Report

The 8-puzzle problem, a classic example in the field of artificial intelligence, involves arranging tiles on a 3x3 grid to reach a specific goal configuration. Solving this problem efficiently requires a structured approach, which led us to employ the Branch and Bound algorithm. Our journey to the solution began with understanding the problem's requirements and the mechanics of the sliding puzzle, where a blank space moves to swap positions with adjacent tiles, ultimately leading to the goal state.

Initially, we focused on state representation. Each state of the puzzle is depicted by a 3x3 matrix, where each element corresponds to a tile or the blank space. The initial state is our starting point, while the goal state is the desired arrangement of the tiles. Actions in this context refer to moving the blank space up, down, left, or right, which generates new states. To measure the closeness of a state to the goal, we used the Hamming distance, which counts the number of misplaced tiles.

To implement the Branch and Bound algorithm, we designed a node structure that captures the essential elements of each state. Each node comprises a matrix representing the current puzzle state, the coordinates of the blank space, the cost to reach this state, the number of moves made (level), and a pointer to the parent node. This structure enables us to trace the path from the initial state to the goal state.

Next, we implemented various functions to manage the puzzle's states and their transitions. The `printMatrix` function allows us to display the current state of the puzzle, aiding in debugging and visualization. The `newNode` function creates a new node by swapping the blank space with an adjacent tile, generating a new state. This function also updates the level and maintains a link to the parent node, which is crucial for reconstructing the solution path.

Calculating the cost of each state is vital for the Branch and Bound algorithm to function correctly. We used the `calculateCost` function to determine the Hamming distance between the current state and the goal state. This cost guides the algorithm in prioritizing which states to explore next. We also implemented the `isSafe` function to ensure that moves remain within the bounds of the puzzle grid, preventing invalid transitions.

To trace the solution path once the goal state is reached, we developed the `printPath` function. This function recursively prints each state from the initial state to the goal state by following the parent pointers. This capability is essential for verifying the correctness of our solution and understanding the sequence of moves.

The core of our solution is the `solve` function, which implements the Branch and Bound algorithm using a priority queue. The priority queue stores nodes, and the algorithm repeatedly extracts the node with the lowest cost, expands it by generating its successors, and inserts these successors back

into the queue. This process continues until the goal state is reached. We ensured that the priority queue is managed efficiently by using the `comp` function to compare nodes based on their cost.

The `solve` function starts by creating the root node from the initial state, calculating its cost, and inserting it into the priority queue. The main loop of the algorithm involves extracting the node with the lowest cost, checking if it is the goal state, and if not, generating its valid successors and inserting them into the queue. This process leverages the priority queue to explore the most promising states first, reducing the number of states that need to be examined.

Throughout the implementation, we maintained a focus on efficiency and correctness. The use of a priority queue ensures that the algorithm explores the most promising states first, which is critical for solving the puzzle in the minimum number of moves. The recursive approach to printing the solution path provides a clear and understandable sequence of moves from the initial state to the goal state.

In conclusion, our approach to solving the 8-puzzle problem using the Branch and Bound algorithm in C involved a series of well-defined steps, from understanding the problem and defining state representation to implementing core functions and the algorithm itself. This structured approach, combined with efficient data structures and algorithms, allowed us to solve the puzzle optimally. The implementation not only achieves the desired goal but also provides a clear pathway for understanding the sequence of moves, ensuring both efficiency and clarity in the solution. This method can be extended to larger instances of sliding puzzles, demonstrating its robustness and applicability in various problem-solving scenarios.