# Report LINFO1361: Assignment 1

**Group N°007**

Student1: Quentin Prieels

Student2: Martin Gyselinck

February 26, 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 *Problem* class. This is used to check if a node (i.e. a state of the problem) is a solution, find all possible actions at this point and perform them. This class define what are the game's rules.

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 BFSg algorithm tends to find the solution with the minimal depth, by trying every childs of a node layer before going to the next layer. The DFSg algorithm will first dive as deep as possible in the node's graph (i.e. a tree where each node is explored only once).

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)**

The graph methods remember which nodes are already visited, while the tree ones don't. It causes the tree algorithms to do a loop and never finish, while the graph algorithms remember them and don't go through twice, giving them the possibility to visit all the 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)**

A Python set is used to implement the closed list (i.e. the list of already visited elements, named `explored` in the code). Every item put in a set needs to be hashable. Basically, a Set is like Java's HashMap, but accepting various types.

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)**

Symmetrical states doesn't need to be visited twice, because they will bring the same children to the open list. If we use the BFSg algorithm, we are certain that the first symmetrical state visited contains the shortest path to him. Therefore, to find the optimal solution, any move to a symmetric node will lead to a less or equal optimized solution.

## 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)**

Each tower of the game can move one item to all the others (assuming that the other towers are not full and that the considered tower contains an element), which means than a disk from a tower can go on $n-1$ towers maximum. As we have n tower, the number of possible branch from a specific node is at most $n*(n-1) = n^2 - n$ branchs.

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 use of breadth first ensures that the first goal found is the one that minimizes the number of moves required to obtain the solution. Its weakness is that we have to visit all the nodes with a depth lower than the solution to find it. The depth first could give a very fast solution of the problem, but it requires a lot of luck in the visiting path, or it could be way longer.

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

Using a graph for this problem resolves the problem of infinite loop in tree algorithms, where reversing the previous action can be a action of the node. Using a graph makes us store which nodes are already visited, and then stop it from going back. The disadvantage of this approach is the use of memory which must store all the nodes already visited, unlike the tree.

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 | | | | | | | | | | | | |
| i_02 | | | | | | | | | | | | |
| i_03 | | | | | | | | | | | | |
| i_04 | | | | | | | | | | | | |
| i_05 | | | | | | | | | | | | |
| i_06 | | | | | | | | | | | | |
| i_07 | | | | | | | | | | | | |
| i_08 | | | | | | | | | | | | |
| i_09 | | | | | | | | | | | | |
| i_10 | | | | | | | | | | | | |

**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)**

We can see than BFSg is the only giving us output in the time interval. As expected, it always finds the optimal solution. This implementation on the tree takes too much memory and time to be finished. Indeed, without memorizing the nodes already visited, due to the large branching factor, the possibilities are too numerous to be quickly analyzed. The DFSt algorithm never finishes. This seems normal, because the algorithm can make a move and then undo it over and over again, creating an infinite loop. With regards to the DFSg algorithm, he tends to finish but it takes too long to be worth. And its solution isn't the optimal one. If he found one, it's most probably due to hazard than something else.

   (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)**

Actually, that's the BFSg algorithm which is probably the most promising, as he check all nodes in ordered depths, without visiting it twice and going too much exponentially at each new depth. It gives us a solution in a reasonable time. One optimization that can be found are to avoid some branch where we are sure than we are destroying a part of the solution, by checking if the some towers are already sorted and fulfilled. (In practice, it seems than calculating that is slower than searching in it, at least in our code where we used `np.unique()`.)