LINGI2261 : Artificial Intelligence

Professor : Yves Deville
Teaching assistants : Cyrille Dejemeppe and Jean-Baptiste Mairy

Assignement1 : Puzzle Sliding Problem

UCL

Université Catholique de Louvain

Groupe 37

Florentin Rochet
Léonard Debroux

2012-2013

1 Python AIMA

1.1 In order to perform a search, what are the classes that you must define or extend ? What are they used for ?

We have to create a subclass to the class "Problem" given in the AIMA library. This subclass must implement the method successor, and possibly *___init___*, *goal_test*, and *path_cost*. When this is done, we can use the search method on an instance of this subclass.

1.2 In graph_search and tree_search, what is the effect of the instruction fringe.extend(node.expand(problem)). What are the classes and methods involved ?

The method extend is used on a FIFOQueue to add elements at the end of this queue. These elements are the elements returned successively through the yield by node.expand(problem).
The role of the expand method defined in the class Node is in term to reach all the children nodes accessible from one particular node.
The classes involved are FIFOQueue and more precisely the method extend, the class Node to access the method expand and a Problem's subclass.

1.3 Both breadth_first_graph and depth_first_search are making a call to the same function. How s their fundamental difference implemented

The differences lies in the way that fringe is constructed. If we have a FIFOQueue, then the new elements are put in the end of the list, which involve a breadth first traversal. At the opposite, if we have a stack, the traversal will be depth first search because tree_search fill the stack with children nodes to the current. Thus pop call return the child, until the depth one.

1.4 How is the closed list implemented in graph_search ? What is it used for ? What is the technical difference with the fringe ? Why are those two specific structures used ?

The closest list contains True or False (nothing is False) associated with keys which are the node's state. It's used to keep a trace of which node are already explored. The technical difference with fringe is that fringe contains node which are already been explored or unexplored and the closed list is a Dictionary that contains True value for nodes that already been explored.
A specific structure is used for fringe to allow breadth first graph search or a depth first graph search. And closed is a Dictionary which map each node to a boolean value in order to know if they are explored or not.

1.5 How technically are the elements in the closed list searched ? What are the methods involved ? What properties must its elements have ?

They are searched by the instruction "if node.state not in closed" which is the same that "if node.state not in closed.keys()" which search with an iterator among the keys if node.state isn't among them. Implicitly, we think that the methods involved are the method ___iter___ for the iteration, the method keys() that returns the set of keys and the method ___hash___ and ___eq___ of the object node.state that is given as a key to the dictionary. They are handle by Python.
If we use self made objects as keys for the dictionary, we have to redefine the ___hash___ method and the ___eq___ method. This are the properties that the elements must have.

1.6 How technically can you use the implementation of the closed list to deal with symmetrical states ? (hint : if two symmetrical states are considered to be the same, they will not be visited twice)

To deal with symmetrical state, we must have the same hash value for the symmetrical ones. In this situation, the dictionary will perfectly handle the closed list. So we have to choose a way to represent a state and this representation must be the same for symmetrical state. This is this representation that will be hashed to obtain our hash value.

2 The Sliding Puzzle Problem

2.1 Explain the advantages and weaknesses of each search strategy on this problem (depth first, breadth first, depth limited, iterative deepening, uniform cost). Which one would you choose to solve this problem ?

- **depth first** The main advantage of using a depth first search is its space complexity which is better than other because the algorithm is such as only one path must be in memory at a given time. Though, if the tree is infinite, the path also becomes infinite and thus, it is possible to never find the goal and a major part of the tree is unexplored.
- **breadth first** While using this approach, we ensure that we will find the shallowest goal in our tree (which is the one that requires the less actions) but at the cost that all the different states generated during the execution are kept in memory which leads to an exponential space complexity.
- **depth limited** This one acts like the depth first algorithm, though it gets rid of the infinite path problem by setting a maximum depth. It is then possible to explore all the nodes up to a certain depth. Sadly it only works well when we have a good idea of the depth of the goal or at least an upper bound.
- **iterative deepening** The strategy of this one is to perform a search on each level, one after the previous one, using the depth first method, which is following one path at the time. It keeps the advantages of the depth limited way without the disadvantage that we don't know the depth of the goal. The inconvenient of using this method is that it generate slightly more node than the breadth first method.
- **uniform cost** It is a nice way to proceed when some actions are more costly than others. It is basically a breadth first search, although instead of a FIFO Queue, a priority queue is used. While convenient when costs are involved, this method is expensive in space as well as in time.

