College of Engineering, Software Engineering Department

SE 2231/36 Algorithms

Laboratory 4

Slider Puzzle

Objectives:

- Apply the priority queue on a puzzle game simulation.

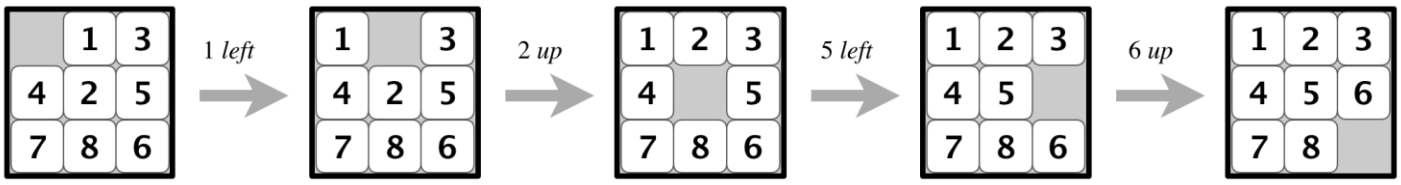
You are to write a Python program that solves the n-by-n sliding puzzle—commonly known as the 8-puzzle for a 3×3 board—by employing the A* search algorithm. The puzzle consists of numbered tiles (from 1 up to $n^2 - 1$) and one empty space (represented by 0 in our implementation) arranged in an n-by-n grid. At each move, you can slide a tile horizontally or vertically into the blank space. The goal is to transform an initial configuration of the board into the goal configuration, where the tiles are in row-major order (i.e., left-to-right, top-to-bottom) with the blank in the bottom-right corner.

Assignment Breakdown

This assignment has two major parts:

1. Implementing the Board Data Type
2. Implementing the Solver Data Type Using A* Search

The problem. The 8-puzzle is a sliding puzzle that is played on a 3-by-3 grid with 8 square tiles labeled 1 through 8, plus a blank square. The goal is to rearrange the tiles so that they are in row-major order, using as few moves as possible. You are permitted to slide tiles either horizontally or vertically into the blank square. The following diagram shows a sequence of moves from an *initial board* (left) to the *goal board* (right).



Part 1: The Board Data Type

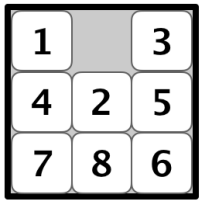
Create a Python class called `Board` that models an n-by-n sliding puzzle board. Your class should be **immutable** in the sense that any method that conceptually modifies the board (like generating neighbors) should return new `Board` instances rather than altering the current board.

Required API and Methods:

- *Constructor*. You may assume that the constructor receives an n-by-n array containing the n^2 integers between 0 and $n^2 - 1$, where 0 represents the blank square. You may also assume that $2 \leqslant n < 128$.

```
def __init__(self, tiles: list[list[int]]):
    """
    Initializes the board from a given n-by-n list of lists.
    Each element is an integer in the range 0 to n^2 - 1, where 0 represents the blank space.
    """
```

- *String Representation*. Implement the `__str__(self)` method such that:
    1. The first line of the returned string contains the board dimension (`n`).
    2. The following `n` lines display the board in row-major order with numbers separated by spaces (use 0 for the blank).



board

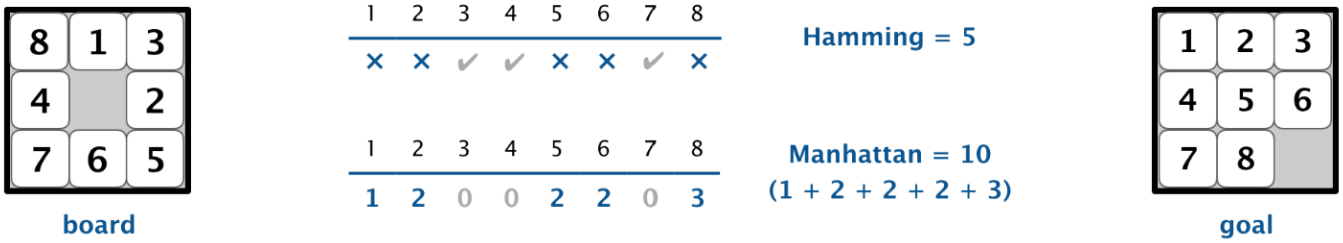
```
3
 1  0  3
 4  2  5
 7  8  6
```

string representation

- *Dimension*

```
def dimension(self) -> int:
    """
    Returns the board dimension n.
    """
```

