COMP2611

Artificial Intelligence

Assignment 1: Search Algorithms

Jinqi Wang — sc22jw3@leeds.ac.uk (Submitter)

Xinyue Wu — sc222xw[@leeds.ac.uk(Submitter)](mailto:sc21kh@leeds.ac.uk(Submitter))

**Declaration A.**

We confirm that we have worked as pair on this project and both of us have made significant contributions to both parts of the assignment.

We are aware that both members of the pair will receive the same grade.

The submitter confirms that they have agreed the final submitted version of this report with the other member of the pair.

The submitter confirms that, after submitting the report to Gradescope, they have added the other member of the pair to the group associated with the submission.

1. **Sliding Blocks Puzzle Search Investigation**

**A1(a) Puzzle Test Cases**

We determine the difficulty of the puzzle according to the size of the puzzle, the number of colors, the number of blocks in initial state and in the goal. After some experimentation we decided to investigate the following cases of Sliding Block Puzzle:

|  |  |  |  |
| --- | --- | --- | --- |
|  | easy\_puzzle | middle\_puzzle | Hard\_puzzle |
| Initial State | IMG_256 | IMG_256 | IMG_256 |
| Goal | IMG_256 | IMG_256 | IMG_256 |

**A1(b) Heuristics**

We designed the following two heuristics for our investigation and testing:

**Preprocessing Step**

First, we preprocessed the data by grouping blocks of the same color and determining their geometric centers. These geometric centers serve as anchor points for heuristic calculations.

**Manhattan Distance**

The Manhattan Distance heuristic calculates the sum of horizontal and vertical distances between each block and its target position, and the formula is shown below.

|  |
| --- |
| *pseudo-code of calculating Manhattan Distance：* |
| *def manhattan\_heuristic(state, goal\_anchors):*  *initialize total\_distance to 0*  *compute state\_anchors from the current state*  *for each block in state\_anchors:*  *if block exists in goal\_anchors:*  *total\_distance += |p1.row - p2.row| + |p1.col - p2.col|*  *return total\_distance* |

**Euclidean Distance (Straight-Line Distance)**

The Euclidean Distance heuristic calculates the direct straight-line distance between a block and its target position, and the formula is shown below.

|  |
| --- |
| *pseudo-code of calculating Euclidean Distance：* |
| *def straight\_line\_distance(state, goal\_anchors):*  *initialize total\_distance to 0*  *compute state\_anchors from the current state*  *for each block in state\_anchors:*  *if block exists in goal\_anchors:*  *total\_distance += sqrt((p1.row - p2.row)² + (p1.col - p2.col)²)*  *return total\_distance* |

This approach ensures efficient heuristic calculations based on geometric centers, improving accuracy while maintaining computational efficiency.

**A1(c) Search Algorithm Test Sequence**

After experimenting with various search options we found that the following sequence of tests gives an informative set of statistics regarding the performance of a wide range of search algorithms and options.

First, we specify the initial and goal states and use these to create an instance of SlidingBlocksPuzzle.

|  |
| --- |
| *Specify the initial and goal states and create an instance of SlidingBlocksPuzzle：* |
| *initial\_state = [*  *[1, 3, 0, 0, 0, 0, 5, 5],*  *[1, 3, 4, 4, 0, 0, 6, 5],*  *[3, 3, 4, 0, 8, 8, 6, 6],*  *[0, 4, 4, 0, 0, 0, 0, 0],*  *[0, 0, 0, 0, 0, 3, 0, 0],*  *[0, 5, 2, 2, 3, 3, 3, 0],*  *[5, 5, 5, 2, 0, 0, 0, 7],*  *[0, 0, 0, 2, 0, 0, 0, 7]*  *]*  *goal\_state = [*  *[7, 6, 0, 0, 0, 0, 8, 8],*  *[7, 6, 6, 0, 0, 0, 0, 0],*  *[0, 0, 0, 0, 0, 0, 0, 0],*  *[0, 0, 0, 0, 0, 0, 0, 0],*  *[0, 0, 0, 0, 0, 0, 0, 0],*  *[0, 0, 0, 0, 0, 0, 0, 0],*  *[0, 0, 0, 0, 0, 0, 0, 1],*  *[0, 0, 0, 0, 0, 0, 0, 1]*  *]*  *puzzle = SlidingBlocksPuzzle(initial\_state,goal\_state)* |

Then, search the puzzle through different ways by the function:

|  |
| --- |
| *Search function：* |
| *search(problem, mode, max\_nodes, loop\_check=False, randomise=False, cost=None, heuristic=None, show\_path=True, show\_state\_path=False, dots=True, return\_info=False)* |

We adjust the search method by adjusting the parameters, and show the search process and time. Some example test cases are shown below.

