



SSE: Fall 2024

CSC 4301: Intro to AI

Project 1: Report

Team: Amine Lahnin, Omar Ismail,

Mohammed Harouak

Table of Contents:

I. Introduction:	3
II. Task 1: <i>Transforming 8-puzzle to 15-puzzle</i>	5
III. Task 2: <i>Implementing Heuristics</i>	10
a) Heuristic 1: <i>Number of Misplaced Tiles</i>	10
b) Heuristic 2: <i>Euclidean Distance</i>	13
c) Heuristic 3: <i>Manhattan Distance</i>	16
d) Heuristic 4: <i>Tiles out of row and column</i>	19
IV. Task 3: <i>Heuristic comparison</i>	22
V. Task 4: <i>Comparing the best heuristic with uninformed algorithms</i>	24
VI. Conclusion:	25

I. Introduction:

This project aims to study and comprehend the use of heuristic-informed search algorithms to solve search problems. The project revolves around the eight-puzzle problem in transforming it to a fifteen-puzzle problem, and then applying various search algorithms to determine which works best, more efficient, and why. Both puzzles share similar criteria that classify them as search problems; they both consist of:

- State Space: Confined in a 3x3 grid for the eight puzzles, and a 4x4 grid for the fifteen puzzles, where the blank tile can move in.

- Successor Function: Which allows the blank tile to move from one state to another within the bounds of the state space.

- Start State: The randomly generated beginning state for each puzzle.

- Goal Test: a test function to assess whether the puzzle is solved. This occurs for the eight puzzle only when the cells are ordered in an ascending order:

	1	2
3	4	5
6	7	8

As for the fifteen puzzle, the solution must have the blank space at the bottom right of the table, with the cells also being ordered in an ascending order:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

The project will therefore cover the following tasks:

- Task 1: Transforming an eight puzzle into a fifteen puzzle.
- Task 2: Implementing four different heuristics for the A* algorithm in the fifteen puzzle.
- Task 3: Comparing the heuristics to determine the most performant.
- Task 4: Comparing the winning heuristic with other uninformed search algorithms (BFS, DFS, and Uniform Cost Search).

II. Task 1: Transforming 8-puzzle to 15-puzzle.

The 8-puzzle operates in a 3x3 grid; 1 blank cell, and 8 other cells each containing a number from 1 to 8. The 15-puzzle on the other hand, should instead operate in a 4x4 grid; 1 blank cell and 15 other cells, hence the name 15-puzzle. To support a 4x4 table, the following code had to be modified as so:

- The row & col count changed both to 4 in the FifteenPuzzleState constructor where the table cells are initialized, and the isGoal(self) function where it handles goal checking of the current state:

```
30 32      def __init__( self, numbers ):  
31 33          ""  
32 34          Constructs a new fifteen puzzle from an ordering of numbers.  
@@ -41,28 +43,21 @@ def __init__( self, numbers ):  
41 43          self.cells = []  
42 44          numbers = numbers[:] # Make a copy so as not to cause side-effects.  
43 45          numbers.reverse()  
46 +          # /*====Start Change Task 1====*/  
44 47          for row in range( 4 ):  
45 48              self.cells.append( [] )  
46 49              for col in range( 4 ):  
47 50                  self.cells[row].append( numbers.pop() )  
48 51                  if self.cells[row][col] == 15:  
52 +                  # /*====End Change Task 1====*/  
49 53          self.blanklocation = row, col
```

Figure 1: FifteenPuzzleState constructor in *fifteenpuzzle.py*

```

51 55      def isGoal( self ):
52      -      """
53      -          Checks to see if the puzzle is in its goal state.
54      -
55      -          -----
56      -          |   | 1 | 2 |
57      -          -----
58      -          | 3 | 4 | 5 |
59      -          -----
60      -          | 6 | 7 | 8 |
61      -          -----
62      -      """
63 56      current = 0
57 +      # /*====Start Change Task 1====*/
64 58      for row in range( 4 ):
65 59      for col in range( 4 ):
66 60 +      # /*====End Change Task 1====*/
66 61      if current != self.cells[row][col]:
67 62      return False
68 63      current += 1

