REPORT

Comp 472

Department of Computer Science and Software Engineering

Mini Project 1

Concordia University

Julia Bazarbachian

2678137

March 14, 2022

The goal of Mini Project 1 was to implement several search algorithms to solve the 8 puzzle problem and to compare their performance. First, breadth-first, depth-first, Best-First, and A* search were implemented using the General Search algorithm. Then, Hamming distance, Manhattan distance, Permutation Inversion and an inadmissible heuristic were implemented for the 8 puzzle problem. Lastly, the path lengths of the outputs were used to compare the search solutions' optimality.

Submitted modules:

Implemented Heuristics for 8-puzzles

All heuristics were implemented within class EightPuzzle.

1. Hamming Distance

```
# returns hamming distance value for a given state
def hamming_distance(self, node):
    sum = 0
    for i in range(1,9): # only considers numbered tiles
        if node.state.index(i) != self.goal.index(i):
            sum += 1
    return sum
```

This function counts how many numbered tiles are out of place. For a given state, it compares the index of each numbered tile with those of the goal state, and outputs the desired value.

2. Manhattan Distance

```
# returns manhattan distance value for a given state
def manhattan_distance(self, node):
    index = [(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)]
    index_goal = dict(zip(self.goal, index))
    index_state = dict(zip(node.state, index))

    sum = 0
    for i in range(1,9): # only considers numbered tiles
        x1, y1 = index_state[i]
        x2, y2 = index_goal[i]
        sum += abs(x2-x1) + abs(y2-y1)
    return sum
```

This function counts the sum of the distances by which the numbered tiles are out of place. For a given state, it compares the coordinate of each numbered tile with the goal state and outputs the sum of these distances.

3. Permutation Inversion

```
# returns permutation inversions value for a given state
def permutation_inversions(self, node):
    goal = self.goal
    state = node.state

    # only considers numbered tiles
    goal = list(goal)
    goal.remove('B')
    state = list(state)
    state.remove('B')

    sum = 0
    for i in range(len(state)):
        value = state[i]
        goal_index = goal.index(value)
        list1 = goal[:goal_index]
        list2 = state[i+1:]
        list3 = [value for value in list1 if value in list2]
        sum += len(list3)
    return sum
```

For each numbered tile of a given state, this function counts how many tiles on its right should appear on its left according to the goal state. It does so by finding the corresponding numbered tile in the goal state and counting the number of elements in common on the right of the tiles in the current state with the left of the tile in the goal state. The sum of these values is then output.

4. Inadmissible heuristic

```
# returns inadmissible heuristic value for a given state
def inadmissible_heuristic(self, node):
    return (self.hamming_distance(node) + 2*self.manhattan_distance(node))
```

For the inadmissible heuristic, I have implemented a function that adds the hamming distance to the double of the Manhattan distance. This heuristic is not admissible since we cannot say that $h_1(n) + 2 * h_2(n) \leq h^*(n)$, where h_1 is the hamming distance and h_2 is the Manhattan distance.

To test whether these heuristics were correctly implemented, I ran the code on the challenge 8-puzzle start states. The results are as follows:

EightPuzzle((2, 8, 3, 1, 6, 4, 7, 'B', 5))

```
challenge.hamming_distance((Node(challenge.initial)))
```

```
4
```

```
challenge.manhattan_distance((Node(challenge.initial)))
```

```
5
```

```
challenge.permutation_inversions((Node(challenge.initial)))
```

```
6
```

```
challenge.inadmissible_heuristic((Node(challenge.initial)))
```

```
14
```

EightPuzzle((5, 1, 4, 7, 'B', 6, 3, 8, 2))

```
greater_challenge.hamming_distance((Node(greater_challenge.initial)))
```

```
8
```

```
greater_challenge.manhattan_distance((Node(greater_challenge.initial)))
```

```
18
```

```
greater_challenge.permutation_inversions((Node(greater_challenge.initial)))
```

```
18
```

```
greater_challenge.inadmissible_heuristic((Node(greater_challenge.initial)))
```

```
44
```

These results are consistent with the calculations I performed manually.

