COM1005: Machines and Intelligence

Assignment: 1

Documentation

Task 2.1 Implementing 8-puzzle search

Breadth-First Search

I have done two solutions for the Breadth-First Search of P1, P2 and P3. The first solution has less iterations and higher efficiency and the second solution has more iterations and lower efficiency.

Puzzle	Iterations	Efficiency
P1	4	1.0
P2	13	0.5385
P3	16	0.5625
P4(hardest)	41	Search Fails
P5(hardest)	41	Search Fails

Table 1. Faster solution of P1, P2 and P3 with Breadth-First Search, the results of P4 and P5 is to be compared with the slower solution

Table 1 shows the number of iterations and the efficiency of the faster solution. This solution does not add all the next possible states as the successors of the current puzzle state. It only adds the state where the empty tile swaps with the adjacent tiles that are not in the correct position. By doing so, it greatly reduced the number of open nodes needed to be checked on each level by the search engine and hence, result in a higher efficiency. However, this solution has a disadvantage compared to the slower solution. The faster solution is not able to solve every solvable puzzle. This is because some solutions require swapping the empty tile with the adjacent tiles that are in the correct position, and the faster solution has excluded this action. Therefore, some solvable puzzles will have “Search Fails” if it is run using the faster solution, as shown by **P4** and **P5** in **Table 1**.

Puzzle	Iterations	Efficiency
P1	11	0.3636
P2	57	0.1228
P3	165	0.05455
P4(hardest)	181439	0.0001764
P5(hardest)	181440	0.0001764

Table 2. Slower solution of P1, P2 and P3 with Breadth-First Search, the results of P4 and P5 is to be compared with A Search algorithm*

Table 2 shows the search results of the slower solution. This solution adds all the next possible states as the successors of the current puzzle state, which will result in a longer searching time and lower efficiency due to the increased number of successor nodes compared to the faster solution. However, it is able to solve **P4** and **P5** but it requires very long time for the search to conclude. Since the requirement of Task 2.1 is to find the solution for **P1**, **P2**, and **P3** only, I have decided to use the faster solution to achieve a better efficiency.

Task 2.2 Experimenting with Search Strategies

A* Search

*Note: The Est. Total Cost is from the first state

Puzzle	Hamming			Manhattan			Global Cost
	Iterations	Efficiency	Est. Total Cost	Iterations	Efficiency	Est. Total Cost	
P1	4	1.0	4	4	1.0	5	3
P2	7	1.0	7	8	0.875	9	6
P3	10	0.9	9	12	0.75	9	8
P4(hardest)	151588	0.0002212	8	22583	0.001575	21	31
P5(hardest)	151588	0.0002212	8	22589	0.001575	21	31

