Intelligent N-Puzzle Solver

Team ID: 76

Project Number: 16

Time Slot: 10:20 Am

القسم	المستوى الدراسي	رقم الطالب	الاسم
CS	المستوى الثالث	20220302	علياء علي حلمي علي
CS	المستوى الثالث	20220193	زينب سيد معتمد قناوي
CS	المستوى الثالث	20220169	رؤى محمد ابراهيم عواد
CS	المستوى الثالث	20220506	منة أحمد هليل عبدالتواب
IS	المستوى الثالث	20220161	رضوى جمال محمد مصطفى
CS	المستوى الثالث	20220333	فاطمة مصطفي عبدالعال نادي

Overview

Project idea

The N-Puzzle solver project aims to provide an efficient and user-friendly solution to the classic N-Puzzle problem, which involves rearranging a set of numbered tiles on a grid to achieve a specific goal configuration. This project uses the Best first search function alongside 5 different heuristic functions to optimize the search for the solution, providing users with a visual representation of the solving process.

The application implements several heuristic functions that guide the search process:

- Misplaced Tiles: counts the number of tiles that are not in their goal positions.
- Misplaced Tiles with Path Cost heuristic: is an extension of the basic
- Misplaced Tiles heuristic. This heuristic not only counts the number of tiles that are out of place but also incorporates the cost associated with the moves taken to reach the current state.
- Manhattan Distance: calculates the total distance each tile must move to reach its goal position.
- Euclidean Distance: computes the straight-line distance from each tile's current position to its goal position.
- Linear Conflict: This advanced heuristic builds upon the Manhattan Distance by adding penalties for tiles that are in the same row or column but in the wrong order.

By incorporating a GUI, built with Tkinter, the application engages users in solving the puzzle allowing them to observe how different heuristics affect the search process. Users can see the current configuration of the tiles and the steps taken by the algorithm to reach the solution.

Similar applications in the market

- N Puzzle (web app):
  https://n-puzzle.baletskyi.me/
- Numpuz: Classic Number Games (mobile app):
  https://play.google.com/store/apps/details?id=com.dopuz.klotski.riddle
- Number Puzzle Mosaic (desktop app):
  https://apps.microsoft.com/detail/9NH0584PFLRM?hl=en-us&gl=EG&ocid=pdpshare

Literature Review of Academic publications

- Investigating the Impact of Different Search Strategies (Breadth First, Depth First, A*, Best First, Iterative Deepening, Hill Climbing) on 8-Puzzle Problem Solving - A Case Study
  https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4378553
- Finding Optimal Solutions to the Twenty-Four Puzzle
  https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=a3b936d9302d793a9438b3d2adbdd1924fb0d42a
- A Comparative Study of A* and Greedy Best-First Search Algorithms in Solving 8-Puzzle Game
  https://www.ijssass.com/index.php/ijssass/article/view/157
- Comparison of A* Algorithm and Greedy Best Search in Searching Fifteen Puzzle Solution
  https://www.journalijisr.com/sites/default/files/issues-pdf/IJISRR-941.pdf
- Experimental Comparison of Uninformed and Heuristic AI Algorithms for N Puzzle and 8 Queen Puzzle Solution
  https://www.academia.edu/download/48301664/IJDIWC_Vol4__Issue1.pdf#page=146

Applied Algorithms

➢ goalState(n)

This function generates the goal state for the N-Puzzle game, where n specifies the total number of tiles (e.g., 8, 15, or 24). The goal state represents the solved configuration of the puzzle.

- Example: 8 tiles: A 3x3 grid with numbers 1–8 arranged in order and a blank space (0) in the last position.

If an unsupported size is provided, the function raises a ValueError with an appropriate error message.

➢ possibleNextState(state)

This function returns an array with the possible next states of that specific tile, the only possible movements of the blank tile is Up, Down, Left and Right, so we check if its possible to move the blank to these positions if so then add it to the "newStates" array and then return it, the reason of using the deepCopy function is that in python if we updated any value in the passed 'state' array this update will be reflected in the original array and we do not want this to happen, so we use deepCopy to just copy the values inside the state array without affecting on its reference value.