Implemented a successor state generator for the 8-puzzle

To implement a successor state generator for a given 8-puzzle start state, the function *Expand* was implemented following the algorithm presented in *Figure 3.7* of the textbook. The function takes a state and outputs a list of possible next states based on the actions that can be taken. The results of testing the successor state generator are as follows:

Testing successor state generator on challenge puzzles

EightPuzzle((2, 8, 3, 1, 6, 4, 7, 'B', 5))

```
children_challenge = Node(challenge.initial).expand(challenge)
```

```
children_challenge
```

```
[(2, 8, 3, 1, 'B', 4, 7, 6, 5),
 (2, 8, 3, 1, 6, 4, 'B', 7, 5),
 (2, 8, 3, 1, 6, 4, 7, 5, 'B')]
```

EightPuzzle((5, 1, 4, 7, 'B', 6, 3, 8, 2))

```
children_greater_challenge = Node(greater_challenge.initial).expand(greater_challenge)
```

```
children_greater_challenge
```

```
[(5, 'B', 4, 7, 1, 6, 3, 8, 2),
 (5, 1, 4, 7, 8, 6, 3, 'B', 2),
 (5, 1, 4, 'B', 7, 6, 3, 8, 2),
 (5, 1, 4, 7, 6, 'B', 3, 8, 2)]
```

For the first challenge puzzle, the blank space can be moved left, right, and up, thereby generating 3 successor states. For the second challenge puzzle, the blank space can be moved left, right, up, and down, thereby generating 4 successor states.

Implemented search for 8-puzzles using the general algorithm

The four following search functions were implemented in accordance with the general search algorithm presented in the slides. The code is nearly identical for all, except for the implementation of the open list, which differs for the search strategies. The order of the nodes in the open list controls the order of the search.

1. Breadth First Search

```
def breadth_first_search(eightpuzzle):

    open_list = deque([(Node(eightpuzzle.initial))])
    closed_list = set()

    while open_list:
        node = open_list.popleft()
        if eightpuzzle.is_goal_state(node.state):
            return node
        closed_list.add(node.state)

        for child in node.expand(eightpuzzle):
            if child.state not in closed_list and child not in open_list:
                open_list.append(child)
    return None
```

The breadth first search function implements the open list as a FIFO queue. New states go to the back of the queue and the shallower old states get expanded first. The states are thereby expanded level by level.

2. Depth First Search

```
def depth_first_search(eightpuzzle):

    open_list = deque([(Node(eightpuzzle.initial))])
    closed_list = set()

    while open_list:
        node = open_list.popleft()
        if eightpuzzle.is_goal_state(node.state):
            return node
        closed_list.add(node.state)

        for child in node.expand(eightpuzzle):
            if child.state not in closed_list and child not in open_list:
                open_list.appendleft(child)
    return None
```

The depth first search function implements the open list as a stack. New states are added to the top of the stack and expanded first. In this search strategy, successor states are expanded before siblings.

3. Best First Search

```
def best_first_search(eightpuzzle, value):

    node = Node(eightpuzzle.initial)
    open_list = [node]
    closed_list = set()

    while open_list:
        open_list.sort(key=value, reverse=True)
        node = EightPuzzle(open_list.pop()).initial
        if eightpuzzle.is_goal_state(node.state):
            return node
        closed_list.add(node.state)

        for child in node.expand(eightpuzzle):
            if child.state not in closed_list and child not in open_list:
                open_list.append(child)
    return None
```

The best first search function implements the open list as a priority queue. In this implementation, the open list is sorted by heuristic value in descending order and the states expanded first are those with the lowest heuristic value. Because the sort() method was used, states with equal heuristic values are sorted so that they retain their original ordering relative to each other. The state that gets expanded first among equal values is therefore the one placed at the rightmost position of the open list.

4. A* Search

```
def a_star_search(eightpuzzle, value):

    node = Node(eightpuzzle.initial)
    open_list = [node]
    closed_list = set()

    while open_list:
        open_list.sort(key=value , reverse=True)
        node = EightPuzzle(open_list.pop()).initial
        if eightpuzzle.is_goal_state(node.state):
            return node
        closed_list.add(node.state)

        for child in node.expand(eightpuzzle):
            if child.state not in closed_list and child not in open_list:
                open_list.append(child)
    return None
```

The a star search function is implemented exactly the same way as the best first search, but the open list is sorted with a different value passed into the function. The following shows output paths generated on the first challenge puzzle when each search function is called:

1. Breadth First Search

```
bfs_challenge_path = breadth_first_search(challenge).path()
print(bfs_challenge_path)
```

```
[(2, 8, 3, 1, 6, 4, 7, 'B', 5), (2, 8, 3, 1, 'B', 4, 7, 6, 5), (2, 'B', 3, 1, 8, 4, 7, 6, 5), ('B', 2, 3, 1, 8, 4, 7, 6, 5),
(1, 2, 3, 'B', 8, 4, 7, 6, 5), (1, 2, 3, 8, 'B', 4, 7, 6, 5)]
```

