**XJCO2611**

**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**

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. Then we designed the following two heuristics for our investigation and testing:

* **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:

* **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.

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.

1. **Test Case Setup**

Initial State: The starting configuration of the puzzle.

Goal State: The target configuration to be achieved.

Puzzle Instance: An instance of the SlidingBlocksPuzzle class created using the initial and goal states.

1. ***Search function***

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)* |

**3. Test Summary**

We conducted 14 tests on three different difficulty levels of the sliding block puzzle. The tests

included variations using the following search algorithms and parameters:

**Search Algorithms:**

* Depth-First Search
* Breadth-First Search
* Best First
* A\*

**Test Parameters:**

* Loop check
* Randomization
* Two different heuristics (Manhattan Distance and Euclidean Distance)

Each test was executed across all three difficulty levels, adjusting parameters like search

method and heuristics. Results were measured in terms of the search process, time and

number of nodes expanded.

**A2. Results**

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Find solution | Path length | Total nodes tested | Time taken(s) |
| BFS | Success | 8 | 185 | 0.6818 |
| DFS(Fixed Order) | Success | 26 | 33 | 1.4006 |
| DFS(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) |
| BFS | Success | 8 | 118659 | 30.4198 |
| DFS(Fixed Order) | Failed | NaN | NaN | NaN |
| DFS(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) |
| BFS | Success | 32 | 56332 | 15.9252 |
| DFS(Fixed Order) | Success | 12399 | 14673 | 11.5494 |
| DFS(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 because the five-minute time limit was exceeded.

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Find solution | Path length | Total nodes tested | Time taken(s) |
| BFS | Failed | NaN | NaN | NaN |
| DFS(Fixed Order) | Failed | NaN | NaN | NaN |
| DFS(Random Order) | Failed | NaN | NaN | NaN |
| Best First(Manhattan) | Success | 81 | 1446 | 17.1941 |
| Best First(Euclidean) | Success | 113 | 1317 | 20.2408 |
| A\*(Manhattan) | Success | 61 | 5513 | 16.836 |
| A\*(Euclidean) | Success | 59 | 75169 | 140.1535 |

For the test of hard\_puzzle when loop check is False, all tests failed because the five-minute time limit was exceeded.

**A3. Observations**

After examining our results, we gained deep understanding of search algorithms. The most interesting observations were as follows:

1. **Factors may affect ‘difficulty’ of the puzzle:** The larger the puzzle, the greater the number of blocks, and the farther the tiles need to be moved from their initial to their goal positions, the more difficult the puzzle becomes. Besides, if the same color blocks appear more than once, the puzzle will also become more difficult.
2. **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.
3. **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.
4. **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.
5. **While DFS with fixed ordering failed in some cases, DFS with random action ordering succeeded**. This is because 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. Randomization helps explore different paths, and overall, it took less time compared to the fixed order.
6. **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.
7. **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.
8. **Heuristic-based search algorithms are better than uninformed search algorithms.** For highly complex puzzles, uninformed search algorithms 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.
9. **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.

**B. Robot Worker Experimentation**

**B1. Robot worker scenario**

After some experimentation we decided to investigate the following cases of Robot worker scenario.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Easy\_case | Middle\_case | Hard\_case |
| Initial State |  |  |  |
| Goal |  |  |  |

**Easy Case:** The robot starts in the store room and needs to navigate through three rooms. All the doors do not require keys, so the robot can freely move between rooms, making the task relatively simple.

**Middle Case:** The robot also starts in the store room, but the task becomes more complex. There are four rooms in this scenario, with two of the doors requiring keys to open. The robot will need to first find the keys and unlock these doors before accessing the other rooms. This setup introduces more steps and path planning for the robot to complete its task efficiently.

**Hard Case:**The robot also starts in the store room, but the scenario is the most complex.The environment consists of seven rooms and four locked doors. Among these doors, three can be opened with keys, but one has no available key. There are ten items in total, with seven of them being target objects.This case is more challenging due to the combination of locked doors, the need to find keys, and multiple target items.

**B2. Heuristics**

First, we let the robot pick up only the objects that appear in the goal scenario, avoiding unnecessary pickups unrelated to the target state. Then we designed the following two heuristics for our investigation and testing:

* **Misplace Heuristic**

The Misplace heuristic is a function that measures the difference between the current state and the goal state. Its core idea is to estimate how far the current state is from the goal state by counting the number of items that are out of place.

This heuristic is simple and fast to compute, but it has a drawback that it may not accurately reflect the true shortest path, as misplaced items do not necessarily affect the shortest path directly and may lead to unnecessary detours.

* **Heuristic Based on Whether Goal Items Are not Carried**

This heuristic method is more dynamic. It calculates the heuristic value based on whether the goal items that not in goal room have been carried by the robot.

The heuristic value will increase in the two cases:

1. The item that not in the goal room have not been carried, indicating that the robot needs to go to the target room to collect it.
2. The item is on the robot and the robot is not in the goal room, suggesting that the robot needs to return to the goal room to put down the item.

This method is particularly useful for situations where multiple items need to be collected at once, and it helps optimize the sequence of item collection and placement based on the previous heuristic.