➢ check_solvability(state)

To check solvability, we used three utility functions:

1. flatten_2d_array(state)
2. inversion_count:
3. blank_pos

1. flatten 2d array:

This function converts a 2D puzzle array into a 1D array
-used as check solvability handles with 1d array and randomize initial state send the puzzle as 2d array to it

ex:

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

output:[1, 2, 3, 4, 0, 5, 6, 7, 8]

----

2. inversion count:

An inversion is when a larger numbered tile precedes a smaller numbered tile in the puzzle. This function counts the total number of such inversions.

Input:1d array of puzzle
output: number of inversions
Ex: input: [3, 2, 1, 4, 0, 5]
Output:(3,2),(3,1),(2,1)=3 inversions

3.blank_pos:
This function calculates the position of the blank tile (denoted as 0) from the bottom of the grid.
Input:1d array of puzzle

- output: index of blank tile from bottom

Steps :
1.calculates grid size sqrt(arrlength)
Ex: 8 puzzle >has length:8+1(0 blank)= 9 therefore sqrt(9)=3
Therefore grid size=3X3
2.calculate the blank from top (0 index) : blank_index // grid_size
3. convert it to index from bottom : grid_size - row_from_top_index
Ex:
Input: [1, 2, 3, 4, 5, 6, 7, 8, 0] output:1
How: 1. Sqrt(8)=3
    2.Index(0)=8, 3.row of blank=8//3=2 (from top)
    3.index from bottom=3-2=1#
Logic:
Puzzle is solvable:
1.when grid size is odd (3X3= 8 puzzle ,5X5 =24 puzzle)
  And inversion count is even
2.when grid size is even (4X4 = 15 puzzle) and inversion count is even and blank pos is odd
3.grid size is even (4X4 =15 puzzle) and inversion count is odd and blank index is even.

- randomizedInitialState(goalState, moves)

This function generates a randomized initial state for the N-Puzzle game based on a given goalState.
Customizability: The number of random moves (moves) can be adjusted as needed to control the level of randomization.
Solvability Check: the function ensures the generated state is solvable. If the state is not solvable, it continues generating and checking new states until a solvable configuration is found.
The function returns a valid, solvable initial state.

- find_position(tile, goalState)

  This function returns the index of the tile in the goal state, as an example if we called this function findPosition(3,goalState), the return of this function is 1,2 means that the tile '3' lies on the second row and the third column in the goal state.

- misplacedTiles(state, goalState)

  This heuristic function returns the number of the misplaced tiles by making a comparison between every index in the passed state with that of the goal state

- misplacedTilesAndPathCost(state, goalState)

  This is the same as the previous one, the difference is the same as I said before in the bestFirstSearch function when it checks if the heuristic function is misplacedTilesAmdPathCost then adds the path cost to the heuristic value of the generated state and then push it into the frontier.

- manhattenDistance(state, goalState)

  This function returns the distance from the tiles to their goal by adding the difference between the absolute values of the state's indexes and its goal's indexes.

- euclideanDistance(state, goalState)

  This heuristic function calculates the total Euclidean distance between the current state and the goal state.

  For each tile in the puzzle (excluding the blank tile, 0), the function computes the straight-line distance (Euclidean distance) from its current position to its goal position. The Euclidean distance for each tile is calculated using the formula:

$$\text{Distance} = \sqrt{(i-igoal)2+(j-jgoal)2}$$

where i, j are the current row and column of the tile in the state, and igoal, jgoal are the row and column of the tile in the goal state.
The function then returns the sum of the Euclidean distances for all tiles, providing an estimate of how far the current state is from the goal state.

