Assignment 1 CS170: Introduction to Artificial Intelligence, Dr. Eamonn Keogh

Aryan Singh
SID-862394968
email-asing301@ucr.edu
Date: May-15-2023
In completing this assignment I consulted:

- The Blind Search and Heuristic Search lecture slides and notes by Professor Eamonn Keogh.
- Python 2.7.14, 3.5, and 3.6 Documentation.
- For the randomly generated puzzles:https://appzaza.com/tile-slide-game

All important code is original.

Outline of this report:

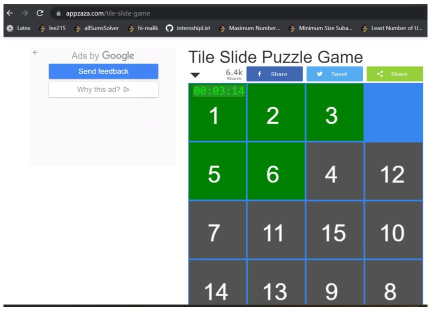
- Cover page: (this page)
- My report: Pages 2 to 7.
- Sample trace on an easy problem, page 8.
- Sample trace on a hard problem, page 10.
- My code pages 11 to 17. Note that in case you want to run my code, here is a URL that points to it online https://github.com/stayaryan/AICS205P1

CS205 : Project 1-The eight-puzzle

Aryan Singh, SID-862394968, May-15-2023

Introduction

The sliding tile 8-puzzle is a popular puzzle game that consists of a 3x3 grid with eight numbered tiles and one empty space. The goal of the game is to rearrange the tiles by sliding them into the empty space in order to achieve a specific target configuration. The tiles can only be moved vertically or horizontally, and the empty space serves as a "buffer" that allows for the movement of tiles.



The image on the side is the first time I saw the 15-puzzle.(https://appzaza.com/tile-slide-game)

The 8-puzzle is a classic example used in the field of artificial intelligence to study search algorithms and heuristic techniques. It serves as a challenging problem due to the large number of possible states and the need to find an optimal solution with the minimum number of moves.

This report documents the completion of the first project in Dr. Eamonn Keogh's Introduction to AI course at the University of California, Riverside, during the Fall 2023 quarter. The project focuses on the implementation of Uniform Cost Search and the application of two heuristics, namely Misplaced Tile and Manhattan Distance, in the context of the A* algorithm. The project was implemented using Python version 3. This write-up provides a comprehensive overview of the project, including the chosen programming language, key concepts, implemented algorithms, and the full code used in the project.

Comparison Of Algorithms

Uniform Cost Search Algorithm:

UCS explores the search space by considering all possible moves from a given state and assigning a cost to each move. In the case of the 8-puzzle, the cost can be defined as the number of moves taken to reach a particular state. The algorithm maintains a priority queue of states, prioritized by their cumulative costs. It starts with the initial state and continues expanding states with the lowest cumulative cost until it reaches the goal state.

During the search, UCS systematically explores the search space, considering all possible moves and their associated costs. It expands states in a breadth-first manner, ensuring that the lowest-cost path is discovered. This property makes UCS optimal for finding the optimal solution to the 8-puzzle problem.

However, UCS may suffer from high time complexity and memory requirements, especially in larger puzzles or when the goal state is far from the initial state. This is because it explores all possible paths, leading to an exponential growth in the number of states to be considered. In such cases, more informed search algorithms like A* with heuristics such as the Misplaced Tile or Manhattan Distance can be more efficient by guiding the search towards more promising paths.

The Misplaced Tile Heuristic:

The Misplaced Tile heuristic works by counting the number of tiles that are not in their goal position. In the 8-puzzle, each tile should ideally occupy a specific position in the grid. If a tile is not in its correct position, it is considered misplaced. The heuristic value is determined by counting the number of misplaced tiles in the current state.

For example, in the goal state of the 8-puzzle:

1 2 3
4 5 6
7 8 0

If a state is:

1 2 3
4 5 0
7 8 6

The Misplaced Tile heuristic would return a value of 2, as the tiles 6 and 0 are in the incorrect positions.

