COMP2611

Artificial Intelligence

Assignment 1: Search Algorithms

Lai Ting Yeung: sc21lty@leeds.ac.uk
Michael D Wiciak: ed203mw@leeds.ac.uk

A. 8-Puzzle Search Investigation

A.1. Search Algorithm Sequence

The initial board states we used are as follows

```
sample_layouts = [[[1, 5, 2], [0, 4, 3], [7, 8, 6]],
                  [[5, 1, 7], [2, 4, 8], [6, 3, 0]],
                  [[2, 6, 3], [4, 0, 5], [1, 8, 7]],
                  [[7, 2, 5], [4, 8, 1], [3, 0, 6]],
                  [[8, 6, 7], [2, 5, 4], [3, 0, 1]], ]
```

Our testing code is as follows, for algorithm, we run all layouts on it.

```
algorithms = [
    {'name': 'Breadth First Search', 'mode': "BF/FIFO", 'cost': None,
        'randomise': False, 'heuristic': None, 'results': [], },
    {'name': 'Depth First Search', 'mode': "DF/LIFO", 'cost': None,
        'randomise': False, 'heuristic': None, 'results': [], },
    {'name': 'Depth First Search (Randomised)', 'mode': "DF/LIFO",
     'cost': None, 'randomise': True, 'heuristic': None, 'results': [], },
    {'name': 'Uniform Cost Search (Path Length)', 'mode': "BF/FIFO",
     'cost': True, 'randomise': False, 'heuristic': None, 'results': [], },
    {'name': 'Greedy Search (Misplaced Tile)', 'mode': "BF/FIFO", 'cost': None,
     'randomise': False, 'heuristic': bb_misplaced_tiles, 'results': [], },
    {'name': 'A* Search (Misplaced Tile)', 'mode': "BF/FIFO", 'cost': True,
     'randomise': False, 'heuristic': bb_misplaced_tiles, 'results': [], },
    {'name': 'Greedy Search (Manhattan)', 'mode': "BF/FIFO", 'cost': None,
     'randomise': False, 'heuristic': bb_manhattan, 'results': [], },
    {'name': 'A* Search (Manhattan)', 'mode': "BF/FIFO", 'cost': True,
     'randomise': False, 'heuristic': bb_manhattan, 'results': [], },
]

max_nodes = 2 ** 19

for index, layout in enumerate(sample_layouts):
    for algorithm in algorithms:
        puzzle = EightPuzzle(layout, NORMAL_GOAL)
        result = search(puzzle,
                        mode=algorithm['mode'],
                        max_nodes=max_nodes,
                        loop_check=True,
                        return_info=True,
                        cost=puzzle.cost_path_length if algorithm['cost'] else None,
                        heuristic=algorithm['heuristic'],
                        dots=False,
                        show_path=False,
                        randomise=algorithm['randomise']
                        )

        algorithm['results'].append(result)
```

A.2. Results

The following tables displays the results from each search algorithm, since we had 5 sample layouts, the following values are the mathematical mean of all search runs. The additional first table is for all the results, the second and third shows results only for success and unsuccess searches respectively only.

Note that 2 of the 5 layouts are unsolvable, hence success rates are all 0.6.

All Searches

Algorithm	Distinct States Seen	Nodes Discarded	Nodes Generated	Nodes Left in Queue	Nodes Tested	Time Taken	Success Rate	Path Length (If found)
Breadth First Search	145050.2	240415.6	385465.8	566.6	144483.6	4.6	0.6	21.3
Depth First Search	82573.0	127094.8	209667.8	4407.8	78165.2	90.5	0.6	8397.3
Depth First Search (Randomised)	101591.2	142964.8	244556.0	10546.2	91045.0	78.6	0.6	25438.7
Uniform Cost Search (Path Length)	145050.2	240415.6	385465.8	566.6	144483.6	12.3	0.6	21.3
Greedy Search (Misplaced Tile)	72775.2	121090.8	193866.0	78.6	72696.6	3.5	0.6	35.3
A* Search (Misplaced Tile)	123465.4	184787.2	308252.6	8109.6	115355.8	5.3	0.6	21.3
Greedy Search (Manhattan)	72738.8	121065.0	193803.8	63.8	72675.0	3.2	0.6	36.7
A* Search (Manhattan)	80163.2	126919.6	207082.8	2457.8	77705.4	3.5	0.6	21.3

