Using Symbolic Search for solving N-Puzzle problem

Introduction

The goal of this practical is to design, implement and analyze the behavior of various AI tree-based Search strategies, both informed and uninformed, for solving 8-puzzle problem. The essence of the problem is to find the cheapest cost path between two connectable 8-puzzle configurations.

1. Instructions (how to run the program and interpret I/O)

The program is interactive. It asks the user to enter the start 8-puzzle configuration and the goal 8-puzzle configuration. The 8-puzzle configuration must me entered as a single line (for example, 734528610). '0' stands for the empty tile. After that the program asks to select the heuristic function that it will use (for example, '1' stands for Manhattan distance) and the search strategy (for example, '4' stands for A*). Image 1 illustrates the interaction window:

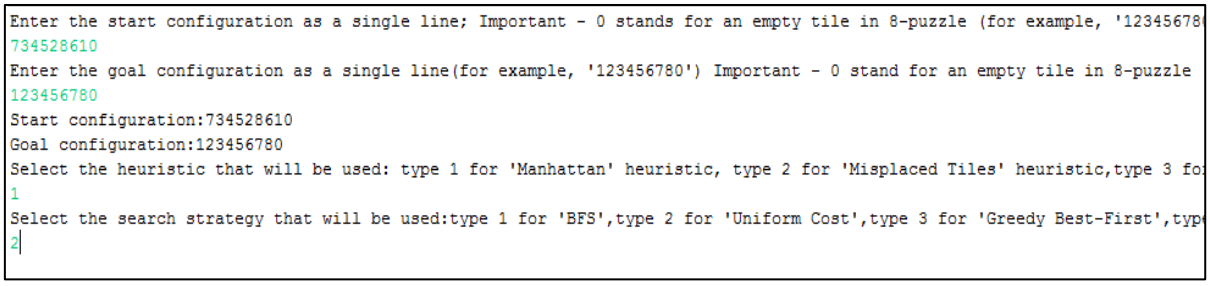

Image 1. I/O window

To run the program, enter the following command: "java –jar Symbolic.jar".

1. Breadth-First Search (BFS)

1.1. Description

BFS is the uninformed search strategy, i.e. the strategy which doesn't have any additional information about the nodes in the tree. For 8-puzzle problem, uninformed search doesn't know about how close the current state is to the goal state. So any uninformed strategy can only blindly expand the nodes in a particular order to get the successors and check if the current state is equal to the goal state.

BFS expands its nodes level by level. This means that the strategy cannot expand nodes at level *n+1* until all the nodes at level *n* have not been expanded (Image 2).

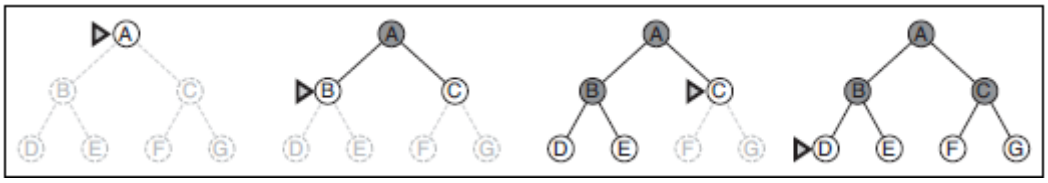

Image 2. Order of nodes' expansion in BFS

How fast and memory-efficient is BFS? The time complexity as well as the memory space needed is $O(b^{d+1})$, where b is the branching factor and d is the depth of the graph. BFS is complete – the algorithm will eventually find the solution if it exists. As for optimality, it is optimal "if the path cost is a nondecreasing function of the depth of the node" [1].

1.2. Design and Implementation

The implementation deviates from the design proposed by "Practical 3 Part A" handout. The reason for that was to provide extensible and flexible design in object-oriented style. There are several general classes that are used not only for BFS but for other implemented search strategies as well:

1. *Node* class represents the 8-puzzle state in a search tree. Each node has a link to his parent node – this helps to construct the solution path. Each node has a *level* attribute which shows how deep in the tree the node is located. The *level* of the found solution node is the depth of the solution.
2. *Problem* class represents the 8-puzzle problem itself – among other attributes it contains the start configuration and the goal configuration.