Table 3. A search results of P1, P2 and P3*

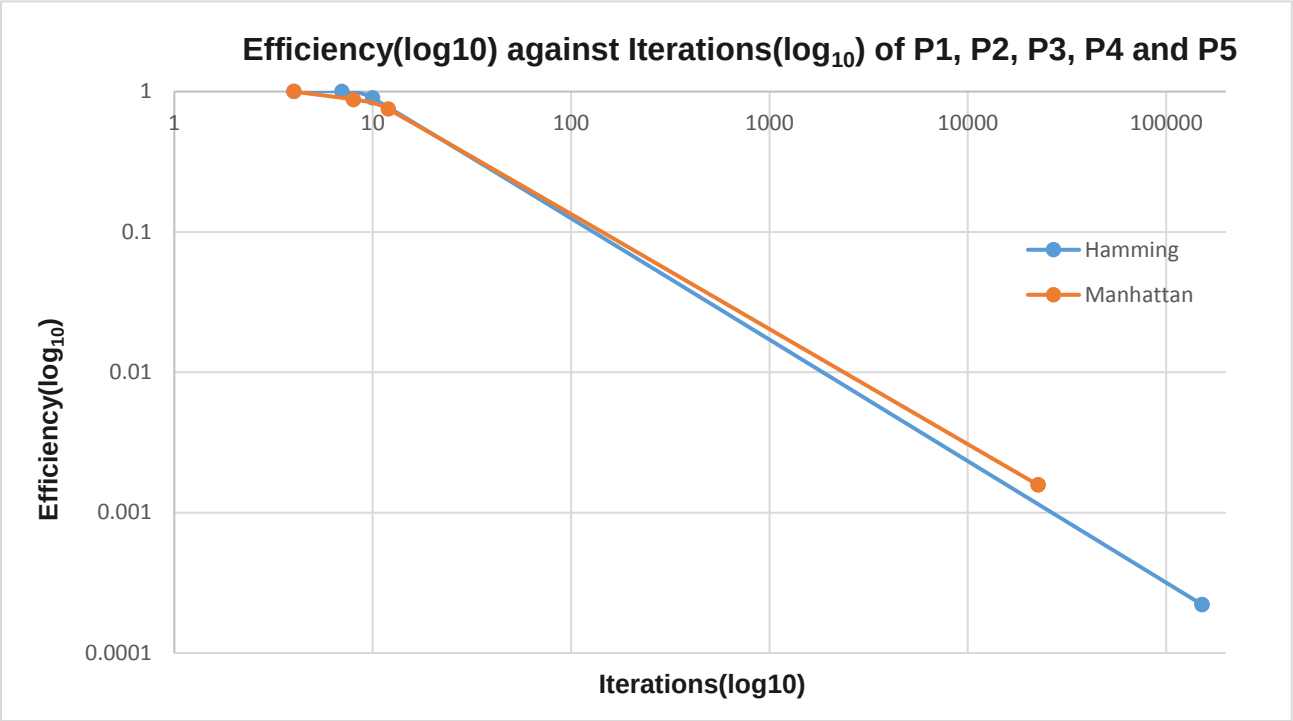


Figure 1.1 Relationship between the efficiency and iterations of Hamming and Manhattan

