UM–SJTU Joint Institute

Fall 2018 VE593 Problem Solving with AI Techniques

Project 1 Report

Xinhao Liao

516370910037

Contents

1 Part 1. Search Algorithms **3**

1.1 Implementation and testing details 3

1.2 Test 1: The real random initial state test 3

1.3 Test 2: Test with random initial states within limited steps 4

2 Part 2. Clickomania **5**

2.1 Algorithm decision 5

3 Part 2. Appendix **5**

1 Part 1. Search Algorithms

1.1 Implementation and testing details

The implementation of the search algorithms are given in *search.py*. All the approaches except the Monte Carlo Tree Search are partly based on the *Graph-search* algorithm. *Graph-search* guarantees that a visited state will never be regenerated again, but much memory is required. To trade-off the memory cost with the repeatability, the *Graph-search* is slightly modified. Only 50000 most recently visited states are stored in memory to be checked with.

The test graphs are slightly modified and given in *testgraphs.py*. The *isGoal* function of *nPuzzlegraph* is modified to apply for all possible odd number n. And the *successors* functions are modified to have the same type of returned result no matter whether the graph is valued or not

Two testing programs are given in *testtime.py* and *testtime2.py*. The code in *testtime.py* conforms to the original instructions for generating random solvable nPuzzle graphs and evaluate the cost of time. And the code in *testtime2.py* generates a random state within limited steps from the goal state.

1.2 Test 1: The real random initial state test

The original test randomly generate solvable initial state of nPuzzle graphs. Every method is tested with 5 initial states. Once a search approach fails to reach the goal within the limited time (and within limited 20 depth for DLS), this approach will be skipped in the test.

Unfortunately, the program fails to collect enough data for a plot of any method. Only some data for $n = 3$ are able to be collected within the time limit. When n reaches 5, all the approaches fail to find out the answer within an hour. The data collected are given in the table below.

Method	1	2	3	4	5	Average
BFS	110.931s	1157.878s	157.257s	224.425s	143.705s	358.839s
UCS	128.532s	1433.301s	190.727s	274.099s	177.538s	440.839s
DFS	NA	skipped	skipped	skipped	skipped	NA
DLS	NA	skipped	skipped	skipped	skipped	NA
IDS	8.611s	4.165s	2.867s	1.008s	5.549s	4.440s
A*	1.590s	28.473s	1.749s	2.388s	0.733s	6.987s
MCTS	2123.232s	814.542s	1428.519s	188.902s	317.615s	976.362s

Table 1: Result for sorting with different algorithms.

This is reasonable since the number of possible states grow sharply when n increases. When $n = 3$, there are $9! = 362880$ random states. Among them, half of them are solvable with even number of inversions. That is 181440 possible initial states. That's a huge number of states to check. And when n increases to 5, there will be $25!/2 = 7.755605022 \times 10^{24}$ possible initial states, which is 4.274×10^{19} times of that when $n = 3$. It requires a lot of time even though only part of these states need to be checked before the goal is reached.

1.3 Test 2: Test with random initial states within limited steps

To help increase the possibility of finding the goal state in the limited time, a different method is used to generate the random solvable initial state. In this test, the initial state is generated by letting the goal state randomly move limited steps. Then at least for the optimal solution, its depth is limited. But the generated state in this way is more likely to have lower depth. The probability for generating various states are not equal.

Actually, this is not fair for some approaches. Since the generated initial states are more likely to be close to the goal state, BFS and UCS, which search in the order of the step and path cost, are more likely to find out the answer.

The limit of number of random moves from the goal is set to be 20. The depth limit of DLS is also set to be 20, since there is at least a solution within 20 steps. In this way, DLS is more likely to find the solution. And the only possible reason that DLS fails should be some time cost exceeds 1 hour or the average time cost exceeds 30 minutes.

Method	n=3	n=5	n=7	n=9	n=11
BFS	0.0137s	369.0262s	NA	NA	NA
UCS	0.0157s	464.8742s	NA	NA	NA
DFS	NA	NA	NA	NA	NA
DLS	0.0752s	157.1175s	NA	NA	NA
IDS	0.0060s	0.9785s	18.6813s	55.6719s	115.1938s
A*	0.0011s	0.0100s	0.0331s	0.5681s	2.5726s
MCTS	908.5848s	NA	NA	NA	NA

Table 2: Average computation time of different approaches in Test 2.