- *Hamming Distance*. To measure how close a board is to the goal board, we define two notions of distance. The Hamming distance between a board and the goal board is the number of tiles in the wrong position. The Manhattan distance between a board and the goal board is the sum of the Manhattan distances (sum of the vertical and horizontal distance) from the tiles to their goal positions.




```
def hamming(self) -> int:
    """
    Returns the number of tiles that are not in their goal position.
    Do not count the blank (0) in the Hamming score.
    """
```

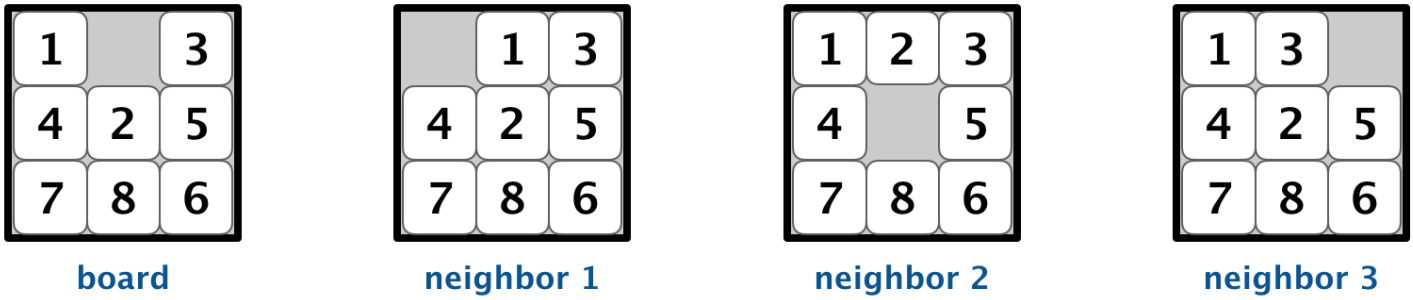
- *Manhattan Distance*

```
def manhattan(self) -> int:
    """
    Returns the sum of the Manhattan distances (vertical + horizontal)
    from the tiles to their goal positions.
    """
```

- *Goal Test*

```
def is_goal(self) -> bool:
    """
    Returns True if the board is the goal board.
    """
```

- *Equality Test*. Implement the `__eq__(self, other)` method to check if two boards are equal. Two boards are equal if:
    1. They have the same dimension.
    2. Their corresponding tiles are identical.
- `Neighbors`. The `neighbors()` method returns a list of `Board` objects which are the neighbors of the board. Depending on the location of the blank square, a board can have 2, 3, or 4 neighbors.




```
def neighbors(self) -> list[Board]: # you might have to import annotations from __future__
    """
    Returns an iterable (e.g., generator) of all neighboring boards.
    A neighbor is a board obtained by sliding a tile into the empty space.
    Depending on the position of the blank, there will be 2, 3, or 4 neighbors.
    """
```

- *Twin Board*

```
def twin(self) -> Board:
    """
    Returns a board that is a twin of the current board – obtained by swapping any pair
    of tiles (the blank should not be swapped).
    This is useful for detecting unsolvable puzzles.
    """
```

- *Unit Testing*

```
if __name__ == '__main__':
    # Add some basic tests to verify your Board methods.
    # For example, read a board from an input, print it, and check distances.
```

Save your `Board` class (and support functions) in a file called `board.py`.

Performance requirements. Your implementation should support all `Board` methods in time proportional to n^2 (or better) in the worst case.

A* search. Now, we describe a solution to the 8-puzzle problem that illustrates a general artificial intelligence methodology known as the A* search algorithm. We define a *search node* of the game to be a board, the number of moves made to reach the board, and the previous search node. First, insert the initial search node (the initial board, 0 moves, and a null previous search node) into a priority queue. Then, delete from the priority queue the search node with the minimum priority, and insert onto the priority queue all neighboring search nodes (those that can be reached in one move from the dequeued search node). Repeat this procedure until the search node dequeued corresponds to the goal board.