BfsStrategy class is the implementation of the BFS search algorithm for 8-puzzle problem. To ensure the correct order of the nodes for expansion, the *Queue* data structure is used. Once the node is expanded, all the children are put into the end of the queue. At each iteration, the first element of the queue is retrieved for expansion. This mechanism guarantees that the tree is traversed level by level. For BFS, once the node is expanded, the children states are checked for the equality with the goal configuration. If any child is equal to the goal configuration, the algorithm terminates.

2. Generalised Search

2.1. Introduction

In this section I'm going to discuss how my solution allows to add new search strategies without writing tons of new code. Also, the comparison of different search strategies in the context of 8-puzzle problem will be carried out.

2.2. Other Search Strategies

To support other search strategies, including Uniform-cost, Greedy best-first and A*, the abstract class *Strategy* is used. *Strategy* has a *Problem* and *Heuristic* (evaluation function). Besides, it stores the solution path and statistic attributes such as depth of a solution, overall number of expanded nodes and number of expanded nodes per level. Each implementation of search strategy inherits from *Strategy* class, which enables to change evaluation function (each evaluation function implements *Heuristic* interface) easily, without code duplication.

Although all implemented search strategies are algorithmically similar, the major difference between them is how they select the next node for expansion:

1) *Uniform-cost* (*UniformCostStrategy* class)

This is uninformed strategy – it takes into account only the past cost of a path when evaluating the cost in the node: $f(n) = g(n)$, where *g(n)* is the cheapest path from the root to *n*. At each iteration, the node with the minimum cost is selected for expansion. To support this mechanism *PriorityQueue* data structure is utilized. Uniform-cost algorithm is complete if the branching factor is finite. Uniform-cost is optimal.

2) *Greedy best-first* (*GreedyStrategy* class)
   This strategy tries to estimate which of the possible nodes to expand is the closest to the goal: $f(n) = h(n)$, where *h(n)* is the heuristic function that estimates the cheapest path from node *n* to the goal. As in the Uniform-cost, *PriorityQueue* is used to store yet unexpanded nodes in the cost-ascending order. Greedy best-first is neither complete nor optimal.
3) *A** (*AStarStrategy* class)
   This search strategy tries to minimize the total estimated solution cost by combining *g(n)* and *h(n)*: $f(n) = g(n) + h(n)$. Thus, "*f(n)* is the estimated cost of the cheapest solution through *n*" [1]. A* tree-search algorithm is complete. It is also optimal if the heuristic used is admissible which means that it never overestimates the cost to reach the goal [1].

2.3. Heuristic functions

Three heuristics were implemented for this Practical to estimate the path cost from the node *n* to the goal:

1) *Manhattan distance* (*Manhattan* class)
   For 8-puzzle state *s*, where x_i and y_i are the *x* and *y* coordinates of the tile *i* and x_g and y_g are the *x* and *y* coordinates of the goal tile, Manhattan distance is calculated the following way:

$$h(s) = \sum_{i=1}^{8} dist = |x_g - x_i| + |y_g - y_i|$$

2) *Misplaced tiles* (*MisplacedTiles* class)
   This heuristic counts the number of tiles that are currently located in the wrong position in 8-puzzle.
3) *Out of row and out of column* (*OutOfRow* class)
   It counts the number of tiles out of row plus the number of tiles out of column.

3. Evaluation

When the search algorithm terminates, performance data can be retrieved. This includes the depth of the solution path, a number of expanded nodes per level and the number of expanded nodes overall. Additionally, the solution path is printed out to the Console.

Ten test problems (various start and goal 8-puzzle configurations) were created to analyze and compare the behavior of various search strategies. Table 1 illustrates the performance of all search strategies, assuming that evaluation function uses Manhattan distance heuristic.

№	BFS	Uniform	Greedy	A*

	Exp. Nodes	Solution Depth	Exp. Nodes	Solution Depth	Exp. Nodes	Solution Depth	Exp. Nodes	Solution Depth
1	68415	23	85744	23	251	93	11475	23
2	10	5	27	5	4	5	10	5
3	2455	15	5464	15	254	79	480	15
4	157799	27	166115	27	489	81	33747	27
5	137327	26	156549	26	357	42	32325	30
6	181411	32	181440	32	70	40	74106	32
7	34	6	46	6	5	6	15	6
8	9	4	16	4	4	4	6	4
9	58	7	85	7	7	7	21	7
10	30	6	52	6	5	6	15	6

Table 1. Performance results of four search strategies using “Manhattan distance” heuristic

As for solution depth, Greedy search stands out by finding very deep solutions. That can be explained by the fact that Greedy best-first is based on future cost only and pays no attention to past cost. BFS, Uniform and A* showed approximately the same depth results.

