Report LINFO1361: Assignment 1

Group N°102

Student1: Antoons Miguel

Student2: Danlee Maxime

March 1, 2023

1 Python AIMA (5 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)**

To perform a search, three classes are used. Problem, State and Node. The State class is used to represent the states of the game at a node in the tree. Each node contains a state object, a parent, the action taken from the parent to reach the actual state, the cost of the path from the initial state to the node and the depth. The Problem class is used to find all the possible actions to reach a new state. It can also generate a new state from an old state and an action.

2. Both *breadth_first_graph_search* and *depth_first_graph_search* have almost the same behaviour. How is their fundamental difference implemented (be explicit)? **(1 pt)**

The difference between depth-first search and breadth-first search is the order in which nodes are visited. In a breadth-first algorithm, nodes at the same depth from the source node are visited before visiting nodes at a deeper depth. In depth-first search nodes are visited as far as possible along a branch before switching to another branch.

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? **(1 pt)**

For the graph search functions, the implementations uses a graph rather than trees such as in the tree search functions. A tree branch only connects parents with its children, this means that there can be no cycles and the leafs of the tree are the goals. A graph can handle cycles but in order to prevent infinite loops, a graph algorithm must keep a data structure (typically a set or a dictionary) containing all the already visited nodes.

4. What kind of structure is used to implement the *closed list*? What properties must thus have the elements that you can put inside the closed list? **(1 pt)**

In python a dictionary or a set can be used. In both cases the keys must be unique so that we can check if an element has already been used.

5. How technically can you use the implementation of the closed 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) **(1 pt)**

We could define an equivalence relation between symmetrical states and only add one of them to the closed list. Since the actions are the same (but just symmetrical) this could save time.

2 The Tower sorting problem (15 pts)

1. **Describe** the set of possible actions your agent will consider at each state. Evaluate the branching factor considering n tower with a maximal size $m$and c colors (the factor is not necessarily impacted by all variables) **(1.5 pts)**

For each state, the agent will consider for each column (if the column is not empty) if it can move the top element to another non-empty column. This means that in the worst case, the branching factor will be of $n * (n - 1)$. In reality, it will be lower since it can't move an element from an empty tower or to a full tower.

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? **(1.5 pts)**

Depth-first search is advantageous in the sense that if it gets the correct the correct branch immediately, it will get the shortest path faster than the breadth-first search. However, since the branching factor is relatively high, the chances of getting the correct branch from at the beginning are small and thus the breadth-first search will be faster in most of the cases. A disadvantage of the breadth-first search is that it will consume exponentially more memory the deeper it has to go in the graph or tree to find the goal.

(b) What are the advantages and disadvantages of using the tree and graph search **for this problem**. Which approach would you choose? **(1 pts)**

The disadvantage of a tree-search is that, since no cycles are allowed, a node can be visited multiple times and a branch could end up looping between two different states. Additionally, two different branches could end up having the same result, a problem that doesn't exist on a graph-search since visited nodes are stored and cannot be visited twice. Furthermore, since in the graph algorithm the new states are evaluated directly rather than put in a list and evaluated later, the graph-search algorithm has a slight speed advantage. Therefore, we would choose the graph-search algorithm rather than the tree-search algorithm.

3. **Implement** a solver for the Tower sorting problem 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 3 minutes. 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	23.4	185194	100020	14.5	114214	70978	124	1053823	714303	180	-	-
i_02	1.68	13745	15083	0.81	6052	7691	125	1062830	704691	180	-	-
i_03	1.16	9804	11772	0.58	4115	5687	125	1091698	677801	180	-	-
i_04	7.31	56636	53236	3.59	27438	29196	34.8	230279	597484	180	-	-
i_05	17.8	76826	175304	5.31	21546	55278	180	-	-	180	-	-
i_06	13.9	60649	128462	4.32	18044	42603	180	-	-	180	-	-
i_07	11.5	49474	108087	3.37	14193	35279	180	-	-	180	-	-
i_08	96.3	267927	884083	20.7	56889	211036	180	-	-	180	-	-
i_09	77.7	228642	666787	19.8	53694	174946	180	-	-	180	-	-
i_10	73.2	204786	678874	16.3	42435	162349	180	-	-	180	-	-

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 only input the path to the instance file of the problem to solve, and print to the standard output a solution to the problem satisfying the format described earlier. Under INGInious (only 45s timeout per instance!),

we expect you to solve at least 10 out of the 15 ones. Solving at least 10 of them will give you all the points for the implementation part of the evaluation. **(6 pts)**

5. **Conclusion.**

   (a) Are your experimental results consistent with the conclusions you drew based on your problem analysis (Q2)? **(0.5 pt)**

Yes, indeed the conclusions we drew at question 2 are consistent with our results.

   (b) Which algorithm seems to be the more promising? Do you see any improvement directions for this algorithm? Note that since we're still in uninformed search, *we're not talking about informed heuristics*). **(0.5 pt)**

The breadth-graph search algorithm seems to give the best results. One way to improve it is to add a condition that verifies if a child is not already in the frontier list. This way, we ensure that no identical states are present in the frontier list.
Finally, we could use a set rather than a dictionary for storing the already visited nodes : it takes less memory and is slightly faster.