```

Figure 2: *isGoal()* function in *fifteenpuzzle.py*

-The `legalMoves(self)` function has been changed to not allow movement beyond the newly extended table borders ($\text{row} < 3$ for down, $\text{col} < 3$ for right):

```

@@ -80,17 +75,21 @@ def legalMoves( self ):
80 75         row, col = self.blankLocation
81 76         if(row != 0):
82 77             moves.append('up')
83 78 +         # /*====Start Change Task 1====*/
84 79         if(row != 3):
85 80 +         # /*====End Change Task 1====*/
86 81             moves.append('down')
87 82         if(col != 0):
88 83             moves.append('left')
89 84 +         # /*====Start Change Task 1====*/
90 85         if(col != 3):
91 86 +         # /*====End Change Task 1====*/
92 87             moves.append('right')
93 88         return moves

```

Figure 3: legalMoves() function in *fifteenpuzzle.py*

-The ASCII drawing function getAsciiString(self) has been modified to handle drawing a larger 4x4 grid:

```

@@ -154,8 +154,10 @@ def __getAsciiString(self):
154 154         for row in self.cells:
155 155             rowLine = '|'
156 156             for col in row:
157 157 +         # /*====Start Change Task 1====*/
158 158             col = col+1
159 159             if col == 16:
160 160 +         # /*====End Change Task 1====*/
161 161             col = ' '
162 162             rowLine = rowLine + ' ' + col.__str__() + ' |'
163 163             lines.append(rowLine)

```

Figure 4: getAsciiString function in *fifteenpuzzle.py*

-The rest of the changes in the fifteenpuzzle.py file represent name refactoring of classes and functions to better suit the FifteenPuzzle context, as well as commenting out the unused loadFifteenPuzzle:

```

17 17
18 18     # Module Classes
19 19
20 + #/*=====Start Change Task 1=====*/
20 21     class FifteenPuzzleState:
21 22         """
22 -         The Eight Puzzle is described in the course textbook on
23 +         The Fifteen Puzzle is described in the course textbook on
23 24         page 64.
24 25
25 26         This class defines the mechanics of the puzzle itself. The
26 27         task of recasting this puzzle as a search problem is left to
27 -         the EightPuzzleSearchProblem class.
28 +         the FifteenPuzzleSearchProblem class.
28 29         """
29 30
31 +         # *=====End Change Task 1 =====*/