The Misplaced Tile heuristic serves as an admissible heuristic, meaning it never overestimates the actual cost to reach the goal. This property guarantees that if the heuristic value is used in an informed search algorithm like A*, the algorithm will always find the optimal solution.

However, the Misplaced Tile heuristic may not be very effective in guiding the search process efficiently. It does not consider the distance between the misplaced tiles and their goal positions, leading to suboptimal search behavior in certain situations. In such cases, heuristics like the

Manhattan Distance can provide more accurate estimates and lead to faster convergence to the optimal solution.

The Manhattan distance Heuristic:

This heuristic calculates the sum of the Manhattan distances between each tile's current position and its goal position. The Manhattan distance is the sum of the horizontal and vertical distances between two points on a grid. In the 8-puzzle, it represents the minimum number of moves required to move each tile to its goal position, considering only horizontal and vertical movements.

For example, in the goal state of the 8-puzzle:

1 2 3
4 5 6
7 8 0

If a state is:

1 2 3
4 5 0
7 8 6

The Manhattan Distance heuristic would calculate the following distances for each misplaced tile:

Tile 0: Manhattan distance = 1 (distance from (2,1) to (2,2))
Tile 6: Manhattan distance = 1 (distance from (2,2) to (2,3))
The heuristic value is the sum of these distances, which in this case would be 2.

The Manhattan Distance heuristic is admissible, meaning it never overestimates the actual cost to reach the goal. It considers both the horizontal and vertical distances between tiles, providing a more accurate estimate compared to the Misplaced Tile heuristic. This allows the informed search algorithm to make more informed decisions about which states to explore, leading to faster convergence to the optimal solution.

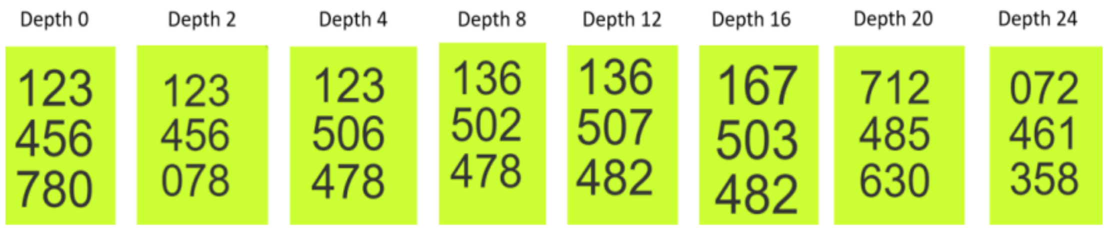
General Working of my Code

1. It prompts the user to choose between using a default puzzle or entering their own puzzle.
2. If the user chooses to enter their own puzzle, they input the values for the puzzle grid.
3. The user then selects the algorithm they want to use: Uniform Cost Search, A* with Misplaced Tile heuristic, or A* with Manhattan Distance heuristic.

4. The main function, generalsearch, is called with the puzzle, chosen algorithm, and the goal state.
5. The program initializes variables for time tracking, queue size, nodes visited, and max queue size.
6. The heuristic value is calculated based on the chosen algorithm.
7. The start node is created, with the puzzle, depth of 0, and heuristic value. It is added to the queue and marked as seen.
8. The main search loop begins:
    a. The queue is sorted based on the heuristic value.
    b. The first node is removed from the queue and its expanded flag is set.
    c. If the goal state is reached, the program returns the solution.
    d. The node is expanded, creating child nodes based on the possible moves of the empty tile.
    e. The child nodes are assigned the depth and heuristic value based on the chosen algorithm.
    f. The child nodes are added to the queue and marked as seen.
    g. The max queue size is updated if needed.

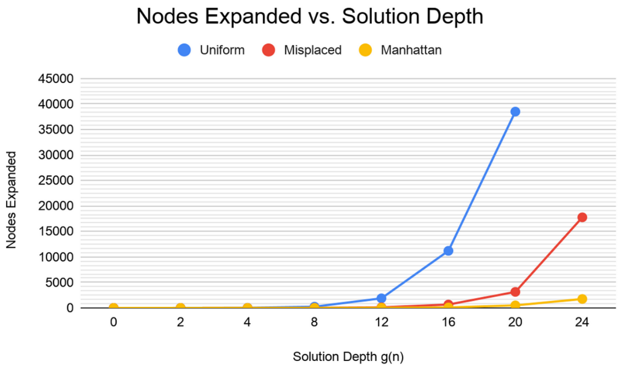
9)If no solution is found, the program returns "No Solution Found".

