The 15 Puzzle

and its Extension for a N*N Grid

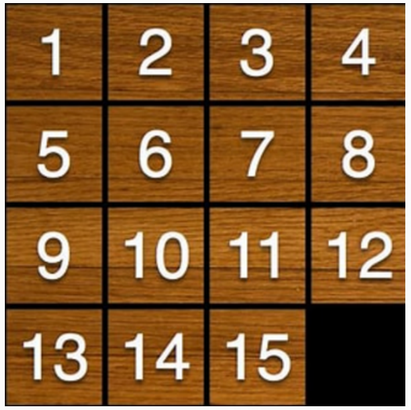
ABOUT

The 15 puzzle game consists of 15 numbered square tiles in a 4x4 grid, with the objective of arranging them in numerical order. Invented in the late 19th century and still popular today, it teaches problem-solving skills and has inspired variations with larger grids, different shapes, and themed designs.

Introduction

THE 15 PUZZLE

The project uses maths to solve the 15 puzzle game and bigger NxN grids using the A* algorithm. The solver checks if the puzzle can be solved before using the algorithm, making it dependable. It works for different puzzle sizes and difficulties, offering a functional approach to even complicated puzzles without compromising their solvability.



Representation of final state of the 15 puzzle

Here the bottom right Square is empty and can be switched with its adjacent block

THE 15 PUZZLE's HISTORY

Puzzle 8 and Puzzle 15 have been important models for studying search strategies for over two decades. They have helped develop and evaluate algorithms and have been used in comprehensive problem-solving programs. The puzzles have been studied in research on two-way search and dynamic loading and are believed to have features that extend beyond their gameplay.

Sam Loyd's Invention and Early Popularity (1878 – 1914)

In 1878, Sam Loyd claimed credit for creating the 15-puzzle. In his publication "Cyclopedia of Puzzles" from 1914, he featured numerous puzzles, including the 15-puzzle, contributing significantly to its widespread popularity as an enjoyable game.

Mathematical Analysis Begins (1879)

Noyes Chapman's paper, "On the Unsolvability of the Fifteen Puzzle" (1879), provided the first significant mathematical analysis of the 15-puzzle, demonstrating that half of the configurations are unsolvable. This marked the beginning of a more rigorous examination of the puzzle's mathematical properties.

Evolution into a Mathematical Challenge (Late 19th Century)

Researchers such as Chapman, Gaston Tarry, William Story, and William Alonzo Rogers expanded on Chapman's work. They provided more systematic approaches to determining the solvability of the 15-puzzle, evolving it from a newspaper pastime to a subject of mathematical investigation.

Graph Theory and Algorithmic Developments (1960s – 1980s)

Noyes Chapman's paper, "On the Unsolvability of the Fifteen Puzzle" (1879), provided the first significant mathematical analysis of the 15-puzzle, demonstrating that half of the configurations are unsolvable. This marked the beginning of a more rigorous examination of the puzzle's mathematical properties.

A* Algorithm Adaptation (1968 – 1980s)

The A* algorithm was introduced by Peter Hart, Nils Nilsson, and Bertram Raphael in "A Formal Basis for the Heuristic Determination of Minimum Cost Paths" (1968). While not initially applied to puzzles, its principles were later adapted for puzzle-solving, including the 15-puzzle and N-puzzle.

Puzzle-Specific Adaptation and Integration with Graph Theory (1980s onward)

Puzzle enthusiasts and researchers in artificial intelligence began adapting A* specifically for solving puzzles. The integration with graph theory became common practice, representing puzzle states as nodes in a graph and using A* to traverse the graph efficiently.

Algorithmic Refinements and Widespread Application (1980s – present)

Over time, researchers introduced refinements and variants of the A* algorithm to enhance its efficiency in solving puzzles. The algorithm became widely used in puzzle-solving software, marking its widespread application in both academic and practical settings.

History Summed Up

01 Sam Loyd claimed credit for inventing the 15-puzzle in 1878, and his "Cyclopedia of Puzzles" publication in 1914 contributed to its widespread popularity as an enjoyable game.

02 Graph theory and advancements in algorithms during the 1960s-70s, such as by Dijkstra and Bellman, provided the basis for applying graph theory to combinatorial problems, encouraging the AI community to explore this area.

03 The A* algorithm was introduced by Peter Hart, Nils Nilsson, and Bertram Raphael in 1968. It was later adapted for puzzle-solving, including the 15-puzzle and N-puzzle.

04 In the 1980s, puzzle enthusiasts and researchers in artificial intelligence integrated the A* search algorithm with graph theory to solve puzzles. They represented puzzle states as nodes in a graph and efficiently traversed the graph.

Solving Puzzles: A Mathematical Approach

- **Project Focus:**
  - Solving the 15 puzzle and extending to NxN grids.
- **Methodology:**
  - A* algorithm for efficient puzzle-solving.
  - Emphasis on solvability criteria.

