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) | Failed |  |  |  |
| 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 |

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

**A3. Observations**

After examining our results, we gained deep understanding of search algorithms. Of the many

interesting observations we made, the most flabbergasting were as follows:

1. Although BFS finds the shortest path with less time and fewer moves, it tests more total nodes compared to DFS. This is because BFS explores all nodes level by level, requiring the storage of all explored nodes, leading to high memory consumption.
2. DFS with fixed action ordering getting stuck in deep paths and failing to find a solution as it tries to go as deep as possible in one branch before backtracking. Our results showed that while DFS with fixed ordering failed in some cases, DFS with random action ordering succeeded. Randomization helps explore different paths, and overall, it took less time compared to the fixed order.
3. Informed search with an appropriate heuristic, outperforms uninformed algorithms (such as BFS and DFS) in terms of both efficiency and solution quality. It significantly reduces the number of explored nodes by prioritizing nodes based on a combination of the actual cost to reach them and the estimated cost to reach the goal.
4. Euclidean distance is more accurate than Manhattan distance but comes with higher computational cost. In low difficulty tests, Euclidean distance often results in more nodes being tested, as its precision doesn’t provide significant advantages for simpler problems. However, in high difficulty tests, it typically tests fewer nodes, as its greater accuracy helps guide the search more efficiently, despite the higher computational cost.
5. Loop checking prevents revisiting explored states, improving efficiency and reducing redundancy. Our results show that enabling it significantly increases the success rate of searches while reducing the total nodes tested and computation time.
6. The balance between the cost and heuristic in A\* search algorithms is essential in determining the search strategy. If the heuristic is given too much weight compared to the cost, the search may prioritize estimated distance over actual cost, resulting in suboptimal paths. Conversely, if the cost is emphasized more, the search can become too greedy and inefficient. Through experimentation, we found a balanced ratio that worked well and provided faster results.
7. The larger the puzzle, the greater the number of tiles, and the farther the tiles need to be moved from their initial to their goal positions, the more difficult the puzzle becomes.
8. As puzzle complexity increases, the path length, number of nodes tested, and computation time generally increase for most algorithms. However, for A\* based on Euclidean distance, the number of tested nodes may not always strictly increase with difficulty. This is because a well-designed heuristic can more effectively guide the search in harder puzzles, sometimes resulting in fewer nodes being explored compared to moderately difficult puzzles.
9. For highly complex puzzles, uninformed search methods like BFS and DFS often fail due to excessive memory usage or time constraints. In contrast, heuristic-based search algorithms, such as Best-First Search and A\*, demonstrate superior performance. By leveraging heuristics, they efficiently prune the search space, significantly reducing the number of explored nodes and computation time, making them viable solutions even for difficult puzzles.