To solve the given problem, the best approach seems to be the breadth first search because we want the solution using the fewer steps and because calculating states and comparing them is expensive. That's why we won't go for the iterative deepening although it was our first choice because it generates more nodes and performs more comparisons. And of course, we'll use the version of the BFS that uses a graph search because we want to avoid symmetrical states in our search.

2.2 Are there symmetrical (equivalent) states in this problem ? What are the potential consequences on the search ?

There may be many situations where symmetrical states occur. The consequences of this in the treatment of this problem is that it maylead to building several times the same subtree and explore it as many times. It would constitute a useless usage of memory and a loss of time considering it generates and analyses the redundant states before reaching the goal state.

2.3 How are you dealing with those symmetrical states ?

One way to deal with those symmetrical states is to keep track of the states already generated and analysed and to avoid to analyse its "twins". It is what performs the graph search compared

to the tree search.
In our implementation, to deal with symmetrical states, we can define an object State which represent a configuration of the puzzle in matrix. To handle symmetrical state, we convert this representation in a new one which will match correctly any twins states.

2.4 What are the advantages and disadvantages of using the tree and graph search for this problem. Which approach would you choose ? Which approach allows you to avoid expending twice symmetrical states ?

As said above, the graph search presents an advantage in comparison to the tree search which treats symmetrical states as different ones. The approach that we shall use is the graph search via the BFS because the redundancy would cause the computing time to explode as will show the tests later on. The graph search is the one of the two that can avoid treating symmetrical states because it tests if the state has already been generated earlier.

2.5 Implement this problem in Python3. Extend the Problem class and implement the necessary methods and other class(es) if necessary. Your file must be named puzzle.py. Your program must print to the standard output a solution to the puzzle satisfying the above format

For the implementation of the solver, we extended the problem class and created three others which are IO, State and WrongDirectionException. The IO is a generic class and has for goal to read and write files while the State class implements an state of the puzzle with all the useful methods. Comments inside the code will help to understand its design.

2.6 Experiments must be realized with the 10 instances of the puzzle problem provided. 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). You must report the time, the number of explored nodes and the number of steps from root to solution. When no solution can be found by a strategy in a reasonable time (3 min), explain the reason (time-out and/or swap of the memory

BFS Graph	Time [s]	Steps	Nodes
init1	10.22	113	80555
init2	0.55	28	5055
init3	9.24	47	77953
init4	13.71	46	111405
init5	27.80	57	222467
init6	10.32	47	82807
init7	25.08	65	195831
init8	10.27	117	81558
init9	13.12	139	101548
init10	15.46	159	121141

BFS Tree	Time [s]	Steps	Nodes
init1	/	/	/
init2	/	/	/
init3	/	/	/
init4	/	/	/
init5	/	/	/
init6	/	/	/
init7	/	/	/
init8	/	/	/
init9	/	/	/
init10	/	/	/

We see that when using BFS with a graph, a solution is always found though with a tree, it never happens. The executions were stopped after 3min without result and we also noticed that the space taken in memory didn't cease to increase (about 3Gb in 3min). We didn't try to fill completly the memory in order to see what would happen. It would probably start to swap with the hdd.

DF Graph	Time [s]	Steps	Nodes
init1	3.36	4192	18692
init2	0.46	626	2320
init3	0.87	1566	3307
init4	3.55	6496	14244
init5	1.75	2991	7747
init6	1.75	3194	7440
init7	2.53	4148	11798
init8	3.36	4122	19024
init9	1.30	2166	6006
init10	1.06	1828	4836

DF Tree	Time [s]	Steps	Nodes
init1	/	/	/
init2	/	/	/
init3	/	/	/
init4	/	/	/
init5	/	/	/
init6	/	/	/
init7	/	/	/
init8	/	/	/
init9	/	/	/
init10	/	/	/

We observe the same result as for the BFS that is the tree search return nothing and fill the memory (after 3 min, 3Gb were used).
We also see that the depth first search is faster but give a solution that involves thousands of actions which is not appropriate to our problem.