has decreased** and **disregard the others**. The **optimal solution will never be found among the children of these removed states**. **These states should be removed from the frontier data structure**, and the **explored set should be reinitialized with only the states where the number of fruits has decreased** to **reduce memory usage**. 2) **Incorporating Pacman's position as an attribute of the State class**, the **necessity to search the grid for Pacman's position for each state is eliminated**. 3) **Retain information about the previous movement and previous available moves**. This facilitates **swift calculation of new possible movements**, optimizing the search process.