Comparison of Algorithms on Sample Puzzles:

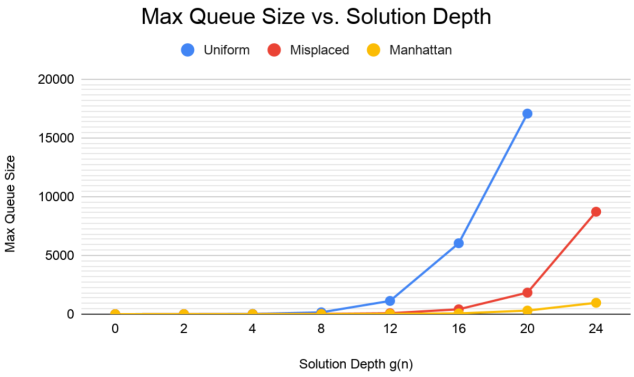


Depth 0	Depth 2	Depth 4	Depth 8	Depth 12	Depth 16	Depth 20	Depth 24
123	123	123	136	136	167	712	072
456	456	506	502	507	503	485	461
780	078	478	478	482	482	630	358

Taking six tests provided by Professor Keogh and running three different algorithms (Uniform Cost Search, A* search with Manhattan Distance heuristic, and A* search with Misplaced Tile heuristic) on them allows for a comprehensive evaluation of the algorithms' performance in solving the 8-puzzle problem.



"Nodes expanded" refers to the number of distinct puzzle configurations that were explored or expanded by the algorithm during its search process.



The "max queue size" metric provides insights into the space requirements of a search algorithm during its execution. It refers to the maximum number of elements present in the search queue at any given point during the algorithm's execution.

There is a certain pattern in max Queue size, number of nodes expanded and time of execution. They all seem to rise exponentially with increase in complexity of the input. In fact in the worst case it is O(b^d) where b is the branching factor, or the number of children nodes a node branches into at a particular depth.

For the depth 24 puzzle:

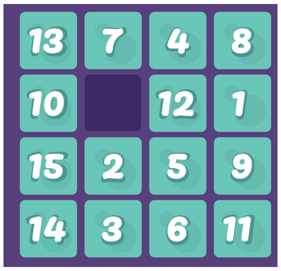
1) **Uniform cost Search:**

->expanded a total of 38294nodes.
->The maximum number of nodes in the queue at any one time was 18321.
->CPU Time: 18 minutes 12 seconds

2) **A* Algorithm with misplaced tile heuristic:**
->expanded a total of 17768 nodes.
->The maximum number of nodes in the queue at any one time was 8717.
->CPU Time: 120.989 seconds or 2 minutes

3) **A* Algorithm with manhattan distance heuristic:**
->expanded a total of 1757 nodes.
->The maximum number of nodes in the queue at any one time was 967.
->CPU Time: 2.814 seconds

Additional Example:



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

As an additional example I have taken a sample 15 puzzle problem (shown beside) that I was given on the website https://15puzzle.netlify.app/ when I hopped on to it.
It took A* search using manhattan distance 23 minutes and 37 seconds to reach the goal state. I did not use Uniform cost search(UCS) since it would take a longer time since all moves are treated equally hence I did not see the point in trying and waiting for UCS to finish.

Conclusion:

For the three algorithms, in conclusion, it can be said that:

- In the case of the 8-puzzle problem, the misplaced tile heuristic, which counts the number of tiles that are not in their correct positions. It assigns a value of 1 for each misplaced tile. This heuristic does not take into account the actual distances the misplaced tiles need to move to reach their correct positions. Due to the more informed nature of the Manhattan distance heuristic, it tends to guide the search more efficiently towards the goal state by prioritizing nodes that are closer to the goal. This often leads to a reduction in the number of expanded nodes compared to the misplaced tile heuristic. Hence heuristic with Manhattan distance will be faster in searching than misplaced tiles. And UCS will be slowest of them all simply owing to the fact that it searches all the unseen cases to make a move.