Successful Searches

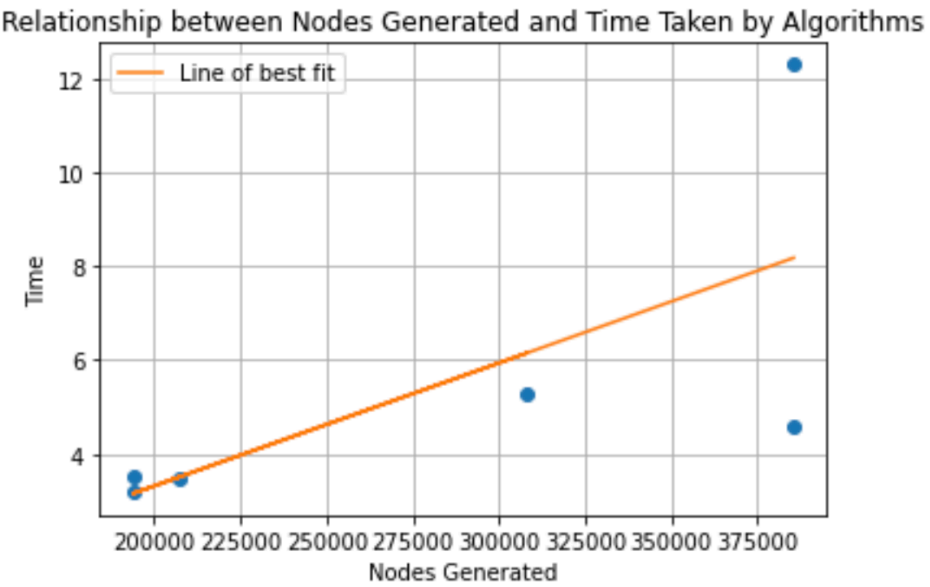
Algorithm	Distinct States Seen	Nodes Discarded	Nodes Generated	Nodes Left in Queue	Nodes Tested	Time Taken	Success Rate	Path Length (If found)
Breadth First Search	120790.3	199092.0	319882.3	944.3	119846.0	3.8	1.0	21.3
Depth First Search	16661.7	10224.0	26885.7	7346.3	9315.3	1.8	1.0	8397.3
Depth First Search (Randomised)	48358.7	36674.0	85032.7	17577.0	30781.7	36.803	1.0	25438.7
Uniform Cost Search (Path Length)	120790.3	199092.0	319882.3	944.3	119846.0	16.094	1.0	21.3
Greedy Search (Misplaced Tile)	332.0	217.3	549.3	131.0	201.0	0.011	1.0	35.3
A* Search (Misplaced Tile)	84815.7	106378.0	191193.7	13516.0	71299.7	3.379	1.0	21.3
Greedy Search (Manhattan)	271.3	174.3	445.7	106.3	165.0	0.007	1.0	36.7
A* Search (Manhattan)	12645.3	9932.0	22577.3	4096.3	8549.0	0.449	1.0	21.3

Unsuccessful Searches

Algorithm	Distinct States Seen	Nodes Discarded	Nodes Generated	Nodes Left in Queue	Nodes Tested	Time Taken	Success Rate	Path Length (If found)
Breadth First Search	181440.0	302401.0	483841.0	0.0	181440.0	5.7	0.0	NaN
Depth First Search	181440.0	302401.0	483841.0	0.0	181440.0	223.5	0.0	NaN
Depth First Search (Randomised)	181440.0	302401.0	483841.0	0.0	181440.0	141.3	0.0	NaN
Uniform Cost Search (Path Length)	181440.0	302401.0	483841.0	0.0	181440.0	6.6	0.0	NaN
Greedy Search (Misplaced Tile)	181440.0	302401.0	483841.0	0.0	181440.0	8.8	0.0	NaN
A* Search (Misplaced Tile)	181440.0	302401.0	483841.0	0.0	181440.0	8.1	0.0	NaN
Greedy Search (Manhattan)	181440.0	302401.0	483841.0	0.0	181440.0	7.9	0.0	NaN
A* Search (Manhattan)	181440.0	302401.0	483841.0	0.0	181440.0	8.0	0.0	NaN