As for the total amount of expanded nodes, Greedy search is much more efficient than three others. Then comes A*, which, in its turn, significantly outperforms BFS and Uniform-cost.

The same tendency remains if search strategies switch to using “Misplaced Tiles” heuristic (Table 2) and “Out of Row and Out of Column” heuristic (Table 3).

№	BFS		Uniform		Greedy		A*	
	Exp. Nodes	Solution Depth	Exp. Nodes	Solution Depth	Exp. Nodes	Solution Depth	Exp. Nodes	Solution Depth
1	68415	23	85744	23	934	103	42728	25
2	10	5	27	5	4	5	11	5
3	2455	15	5464	15	205	45	1330	15
4	157799	27	166115	27	47	47	106870	27
5	137327	26	156549	26	343	64	91585	26
6	181411	32	181440	32	375	112	173601	32
7	34	6	46	6	6	6	17	6
8	9	4	16	4	4	4	6	4
9	58	7	85	7	164	59	30	7
10	30	6	52	6	5	6	17	6

Table 2. Performance results of four search strategies using “Misplaced Tiles” heuristic

№	BFS		Uniform		Greedy		A*	
	Exp. Nodes	Solution Depth	Exp. Nodes	Solution Depth	Exp. Nodes	Solution Depth	Exp. Nodes	Solution Depth
1	68415	23	85744	23	883	107	27856	23

2	10	5	27	5	4	5	10	5
3	2455	15	5464	15	192	55	637	15
4	157799	27	166115	27	368	107	71463	27
5	137327	26	156549	26	199	74	70087	30
6	181411	32	181440	32	115	54	121996	32
7	34	6	46	6	5	6	15	6
8	9	4	16	4	4	4	6	4
9	58	7	85	7	8	7	27	7
10	30	6	52	6	5	6	15	6

Table 3. Performance results of four search strategies using “Out of Row and Out of Column” heuristic

Now, let’s compare relative performance of search strategies as the evaluation function changes. Changing of evaluation function doesn’t affect uninformed strategies – BFS and Uniform-cost. However, it affects informed strategies. Table 4 shows the joint results for A*:

№	Manhattan		Out of Row and Out of Column		Misplaced Tiles	
	Exp. Nodes	Solution Depth	Exp. Nodes	Solution Depth	Exp. Nodes	Solution Depth
1	11475	23	27856	23	42728	25
2	10	5	10	5	11	5
3	480	15	637	15	1330	15
4	33747	27	71463	27	106870	27
5	32325	30	70087	30	91585	26
6	74106	32	121996	32	173601	32
7	15	6	15	6	17	6
8	6	4	6	4	6	4
9	21	7	27	7	30	7
10	15	6	15	6	17	6

Table 4. Performance of A* using various heuristic functions

The number of expanded nodes vividly shows that “Manhattan distance” is the strongest heuristic among those three; then comes “Out of Row and Out of Column” and “Misplaced Tiles” is the weakest. This result is predictable – “Manhattan distance” is theoretically supposed to be more precise in estimating the distance between two 8-puzzle states than other two heuristic functions. “Out of Row and Out of Column” is stronger than “Misplaced Tiles” because it is the extension of “Misplaced Tiles” heuristic and is supposed to be more precise.

The similar table for Greedy search is ignored as the tendency there remains the same so it only confirms previously made conclusions.

4. Unsolvable configurations

It’s important to mention that the unsolvable 8-puzzle configurations exist. It’s not a focus of this work to go into mathematical background of this phenomenon. Briefly speaking, the 8-puzzle configuration is unsolvable if the number of inversions is odd. If my program detects that 8-puzzle is unsolvable, it prints out the respective message to the Console.

Conclusion

To sum up, two informed and two uninformed strategies were implemented to solve 8-puzzle problem. The details of design and implementation of the solution were explained. The proper evaluation of results was conducted to compare the performance of search strategies.

References

[1] Russel S., Norwig P., 2010. Artificial Intelligence:A Modern Approach, Third Edition. Pearson Education 2010, pp. 64-108.