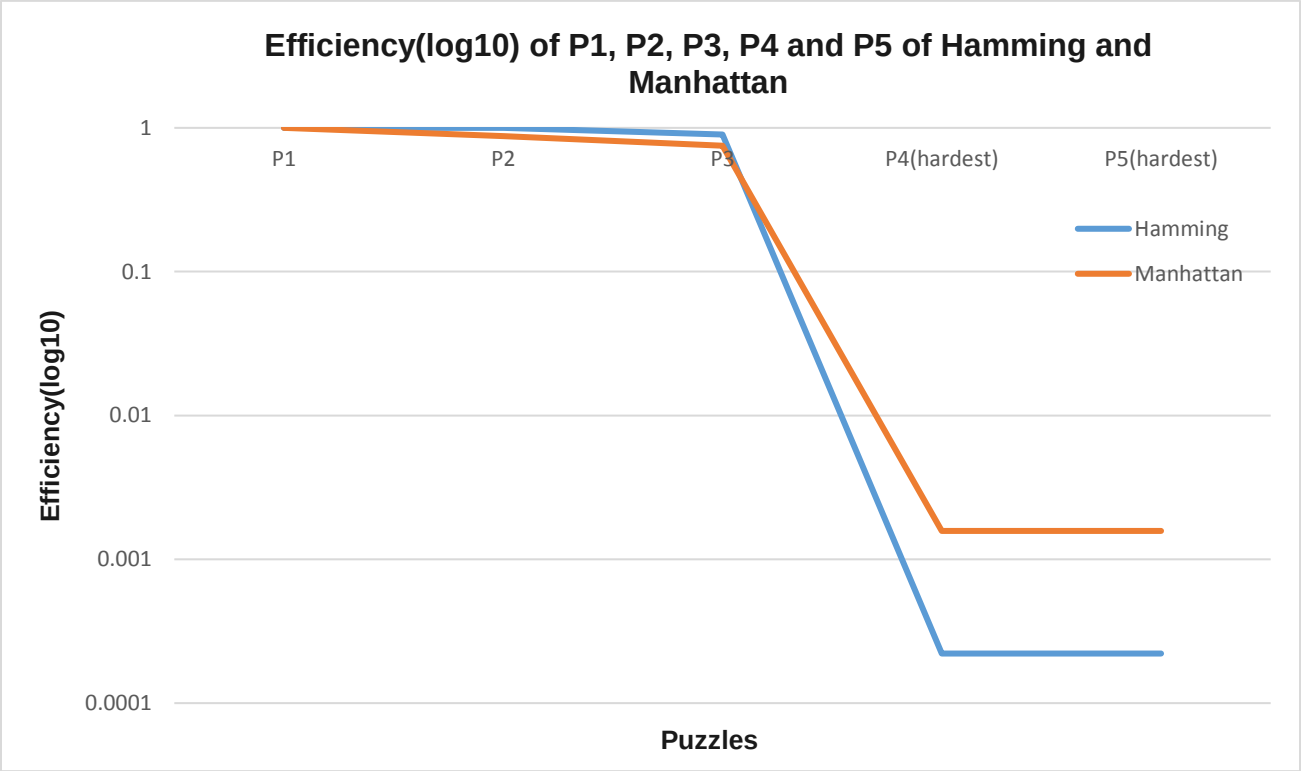


Figure 1.2 Efficiency of P1, P2, P3, P4 and P5 between Hamming and Manhattan

Difficulty (Seed "12345")	Hamming			Manhattan			Global Cost
	Iterations	Efficiency	Est. Total Cost	Iterations	Efficiency	Est. Total Cost	
6	500	0.032	7	287	0.05633	9	15
8	4062	0.005234	8	1096	0.01946	9	20
10	2763	0.007307	7	644	0.03180	9	19
20	13962	0.001753	9	1546	0.01627	15	23

Table 4. A search results of different difficulties of puzzles with seed "12345"*

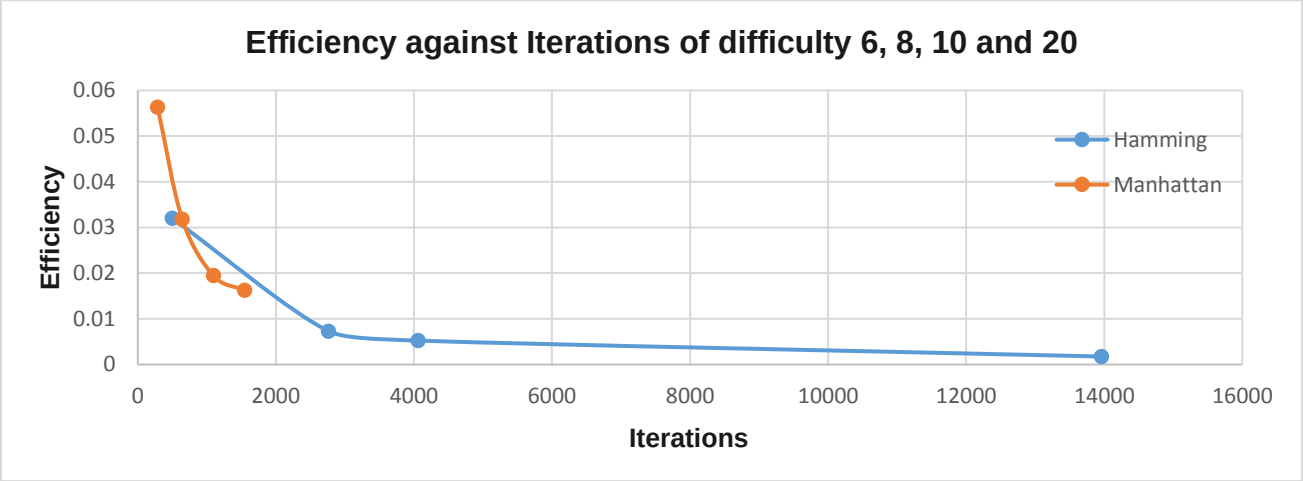


Figure 2.1 Relationship between the efficiency and iterations of Hamming and Manhattan