A.3. Observations

- Algorithms that uses Depth First Search performs poorly in terms of speed, taking on average over a minute to search, over 30 seconds if the puzzle is solvable and over 3 minutes to terminate if not solvable.
- The algorithms that uses the Manhattan heuristic instead of Misplaced Tile heuristic performs better. Manhattan heuristic is much more faster.
- All algorithms successfully find a solution to the puzzle, when it is solvable, note that the max node is set at 524,288. Depth First Search algorithms also found solutions that require a lot more paths, 200 times as much than other algorithms.
- Comparing breadth first search and uniform cost search (cost being the length of path), they both performed equally when excluding time taken. Uniform cost search took nearly 3 times as much time as breadth first search. As seen, the cost function used was not effective.
- A* searches found shorter paths than Greedy's searches with the same heuristic, evidently, the cost function was effective only when paired with a heuristic.
- There seems to be a weak linear correlation between nodes generated and the time taken for the algorithm to find a solution (when excluding DFS algorithms). Suggesting that algorithms that generate the most nodes should usually take a larger amount of time than algorithms that generate less.
- The only algorithms that found the optimal path were Breadth First Search, Uniform Cost Search and A* Searches. The fastest out of all of them was A* Search with Manhattan heuristic.
- Greedy Search with the Manhattan heuristic was the fastest algorithm for this scenario as it was able to find a valid path approximately 8% faster than A* with Manhattan Heuristic. However, the path found was approximately 15 steps longer, which makes the path approximately 71% longer than what A* with Manhattan Heuristic found.
- Depth First Search algorithms takes the longest time to terminate when the puzzle is non-solvable.



A.4. Heuristics

1 Misplaced Tiles

The following function implements the first heuristic for Eight Puzzle, this heuristic counts the number of tiles that are not in their correct place.

```
def heuristic_misplaced(self, state):
    misplaced = 0
    current_layout = state[1]
    goal_layout = self.goal_layout

    for i, row in enumerate(current_layout):
        for j, value in enumerate(row):
            if value != goal_layout[i][j]:
                misplaced += 1

    return misplaced
```

The other heuristic we implemented uses the Manhattan distance to estimate how far we are from the solution. This heuristic counts how far each tile is from their goal position, and sums their Manhattan distance together.

2 Manhattan

```
def heuristic_manhattan(self, state):
    total_distance = 0
    current_layout = state[1]
    goal_layout = self.goal_layout

    for i, row in enumerate(current_layout):
        for j, value in enumerate(row):
            goal_pos = EightPuzzle.number_position_in_layout(
                value, goal_layout
            )
            goal_row, goal_col = goal_pos

            distance = abs(goal_row - i) + abs(goal_col - j)
            total_distance += distance

    return total_distance
```

B. Robot Worker Scenario

B.1. Our Robot Scenario

The robot is inside a castle-like structure. The old castle contains many secret passageways that are locked. They can only be accessed by first finding a secret key and then opening the passageway.

Passageways (denoted D) can be locked or unlocked. The castle contains 6 rooms (denoted R) and the robot starts in R0.

Each room may or may not contain one or more chests (denoted C), which contains valuable items.

The goal of the robot is to traverse through this castle and pick up all of the valuables from the chests, which are the keys, then place it in R0. However, the robot should traverse the castle in a way such that it performs the least amount of actions.

Passageways D1, D2, D3, D6, and D7 are unlocked by default. Passageways D4, D5, and D8 are locked.

Chest C2 contains the D4's key. C1 contains the D5's key. C3 contains the D8's key.