Conditions for Solvability

Odd Width

The puzzle is solvable if the number of inversions is even.

Even Width

If the blank space is on an even row from the bottom, the puzzle is solvable if the number of inversions is odd.

Even Width

If the blank space is on an odd row from the bottom, the puzzle is solvable if the number of inversions is even.

What is Inversion?

Inversion refers to a pair of tiles (a, b) where a appears before b in a linear arrangement, but a > b.

Example

In a 4×4 board with 15 numbered tiles and one empty space, in the arrangement: 2 1 3 4 5 6 7 8 9 10 11 12 13 14 15 X, there is only one inversion: (2, 1).

Solvability Criteria

Illustration for 15 puzzle ie 4 x 4 grid –

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

N = 4 (Even)
Position of X from bottom = 2 (Even)
Inversion Count = 41 (Odd)
➔ Solvable

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

N = 4 (Even)
Position of X from bottom = 2 (Even)
Inversion Count = 56 (Even)
➔ Not Solvable

Assumptions and Definitions:

Assume S is a solvable configuration of the 15-puzzle.
There exists a sequence of moves M1, M2, ..., Mk that transforms S into the goal state.
Define I(i) as the number of inversions after the ith move Mi.
Define R(i) as the row number (1-4) of the blank space after the ith move Mi.
The goal state has I(k) = 0 (no inversions) and R(k) in the bottom row (even row number).

Key Points of the Proof:

Parity Change with Vertical Moves:

Moving the blank space vertically changes the inversion count by an odd number.

Moving the blank space vertically changes R by ±1, which is an odd change.

Therefore, the sum I(i) + R(i) changes by an even number after a vertical move.

Key Points of the Proof:

Parity Change with Horizontal Moves:

Moving the blank space horizontally changes the inversion count by an even number.
Moving the blank space horizontally does not change R.
Thus, the sum I(i) + R(i) remains even after a horizontal move.

Key Points of the Proof:

Total Moves and Parity Preservation:

Each move, whether vertical or horizontal, preserves the parity of I(i) + R(i).
Since S is solvable, it must start with an even I(0) + R(0) (as the goal state has an even I(k) + R(k)).
Therefore, the total number of moves k to reach the goal state preserves the even parity of I + R.

Proof Conclusion:

At the goal state, I(k) = 0 (no inversions) and R(k) is in the bottom row (an even number).

Hence, I(k) + R(k) is even at the goal state.
Since every move preserves the even parity of I + R, it's concluded that any solvable configuration must start with an even I + R.

Efficient Puzzle-Solving with A*

- **Introduction to A***
  - A more efficient version of Dijkstra's algorithm.
- **Algorithmic Process:**
  - Priority queue, heuristic function, and node expansion.
  - Iterative selection and expansion of nodes.
- **Heuristic Function:**
  - Detailed explanation of Misplaced Tiles and Manhattan Distance.
  - Cost function calculation.
- **Optimality and Performance:**
  - A* algorithm's superiority in problem-solving scenarios.

Introduction to A* Algortihm

The A* algorithm is a pathfinding algorithm that finds the shortest path between two points on a graph. It works by using a heuristic function to estimate the cost of reaching the goal from each neighbour of the current node (Manhattan for our case). The algorithm then chooses the neighbour with the lowest cost and examines its neighbors, repeating the process until the goal is reached. The heuristic function is used to prioritise the search and guide the algorithm towards the goal. The A* algorithm is widely used in games, robotics, and other applications where finding the shortest path is important.

What exactly is A*?

A* is a more efficient version of Dijkstra's algorithm, especially useful for solving problems like the N puzzle. In this algorithm, the puzzle states form a graph, and A* aims to find the best path from the initial state to the goal state.

Unlike Dijkstra's algorithm, A* uses a heuristic function to estimate the cost of reaching the goal from a given state. This additional information guides the algorithm to explore paths that seem more likely to lead to the goal, improving efficiency.

What exactly is A*?

The algorithm starts by initialising a priority queue with the initial state and its associated cost. This cost combines the actual cost from the start state and the heuristic estimate to the goal. A* then iteratively selects and expands nodes from the priority queue until it reaches the goal or the queue is empty.

A* prioritises paths that appear promising, striking a balance between exploration and exploitation. This heuristic-guided approach makes A* highly effective for navigating large state spaces, such as those found in the N puzzle problem.

