Report LINFO1361: Assignment 1

Group N°34

Student1: Diego Troch

Student2: David Lefebvre

March 2, 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)**

We must extend the class *Problem* in the class *TowerSorting* and define the functions *actions*, *results* and *goaltest*. The function *actions* gives a list of the different possible actions (to speed up our algorithm, we decided not to directly return the list but rather to give the different possible actions through the yield keyword). The function *results* allows to apply a particular action on a given state. The function *goaltest* allows us to check if we've reached our goal. (For the goal test, instead of using *lists*, we used the *set* built-in data structure).

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 fundamental difference between breadth first graph search *(BFS)* and depth first graph search *(DFS)* is the data structure it uses to keep track of the visited nodes. *DFS* uses a stack, which means it explores nodes in a LIFO (last-in first-out) order. Furthermore, *BFS* uses a queue, which means it explores nodes in a FIFO (first-in first-out) order. The path found by both algorithms are different because they explore the graph differently.

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)**

Graph_search keeps track of the nodes that have already been visited, while *tree_search* does not. *Graph_search* uses a closed list to store the visited nodes and *tree_search* only uses a list called the frontier to store the nodes that are waiting to be expanded. *Graph_search* uses more memory than *tree_search*, but it avoids revisiting and expanding the same node multiple times. *Graph_search* is also complete, meaning it can find a solution if one exists, while *tree_search* may get stuck in an infinite loop if there are cycles in the tree.

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)**

We need to use the closed lists in order to find if a node has already been visited or not. In order to check whether a node has already been visited with the lowest possible time complexity, we use the data structure dictionary. For this purpose, a hash and equal function had to be implemented to represent the state. Only the grid attribute of the state object is interesting when we want to represent it in a hashed version.

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)**

Because every node is identified by a hash, all we have to do is check if the hash of the node we are visiting is already in our closed list. The closed list can be implemented as a python dictionary or a python set where the key is the correspondent hash and the value is a boolean indicating if the node has been visited (goal-tested).

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)**

The branching factor allows us to have an idea of the number of child nodes that can be obtained after extending the corresponding parent node. In this context it corresponds to the number of moves that can be made for a certain state. Depending on the state, we can end up with an upper and lower bound of possible factors. Upper bound: if none of the towers is filled to its maximum and none is empty the branching factor will be n*(n-1). Lower bound : there is only one empty tower and all the others are filled, the branching factor will be (n-1)

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)**

The weakness of the depth first algorithm is that we encounter a cycle which causes a timeout when we benchmark our program with a maximum execution time of 3 minutes whether we are on a tree or a graph implementation. The breadth-first algorithm will also renounce this problem when used with a *tree_search* but not with a *graph_search*. An interesting approach would be to first use a DFS because if the solution is close to the root, it will be found quickly and then, if no solutions are found, run a BFS which will find the solution in all cases.

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

We used the *graph_search* because it allows us to find a solution and to avoid cycles. This is due to the fact that the *graph_search* will not "goal test" nodes that have already been visited. The problem with *tree_search* is that the algorithm may visit the same node several times, because we don't keep track of the visited nodes, and therefore be inefficient if we encounter a cycle.

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	TimeOut			18.97	208077	450205	TimeOut			TimeOut		
i_02	TimeOut			0.34	6070	17518	TimeOut			TimeOut		
i_03	TimeOut			0.27	4118	12516	TimeOut			TimeOut		
i_04	TimeOut			2.21	28334	73885	TimeOut			TimeOut		
i_05	TimeOut			2.85	21755	131559	TimeOut			TimeOut		
i_06	TimeOut			2.43	18050	105411	TimeOut			TimeOut		
i_07	TimeOut			1.92	14196	86069	TimeOut			TimeOut		
i_08	TimeOut			12.57	56938	528852	TimeOut			TimeOut		
i_09	TimeOut			11.98	55583	498972	TimeOut			TimeOut		
i_10	TimeOut			9.54	42437	410851	TimeOut			TimeOut		

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, completely. We have indeed experienced cycles with the tree approach, the *BFS_tree* and *DFS_tree* implementation have cycles which prevent us from finding the solution. Since the dfs algorithm is not complete (neither the "graph" version, nor the "tree" version), it did not allow us to find a solution for any of the problem instances. After the implementation of the "tower sorting" problem, we could experiment the fact that the *BFS_graph* algorithm is indeed a complete algorithm allowing to reach a solution.

   (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)**

Logically, we tend to say that the *BFS_graph* is more promising, but we should not forget the advantages of the *DFS* approach because it is possible to find a solution faster than the *BFS* if the solution is close to the root (the initial problem). One possible way to improve *BFS_graph* would be to limit the maximum depth that *BFS_graph* can reach. This could reduce the search space and therefore speed up the search. However, the solution might not be found if the depth limit is too short but we could imagine a solution to this problem by iteratively increasing the maximum depth. These two proposed solutions are currently known respectively as *DLS* (Depth Limited Search) and *IDDFS* (Iterative Deepening Depth First Search).