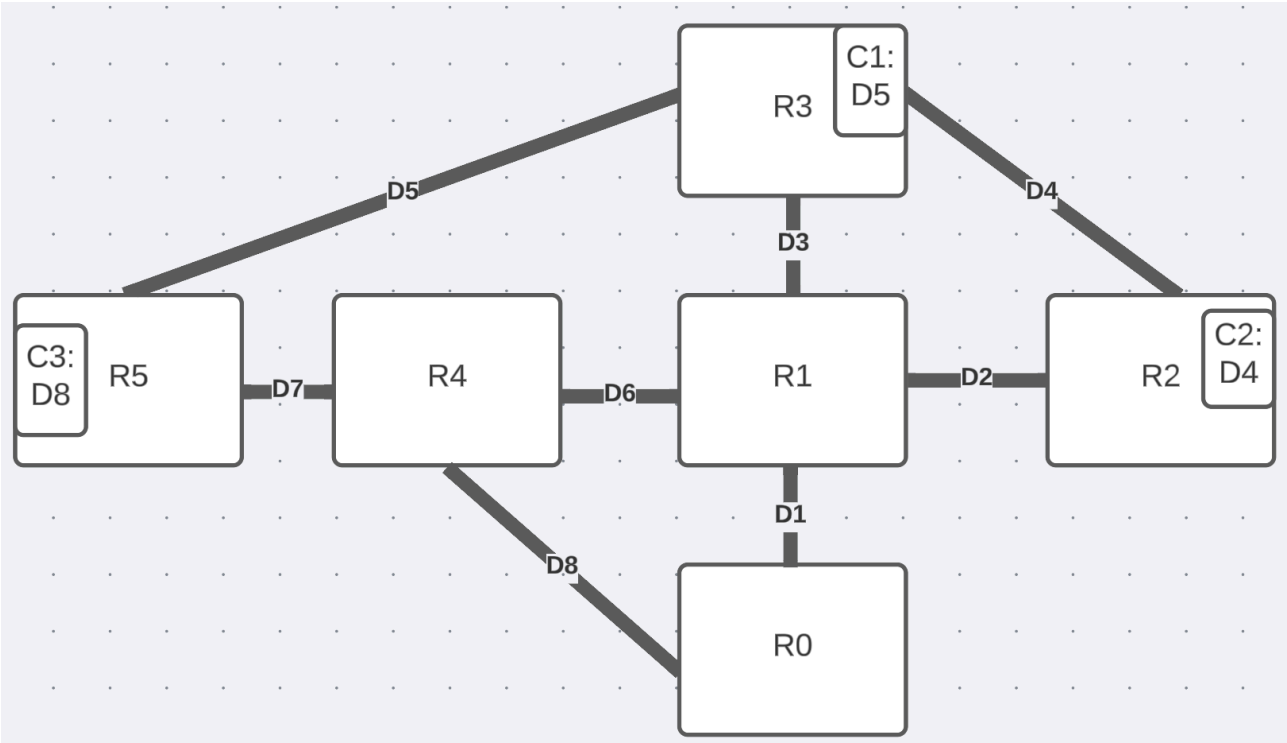
The action of opening a chest is abstracted away and not counted as an action. As such, the robot can open the chest and pick up the key in the same step.

The robot can change location from one room to another by using a passageway.

The robot can unlock locked passageways if it has the correct key in its contents. This step counts as an action for the robot.

The robot can pick up items from the chest if there is a chest in the current room which contains items.

The robot can put down items only in R0 since its goal is to fetch every item and place it in R0. This is done to reduce the number of possibilities the robot could go through.



B.2. Heuristics

1. Count Items

The first heuristic is quite simple. It counts how many of the goal items are not in the correct room. The lower the value, the more items from all the rooms are in the correct goal room. This heuristic assumes that is it advisable for the robot to pick up the item(s) from the current room as soon as possible and the is no benefit of waiting. It is admissible because it never overestimates the cost of reaching a goal.

```
def countItemsHeuristic(self, state):
    # count the number of items in the wrong room
    count = 0
    for room, contents in self.goal_item_locations.items():
        for i in contents:
            if not i in state.room_contents[room]:
                count += 1
    return count
```