```

Figure 5: FifteenPuzzleState constructor in *fifteenpuzzle.py*

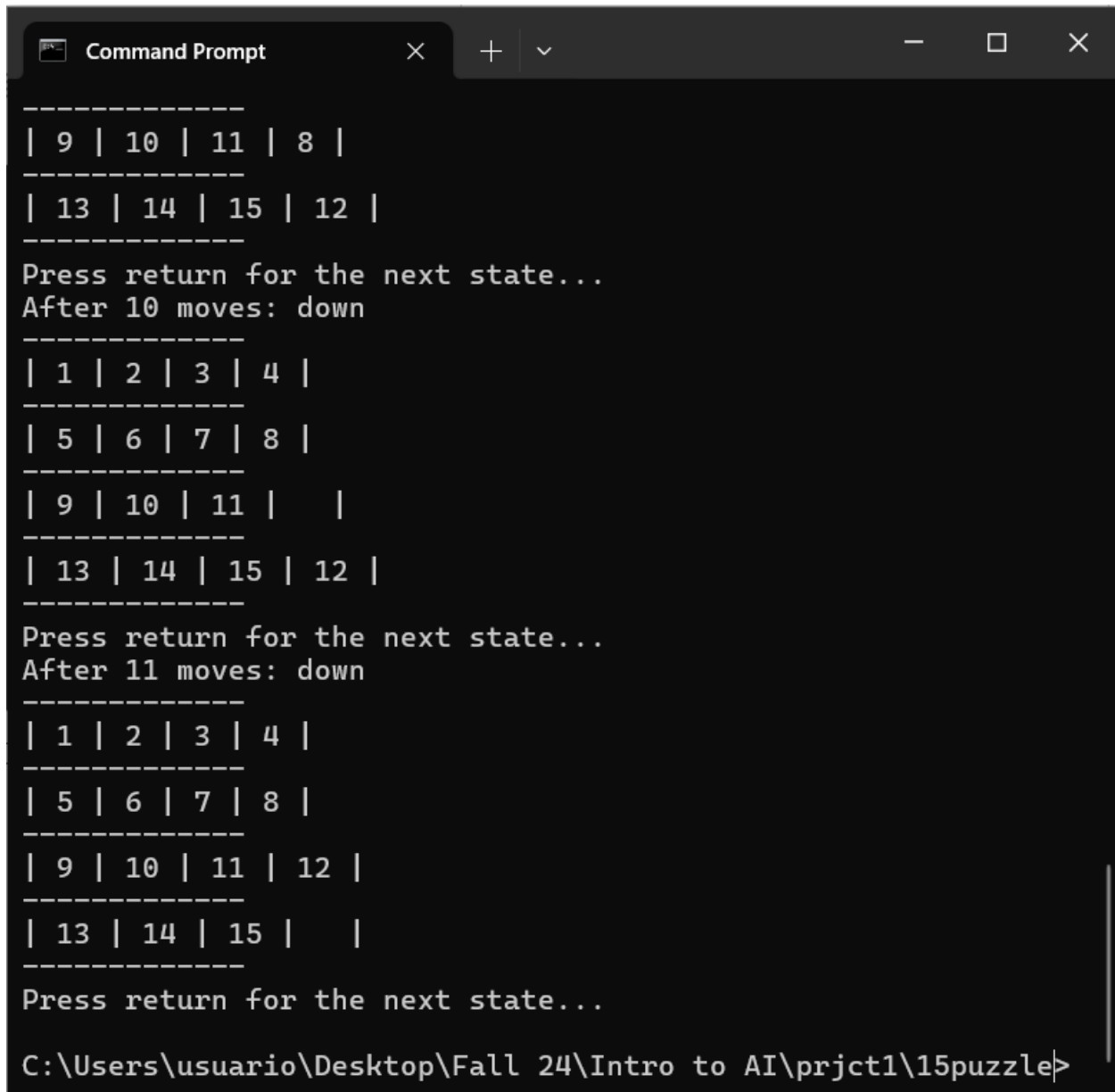
-Execution Trace:

Running the fifteenpuzzle.py file:


```
Command Prompt - python fi × + ▾
C:\Users\usuario\Desktop\Fall 24\Intro to AI\prjct1\15puzzle>
python fifteenpuzzle.py
A random puzzle:
-----
| 1 | 2 | 3 | 4 |
-----
| 5 | 10 | 6 | 7 |
-----
| 9 |   | 8 | 12 |
-----
| 13 | 11 | 14 | 15 |
-----
A* found a path of 11 moves: ['down', 'right', 'right', 'up',
'left', 'left', 'up', 'right', 'right', 'down', 'down']
After 1 move: down
-----
| 1 | 2 | 3 | 4 |
-----
| 5 | 10 | 6 | 7 |
-----
| 9 | 11 | 8 | 12 |
-----
| 13 |   | 14 | 15 |
-----
Press return for the next state...|
```

Figure 6: Running the fifteenpuzzle.py

-Reaching the goal state:



```
-----  
| 9 | 10 | 11 | 8 |  
-----  
| 13 | 14 | 15 | 12 |  
-----  
Press return for the next state...  
After 10 moves: down  
-----  
| 1 | 2 | 3 | 4 |  
-----  
| 5 | 6 | 7 | 8 |  
-----  
| 9 | 10 | 11 |  |  
-----  
| 13 | 14 | 15 | 12 |  
-----  
Press return for the next state...  
After 11 moves: down  
-----  
| 1 | 2 | 3 | 4 |  
-----  
| 5 | 6 | 7 | 8 |  
-----  
| 9 | 10 | 11 | 12 |  
-----  
| 13 | 14 | 15 |  |  
-----  
Press return for the next state...  
C:\Users\usuario\Desktop\Fall 24\Intro to AI\prjct1\15puzzle>
```