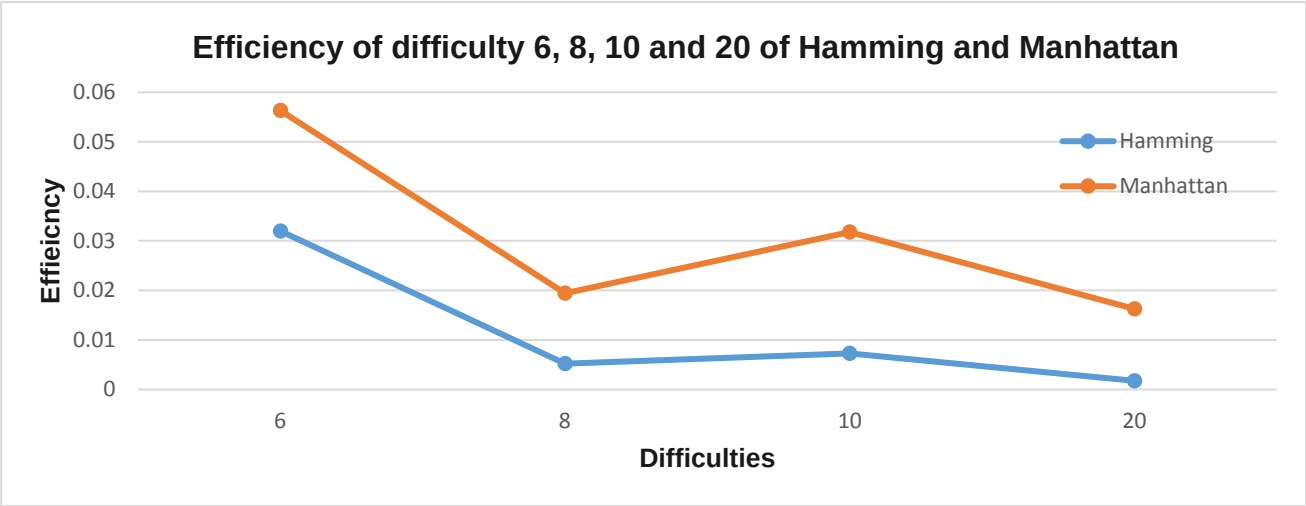


Figure 2.2 Comparison of efficiency between puzzles of different difficulties with seed "12345"

Difficulty (Seed "54321")	Hamming			Manhattan			Global Cost
	Iterations	Efficiency	Est. Total Cost	Iterations	Efficiency	Est. Total Cost	
6	1315	0.01379	6	726	0.02531	9	17
8	263	0.05703	7	144	0.1049	9	14
10	61538	0.0004701	7	12053	0.002536	13	27
12	14969	0.001639	6	2131	0.01171	11	23

Table 5. A search results of different difficulties of puzzles with seed "54321"*