- Space complexity wise, all three algorithms are approximately the same, but Manhattan distance heuristic and Misplaced tile heuristic will operate on more memory at a given time because it has to store the heuristic about current state as well as the heuristics of previous states in a queue along with a lot of metadata.

Trace path On the hard depth 24 problem given by professor:
Using A* manhattan distance heuristic

```
The best state to expand with a g(n) = 0 and h(n) = 14 is...
[[0, 7, 2], [4, 6, 1], [3, 5, 8]]      Expanding this node...

The best state to expand with a g(n) = 1 and h(n) = 13 is...
[[7, 0, 2], [4, 6, 1], [3, 5, 8]]      Expanding this node...

The best state to expand with a g(n) = 2 and h(n) = 12 is...
[[7, 2, 0], [4, 6, 1], [3, 5, 8]]      Expanding this node...

The best state to expand with a g(n) = 3 and h(n) = 11 is...
[[7, 2, 1], [4, 6, 0], [3, 5, 8]]      Expanding this node...

The best state to expand with a g(n) = 4 and h(n) = 10 is...
[[7, 2, 1], [4, 0, 6], [3, 5, 8]]      Expanding this node...

The best state to expand with a g(n) = 5 and h(n) = 9 is...
[[7, 2, 1], [4, 5, 6], [3, 0, 8]]      Expanding this node...

The best state to expand with a g(n) = 6 and h(n) = 8 is...
[[7, 2, 1], [4, 5, 6], [0, 3, 8]]      Expanding this node...
```

```
The best state to expand with a g(n) = 6 and h(n) = 8 is...
[[7, 2, 1], [4, 5, 6], [3, 8, 0]]      Expanding this node...

The best state to expand with a g(n) = 1 and h(n) = 15 is...
[[4, 7, 2], [0, 6, 1], [3, 5, 8]]      Expanding this node...

The best state to expand with a g(n) = 2 and h(n) = 14 is...
[[7, 6, 2], [4, 0, 1], [3, 5, 8]]      Expanding this node...

The best state to expand with a g(n) = 2 and h(n) = 14 is...
[[4, 7, 2], [3, 6, 1], [0, 5, 8]]      Expanding this node...

The best state to expand with a g(n) = 3 and h(n) = 13 is...
[[7, 6, 2], [4, 1, 0], [3, 5, 8]]      Expanding this node...
```

I've left a lot of traces

```
The best state to expand with a g(n) = 20 and h(n) = 4 is...
[[1, 3, 6], [4, 5, 2], [7, 8, 0]]      Expanding this node...

The best state to expand with a g(n) = 20 and h(n) = 4 is...
[[1, 3, 2], [4, 8, 6], [7, 5, 0]]      Expanding this node...

The best state to expand with a g(n) = 20 and h(n) = 4 is...
[[1, 2, 3], [4, 0, 6], [5, 8, 7]]      Expanding this node...

The best state to expand with a g(n) = 21 and h(n) = 3 is...
[[1, 2, 3], [0, 4, 6], [7, 5, 8]]      Expanding this node...

The best state to expand with a g(n) = 21 and h(n) = 3 is...
[[1, 0, 3], [4, 2, 6], [7, 5, 8]]      Expanding this node...

The best state to expand with a g(n) = 22 and h(n) = 2 is...
[[1, 2, 3], [4, 0, 6], [7, 5, 8]]      Expanding this node...

The best state to expand with a g(n) = 23 and h(n) = 1 is...
[[1, 2, 3], [4, 5, 6], [7, 0, 8]]      Expanding this node...
```

```
Goal!!

To solve this problem the search algorithm expanded a total of 1757 nodes.
The maximum number of nodes in the queue at any one time was 967.
The depth of the goal node was 24

CPU Time: 2.522892713546753 seconds
```