Figure 7: Puzzle solution execution trace

III. Task 2: *Implementing Heuristics.*

a) **Heuristic 1:** *Number of Misplaced Tiles.*

The first heuristic consists of calculating the number of misplaced tiles of the current state compared to the goal state.

This is achievable by looping through each cell of the grid and checking whether the value of that current cell matches the intended value in the goal state. The current variable increments in each iteration of the loop, and then the function performs a check whether the goal state cell value matches with the current cell value ; if not true, then the number of misplaced tiles should increment, which would represent the value of the heuristic.

```
def misplacedTilesHeuristic(state, problem=None):
    """
    This heuristic function takes a state and estimates the cost to the nearest
    goal as equal to the number of misplaced tiles
    """
    current = 0
    misplaced_tile_count = 0
    for row in state.cells:
        for col in row:
            if(col != current):
                misplaced_tile_count += 1
            current += 1
    return misplaced_tile_count
```

*Figure 8: Heuristic 1 function implementation in **search.py***

Usage in the A* function:

For all the heuristics, replacing the heuristic call with the intended one shall suffice:

```

if not already_explored:
    frontier.push(newNode, newCost + heuristic(succState, problem))

    frontier.push(newNode, newCost + misplacedTilesHeuristic(succState, problem))

    exploredNodes.append((succState, newCost))
    # /*****Start Change Task 3*****/
    maxFringeSize = max(maxFringeSize, len(frontier))

```

Figure 9: Heuristic integration in `aStarSearch()` function of `search.py`

-Execution Trace:

```

Command Prompt - python fi  X + v

C:\Users\usuario\Desktop\Fall 24\Intro to AI\prjct1\15puzzle>python fifteenpuzzle.py
A random puzzle:
-----
| 1 | 6 | 2 | 3 |
-----
| 5 | 8 | 7 | 4 |
-----
| 9 | 10 | 15 | 11 |
-----
| 13 | 14 |   | 12 |
-----
A* found a path of 11 moves: ['up', 'up', 'left', 'up', 'right', 'right',
'down', 'left', 'down', 'right', 'down']
After 1 move: up
-----
| 1 | 6 | 2 | 3 |
-----
| 5 | 8 | 7 | 4 |
-----
| 9 | 10 |   | 11 |
-----
| 13 | 14 | 15 | 12 |
-----
Press return for the next state...|

```

Figure 10: `fifteenpuzzle.py` execution with heuristic 1

```
Command Prompt
-----
| 9 | 10 |   | 11 |
-----
| 13 | 14 | 15 | 12 |
-----
Press return for the next state...
After 10 moves: right
-----
| 1 | 2 | 3 | 4 |
-----
| 5 | 6 | 7 | 8 |
-----
| 9 | 10 | 11 |   |
-----
| 13 | 14 | 15 | 12 |
-----
Press return for the next state...
After 11 moves: down
-----
| 1 | 2 | 3 | 4 |
-----
| 5 | 6 | 7 | 8 |
-----
| 9 | 10 | 11 | 12 |
-----
| 13 | 14 | 15 |   |
-----
Press return for the next state...
C:\Users\usuario\Desktop\Fall 24\Intro to AI\prjct1\15puzzle>
```

Figure 11: *fifteenpuzzle.py* completion with heuristic 1

b) Heuristic 2: *Euclidean Distance*.

The second heuristic utilizes Euclidean distance to measure the distance between each cell in the current state, and its goal destination in the goal state. The value of the heuristic is the sum of all the latter values measured using Euclidean distance.