2. Depth First Search

```
dfs_challenge_path = depth_first_search(challenge).path()
print(dfs_challenge_path)
```

```
[(2, 8, 3, 1, 6, 4, 7, 'B', 5), (2, 8, 3, 1, 6, 4, 7, 5, 'B'), (2, 8, 3, 1, 6, 'B', 7, 5, 4), (2, 8, 3, 1, 'B', 6, 7, 5, 4),
(2, 8, 3, 'B', 1, 6, 7, 5, 4), (2, 8, 3, 7, 1, 6, 'B', 5, 4), (2, 8, 3, 7, 1, 6, 5, 'B', 4), (2, 8, 3, 7, 1, 6, 5, 4, 'B'),
(2, 8, 3, 7, 1, 'B', 5, 4, 6), (2, 8, 3, 7, 'B', 1, 5, 4, 6), (2, 8, 3, 'B', 7, 1, 5, 4, 6), (2, 8, 3, 5, 7, 1, 'B', 4, 6),
(2, 8, 3, 5, 7, 1, 4, 'B', 6), (2, 8, 3, 5, 7, 1, 4, 6, 'B'), (2, 8, 3, 5, 7, 'B', 4, 6, 1), (2, 8, 3, 5, 'B', 7, 4, 6, 1),
(2, 8, 3, 'B', 5, 7, 4, 6, 1), (2, 8, 3, 4, 5, 7, 'B', 6, 1), (2, 8, 3, 4, 5, 7, 6, 'B', 1), (2, 8, 3, 4, 5, 7, 6, 1, 'B'),
(2, 8, 3, 4, 5, 'B', 6, 1, 7), (2, 8, 3, 4, 'B', 5, 6, 1, 7), (2, 8, 3, 'B', 4, 5, 6, 1, 7), (2, 8, 3, 6, 4, 5, 'B', 1, 7),
(2, 8, 3, 6, 4, 5, 1, 'B', 7), (2, 8, 3, 6, 4, 5, 1, 7, 'B'), (2, 8, 3, 6, 4, 'B', 1, 7, 5), (2, 8, 3, 6, 'B', 4, 1, 7, 5),
(2, 8, 3, 'B', 6, 4, 1, 7, 5), ('B', 8, 3, 2, 6, 4, 1, 7, 5), (8, 'B', 3, 2, 6, 4, 1, 7, 5), (8, 3, 'B', 2, 6, 4, 1, 7, 5),
(8, 3, 4, 2, 6, 'B', 1, 7, 5), (8, 3, 4, 2, 'B', 6, 1, 7, 5), (8, 3, 4, 'B', 2, 6, 1, 7, 5), (8, 3, 4, 1, 2, 6, 'B', 7, 5),
(8, 3, 4, 1, 2, 6, 7, 'B', 5), (8, 3, 4, 1, 2, 6, 7, 5, 'B'), (8, 3, 4, 1, 2, 'B', 7, 5, 6), (8, 3, 4, 1, 'B', 2, 7, 5, 6),
(8, 3, 4, 'B', 1, 2, 7, 5, 6), (8, 3, 4, 7, 1, 2, 'B', 5, 6), (8, 3, 4, 7, 1, 2, 5, 'B', 6), (8, 3, 4, 7, 1, 2, 5, 6, 'B'),
(8, 3, 4, 7, 1, 'B', 5, 6, 2), (8, 3, 4, 7, 'B', 1, 5, 6, 2), (8, 3, 4, 'B', 7, 1, 5, 6, 2), (8, 3, 4, 5, 7, 1, 'B', 6, 2),
(8, 3, 4, 5, 7, 1, 6, 'B', 2), (8, 3, 4, 5, 7, 1, 6, 2, 'B'), (8, 3, 4, 5, 7, 'B', 6, 2, 1), (8, 3, 4, 5, 'B', 7, 6, 2, 1),
(8, 3, 4, 'B', 5, 7, 6, 2, 1), (8, 3, 4, 6, 5, 7, 'B', 2, 1), (8, 3, 4, 6, 5, 7, 2, 'B', 1), (8, 3, 4, 6, 5, 7, 2, 1, 'B'),
(8, 3, 4, 6, 5, 'B', 2, 1, 7), (8, 3, 4, 6, 'B', 5, 2, 1, 7), (8, 3, 4, 'B', 6, 5, 2, 1, 7), (8, 3, 4, 2, 6, 5, 'B', 1, 7),
(8, 3, 4, 2, 6, 5, 1, 'B', 7), (8, 3, 4, 2, 'B', 5, 1, 6, 7), (8, 3, 4, 2, 5, 'B', 1, 6, 7), (8, 3, 4, 2, 5, 7, 1, 6, 'B'),
(8, 3, 4, 2, 5, 7, 1, 'B', 6), (8, 3, 4, 2, 5, 7, 'B', 1, 6), (8, 3, 4, 'B', 5, 7, 2, 1, 6), (8, 3, 4, 5, 'B', 7, 2, 1, 6),
(8, 3, 4, 5, 7, 'B', 2, 1, 6), (8, 3, 4, 5, 7, 6, 2, 1, 'B'), (8, 3, 4, 5, 7, 6, 2, 'B', 1), (8, 3, 4, 5, 7, 6, 'B', 2, 1),
(8, 3, 4, 'B', 7, 6, 5, 2, 1), (8, 3, 4, 7, 'B', 6, 5, 2, 1), (8, 3, 4, 7, 6, 'B', 5, 2, 1), (8, 3, 4, 7, 6, 1, 5, 2, 'B'),
```