➢ linear_conflit(state, goalState)

This heuristic function extends the Manhattan Distance by adding penalties for linear conflicts—situations where two tiles are in the same row or column but in the wrong order, which increases the number of moves required to solve the puzzle.
Input: initial state(2d array) and goal state
Steps :

1. Initialize a goal state dictionary for each tile to its position in goal state

Ex: 1:(0,0) tile 1
 2:(0,1) tile 2
 0:(2,2) >blank

2.check for row conflict:

2.1. if tile is not blank then check if goal row is the same as current row

i==goalCurrentTileRow

2.1.1 If yes compare with the other tiles in the same row (next columns)

for q in range(j+1,grid_size) >>j =column

2.2 If goal of tile in current column > goal of tile in next column and goal row =current row therefore : row conflict happens

Ex: Current Row: [2, 1, 3]
 Goal Row: [1, 2, 3]
Row conflict 2 and 1 out of order

```
3.check for column conflict:
    3.1 if tile not blank check if tile column is goal column
                j == goalCurrentTileCol
        3.1.1 if yes compare with other tiles in the same column (next
        rows)
                for k in range(i + 1, grid_size)
            3.1.1.1 If goal column= current column and goal of tile in
            next row<current tile
            Column conflict happens
            Ex:  Current Column: [2, 1, 3]
            Goal Column: [1, 2, 3]
            Conflict: Tiles 2 and 1 are out of order.
4. Combine Manhattan Distance and Linear Conflict
    4.1Compute Manhattan Distance using the function
    manhattanDistance(state, goal)
    4.2 get total conflict= row conflict+column conflict
    4.3 add penality   :  total_penlity = (conflict) * 2 + manhattan_distance
    Why *2? As each conflict requires atleast two moves to go to its goal
    position on same row or column.
```

- bestFirstSearch(initialSatet, goalState, heuristicFunction)

This function is responsible for choosing the best path or solution for the search. At first we create an empty array of the path (we will need it later in our code), then initialize the pathCost to 0, and we will initialize a frontier to be the list that contains all the states that have been reached in the frontier and not have been visited, also contains the heuristic value of that state (will be needed later), also contain the path (will be the solution at the end) and pathCost, then we will turn this frontier into a heap to make it easy to extract/pop the minimum value inside that list, then we will loop over this heap frontier until it becomes empty, in each iteration we will pop that queue with the minimum value according to the heauristic value of the states inside that queue/list, if we found out that this popped state is the goal state then we will return the path plus that state as the solution path, if not then we will add this state to the visitedStates array to not return back to that state and make a loop, then we will call the “possibleNextStates” function and pass the popped state to it so it will return back all the possible states that the blank tile can go through, then we will loop on that returned array and check if that generatedState is visited then skip, if not then (lets at first assume that the heuristic function is not misplacedTilesAndPathCost to make it easier to understand for now) push that generatedState into the frontierQ with its heuristic value, and its path (add that current state that we have sent to the function possibleNextStates earlier to the path then pass it) and pathCost as 0, else if the heuristic function is misplacedTilesAndPathCost then we will need the depth/path cost of that state because it’s essential in that type of heuristic functions, so every time we enter this if condition we increment the pathCost by 1 and also push that generatedState to the frontier with that path cost and the heuristic value of that function + that path cost, and the path is the old path plus the current state as before ,we will keep doing those steps until we find the goal state and then we will return the path + that state as the solution path of that popped state.

Experiments and Results

In order to be able to extract results from the five heuristic function to compare there performance, we applied a fixed initial state for the five of them during execution, one state for each puzzle size.

Results of 8-puzzle

Heuristic function	Number of steps
Misplaced Tiles	30
Misplaced Tiles and Path Cost	16
Manhattan Distance	16
Euclidean Distance	44
Linear Conflict	16

Results of 15-puzzle

Heuristic function	Number of steps
Misplaced Tiles	84
Misplaced Tiles and Path Cost	58
Manhattan Distance	56
Euclidean Distance	86
Linear Conflict	56

Results of 24-puzzle

Heuristic function	Number of steps
Misplaced Tiles	220
Misplaced Tiles and Path Cost	110
Manhattan Distance	108
Euclidean Distance	259
Linear Conflict	105

Analysis

Insights

Which heuristic is faster?