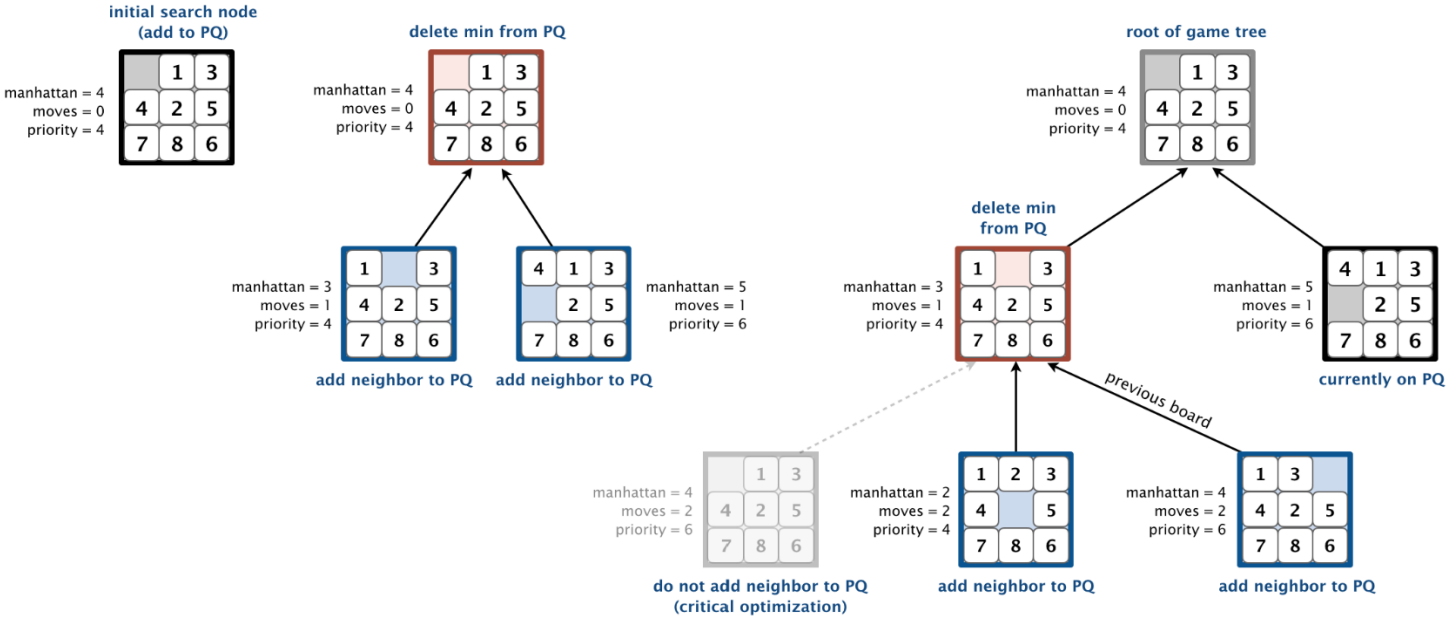
The efficacy of this approach hinges on the choice of *priority function* for a search node. We consider two priority functions:

- The *Hamming priority function* is the Hamming distance of a board plus the number of moves made so far to get to the search node. Intuitively, a search node with a small number of tiles in the wrong position is close to the goal, and we prefer a search node if has been reached using a small number of moves.
- The *Manhattan priority function* is the Manhattan distance of a board plus the number of moves made so far to get to the search node.

To solve the puzzle from a given search node on the priority queue, the total number of moves we need to make (including those already made) is at least its priority, using either the Hamming or Manhattan priority function. Consequently, when the goal board is dequeued, we have discovered not only a sequence of moves from the initial board to the goal board, but one that makes the *fewest* moves.

Game tree. One way to view the computation is as a *game tree*, where each search node is a node in the game tree and the children of a node correspond to its neighboring search nodes. The root of the game tree is the initial search node; the internal nodes have already been processed; the leaf nodes are maintained in a *priority queue*; at each step, the A* algorithm removes the node with the smallest priority from the priority queue and processes it (by adding its children to both the game tree and the priority queue).

For example, the following diagram illustrates the game tree after each of the first three steps of running the A* search algorithm on a 3-by-3 puzzle using the Manhattan priority function.



Part 2: The Solver Data Type

Implement a Python class called `Solver` that uses the A* search algorithm to solve the sliding puzzle. Your solver will search for the minimum sequence of moves that leads from the initial board to the goal.

Required API and Methods:

- *Constructor*

```
class Solver: # you might have to import annotations from __future__
    def __init__(self, initial: Board):
        """
        Find the solution to the initial board using the A* algorithm.
        If the initial board is None, raise a ValueError.
        """
```

- *Solvability check*

```
def is_solvable(self) -> bool:
    """
    Returns True if the initial board is solvable.
    A well-known fact: the puzzle is solvable if and only if the goal board is
    reachable from the initial board. One efficient approach is to run two simultaneous
    A* searches: one on the initial board and one on its twin (obtained by swapping any
    pair of non-blank tiles). Exactly one of these searches will yield a solution.
    """
```

- *Moves Count*

```
def moves(self) -> int:
    """
    Returns the minimum number of moves required to solve the puzzle,
    or -1 if the puzzle is unsolvable.
    """
```

- *Solution Path*

```
def solution(self) -> list[Board]:
    """
    Returns a list of Board objects representing the sequence of moves from the
    initial board to the goal board, if the puzzle is solvable. Otherwise, return None.
    """
```