Trace path on easy problem (depth 2) given by professor:
Using Uniform cost search algorithm:

```
The best state to expand with a g(n) = 0 and h(n) = 0 is...
[[1, 2, 3], [4, 5, 6], [0, 7, 8]]      Expanding this node...

The best state to expand with a g(n) = 1 and h(n) = 0 is...
[[1, 2, 3], [4, 5, 6], [7, 0, 8]]      Expanding this node...

The best state to expand with a g(n) = 1 and h(n) = 0 is...
[[1, 2, 3], [0, 5, 6], [4, 7, 8]]      Expanding this node...

Goal!!

To solve this problem the search algorithm expanded a total of 3 nodes.
The maximum number of nodes in the queue at any one time was 4.
The depth of the goal node was 2

CPU Time: 0.0025463104248046875 seconds
```

MY CODE:

```
import time
import sys
import math
from node import node
from heuristics import manhattanDistanceHeuristic, misplacedTileheuristic
from expand import expansionOfStates
def main():
    # Getting user input for if they want to use default or their own
    choice = int(input('Please type “1” to use a default puzzle, or “2” to enter your own puzzle!
Make sure to enter a valid puzzle\n Press enter after enterin one number\n'))

    # Setting up puzzle if user uses a custom puzzle
    if choice == 1:
        puzzle = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]
        goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    elif choice == 2:
        print('Enter N of the N-puzzle')
        N = int(input())
        n = int(math.sqrt(N+1))
        k = 1
        goal = []
        puzzle = []
        for j in range(1, n+1):
            temp = []
            tempGoal = []
            for i in range(1, n+1):
                tempGoal.append(k)
                k+=1
                element = int(input())
                temp.append(element)
            puzzle.append(temp)
            goal.append(tempGoal)
        goal[n-1][n-1] = 0

        print('\n')
    # Allowing the user to choose heuristic
    qf = int(input('Enter your choice of algorithm \n1. Uniform Cost Search '
                '\n2. A* with the Misplaced Tile heuristic. \n3. A* with the Manhattan distance
heuristic\n'))

    # Running the program and printing the output
    print(generalsearch(puzzle, qf, goal))

def generalsearch(problem, queueingFunction, goal):

    # The code snippet captures the starting time of the general search algorithm
    InitTime = time.time()
#'q' represents the queue that holds the nodes to be explored in the search algorithm.
#'statesSeen' is a list that keeps track of the puzzle states that have already been encountered
during the search.
#'numberOfNodesVisited' counts the total number of nodes that have been visited or expanded
during the search.
#'queueSize' keeps track of the current size of the queue.
#'maxQ' stores the maximum size that the queue has reached at any point during the search.
#These variables play essential roles in managing the search process and gathering statistics
about the search algorithm's performance.
    q = []
    statesSeen = []
    numberOfNodesVisited = -1
    queueSize = 0
    maxQ = -1
    # Calculating heuristic based on the user inputted heuristic
    if queueingFunction == 1:
        heuristicVal = 0
    if queueingFunction == 2:
        heuristicVal = misplacedTileheuristic(problem, goal)
    if queueingFunction == 3:
        heuristicVal = manhattanDistanceHeuristic(problem, goal)

#creates the initial node for the search algorithm. It includes the puzzle configuration,
#  sets the depth of the node to 0, and calculates the heuristic value. The node is then
# added to the queue and included in the statesSeen array to keep track of visited states.
    n = node(problem)
    n.heuristicCost= heuristicVal
    n.depth = 0
    q.append(n)

    statesSeen.append(n.puzzle)
    queueSize +=1
    maxQ += 1

    while True:
#Instead of using a conventional sorting approach, the code utilizes a lambda function to
# perform a faster sorting based on a combination of the heuristic value (h(n)) and the depth
#  (g(n)) of each node. The sorting prioritizes the lowest value of h(n) + g(n). In case of a tie,
# the sorting considers the depth as a secondary criterion.
        if queueingFunction != 1:
            q = sorted(q, key=lambda x: (x.depth + x.heuristicCost, x.depth))

        # An empty queue will lead to a dead end
        if len(q) == 0:
            return 'Cannot find Solution'

        # The code removes the first node from the queue, increments the count of visited nodes,
a