- In terms of search speed:

    1. Linear conflict
    2. Manhattan Distance
    3. Misplaced tiles and path cost
    4. Misplaced tiles and Euclidean

Linear conflict is fastest (in terms of search not computation) because it reduces the number of nodes expanded by providing the most informed estimate. But in terms of computation, it requires a lot of comparing per tile.

- In terms of computation (simplicity of computation):

    1. Misplaced Tiles
    2. Misplaced Tiles with Path Cost
    3. Manhattan Distance
    4. Euclidean Distance
    5. Linear conflict

- Larger puzzles (search space):

So linear conflict is more useful in larger search space like 15 and 24 puzzle as it is more accurate and reduces the search required also Manhattan Distance can be applied to that. While misplaced tiles and Euclidean is too slow In search an almost give no information or guidance in search tree.

Advantage and Disadvantage

Advantages and disadvantages of N-Puzzle depend on the heuristic function used:

Misplaced tiles:

Advantages:

- ❖ Simple and fast: It requires minimal computation, making it efficient for smaller puzzles or as a baseline heuristic.
- ❖ Memory-friendly: Requires only a simple loop to count misplaced tiles.

Disadvantages:

- ❖ Less informative: It ignores the distance tiles needed to travel to reach their goal positions, so it often underestimates the cost.
- ❖ Deeper search tree: The lack of precision means the algorithm explores many unnecessary paths, increasing the search depth.
- ❖ Speed comparison: simplicity requires longer search time.

Manhattan Distance

Advantages:

- ❖ More accurate than misplaced tiles: Provides a better estimate of the actual cost to solve the puzzle.
- ❖ Efficient: Computational cost is low since it only calculates absolute differences in row and column positions.

Disadvantages:

- ❖ Ignores conflicts: Does not account for situations where tiles block each other, leading to slight underestimation.
- ❖ Still explores redundant paths: May not handle puzzles with many constraints optimally.
- ❖ Speed Comparison: Faster than Linear Conflict but slightly slower than Misplaced Tiles due to additional computations for each tile.

Euclidean Distance

Advantages:

- ❖ Precise for real-world scenarios: If diagonal moves were allowed, this would be more accurate.
- ❖ Captures distance better than Manhattan in some layouts: Especially in cases where tiles are diagonally displaced.

Disadvantages:

- ❖ More computationally expensive: Requires square root calculations for each tile, which is slower than Manhattan distance.
- ❖ Overhead for sliding puzzles: Sliding puzzles don't allow diagonal moves, making Euclidean less relevant and often redundant compared to Manhattan.
- ❖ Speed Comparison: Slower than Manhattan for larger puzzles due to its complexity, and it offers little additional value in the N-Puzzle context.

Linear Conflict

Advantages:

- ❖ More informed heuristic: Provides a better estimate of the cost by considering conflicts, reducing unnecessary paths.
- ❖ Significantly improves pruning: Especially effective for larger puzzles (e.g., 8x8 or 15-puzzle), where many tiles are likely to conflict.

Disadvantages:

- ❖ Computationally expensive: Requires checking each row and column for conflicts, increasing processing time.
- ❖ Memory overhead: The added complexity can lead to higher memory usage compared to simpler heuristics.
- ❖ Speed Comparison: Slower than Manhattan Distance but more accurate. The improved accuracy often results in fewer nodes being expanded, compensating for the added computational cost.

Misplaced Tiles and Path Cost

Advantages:

- ❖ Balanced Search Guidance: By combining misplaced tiles and path cost, it ensures a more informed search, leading to better performance in moderately complex puzzles.
- ❖ Improved Solution Quality: Helps in avoiding suboptimal paths by considering the cost of moves, which is beneficial for puzzles of intermediate difficulty.

Disadvantages:

- ❖ Higher Computational Cost: Adding path cost increases the time and memory needed, making it slower compared to simpler heuristics like misplaced tiles.
- ❖ Less Effective for Larger Puzzles: For very large puzzles, the added complexity can slow down the search significantly without ensuring better solutions compared to heuristics like Manhattan Distance or Linear Conflict.