To achieve such results, we implemented two functions:

-euclideanDistance function which takes in two coordinates of points (in this case the cells in the 2D plane which is the tab represented by the puzzle's grid), and calculates the distance between the two points using the following formula:

$$d(p, q) = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2}.$$

With (p, q) representing a coordinate system.

```
#!/=====Start Change Task 2=====*/

def euclideanDistance(xy1, xy2):
    distance_squared = ((xy1[0] - xy2[0])**2 + (xy1[1] - xy2[1])**2)
    distance = math.sqrt(distance_squared)
    return distance
```

Figure 12: euclideanDistance function in *util.py*

-getFinalPosition function which transforms a particular cell into two coordinates used for the euclideanDistance function:

```
def getFinalPosition(val, size=4):
    """
    This is a function that takes in a number and return a tuple representing the
    coordinates of that value in a game of the given size
    """
    row = val // size
    col = (val%size)
    return (row,col)
#!/=====End Change Task 2=====*/
```

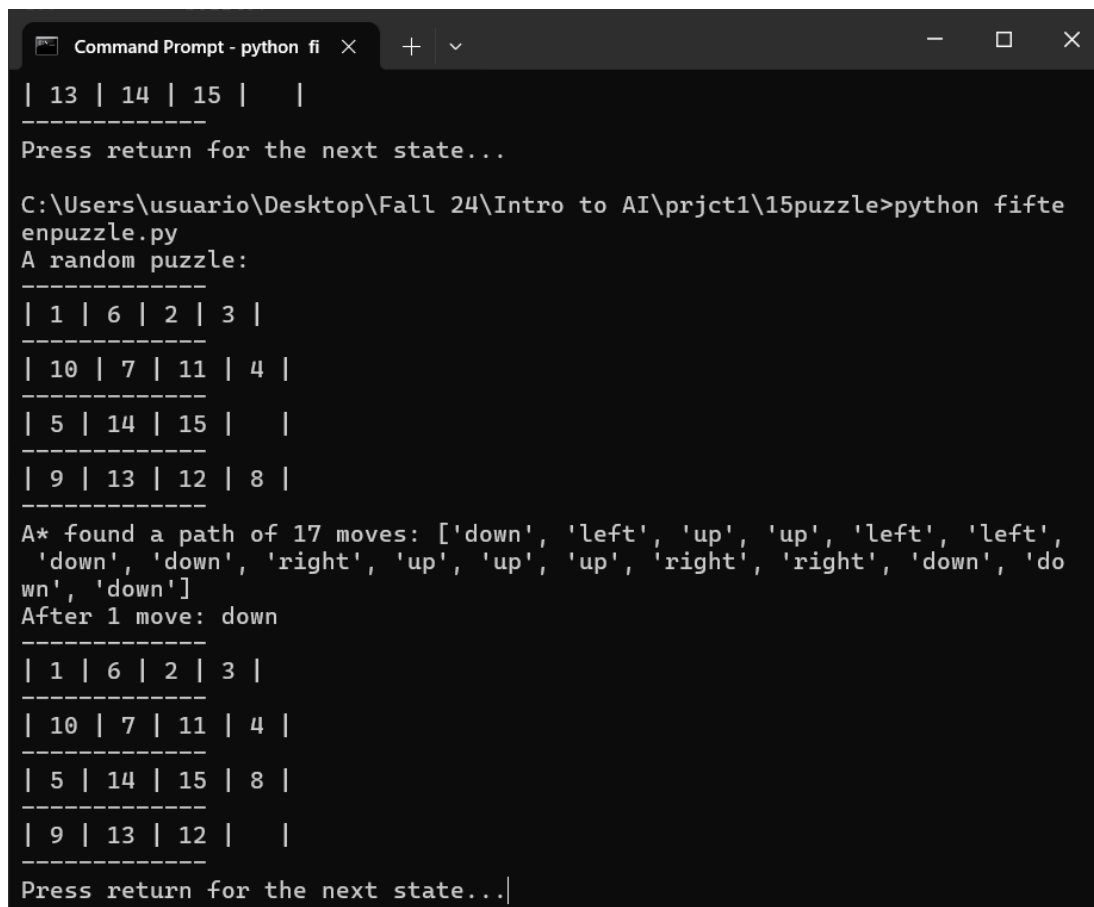
Figure 13: getFinalPosition function in *util.py*