        # nd decreases the size of the queue. This process is performed to track the progress of
the
        # search algorithm and manage the queue effectively.
        nodeDefinition = q.pop(0)
        if nodeDefinition.expanded is False:
            numberOfNodesVisited += 1
            nodeDefinition.expanded = True
        queueSize -= 1

        # Printinf statistics when we reach goal state
        if isGoal(nodeDefinition.puzzle, goal):
            return ('Goal!! \n\nTo solve this problem the search algorithm expanded a total of ' +
                  str(numberOfNodesVisited) + ' nodes.\nThe maximum number of nodes in the queue
at any one time was '
                  + str(maxQ) + '.\nThe depth of the goal node was ' + str(nodeDefinition.depth) +
'\n\nCPU Time: ' +
                    str(time.time()-InitTime) + ' seconds')

        print('The best state to expand with a g(n) = ' + str(nodeDefinition.depth) + ' and h(n) = ' +
str(nodeDefinition.heuristicCost)
                  + ' is...\n' + str(nodeDefinition.puzzle) + '\tExpanding this node...\n')
        # Expand all possible states from the node popped off the queue and put them into child
nodes
        branchedNodes = expansionOfStates(nodeDefinition, statesSeen)

        # The code iterates through the array of child nodes and adjusts their statistics based on
the
        # selected heuristics. The depth of each child node is set to the depth of its parent node
        # (the node that was popped off the queue) plus 1. This ensures that the depth of each
child
        #  node is incremented correctly in relation to its parent.
        arrayOfChildren = [branchedNodes.leftChild, branchedNodes.rightChild,
branchedNodes.downChild, branchedNodes.upChild]
```

```
        for i in arrayOfChildren:
            if i is not None:
                if queueingFunction == 1:
                    i.depth = nodeDefinition.depth + 1
                    i.heuristicCost = 0
                elif queueingFunction == 2:
                    i.depth = nodeDefinition.depth + 1
                    i.heuristicCost = misplacedTileheuristic(i.puzzle, goal)
                elif queueingFunction == 3:
                    i.depth = nodeDefinition.depth + 1
                    i.heuristicCost = manhattanDistanceHeuristic(i.puzzle, goal)

                # Add these states to the queue and add them to a list of states we have now seen
                q.append(i)
                statesSeen.append(i.puzzle)
                queueSize += 1

        # Change the max queue size if it has been surpassed
        if queueSize > maxQ:
            maxQ = queueSize

def isGoal(puzzle, goal):
    if puzzle == goal:
        return True
    return False
#The function isGoal(puzzle, goal) checks whether the given puzzle configuration matches the
# goal configuration. It compares the puzzle and goal to determine if they are identical. If they
# are the same, indicating that the puzzle has reached the goal state, the function returns True.
# Otherwise, it returns False, indicating that the puzzle is not yet in the goal state.
if __name__ == "__main__":
    main()

class node:
    def __init__(self, puzzle):
        self.puzzle = puzzle
        self.heuristicCost = 0
        self.depth = 0
        self.leftChild = None
        self.rightChild = None
        self.upChild = None
        self.downChild = None
        self.expanded = False
#The node class in this program represents a state in the 8-puzzle problem. It stores the
# current puzzle configuration, the depth of the node in the search tree, the heuristic
# cost associated with the node, and four child nodes. The four child nodes correspond to the
# possible moves of the empty tile (0) in the puzzle. Additionally, the node has an expanded
# boolean flag that indicates whether it has been expanded during the search process.

def manhattanDistanceHeuristic(puzzle, goal):
    count = 0
    goalRow, goalColumn, puzzleRow, puzzleColumn = 0, 0, 0, 0

    for val in range(1, len(puzzle)*len(puzzle)):
        for i in range(len(puzzle)):
            for j in range(len(puzzle)):
                if int(puzzle[i][j]) == val:
                    puzzleRow = i
                    puzzleColumn = j
                if goal[i][j] == val:
                    goalRow = i
                    goalColumn = j
        count += abs(goalRow-puzzleRow) + abs(goalColumn-puzzleColumn)
    return count
#The function manhattanDistanceHeuristic(puzzle, goal) calculates the Manhattan distance