|  |
| --- |
| *Depth-First(Random Action Choice Order：* |
| *search(puzzle, 'DF/LIFO', 10000000, loop\_check=True, randomise=False, show\_state\_path=False, return\_info=True)* |

|  |
| --- |
| *A\*(Euclidean Distance)：* |
| *search( puzzle, 'BF/FIFO', 10000000, heuristic = straight\_line\_distance, loop\_check=True, randomise=False,cost=thecost, show\_state\_path=False, return\_info=True)* |

**A2. Results**

**Test result for simple\_puzzle (loop=True)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Find solution | Path length | Total nodes tested | Time taken(s) |
| Breadth-First | Success | 8 | 185 | 0.6818 |
| Depth-First(Fixed Order) | Success | 26 | 33 | 1.4006 |
| Depth-First(Random Order) | Success | 16 | 19 | 1.2411 |
| Best First(Manhattan) | Success | 8 | 45 | 0.4096 |
| Best First(Euclidean) | Success | 8 | 45 | 0.4945 |
| A\*(Manhattan) | Success | 8 | 55 | 0.532 |
| A\*(Euclidean) | Success | 8 | 71 | 0.4633 |

**Test result for simple\_puzzle (loop=False)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Find solution | Path length | Total nodes tested | Time taken(s) |
| Breadth-First | Success | 8 | 118659 | 30.4198 |
| Depth-First(Fixed Order) | Failed |  |  |  |
| Depth-First(Random Order) | Success | 1099 | 1100 | 49.8496 |
| Best First(Manhattan) | Success | 8 | 798 | 0.5972 |
| Best First(Euclidean) | Success | 8 | 798 | 0.9699 |
| A\*(Manhattan) | Success | 8 | 2938 | 1.2309 |
| A\*(Euclidean) | Success | 8 | 3122 | 1.2316 |

**Test result for middle\_puzzle (loop=True)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Find solution | Path length | Total nodes tested | Time taken(s) |
| Breadth-First | Success | 32 | 56332 | 15.9252 |
| Depth-First(Fixed Order) | Success | 12399 | 14673 | 11.5494 |
| Depth-First(Random Order) | Success | 4851 | 5409 | 3.1426 |
| Best First(Manhattan) | Success | 45 | 274 | 3.1357 |
| Best First(Euclidean) | Success | 59 | 425 | 3.8042 |
| A\*(Manhattan) | Success | 36 | 961 | 2.5587 |
| A\*(Euclidean) | Success | 35 | 4763 | 3.5421 |

For the test of middle\_puzzle when loop check is False, all tests failed.

**Test result for hard\_puzzle (loop=True)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Find solution | Path length | Total nodes tested | Time taken(s) |
| Breadth-First | Failed |  |  |  |
| Depth-First(Fixed Order) | Failed |  |  |  |
| Depth-First(Random Order) |  |  |  |  |
| Best First(Manhattan) | Success | 89 | 12666 | 26.1342 |
| Best First(Euclidean) | Success | 111 | 6963 | 17.5528 |
| A\*(Manhattan) | Success | 62 | 6066 | 9.5629 |
| A\*(Euclidean) | Success | 51 | 2334 | 8.4967 |

**Test result for hard\_puzzle (loop=False)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Find solution | Path length | Total nodes tested | Time taken(s) |
| Breadth-First |  |  |  |  |
| Depth-First(Fixed Order) |  |  |  |  |
| Depth-First(Random Order) |  |  |  |  |
| Best First(Manhattan) |  |  |  |  |
| Best First(Euclidean) |  |  |  |  |
| A\*(Manhattan) |  |  |  |  |
| A\*(Euclidean) |  |  |  |  |

**A3. Observations**

大于等于8个

Different basic search algorithms比较

两种启发式算法比较

Loop check 有无的区别

Puzzle的难易程度

Random Action Choice Order

1 BFS path 短，node多

2 DFS Fixed Action Ordering

3 A\* search, when paired with an appropriate heuristic (like Manhattan distance), outperforms uninformed algorithms (such as BFS and DFS) in terms of both efficiency and solution quality.

4 欧几里得距离（Euclidean Distance）：计算每个块当前的位置与目标位置之间的直线距离。比曼哈顿距离更精确，但计算开销较大。

5 Loop check

6 启发式函数与代价（cost）分配比例

7 Puzzle越大、初始状态和目标状态中的方块数量越多、相同滑块在初始状态和目标状态中距离越远，谜题的难度就更大

8 面对puzzle难度的增加，每个算法得到的path，生成的节点和花费的时间都显著增加

9 面对复杂puzzle，BFS和DFS会因为超时而无法产生结果。但基于启发式的搜索（best first , A\*,）可以成功搜索