-We use both functions to implement the euclideanHeuristic. Looping through each cell of the current state, and calculating its euclidean distance with the cell of the goal state, then summing the values iteratively for all cells in that table.s

```
def euclideanHeuristic(state, problem=None):
    total_distance = 0
    for (row_count,row) in enumerate(state.cells):
        for (col_count,col) in enumerate(row):
            # get the final position of this value
            eventual_row, eventual_col = util.getFinalPosition(col)
            total_distance += util.euclideanDistance((eventual_row, eventual_col),(row_count,col_count))
    return total_distance
```

Figure 14: euclideanHeuristic function in *search.py*

-Execution Trace:



```
Command Prompt - python fi
| 13 | 14 | 15 |  |
-----
Press return for the next state...

C:\Users\usuario\Desktop\Fall 24\Intro to AI\prjct1\15puzzle>python fifteenpuzzle.py
A random puzzle:
-----
| 1 | 6 | 2 | 3 |
-----
| 10 | 7 | 11 | 4 |
-----
| 5 | 14 | 15 |  |
-----
| 9 | 13 | 12 | 8 |
-----
A* found a path of 17 moves: ['down', 'left', 'up', 'up', 'left', 'left',
'down', 'down', 'right', 'up', 'up', 'up', 'right', 'right', 'down', 'down']
After 1 move: down
-----
| 1 | 6 | 2 | 3 |
-----
| 10 | 7 | 11 | 4 |
-----
| 5 | 14 | 15 | 8 |
-----
| 9 | 13 | 12 |  |
-----
Press return for the next state...|
```

Figure 15: *fifteenpuzzle.py* execution with heuristic 2

```
Command Prompt
-----
| 9 | 10 | 11 | 8 |
-----
| 13 | 14 | 15 | 12 |
-----
Press return for the next state...
After 16 moves: down
-----
| 1 | 2 | 3 | 4 |
-----
| 5 | 6 | 7 | 8 |
-----
| 9 | 10 | 11 |   |
-----
| 13 | 14 | 15 | 12 |
-----
Press return for the next state...
After 17 moves: down
-----
| 1 | 2 | 3 | 4 |
-----
| 5 | 6 | 7 | 8 |
-----
| 9 | 10 | 11 | 12 |
-----
| 13 | 14 | 15 |   |
-----
Press return for the next state...
C:\Users\usuario\Desktop\Fall 24\Intro to AI\prjct1\15puzzle>
```

Figure 16: *fifteenpuzzle.py* completion with heuristic 2

c) Heuristic 3: *Manhattan Distance*.

The third heuristic shares a similar approach to that of the second heuristic. The only difference being the use of manhattan distance rather than the euclidean one.

Which calculates the number of increments (i.e. movements) of a given cell in the current state necessary to reach the final position of the cell in the goal state. To do

this we can leverage the coordinate system we have used to create the manhattan distance function following this formula:

$$|x_1 - x_2| + |y_1 - y_2|$$

With (x1, y1) and (x2, y2) representing two points in the coordinate system.

```
def manhattanDistance( xy1, xy2 ):
    "Returns the Manhattan distance between points xy1 and xy2"
    return abs( xy1[0] - xy2[0] ) + abs( xy1[1] - xy2[1] )
```

