UCLouvain

École polytechnique de Louvain

Artificial Intelligence

Assignment 1 : Solving Problems with Uninformed Search

Pasture Guillaume - 32901900
Defrère Sacha - 51621900

Groupe 151

Année académique 2021 - 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)**

There are 2 (or 3 in some cases) main classes we have to implement in order to perform a search.
The first required one is obviously the Node class, which will hold information about a state such as the state itself, its parents, the last action taken or the total cost. This will be needed in order to go from Node to Node while iterating for the search.
The second and optional one is a kind of data structure that should hold the different nodes that compose the frontier of our current search, such as a queue or a stack.
The third and last required one is the Problem class which must be extended to fit the real problem's specifications. This is needed for multiple reasons : to get the initial state of the problem at the beginning of the algorithm, to check whether or not we have reached a solution with each new state of the iteration, and lastly to extend our frontier each iteration with the new nodes that should be goal-tested.

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 main difference between those two algorithms is the order of picking new vertices. DFSg explores the graph by looking for new vertices far away from the initial state, taking closer vertices only when dead ends are encountered. On the contrary, BFSg completely covers the area close to the starting vertice, moving farther away only when everything nearby has been checked. DFS paths tend to be long and winding whereas BFS paths are short and direct.
In the code, this is simply implemented using a FIFO queue for the BFSg and a LIFO stack for the DFSg.

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)**

A *..._tree_search* assumes its search base includes no cycle. When the algorithm enters in a branch, it only sees vertices never examined. Whereas in *..._graph_search*, it assumes vertices are interconnected and keeps an indicator which shows if the vertice has already been examinated.
In the code, this is implemented using a simple set or collection often called *explored* to track which nodes have been visited.

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 main goal of our closed list will be to check whether or not we already have visited the vertice (state) we are currently in, to know if we can simply ignore it. For this problem, we decided to use an implicite graph and the easiest way to implement its closed list is with a hash table. Given the large amount of states that are reachable and the number of nodes that needs to be stored, rather than remembering each encountered grid, it is more optimal to simply hash the grid's value and compare the next states by hashing their own grids and checking in the hash table.

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)**

The hash table takes the grid to see if a certain state is in the closed list or not. So, if by another path of actions the algorithm arrives to a grid that has already been examinated, it will check in the closed list's hash table and see that this state has already been reached and will not visit the vertice a second time.

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)**

In this particular problem, for a given initial grid, the set of possible actions will never vary : each and every column (n columns) and row (m rows) will always be able to be circled around 1 to m-1 and n-1 respectively. For that reason, the branching factor is equals to $m(n-1)+n(m-1)$.

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 ? **(2 pts)**

The main drawback of DFS is that it can explore very distant states before even looking at a grid that is very similar to the initial grid. It can then take a very long time to solve an easy problem, juste because it started with the wrong action. The only advantage of DFS occurs if, by chance, the algorithm finds the correct solution on its first guess. In this case only, it can be faster than BFS.

The advantage of BFS is that it ensures to find the path with as few actions as possible. In some very specific cases, BFS may be slower to find the solution than DFS but it is much faster on average.

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

For this specific problem, the advantages and disadvantages of using the tree search are the following : because it doesn't store the nodes it visited in the past in its memory, this algorithm indeed uses less memory space than the graph search, but it also takes a lot more time to find the solution, as it visits again and again states it was already in.

As for the graph search, it implements the closed list structure, using memory space to store the visited nodes, which allows it to avoid going back to those previous states and to gain a lot of processing time.

To solve this problem, the optimal method is the graph search, given that different actions can lead to the same exact state, creating cycles in our tree that could cause the tree search to loop uselessly.

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)**

[7cm]

3*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	2.48	22443	516166	$1{,}48\times10^{-2}$	203	2601	-1	-1	-1	-1	-1	-1
i_02	13.83	120267	2766118	$7{,}34\times10^{-2}$	894	13088	-1	-1	-1	-1	-1	-1
i_03	-1	-1	-1	1.09	9951	188117	-1	-1	-1	-1	-1	-1
i_04	304.89	1256462	28898603	1.40	11734	208730	-1	-1	-1	-1	-1	-1
i_05	-1	-1	-1	11.12	83166	1316617	-1	-1	-1	-1	-1	-1
i_06	$3{,}92\times10^{-3}$	27	1014	$1{,}70\times10^{-4}$	1	17	-1	-1	-1	-1	-1	-1
i_07	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
i_08	-1	-1	-1	20.04	111704	1597354	-1	-1	-1	-1	-1	-1
i_09	-1	-1	-1	2.54	11955	354134	-1	-1	-1	-1	-1	-1
i_10	-1	-1	-1	11.35	67339	1108059	-1	-1	-1	-1	-1	-1

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)**

This problem's analysis yielded two main assumptions : that the breadth first strategy would be optimal in order to find (on average) the shortest solution as fast as possible, and that the graph method would be preferred because of how it handles symmetrical states, which are omnipresent in this problem.

As expected, the results prove those assumptions correct : while a drastic performance improvement can be observed when switching from tree to graph with breadth-first, showing how our problem is better represented as a grpah than as a tree, the difference is even more remarkable when switching from breadth-first to depth-first : with the depth-first strategy, no solutions were ever found without a memory overflow or a timeout, showing how unsuited it is for this problem.

   (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 BFS with graph is clearly the more promising algorithm. We can see that it is optimal both in terms of time and memory usage as it takes less time and explores less nodes to find the shortest solution.

A few improvements that come to mind :

— Deleting useless moves : the opposite action could always be substracted from the next state's possible action : for example, if the last state (on 4x4 grid) was obtained with a down-1 move on the 0th column, this state's possible actions could be the whole set of actions except for the down-3 move on the 0th column.

— Concatenating similar moves : since at each step every possible move creates a new state, doing three times in a row a right-1 move on the 2nd row is the same as doing a right-3 move on that column. We could suppress some moves of each state's possible moves with this in mind.