# Report LINFO1361: Assignment 1

**Group N°59**

Student1: Cosyns Mathieu

Student2: Ngoran Ntam Brandon

February 23, 2022

## 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)**

We have to extend the Problem class from the search.py file because this class will provide the necessary information that the different algorithms will use such as the initial state, actions, actions results, the goal, the path costs etc. We can also define the State class to help view our states.

2. Both *breadth_first_graph_search* and *depth_first_graph_search* have almost the same behaviour. How is their fundamental difference implemented (be explicit)? **(0.5 pt)**

The fundamental difference between both algorithms is that the breadth_first_graph_search will evaluate the nodes of the tree by levels whereas the depth_first_graph_search will evaluate the nodes by visiting the deepest node first. 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)**

When we use the graph_searches, we remember the previous nodes that we visited in the search. These nodes are kept in either a set or a dictionary. In the tree_searches, the explored nodes are not kept in any data structure. This means that we can visit these nodes multiple times in a single search.

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? **(0.5 pt)**

A closed list can either be a set or a dictionary that acts like a set data structure. The nodes in the closed list must be expanded and unique. This means that they are nodes that we have visited in the search and can be reachable from the initial state.

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) **(0.5 pt)**

In case of breadth and depth searches, the first path that leads to a symmetrical state will be kept.

## 2 The 2D Rubik's square problem (17 pts)

1. **Describe** the set of possible actions your agent will consider at each state. Evaluate the branching factor considering a grid of size $m \times n$. **(2 pts)**

The branching factor will be $m \times (n-1) + n \times (m-1)$ because we have $n-1$ possible action for each row (going to the right) and $m-1$ possible action for each column(going downwards) between each state.

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 choose the breadth first strategy because the aim of the Rubik's cube solver is not only to find the solution to the objective state but to find the solution in the least amount of moves possible. By using the breadth first, we check if each node per level is equal to the goal state. Each level represents a move which means that as soon as we find the goal, we will have the minimal amount of moves to the final state. If we went for the depth first strategy, we would have to find the paths leading to the final state and then compare the paths to find the shortest one. This isn't efficient.

(b) What are the advantages and disadvantages of using the tree and graph search **for this problem**. Which approach would you choose? **(2 pts)**

Since we are using using the breadth first search, we don't need to revisit the nodes that we know won't bring us to the optimal solution. This means that the tree searches won't be very useful to us in this situation because if we do revisit them, we will automatically have more additional moves to arrive at the solution. By using the graph search, we won't have to revisit these nodes since they will be saved in our closed list.

3. **Implement** a 2D Rubik's square 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 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 | 1.48 | 6280 | 105892 | 0.04 | 203 | 2592 | ∞ | | | 117.10 | 4610 | 6304 |
| i_02 | 21.00 | 73307 | 1505720 | 0.19 | 894 | 12699 | ∞ | | | 178.22 | 5287 | 7579 |
| i_03 | ∞ | | | 2.93 | 9951 | 185581 | ∞ | | | ∞ | | |
| i_04 | ∞ | | | 3.66 | 11534 | 203859 | ∞ | | | ∞ | | |
| i_05 | ∞ | | | 28.86 | 83166 | 1304142 | ∞ | | | ∞ | | |
| i_06 | 0.006 | 19 | 494 | 0.0009 | 1 | 17 | ∞ | | | ∞ | | |
| i_07 | ∞ | | | ∞ | | | ∞ | | | ∞ | | |
| i_08 | ∞ | | | 54.35 | 111704 | 1597354 | ∞ | | | ∞ | | |
| i_09 | ∞ | | | 8.27 | 11955 | 354134 | ∞ | | | ∞ | | |
| i_10 | ∞ | | | 34.82 | 67339 | 1108059 | ∞ | | | ∞ | | |

**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. We chose the breadth first graph strategy and it ended up being the most optimal strategy out of the four.

   (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)**

Breadth first seems to be the most promising because it solves most of the cubes under 3 mins whereas the others algorithms take more time and more memory to be solved. Since it takes a lot of memory, we could improve it by finding a way to reduce the frontier of possible states and the number of possible actions