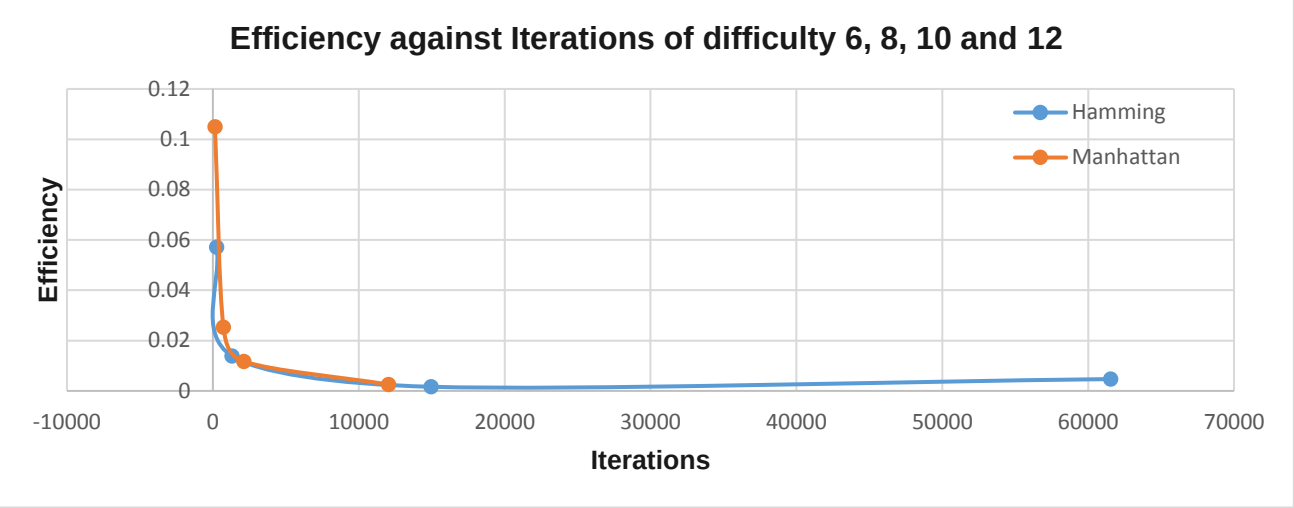


Figure 3.1 Relationship between the efficiency and iterations of Hamming and Manhattan

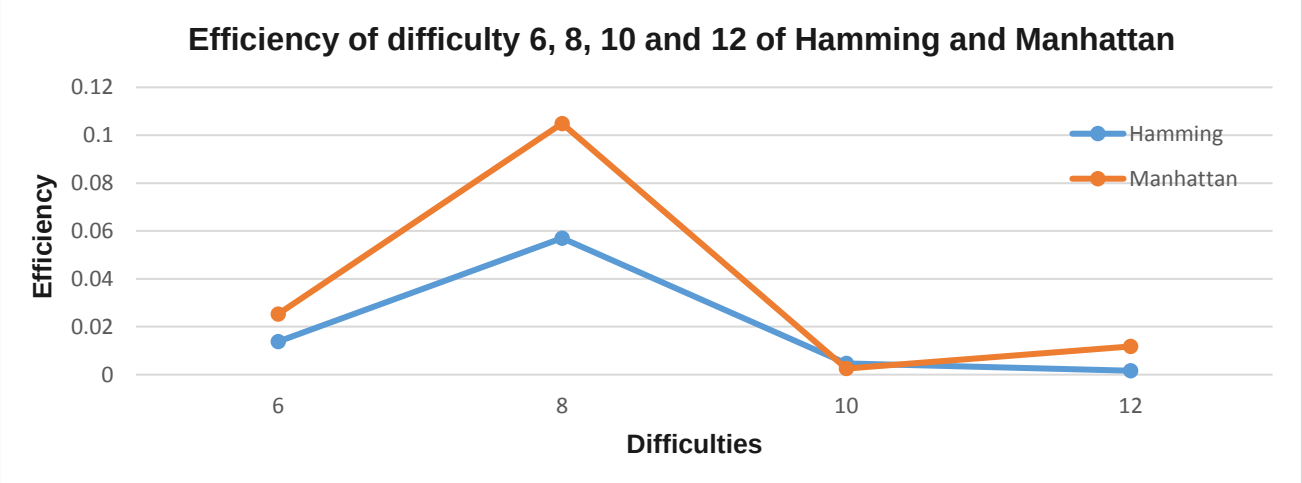


Figure 3.2 Comparison of efficiency between puzzles of different difficulties with seed "54321"

First, by comparing the results in **Table 1**, **Table 2**, and **Table 3**, it has been shown that A* Search is more efficient than Breadth-First Search. Both Hamming and Manhattan have higher efficiency and require less iteration to solve **P1**, **P2**, and **P3**. This is because the A* Search Algorithm calculates the estimated total cost required from the initial state to the goal state, and select the path with the minimum estimated total cost. Breadth-First Search is inefficient for eight puzzle because it goes through all the nodes one by one, and for the harder puzzle like **P4** and **P5** which require more moves in **Table 2**, the search tree is large and requires about 30 minutes for the search to conclude.

