Report LINGI2261: Assignment 1

Group N°29

Student1: Ben Haddou Mehdi

Student2: Hennebo Eliot

February 25, 2021

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)**

In order to perform a search, the class Problem need to be extended and Node need to be defined. The class which extend Problem must implement at least the following method ;
- successor : gives a sequence of pairs reachable from a given state
- goal_test : tests if the given state is a goal state
The methods tree_search tests if the node generated by the method successor are goals via the test_goal method and if it is not a goal, it expands the node (put it in the stack/queue).

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 only difference between the two algorithms is the data structure used to store the next node to be expanded. BFS uses FIFO data structure (Queue) and DFS uses LIFO data structure (Stack).

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 graph_search method keeps in memory nodes already visited to avoid cycle. It's an extra cost for space as the space complexity becomes exponential. It is not the case in tree_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)**

The Data Structure is a set and we need to know in O(1) if an element is in the set, so we need to compute a hash for each element of the set instead of performing a O(n) linear search. Data structure that uses hash like HashSet will do the job. The structure must perform search operation in O(1) and element must be hashable.

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)**

By remembering the states that have already been seen and putting them in a Data Structure like HashSet for example. For this to work we just need to create a method that can recognize symmetrical states and use it to check if a state is already in the set.

2 The Knight's tour Problem (17 pts)

1. **Describe** the set of possible actions your agent will consider at each state. Evaluate the branching factor. **(2 pts)**

The knight can move to 8 different positions in each state (legal moves of a knight in chess). This means that he will potentially find himself in certain states with fewer move possibilities, since he will be constrained by the rules of the problem to stay on the board and not to move twice on the same square. We can still consider that the Branching Factor is 8, since this represents an extreme case, even if it is slightly overestimated for the reasons explained above (board limit and passage per square), we cannot provide a precise Branching Factor since the size of the board may vary.

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)**

For this problem, we know the goal depth which is always equals nRows * nCols and which is always the maximum depth. BFS will in any case go through all the nodes except maybe some of the last depth while DFS can fall on the ideal path at any time as it travels the path until it is blocked/arrived at the target state. In addition, DFS has the advantage of low space complexity, unlike BFS, which uses exponential space complexity. Finally, DFS cannot lock itself into an infinite loop due to the finite depth of the problem, which makes it optimal.

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

Tree search has for advantage to use less memory since nodes are not stored in memory but this allow to visit the same node multiple times.
Graph search has for advantage to browse less nodes but for disadvantage to keep in memory each visited nodes (and thus to compute hash for each node). But as in our algorithm, most of the paths are found very quickly and therefore by traversing very few nodes, this performance gain is not remarkable and therefore tree search finally seems to be faster in our tests. Moreover, since the problem is in the form of a tree, the optimization provided by Graph Search seems to slow it down in this situation because of the additional operations it provides, which are therefore useless.

3. **Implement** a Knight's tour 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(ms)	EN	RNQ	T(ms)	EN	RNQ
i_01	180	750k	959k	180	759k	965k	0.735	30	33	1.60	30	33
i_02	180	518k	1.20m	180	519k	1.21m	60.31	5534	53	77.77	5534	53
i_03	180	1.01m	777k	180	1.02m	786k	3305	265572	24	4507	265572	24
i_04	180	558k	1.17m	180	562k	1.18m	180k	14.4m	46	180k	10m	47
i_05	180	482k	1.27m	180	484k	1.28m	5.068	427	65	5.892	427	65
i_06	100	1.46m	55	97	1.46m	55	0.669	58	23	0.752	58	23
i_07	180	549k	1.17m	180	547k	1.17m	27.70	2342	46	50.99	2342	46
i_08	180	544k	1.19m	180	539k	1.18m	1.974	156	54	3.119	156	54
i_09	180	490k	1.29m	180	484k	1.28m	1.011	54	68	1.096	54	68
i_10	180	668k	1.05m	180	666k	1.05m	3.787	347	41	6.526	347	41

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 number 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 45s timeout per instance!), we expect you to solve at least 12 out of the 15 ones. **(6 pts)**

5. **Conclusion.** 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*). **(1 pt)**

At each move, our algorithm will generate a number of successors between 0 and 8, for all these successors, it will sort them according to the distance that separates the position of the Knight from the closest borders in that given state. It should be possible to be more precise in the calculation of the distance to the border, since the usefulness of this strategy is to restrict the possibilities of the knight, one could imagine algorithms that favor this system and that instead of sorting distances, would sort the states according to their number of legal moves allowed as explained in Warnsdorff's rule which seems to be one of the most optimal strategies, but which we have not used.