2. Estimate Distance

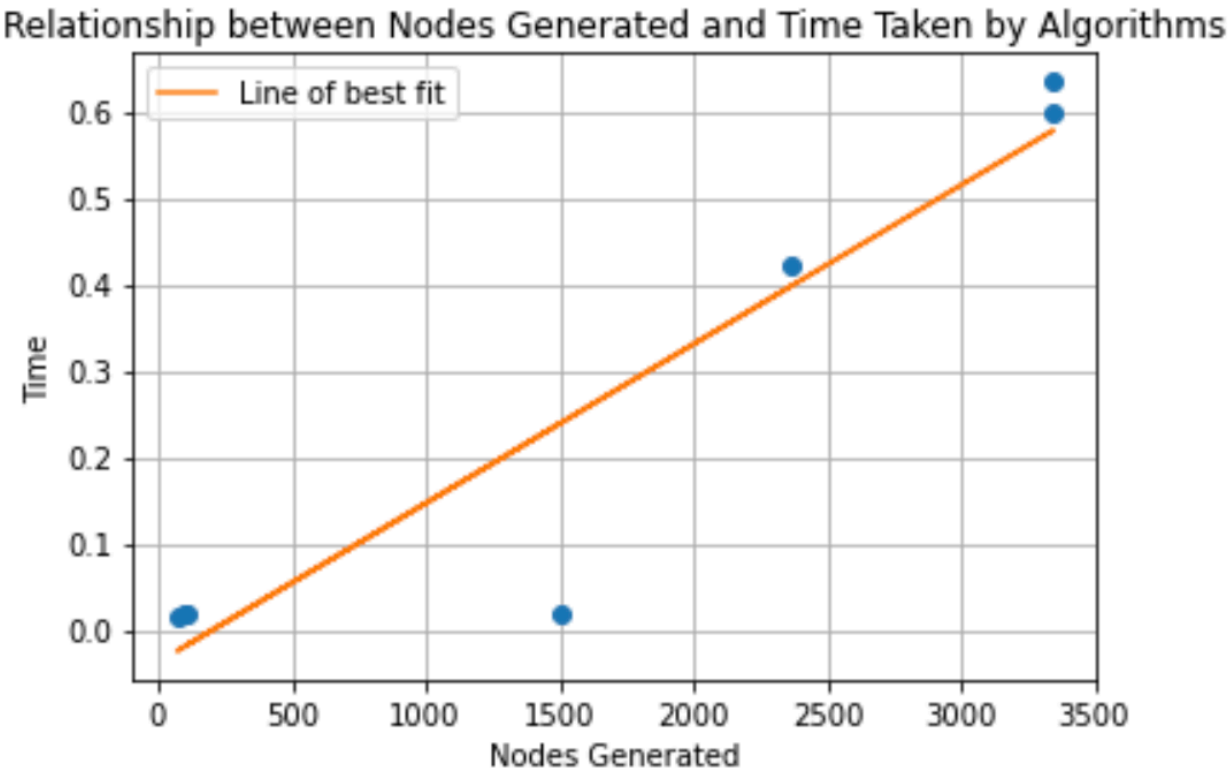
The second heuristic is more complex. It tries to estimate the cost of reaching a goal by considering the distance from the robot's current location to the closest item it has to pick up. So it outputs the cumulative distance it will take the robot to travel from the current location to the closest goal item plus the number of items the robot currently has in its possession. It uses a predefined dictionary to fetch the shortest path between two rooms. As such, the lower this heuristic, the closer the robot is to achieving its goal of putting an item in the correct goal room. It is admissible because it never overestimates the cost of reaching a goal.

```
def estimateDistanceHeuristic(self, state):
    min_dist = 1000000
    for room, contents in self.goal_item_locations.items():
        for i in contents:
            if not i in state.room_contents[room]:
                robotLocation = state.robot.location
                roomNumber = int(room[1:])
                global distance
                dist = distance[room]
                dist = dist[roomNumber]
                if dist < min_dist:
                    min_dist = dist
    # Now add the distance to the nearest goal item from the robot's current location
    # to the number of items the robot is carrying
    return min_dist + len(state.robot.carried_items)
```

B.3. Results

The following table displays the results from each search algorithm

Algorithm	Nodes Generated	Nodes tested	Nodes Discarded	Distinct States Seen	Nodes Left in Queue	Time Taken	Path Length
Breadth First Search	2360	797	1382	978	181	0.421	15
Depth First Search	95	35	41	54	19	0.020	23
Depth First Search (Randomised)	71	27	25	46	19	0.015	22
Greedy Search (Estimate Distance)	3344	1159	2178	1166	7	0.634	20
A* Search (Estimate Distance)	3344	1159	2178	1166	7	0.599	15
Greedy Search (Count Items)	106	43	36	70	27	0.020	20
A* Search (Count Items)	1500	535	774	726	191	0.020	15



B.4. Key Findings

- Depth First Search found a valid path faster than Breadth-First Search algorithms in terms of speed. Depth First Search found a path approximately 21 times faster than Breadth-First Search. This might be because Depth First Search generated approximately 24 times fewer nodes than Breadth-First Search.
- However, Breadth First Search algorithms found the optimal path of 15 whereas Depth First search found a path that is 8 longer, making it much less efficient for this scenario when looking at it in terms of correctness.
- All algorithms found a valid path when they were limited to a maximum of 100000 nodes.
- The estimateDistanceHeuristic when applied to A* Search allowed it to find the same path as Breadth First Search but it took significantly longer (approximately 33%), meaning this heuristic wasn't efficient for this scenario. The same could be said about Greedy Search with this heuristic where it found a better path than Depth First Search but took the longest compared to every other algorithm in this list. It seems that for these small scenarios, the computation overhead associated with calculating estimateDistanceHeuristic is greater than any benefit it provides.
- However, counterItemsHeuristic seemed to have worked well for this scenario. When applied to the Greedy Search, it found a more optimal path than Depth First Search (by three steps) and found it in a very similar time, making it a very efficient heuristic for this scenario.
- When countItemsHeuristic was applied to A* search, it also was a significant improvement compared to Breadth First Search. It not only also found the optimal path of 15 but found it nearly 35% faster.
- There seems to be a strong linear correlation between nodes generated and the time taken for the algorithm to find a solution (including DFS). Suggesting that algorithms that generate the most nodes should usually take a larger amount of time than algorithms that generate less for this specific scenario.