Report LINFO1361: Assignment 1

Group N°1 (Moodle), 39 (INGInious)

Student1: Victor Carballes Cordoba (NOMA : 34472100)

Student2: Krystian Targonski (NOMA : 42942000)

April 6, 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. State : This class is used to represent / store the current state of the search problem.
> 2. Node : This class is used to construct and keep track of the exploration of the problem.
> 3. Problem : This class is used to define the problem in its entirety. It contains the initial state, the goal test, the possible actions as well as the way to apply the actions.

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 key difference is the data structure used for the frontier. In the case of *breadth_first_graph_search* (BFS), the frontier is implemented as a FIFO queue whereas in the case of *depth_first_graph_search* (DFS), the frontier is implemented as a stack. This difference changes the way the nodes are explored, the BFS explores the nodes level by level while the DFS fully explores one node and its children before moving to the next one.

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 difference between those 2 methods lies within the method used to handle repeated states or loops in the search. The *tree_search* method does not handle them which can cause infinite loops. The *graph_search* method on the other hand, keeps track of the states that have been explored and will not add them to the frontier if they are already present thus avoiding loops. As for the impact, *tree_search* may run indefinitely if the search space contains loops while *graph_search* will not.

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 this case, the structure will be used to store the explored nodes so it can be done with either a set or dictionary. The propreties of elements to be put in those structure should be : a. They should be hashable b. They should be unique c. They should have a defined equality operator

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)**

If a symetry is defined, we can use a unique representation for a state that is independant of any symetries. If we have this representation, we can just transform the state into it and then compare with what is already in the reached nodes minus the frontier list and if it is, we ignore it.

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 directions at each state: up, down, left and right. For each one, it checks how far it can move before finding a wall or fruit. If the later is detected, then only the action that leads to that fruit will be given. An optimisation we did to reduce the branching factor, was allow an agent to rush in a direction if he found himself in a tunnel (be it between walls or something else) effectively considering less possibilities. Which leads us to the worst branching factor, N+M, where N is the number of rows and M the number of columns of the map, and the best, 1.

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

We would first create a function that checks if the position is within the map's bounds and not inside a wall. Then, using this function, we iterate forward updating the offset of the observation, up until a wall or the end of the map is found.

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)**

We would prefer the breadth first search strategy, as it is guaranteed to find the shortest path to any goal using techniques such as marking visited nodes (in a separate set). If we choose the depth first search algorithm, its very likely for it to get stuck in infinite loops, as the algorithm is incapable of avoiding previously explored nodes. If it did, then it would avoid its own tail, and the algorithm would never end or would get stuck.

   (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 Tree search, first the advantages would be : a. Simplicity to implement and b. When it works, the result would be given quickly. However there would be a big disadvantage, the algorithm would not be able to avoid loops which would lead to either infinite loops / algorithm getting stuck or the algorithm eating twice the same fruit or something of the sort which isn't easy to handle.
For Graph search, the disadvantages would be : a. More complicated to implement and b. More memory usage / overall hit on performance. HOWEVER, the advantages would outweight those. The algorithm would avoid loops witch guarantees the algorithm to end and to not eat the same fruit more than once. Hence why, in this case, graph search would be preferable.

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	0,007	50	334	0,001	6	43	/	/	/	/	/	/
i_02	0,010	109	681	0,001	13	95	0,00	4	12	0,00	4	12
i_03	0,793	14379	39702	0,210	3795	10583	/	/	/	/	/	/
i_04	4,617	45803	234229	0,753	7158	38644	/	/	/	/	/	/
i_05	1,128	9553	56748	0,158	1409	8143	/	/	/	/	/	/
i_06	0,001	22	27	0,001	8	13	/	/	/	/	/	/
i_07	0,072	1017	4392	0,014	180	836	/	/	/	/	/	/
i_08	0,001	13	7	0,001	6	6	0,00	5	3	0,00	5	3
i_09	0,021	195	1133	0,003	32	162	/	/	/	/	/	/
i_10	0,010	104	681	0,002	13	95	0,00	4	12	0,00	4	12

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)**

Just add a check in the actions, if the fruit is poisonous, then the agent should try to jump over it or just avoid it.

   (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 improve the algorithm by using binary tables to store the different elements in a fast, vectorizable way. This would allow us to avoid using loops and would speed up the algorithm overall. Another improvement would be to avoid using dictionaries and instead use a binary table to store the visited nodes. This would reduce the memory footprint and allow for faster lookups. Lastly, we could keep track of already eaten fruits in other instances, but although it would be better for memory and branching factor, this would become more of an informed search solution.