In **Table 3**, Hamming has higher efficiency than Manhattan for **P1**, **P2**, and **P3**, but lower for **P4** and **P5**. By comparing the estimated total cost against the global cost of Hamming of Manhattan, it has been clearly shown that the closer the estimated total cost is to the global cost, the higher the efficiency. For the easier puzzle like **P1**, **P2**, and **P3**, both Hamming and Manhattan have overestimated the cost but Manhattan's estimate is worse than Hamming, which results in a lower efficiency and more iterations.

Apart from **P1**, **P2**, and **P3**, Manhattan has better estimates, higher efficiency, and less iteration in all other tests. There is no evidence that the estimated total cost is closer to the global cost when the difficulty of the puzzle increases. According to **Figure 1.1**, **Figure 2.1**, and **Figure 3.1**, the rate of decrease of efficiency of Manhattan is lower than Hamming, which can be concluded that Manhattan is more suitable in solving harder puzzles than Hamming, and it takes a shorter time for the search to complete.

Average Efficiency Comparison

$$Average\ \ Efficiency\ \ of\ \ Hamming = \frac{\sum Efficiency}{Number\ \ of\ \ tests} = \frac{0.1192231}{8}$$

$$Average\ \ Efficiency\ \ of\ \ Manhattan = \frac{\sum Efficiency}{Number\ \ of\ \ tests} = \frac{0.268316}{8}$$

$$\therefore \frac{Average\ \ efficiency\ \ of\ \ Manhattan}{Average\ \ efficiency\ \ of\ \ Hamming} = \frac{0.268316}{0.1192231} \approx 2.251$$

The average efficiency of Manhattan is 2.25 times faster than the average efficiency of Hamming. Therefore, it has been proved that Manhattan is more efficient than Hamming because it has a closer value of estimated remaining cost.

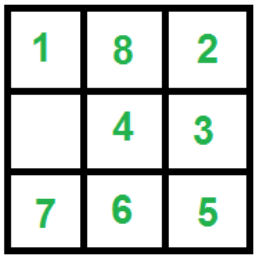
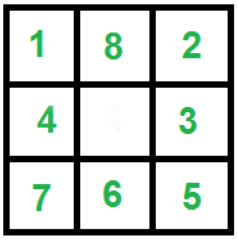


Image 1. An eight puzzle example

Hamming			Manhattan			Global Cost
Iterations	Efficiency	Est. Total Cost	Iterations	Efficiency	Est. Total Cost	
31	0.3226	7	12	0.8333	11	9

Table 6. A search results of puzzle in Image 1*

To improve the estimated total cost, one possible method is by comparing the estimated total cost of the parent state with the current state. As the local cost of the eight puzzle is always 1, this indicates that the difference between the previous and current state is at most one tile. For example, in **Image 1** above, the next two possible states are:



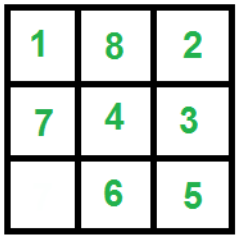


Image 2 and Image 3. First and second possible states

The estimated total cost of both **Image 2** and **Image 3** must be at most 1 more than the estimated total cost of **Image 1**. During the experiments, most of the time the estimated total cost of the current state is at least 1 more than the previous state, which causes the estimated total cost to be overestimated sometime. Below is an example of a searching process:

======Start of Node======
Parent(previous state):
5 2 3
1 0 4
7 8 6

State(current):