3. Best First Search

```
bf_challenge_hamming_path = best_first_search(challenge, lambda x: challenge.hamming_distance(x)).path()
print(bf_challenge_hamming_path)
```

```
[(2, 8, 3, 1, 6, 4, 7, 'B', 5), (2, 8, 3, 1, 'B', 4, 7, 6, 5), (2, 8, 3, 'B', 1, 4, 7, 6, 5), ('B', 8, 3, 2, 1, 4, 7, 6, 5),
(8, 'B', 3, 2, 1, 4, 7, 6, 5), (8, 1, 3, 2, 'B', 4, 7, 6, 5), (8, 1, 3, 'B', 2, 4, 7, 6, 5), ('B', 1, 3, 8, 2, 4, 7, 6, 5), (1,
'B', 3, 8, 2, 4, 7, 6, 5), (1, 2, 3, 8, 'B', 4, 7, 6, 5)]
```

```
bf_challenge_manhattan_path = best_first_search(challenge, lambda x: challenge.manhattan_distance(x)).path()
print(bf_challenge_manhattan_path)
```

```
[(2, 8, 3, 1, 6, 4, 7, 'B', 5), (2, 8, 3, 1, 'B', 4, 7, 6, 5), (2, 'B', 3, 1, 8, 4, 7, 6, 5), ('B', 2, 3, 1, 8, 4, 7, 6, 5),
(1, 2, 3, 'B', 8, 4, 7, 6, 5), (1, 2, 3, 8, 'B', 4, 7, 6, 5)]
```

```
bf_challenge_permutation_inversions_path = best_first_search(challenge, lambda x: challenge.permutation_inversions(x)).path()
print(bf_challenge_permutation_inversions_path)
```

```
[(2, 8, 3, 1, 6, 4, 7, 'B', 5), (2, 8, 3, 1, 'B', 4, 7, 6, 5), (2, 'B', 3, 1, 8, 4, 7, 6, 5), ('B', 2, 3, 1, 8, 4, 7, 6, 5),
(1, 2, 3, 'B', 8, 4, 7, 6, 5), (1, 2, 3, 8, 'B', 4, 7, 6, 5)]
```

```
bf_challenge_inadmissible_heuristic_path = best_first_search(challenge, lambda x: challenge.inadmissible_heuristic(x)).path()
print(bf_challenge_inadmissible_heuristic_path)
```

```
[(2, 8, 3, 1, 6, 4, 7, 'B', 5), (2, 8, 3, 1, 'B', 4, 7, 6, 5), (2, 'B', 3, 1, 8, 4, 7, 6, 5), ('B', 2, 3, 1, 8, 4, 7, 6, 5),
(1, 2, 3, 'B', 8, 4, 7, 6, 5), (1, 2, 3, 8, 'B', 4, 7, 6, 5)]
```