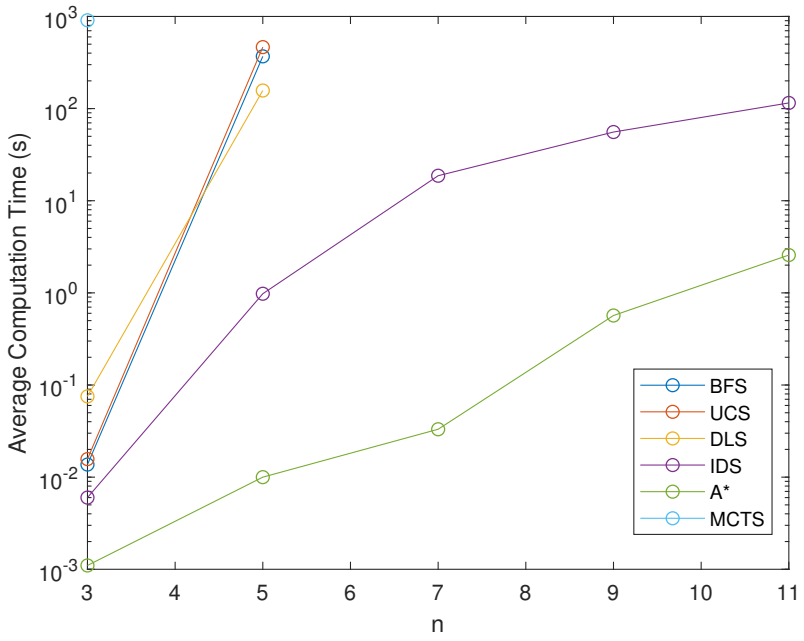

Figure 1: Plot of results in Test 2.

As we can see from the plot above, A* has the best performance in this game, and second best one is IDS. The max number of odd n for BFS, UCS, and DLS are all 5. And that number for MCTS is only 3. And for DFS, it fails to find within the limited time even when $n = 3$.

BFS and UCS has similar performance. This is resonable since for this game, they both search in the order of the path cost, with any step costing 1.

The reason why DFS fails may be that, the maximum depth of this game is super big. It is hard for DFS to turn backward and search among the nodes with smaller depth. But we know that at least a solution is within only the depth of 20. It's harder for DFS to find that solution than for other approaches.

As for DLS, alothough there must be a solution within its depth limit in this test, there are more and more possible states within that depth limit when n increases. The time cost finally exceeds the time limit.

The reason why MCTS has bad performance may be that, its default reward computation is not efficient for the nPuzzlegraph. With so many possible states, it hard to find the goal state in a random simulation. So the reward will often be 0 after a simulation.

2 Part 2. Clickomania

2.1 Algorithm decision

For this game, notice that the goal is not a particular state. Our goal is to reach a state with as high score as possible. The time and path cost is not so important.

Since every state corresponds to a score, it's natural to consider using the Monte Carlo Tree Search since this approach chooses the next state with the highest score based on simulation.

And it's clear that breadth-first search, uniform-cost search, depth-first search, depth-limited search, and iterative-deepening search are not good choices. The reason is that we are not supposed to find a particular goal state or a random goal state, but a goal state with maximized score. These approaches can find a state that terminates the searching, maybe with limited or minimized time, but the score can be very low.

As for greedy-best first search and A star algorithm, they are originally supposed to find a goal state with less path cost. We may somehow modify them to make them search for higher score, with some good heuristic function. But that seems to be not easy, and the heuristic function needs to be carefully designed.

To optimize the result and trade-off with the time cost limitation, we may adjust the budget.

3 Part 2. Appendix

The implemantation of search approaches are in *search.py*, and the graphs for testing are in *testgraphs.py*. The testing program for Test 1 and Test 2 are in *testtime.py* and *testtime2.py*.

For the game of clickomania, besides the *clickomania.py* and *clickomaniaplayer.py*, a test program is given in *clickomaniatest.py*.