Figure 17: manhattanDistance function in util.py

-The implementation of the Manhattan distance heuristic thus becomes almost identical to that of the Euclidian one, with the only difference being the use of the manhattanDistance function.

```
def manhattanHeuristic(state, problem=None):
    total_distance = 0
    for (row_count, row) in enumerate(state.cells):
        for (col_count, col) in enumerate(row):
            # get the final position of this value
            if(col == 15):
                continue
            eventual_row, eventual_col = util.getFinalPosition(col)
            total_distance += util.manhattanDistance((eventual_row, eventual_col), (row_count, col_count))
    # print(state)
    # print(total_distance)
    return total_distance
```

Figure 18: manhattanHeuristic function in search.py

-Execution Trace:

```
Command Prompt - python fi  X + v - □ X
-----
| 13 | 14 | 15 |   |
-----
Press return for the next state...

C:\Users\usuario\Desktop\Fall 24\Intro to AI\prjct1\15puzzle>python fifte
enpuzzle.py
A random puzzle:
-----
| 1 | 6 | 2 | 4 |
-----
|   | 5 | 3 | 8 |
-----
| 9 | 14 | 7 | 12 |
-----
| 13 | 11 | 10 | 15 |
-----
A* found a path of 11 moves: ['right', 'up', 'right', 'down', 'down', 'do
wn', 'left', 'up', 'right', 'down', 'right']
After 1 move: right
-----
| 1 | 6 | 2 | 4 |
-----
| 5 |   | 3 | 8 |
-----
| 9 | 14 | 7 | 12 |
-----
| 13 | 11 | 10 | 15 |
-----
Press return for the next state...|
```

Figure 19: *fifteenpuzzle.py* execution with heuristic 3

```
Command Prompt
-----
| 9 | 10 |   | 12 |
-----
| 13 | 14 | 11 | 15 |
-----
Press return for the next state...
After 10 moves: down
-----
| 1 | 2 | 3 | 4 |
-----
| 5 | 6 | 7 | 8 |
-----
| 9 | 10 | 11 | 12 |
-----
| 13 | 14 |   | 15 |
-----
Press return for the next state...
After 11 moves: right
-----
| 1 | 2 | 3 | 4 |
-----
| 5 | 6 | 7 | 8 |
-----
| 9 | 10 | 11 | 12 |
-----
| 13 | 14 | 15 |   |
-----
Press return for the next state...
C:\Users\usuario\Desktop\Fall 24\Intro to AI\prjct1\15puzzle>
```

Figure 20: *fifteenpuzzle.py* completion with heuristic 3

d) Heuristic 4: Tiles out of row and column

The fourth heuristic function calculates the sum of misplaced tiles in terms of the row and column positions. For each difference in the row and column placement of the tiles compared to their final position, the total estimated cost is incremented by 1. We can calculate the heuristic by looping through each cell of the current state

whilst checking whether the current cells are positioned in the same row and column as the goal state's counterpart cells. If not, then the estimated cost will increment for each difference.

```
def tilesOutOfRowAndColHeuristic(state, problem=None):
    total_estimated_cost = 0
    for (row_count, row) in enumerate(state.cells):
        for (col_count, col) in enumerate(row):
            # get the final position of this value
            eventual_row, eventual_col = util.getFinalPosition(col)
            if(eventual_row != row_count):
                total_estimated_cost += 1
            if(eventual_col != col_count):
                total_estimated_cost += 1
    return total_estimated_cost
```

Figure 21: tilesOutOfRowAndColHeuristic function in search.py

-Execution Trace:

```
Command Prompt - python fi X + v
-----
| 13 | 14 | 15 |  |
-----
Press return for the next state...

C:\Users\usuario\Desktop\Fall 24\Intro to AI\prjct1\15puzzle>python fifteenpuzzle.py
A random puzzle:
-----
| 1 | 2 | 3 | 4 |
-----
| 6 | 9 | 7 | 8 |
-----
| 5 | 14 | 10 | 12 |
-----
| 13 | 11 |  | 15 |
-----
A* found a path of 9 moves: ['left', 'up', 'up', 'left', 'down', 'right',
'right', 'down', 'right']
After 1 move: left
-----
| 1 | 2 | 3 | 4 |
-----
| 6 | 9 | 7 | 8 |
-----
| 5 | 14 | 10 | 12 |
-----
| 13 |  | 11 | 15 |
-----
Press return for the next state...|
```