heuristic
# for the given puzzle configuration. It computes the total number of moves required to bring
each
# tile (except the empty tile) from its current position to its correct position in the goal
# configuration. The function iterates through the puzzle and goal configurations, tracking the
# positions of each value. For each value from 1 to 9 (or the size of the puzzle), it calculates
# the vertical and horizontal distance between the current position in the puzzle and the
# corresponding position in the goal configuration. These distances are summed up to obtain
the
#  total Manhattan distance. Finally, the function returns the total count as the heuristic value.

def misplacedTileheuristic(puzzle, goal):
    count = 0

    for i in range(len(puzzle)):
        for j in range(len(puzzle)):
            if int(puzzle[i][j]) != goal[i][j] and int(puzzle[i][j]) != 0:
                count+=1
    return count
#The function misplacedTileHeuristic(puzzle, goal) calculates the misplaced tile heuristic for the
# given puzzle configuration. It counts the number of tiles that are not in their correct
# positions in relation to the corresponding positions in the goal configuration. The function
#  iterates through the puzzle and goal configurations, comparing the values at each position.
# If a tile in the puzzle is different from the corresponding tile in the goal configuration and
# it is not the empty tile (0), the count is incremented. Finally, the function returns the
# total count of misplaced tiles as the heuristic value.

from node import node
import copy
#The function expansionOfStates(nodeDefinition, s) expands the given node by generating its
child
# nodes based on the possible moves of the empty tile (0) in the puzzle configuration.
def expansionOfStates(nodeDefinition, s):
    row = 0
    col = 0
#First, it determines the position of the empty tile in the puzzle. Then, it checks four possible
moves:
#  left, right, up, and down.
    for i in range(len(nodeDefinition.puzzle)):
        for j in range(len(nodeDefinition.puzzle)):
            if nodeDefinition.puzzle[i][j] == 0:
                row = i
                col = j
#If the empty tile is not in the first column, it creates a new child node by swapping the empty
tile
# with the tile to its left (column-wise). The new puzzle configuration is stored in the left variable,
#  and if this configuration has not been encountered before (not present in set s), a new child
node
# is created with the left configuration.
    if col > 0:
        left = copy.deepcopy(nodeDefinition.puzzle)
        temp = left[row][col]
        left[row][col] = left[row][col - 1]
        left[row][col - 1] = temp

        if left not in s:
            nodeDefinition.leftChild = node(left)
#If the empty tile is not in the last column, it creates a new child node by swapping the empty tile
# with the tile to its right (column-wise). The new puzzle configuration is stored in the right
variable,
#  and if it is not in set s, a new child node is created
    if col < len(nodeDefinition.puzzle)-1:
        right = copy.deepcopy(nodeDefinition.puzzle)
        temp = right[row][col]
        right[row][col] = right[row][col+1]
        right[row][col+1] = temp

        if right not in s:
            nodeDefinition.rightChild = node(right)
#If the empty tile is not in the first row, it creates a new child node by swapping the empty tile
with
# the tile above it (row-wise). The new puzzle configuration is stored in the up variable, and if it
is
# not in set s, a new child node is created.
    if row > 0:
        up = copy.deepcopy(nodeDefinition.puzzle)
        temp = up[row][col]
        up[row][col] = up[row - 1][col]
        up[row - 1][col] = temp

        if up not in s:
            nodeDefinition.downChild = node(up)
#If the empty tile is not in the last row, it creates a new child node by swapping the empty tile
with
# the tile below it (row-wise). The new puzzle configuration is stored in the down variable, and if
it
# is not in set s, a new child node is created.
    if row < len(nodeDefinition.puzzle) - 1:
        down = copy.deepcopy(nodeDefinition.puzzle)
        temp = down[row][col]
        down[row][col] = down[row + 1][col]
        down[row + 1][col] = temp

        if down not in s:
            nodeDefinition.upChild = node(down)
    #Finally, the function returns the updated nodeDefinition with the generated child nodes.
    return nodeDefinition
```