4. A* Search

```
a_star_challenge_hamming_path = a_star_search(challenge, lambda x: challenge.hamming_distance(x) + x.path_cost).path()
print(bf_challenge_hamming_path)
```

```
[(2, 8, 3, 1, 6, 4, 7, 'B', 5), (2, 8, 3, 1, 'B', 4, 7, 6, 5), (2, 8, 3, 'B', 1, 4, 7, 6, 5), ('B', 8, 3, 2, 1, 4, 7, 6, 5),
(8, 'B', 3, 2, 1, 4, 7, 6, 5), (8, 1, 3, 2, 'B', 4, 7, 6, 5), (8, 1, 3, 'B', 2, 4, 7, 6, 5), ('B', 1, 3, 8, 2, 4, 7, 6, 5), (1,
'B', 3, 8, 2, 4, 7, 6, 5), (1, 2, 3, 8, 'B', 4, 7, 6, 5)]
```

```
a_star_challenge_manhattan_path = a_star_search(challenge, lambda x: challenge.manhattan_distance(x) + x.path_cost).path()
print(bf_challenge_manhattan_path)
```

```
[(2, 8, 3, 1, 6, 4, 7, 'B', 5), (2, 8, 3, 1, 'B', 4, 7, 6, 5), (2, 'B', 3, 1, 8, 4, 7, 6, 5), ('B', 2, 3, 1, 8, 4, 7, 6, 5),
(1, 2, 3, 'B', 8, 4, 7, 6, 5), (1, 2, 3, 8, 'B', 4, 7, 6, 5)]
```

```
a_star_challenge_permutation_inversions_path = a_star_search(challenge, lambda x: challenge.permutation_inversions(x) + x.path_cc
print(bf_challenge_permutation_inversions_path)
```

```
[(2, 8, 3, 1, 6, 4, 7, 'B', 5), (2, 8, 3, 1, 'B', 4, 7, 6, 5), (2, 'B', 3, 1, 8, 4, 7, 6, 5), ('B', 2, 3, 1, 8, 4, 7, 6, 5),
(1, 2, 3, 'B', 8, 4, 7, 6, 5), (1, 2, 3, 8, 'B', 4, 7, 6, 5)]
```

```
tar_challenge_inadmissible_heuristic_path = a_star_search(challenge, lambda x: challenge.inadmissible_heuristic(x) + x.path_cost)
nt(bf_challenge_inadmissible_heuristic_path)
```

```
[(2, 8, 3, 1, 6, 4, 7, 'B', 5), (2, 8, 3, 1, 'B', 4, 7, 6, 5), (2, 'B', 3, 1, 8, 4, 7, 6, 5), ('B', 2, 3, 1, 8, 4, 7, 6, 5),
(1, 2, 3, 'B', 8, 4, 7, 6, 5), (1, 2, 3, 8, 'B', 4, 7, 6, 5)]
```

The search functions output a list of successive states that start with the given start state and end with the goal state, with all the traversed states in between. Similarly, outputs for the second challenge question were also generated and can be found in the source code file.

Performance

Path Length

Search	**Hamming**	**Manhattan**	**Inversions**	**Inadmissible**	**Initial State**
Best-First	10	6	6	6	(2, 8, 3, 1, 6, 4, 7, 'B', 5)
A*	6	6	6	6	(2, 8, 3, 1, 6, 4, 7, 'B', 5)
Best-First	117	79	141	57	(5, 1, 4, 7, 'B', 6, 3, 8, 2)
A*	21	21	21	29	(5, 1, 4, 7, 'B', 6, 3, 8, 2)
Best-First	7	7	7	7	('B', 2, 3, 1, 4, 5, 8, 7, 6)
A*	7	7	7	7	('B', 2, 3, 1, 4, 5, 8, 7, 6)
Best-First	10	6	6	6	(1, 4, 2, 'B', 8, 3, 7, 6, 5)
A*	6	6	6	6	(1, 4, 2, 'B', 8, 3, 7, 6, 5)
Best-First	44	42	46	24	(2, 8, 1, 'B', 4, 3, 7, 6, 5)
A*	10	10	10	10	(2, 8, 1, 'B', 4, 3, 7, 6, 5)

In the table above, we have 5 tests performed on different initial states. The first two rows show results for the challenge puzzle, and the following two for the greater challenge puzzle. In all cases, the paths of the A* algorithm are less than or equal to those of the Best-First Search. For the inadmissible heuristic, we can observe from row 4 (greater challenge puzzle) that the shortest search path isn't guaranteed. Since a more informed heuristic expands fewer nodes, we can also observe the Manhattan distance is more informative than the hamming distance, which is more informative than the permutation inversions heuristic.

REFERENCE

Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, 4th edition, Prentice Hall.