Figure 22: *fifteenpuzzle.py* execution with heuristic 4

```
Command Prompt
-----
| 9 | 10 |  | 12 |
-----
| 13 | 14 | 11 | 15 |
-----
Press return for the next state...
After 8 moves: down
-----
| 1 | 2 | 3 | 4 |
-----
| 5 | 6 | 7 | 8 |
-----
| 9 | 10 | 11 | 12 |
-----
| 13 | 14 |  | 15 |
-----
Press return for the next state...
After 9 moves: right
-----
| 1 | 2 | 3 | 4 |
-----
| 5 | 6 | 7 | 8 |
-----
| 9 | 10 | 11 | 12 |
-----
| 13 | 14 | 15 |  |
-----
Press return for the next state...
C:\Users\usuario\Desktop\Fall 24\Intro to AI\prjct1\15puzzle>
```

Figure 23: *fifteenpuzzle.py* completion using heuristic 4

IV. Task 3: *Heuristic comparison.*

For this task we created a function called `automate_heuristics_tests` that generates a bunch of random 15 puzzles and solves each puzzle with the a start using all the heuristics mentioned above. The max fringe size, time of execution, number of nodes expanded, and maximum depth of each iteration was recorded. Then the average of each heuristic was taken across all puzzles.

We created 1000 puzzles to make the results statistically significant. The following table summarizes the findings:

Heuristic	Average max fringe size	Average tree depth	Average Execution time (s)	Average # nodes expanded
Misplaced Tiles	90.75	9.490	0.0540	173
Euclidean Heuristic	42.22	9.490	0.0054	79
Manhattan heuristic	26.57	9.498	0.0010	48
Tiles out of row and col count	35.33	9.498	0.0049	67.35

From the table it appears that the Manhattan heuristic is the fastest and uses the minimum fringe size and expands less nodes than all other heuristics. This could be attributed to the fact that it is the exact distance the tile needs to reach its destination had there been no movement restrictions. So, it gives a good approximation of how much is left to actually reach its final position. The results show a close tree depth between all heuristics. This could be explained by noting that the heuristics are all searching within the same depth. But the Manhattan heuristic allows the algorithm to expand less siblings to reach its goal.

V. **Task 4:** *Comparing the best heuristic with uninformed algorithms*

For this part we compared the best performing heuristic which is the Manhattan Heuristic to uninformed search algorithms. Namely, DFS, BFS, and UCS (Uniform Cost Search). The max fringe size, tree depth, time spent, and # nodes expanded were recorded. The algorithms were initially compared on one puzzle that was created by applying 3 moves on a solved state. The algorithms were run on an intel core i7 for 4 hours without converging. Which shows how slow uninformed algorithms are compared to informed search. To have concrete comparison values the same problem was run on one puzzle with 2 applied moves and the following values were recorded.

algorithm	Maximum fringe size	Tree depth	Execution time (ms)	# nodes expanded
dfs	60	38	0.0065	118
bfs	6	2	0.0000	8
ucs	6	2	0.0000	8
A* with manhattan	3	1	0.0000	4

This simple example is enough to highlight the significant deficiency of uninformed search in terms of performance. While A* needs only to expand 4 nodes and use 3 slots at the fringe at most. DFS needed to expand over 100 nodes and use 60 slots on the fringe to find the solution although it is one move away

from the initial state. BFS and UCS show better performance because the solution is closer to the initial state. But they still are behind A* because they need to search every sibling per level. In this case UCS and BFS become identical because there is no cost function which reduces the priority queue used for UCS to a normal queue.

VI. Conclusion:

In conclusion, it is apparent that informed search is much superior to uninformed search across all metrics. Running 1000 samples using different heuristics for A* was much faster than solving 1 puzzle using DFS. Across all heuristics the Manhattan Distance Heuristic was by far the best performing one.