**B3.Results**

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Find solution | Path length | Total nodes tested | Time taken(s) |
| BFS | Success | 10 | 477 | 0.1267 |
| DFS(Fixed Order) | Success | 248 | 976 | 0.2495 |
| DFS(Random Order) | Success | 40 | 43 | 0.0136 |
| Best First(misplace) | Success | 12 | 27 | 0.0072 |
| Best First(carry\_item) | Success | 12 | 20 | 0.0050 |
| A\*(misplace) | Success | 12 | 64 | 0.0150 |
| A\*(carry\_item) | Success | 12 | 49 | 0.0128 |

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Find solution | Path length | Total nodes tested | Time taken(s) |
| BFS | Success | 10 | 46283 | 16.4911 |
| DFS(Fixed Order) | Failed | NaN | NaN | NaN |
| DFS(Random Order) | Success | 216 | 217 | 0.052 |
| Best First(misplace) | Success | 12 | 45 | 0.0093 |
| Best First(carry\_item) | Success | 12 | 28 | 0.0061 |
| A\*(misplace) | Success | 12 | 132 | 0.0278 |
| A\*(carry\_item) | Success | 10 | 97 | 0.0236 |

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Find solution | Path length | Total nodes tested | Time taken(s) |
| BFS | Failed | NaN | NaN | NaN |
| DFS(Fixed Order) | Success | 34749 | 36055 | 147.9692 |
| DFS(Random Order) | Success | 8522 | 8794 | 19.7387 |
| Best First(misplace) | Success | 22 | 206 | 0.1146 |
| Best First(carry\_item) | Success | 22 | 124 | 0.0684 |
| A\*(misplace) | Success | 22 | 609 | 0.3354 |
| A\*(carry\_item) | Success | 22 | 550 | 0.3176 |

**Test result for middle\_robot (loop=False)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Find solution | Path length | Total nodes tested | Time taken(s) |
| BFS | Failed | NaN | NaN | NaN |
| DFS(Fixed Order) | Failed | NaN | NaN | NaN |
| DFS(Random Order) | Failed | NaN | NaN | NaN |
| Best First(misplace) | Success | 22 | 626 | 0.2818 |
| Best First(carry\_item) | Success | 22 | 613 | 0.2749 |
| A\*(misplace) | Success | 22 | 19057 | 16.3449 |
| A\*(carry\_item) | Success | 22 | 17364 | 14.0053 |

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Find solution | Path length | Total nodes tested | Time taken(s) |
| BFS | Failed | NaN | NaN | NaN |
| DFS(Fixed Order) | Failed | NaN | NaN | NaN |
| DFS(Random Order) | Failed | NaN | NaN | NaN |
| Best First(misplace) | Success | 56 | 4490 | 6.5715 |
| Best First(carry\_item) | Success | 56 | 7560 | 8.6286 |
| A\*(misplace) | Success | 41 | 78849 | 89.9099 |
| A\*(carry\_item) | Success | 41 | 93798 | 147.1164 |

For the test of hard\_puzzle when loop check is False, all tests failed because the five-minute time limit was exceeded.

**B4. Key findings**

After examining our results, we gained deep understanding of search algorithms. The most interesting observations were as follows:

1. **Factors may affect ‘difficulty’ of the scenario:** The scenario's difficulty increases with more rooms, objects, locked doors, as well as longer transport distances and smaller strength of robot, because a larger decision space raises computational complexity. Besides, when there is a door that is locked but has no corresponding key to open, it also increases the difficulty of the scene.
2. **As scenario complexity increases, the path length, number of nodes tested, and computation time significantly increase for all algorithms that can find the path**.
3. **A\* algorithm finds the optimal path but is slower and tests more nodes than Best Fit.** A\* uses f(n)=g(n)+h(n) to prioritize node expansion based on both the cost and the heuristic. This leads to more node expansions, higher memory usage, and longer computation times, especially in large search spaces. In contrast, the Best Fit algorithm relies solely on the heuristic f(n) = h(n), making it faster and more memory-efficient by focusing only on the most promising paths. However, it sacrifices optimality and may not find the shortest path.

1. **Compared to DFS, BFS performs better in simple case, but middle BFS fails in hard case.**

In the simple case, BFS works well as there are no obstacles, and it can easily explore the search space level by level. However, in the middle case, BFS struggles because it explores all paths, including those with locked doors, leading to a large, inefficient search space. DFS, on the other hand, explores one path deeply before backtracking, which allows it to more efficiently find the keys and doors' correct order in the middle case. As a result, DFS can sometimes find the solution faster, especially when the goal is deeper or fewer paths need to be explored.

1. **Although Misplace and Carry\_item heuristics have the same path length, Carry\_item is faster in simpler cases, while in the Hard Case, its results in longer computation time.** In simpler case, Carry\_item is faster because it provides more specific guidance by considering the robot's position and carried items, reducing unnecessary path exploration. Misplace is simpler and only checks if items are in the correct rooms, leading to a larger search space, which increases computation time. In the hard case, Carry\_item face longer runtimes due to detailed nature, while Misplace is faster because it provides a simpler heuristic, which results in fewer calculations and quicker decision-making despite a larger search space.