- *Main Function / Test Client*. Your solver's test client should:
    1. Read the board data from a file.
    2. Create an initial Board.
    3. Instantiate a Solver with that initial board.
    4. Print the minimum number of moves required to solve the puzzle.
    5. If the puzzle is solvable, print out each board state from start to goal. Otherwise, print a message stating that the puzzle is unsolvable.

```
import sys

def main():
    if len(sys.argv) != 2:
        print("Usage: python solver.py [input_file]")
        return

    with open(sys.argv[1], 'r') as f:
        # The first integer is the board dimension n.
        n = int(f.readline().strip())
        tiles = []
        for _ in range(n):
            row = list(map(int, f.readline().split()))
            tiles.append(row)

    initial_board = Board(tiles)
    solver = Solver(initial_board)

    if not solver.is_solvable():
        print("No solution possible")
    else:
        print("Minimum number of moves =", solver.moves())
        for board in solver.solution():
            print(board)

if __name__ == '__main__':
    main()
```

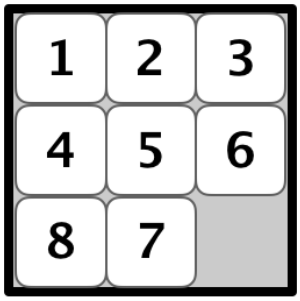
Save your *Solver* class (and support functions) in a file called *solver.py*.

Algorithm and Performance Considerations

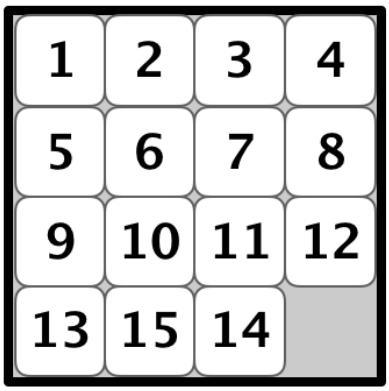
1. *A** Search & Priority Functions:
    - Use the Manhattan priority function: **priority = Manhattan distance of board + number of moves taken so far.**
    - (Optionally, you can also compute the Hamming priority for reference, but the assignment requires using the Manhattan heuristic.)
2. **Priority Queue Implementation:**
    - Use Python's `heapq` module to maintain a priority queue of search nodes.
    - Each search node should contain a board, the number of moves made to reach the board, a precomputed Manhattan (or Hamming) value, and optionally a reference to the previous search node (to reconstruct the path later).
3. **Critical Optimization:**
    - When considering neighbors for a search node, do not enqueue a neighbor whose board is identical to the board of the previous search node. This avoids redundant exploration and speeds up the search.

4. **Caching Priorities:**
    - Cache the Manhattan and Hamming values in your search nodes at the time of creation to avoid repeated recomputation during priority queue operations.

Handling Unsolvable Boards



unsolvable



unsolvable

Not every board is solvable. One way to detect unsolvability is to run two simultaneous searches:

- One starting with the initial board.
- The other starting with a "twin" board (generated by swapping any two non-blank tiles).

Because the board states split into two equivalence classes, exactly one of these searches will eventually find a solution (if one exists). If the twin board's search finds the goal before the initial board's search, then the initial board is unsolvable.

The input file contains the board size *n*, followed by the *n*-by-*n* grid of tiles, using 0 to designate the blank square. Output would somehow look like the following:

```
~/Desktop/8puzzle> cat puzzle04.txt
3
 0  1  3
 4  2  5
 7  8  6

~/Desktop/8puzzle> java-algs4 Solver puzzle04.txt
Minimum number of moves = 4

3
 0  1  3
 4  2  5
 7  8  6

3
 1  0  3
 4  2  5
 7  8  6

3
 1  2  3
 4  0  5
 7  8  6

3
 1  2  3
 4  5  0
 7  8  6

3
 1  2  3
 4  5  6
 7  8  0
```

```
~/Desktop/8puzzle> cat puzzle3x3-unsolvable.txt
3
 1  2  3
 4  5  6
 8  7  0

~/Desktop/8puzzle> java-algs4 Solver puzzle3x3-unsolvable.txt
Unsolvable puzzle
```

Submission Requirements

- Submit a .zip file containing the following files:
  - board.py — Contains your Board class implementation.
  - solver.py — Contains your Solver class implementation and the test client.
- Restrictions:
  - You may only import modules from Python's standard library (e.g., heapq, sys).
  - Do not use any external libraries.

Submit to https://www.dropbox.com/request/Z7SCoevAeFPpfciKZ8r4