Pseudocode

```
# A* (star) Pathfinding Algorithm

# Initialization
def initialize(node):
    node.g = infinity
    node.h = heuristic(node)
    node.f = node.g + node.h

# Heuristic function
def heuristic(node):
    return distance(node, goal)

# Add the start node
initialize(startNode)
startNode.g = 0
put startNode in openList

# A* Algorithm
while openList is not empty:
    # Extract node with the minimum f value
    currentNode = extractMin(openList)
```

Pseudocode

```
# Goal check
if currentNode is goal:
    Congratz! You've found the end! Backtrack to get the path
    break

# Generate children
children = adjacent nodes of currentNode

for each child in children:
    # Update values if better path is found
    tentative_g = currentNode.g + distance(currentNode, child)
    if tentative_g < child.g:
        child.g = tentative_g
        child.f = child.g + child.h

        # Add to openList if not already present
        if child not in openList:
            put child in openList
```

Optimising Puzzle-Solving with Heuristics

- **Introduction:**
  - Utilising heuristic functions within the A* algorithm for effective puzzle-solving.
- **Heuristic Function Framework:**
  - Considers both a cost function and a heuristic for decision-making.
  - Cost function includes heuristic cost and the number of steps to reach the current state.

Heuristic Function in A* Algorithm

- **Two Heuristics Employed -**
  - **Misplaced Tiles Heuristic (h1):**
    - Assesses the number of tiles out of their intended positions.
    - Example: Misplaced tiles calculated as 8.
  - **Manhattan Distance Heuristic (h2):**
    - Computes the cumulative displacement of each tile from its original position.
    - Example: Manhattan distance individually calculated for each tile and summed.

Heuristic Function

- In adherence to the principles of the A* algorithm, the maximum of the two heuristics is selected, as the maximum heuristic ensures that the cost function never underestimates the actual cost of reaching the desired end state, i.e., the initial configuration of the 8-puzzle.

- Illustration - if it took 10 moves to reach the current state, the cost function for a given state cost is computed as follows -
  cost =max(h1, h2) + 10 = max(14, 8) + 10 = 24

Optimizing Puzzle-Solving with Heuristics

- **Cost Function Calculation:**
  - Adheres to A* algorithm principles.
  - Selects the maximum of the two heuristics to ensure the cost function never underestimates the actual cost.
- **Benefits of Maximum Heuristic:**
  - Ensures an upper bound on the cost function.
  - Helps guide A* algorithm efficiently toward the goal state.
- **Application in Puzzle-Solving:**
  - Essential for evaluating the desirability of a state.
  - Optimises A* algorithm's decision-making process.

Our A* Algo Code

A* Algorithm

The A* algorithm is a heuristic-based search algorithm widely used in pathfinding and graph traversal. It intelligently balances the actual cost to reach a state with a heuristic estimation of the cost to reach the goal. This implementation maintains a priority queue to efficiently explore states.

Code Structure

`Node` Class

- Represents a node in the search space.
- Stores information about the puzzle state, parent node, action taken, and cost.

`Puzzle` Class

- Represents the n x n puzzle configuration.
- Provides methods for puzzle-solving logic, generating possible moves, calculating heuristics, and puzzle manipulation.

`Solver` Class

- Implements the A* algorithm to find the optimal solution.
- Utilizes a priority queue to explore states in order of increasing cost.

Example Application

```
# Example of use
board = [[0, 3, 8], [4, 1, 7], [2, 6, 5]]
if isSolvable(board):
    print("The puzzle is solvable.")
else:
    print("The puzzle is not solvable. Please adjust the initial configuration.")
    exit()

puzzle = Puzzle(board)
solver = Solver(puzzle)
solution = solver.solve()

# Print solution steps
for node in solution:
    print(node.action)
    node.puzzle.pprint()

# Print additional information
print("Total number of steps in the solution: " + str(len(solution)))
```

We will now explain our python code in depth :)

Advanced Approaches: Subgoal Analysis

- **Collaborative Puzzle-Solving:**
  - Breaking down complex puzzles into subgoals for adaptability.
- **Real-Time Adaptability:**
  - Inspired by natural systems for efficient problem-solving.
- **Example**
  - Analogous to breaking down a journey into subgoals like reaching specific cities.

Using Reinforcement Learning (RL)

- **RL in N-Puzzle:**
  - Efficient handling of large state spaces.
- **Agent Training:**
  - Introduction to Q-learning and policy gradient methods.
  - Role of reward function in agent training.
- **Balancing Exploration and Exploitation:**
  - Striking a balance for effective agent training.

Question and Answer...

Project Summary

- **Achievements:**
  - Successful solvability checks for 8, 15, and NxN puzzles.
  - Analysis using Discrete Math concepts (Graph theory).
- **Extensions:**
  - Exploring solutions for NNN puzzles.
  - Integration with Machine and Reinforcement Learning.
- **Comprehensive Understanding:**
  - Covering concepts from inversions and parity to heuristics, graph theory, and A* algorithm.

Additional Comments For Future Research

- **Optimising A***
  - Seeking faster solutions for larger puzzles.
- **Dimensional Expansion:**
  - Exploration of 3D puzzles (NNN).
- **Machine Learning Integration:**
  - Leveraging advanced algorithms for improved efficiency.
- **Continuous Improvement:**
  - Acknowledging the potential for future refinements and advancements.

Thank You

By Angadjeet Singh
Parth Sandeep Rastogi
Aryan Jain :)