5 2 3
1 4 0
7 8 6
Local cost: 1
Global cost: 6
Est. remaining cost: 6
Est. total cost: 12
=======End of Node=======

======Start of Node======
Parent(previous state):
5 2 3
1 4 0
7 8 6

State(current):
5 2 0
1 4 3
7 8 6
Local cost: 1
Global cost: 7
Est. remaining cost: 8
Est. total cost: 15
=======End of Node=======

======Start of Node======
Parent(previous state):
5 2 0
1 4 3
7 8 6

State(current):
5 0 2
1 4 3
7 8 6
Local cost: 1
Global cost: 8
Est. remaining cost: 10
Est. total cost: 18
=======End of Node=======

The estimated total cost increased from 12 to 15, and then 15 to 18, which might suggest that the starting path chosen may not be the most optimal. If we use this limitation, then the search algorithm may choose the path where all the difference between estimated total cost of previous and current state is at most 1, this could prevent the estimated total cost to be overestimated.

Testing (results copied from command line)

RunMethodTest.java in Part1 folder:

Testing the methods in PuzzleSearch.java

Testing method getTargetPuzzle(), test print of TARGET_TEST
1 2 3
4 5 6
7 8 0

==========
Testing the methods in PuzzleState.java

Testing method getCurrentPuzzle(), test print of TEST_1
1 0 3
4 2 6
7 5 8
The empty tile row of TEST_1 is: 0 // Row 0
The empty tile column of TEST_1 is: 1 // Column 1
Testing swapTile() method, swap (0, 1) with (1, 1) :
1 2 3
4 0 6
7 5 8

Testing method getCurrentPuzzle(), test print of TEST_2
0 1 2
3 4 5
6 7 8
The empty tile row of TEST_2 is: 0 // Row 0
The empty tile column of TEST_2 is: 0 // Column 0
Testing swapTile() method, swap (0, 0) with (0, 1) :
1 0 2
3 4 5
6 7 8
Testing method copyPuzzle(), which returns a deep copy of TEST_2
0 1 2
3 4 5
6 7 8
The empty tile row of test3 is: 0 // Row 0, same as TEST_2
The empty tile column of test is: 0 // Column 0, same as TEST_2

Testing the sameState() method, using test3:
Is TEST_1 same as TEST_1 : true // same puzzle

```
Is TEST_1 same as TEST_2 : false        // two different puzzles
Is TEST_1 same as test3 : false         // two different puzzles
Is TEST_2 same as test3 : false         // test3 is a deep copy of TEST_2

Testing the goalP() method
Is TEST_1 a goal : false
Is TEST_2 a goal : false
Is TARGET_TEST a goal : true
```

RunMethodTest.java in Part2 folder, duplicates are removed in here:

```
Testing method getDistanceMethod()
The testPuzzleSearch1 distance method is: hamming
The testPuzzleSearch2 distance method is: manhattan

=========================
Testing the methods in PuzzleState.java

Testing method getCurrentPuzzle(), test print of TEST_2
8 6 7
2 5 4
3 0 1
The empty tile row of TEST_2 is: 2
The empty tile column of TEST_2 is: 1
Testing swapTile() method, swap (2, 1) with (1, 1):
8 6 7
2 0 4
3 5 1

Testing method copyPuzzle(), which returns a deep copy of TEST_2
8 6 7
2 5 4
3 0 1
The empty tile row of test3 is: 2        // Row 2, same as TEST_2
The empty tile column of test is: 1      // Column 1, same as TEST_2
getHammingDistance() method test
This is the target state: 0        // 0
This is the puzzle used in the assignment document: 7        // 7
This is one of the hardest puzzle (TEST_2): 8        // 8

getManhattanDistance() method test
This is the target state: 0        // 0
This is the puzzle used in the assignment document: 16        // 16
This is one of the hardest puzzle (TEST_2): 22        // 22
```