Technical Report: Ball Sort Puzzle Solver

1. Executive Summary

This report details the architecture and functionality of software solution designed to solve the Ball Sort Puzzle. The application models the puzzle as a graph traversal problem and utilizes the A* search algorithm to find the most efficient sequence of moves to sort colored balls into uniform stacks. The system is designed to handle user-defined configurations, allowing for variable numbers of stacks and capacities and handles the case when there is no possible soliution.

2. Data Structure Architecture

A combination of structs and classes is used to solve the problem. The structs are used to provide a simple collections of data without the need for encapsulation.

- **Move Structure:** A structure designed to record a single action. It holds two integer values identifying the source stack and the target stack. Its primary purpose is to help track and print the step-by-step solution.
- **GameState Structure:** This structure represents a snapshot of the game board at a specific moment.
    - **Functionality:** It stores the arrangement of balls and the cost metrics required for the A* algorithm. These metrics include the cost so far and a heuristic value which estimates the moves remainign.

    - **Usage:** It includes logic to calculate a total score summing the cost and heuristic and provides comparison operators. These operators allow the Priority Queue to automatically sort game states, ensuring the solver always processes the most promising state first.
- **StateHash Structure:** This structure is responsible for memory optimization.
    - **Functionality:** It converts a complex game state into a single unique integer. It achieves this by iterating through every ball and applying bitwise operations to generate a digital fingerprint of the board.
    - **Usage:** This hash is used to quickly identify if a board configuration has been encountered previously, preventing the solver from entering infinite loops without storing the full vector data for every visited state.
- **Priority Queue:** The solver uses a Min-Heap priority queue to manage the open set of the A* algorithm. It automatically organizes discovered game states so that the state with the lowest estimated solution cost is always at the top, ensuring the search is directed and efficient.
- **Set (The Visited List):** A Set container is used to store the hash values of all game states that have already been processed. This acts as a closed set, ensuring the algorithm never wastes time re-analyzing the same board configuration.
- **Map (Path tracking):** A Map is utilized to record the history of the search. It links a specific board configuration to the configuration that preceded it, effectively creating a breadcrumb trail. Once the solution is found, this map is used to backtrack from the goal to the start to reconstruct the winning moves.

3. Functional Breakdown

The software logic is compartmentalized into several key functions, each serving a distinct role in the solving process.

- **mismatch_heuristic:** This function calculates the "Heuristic" score for the A* algorithm. It scans the stacks and counts how many balls are placed directly on top of a ball of a different color. This provides a numerical estimate of how disordered the current state is, guiding the solver toward states that look more sorted.
- **is_goal:** This is the validation function that determines if the puzzle is solved. It iterates through all stacks to ensure two conditions are met: first, that every stack is either completely empty or contains only balls of a single color; and second, that no two stacks contain the same completed color.
- **print_visual:** This helper function handles the user interface. It renders the current state of the stacks into the console using ASCII art, allowing the user to visualize the ball arrangements and the steps of the solution.
- **BallSortSolver (Constructor):** Initializes the solver instance with the user-defined parameters for the number of stacks and the maximum capacity of each stack.
- **solve:** This is the core engine of the program.
    - It initializes the search by placing the starting configuration into the Priority Queue and the Visited Set.
    - It enters a loop that continues until a solution is found or no moves remain. Inside the loop, it extracts the best state, checks if it is the goal, and then generates all possible legal moves (moving a ball from one stack to another).

    - Valid moves are determined by checking stack capacities and color matching rules. New states resulting from valid moves are hashed, checked against the visited list, and added to the queue if they are new.
- **print_solution:** Once the goal is reached, this function reconstructs the path. It uses the Map history to trace backward from the solved state to the initial state. It then reverses this list to get the chronological order of moves and iterates through them, updating a temporary board and printing the visual state at every step.
- **main:** The entry point of the program. It handles user input, asking for the number of stacks, stack capacity, and the specific arrangement of balls. It validates the input (checking for capacity overflows) and then instantiates the solver to begin the process.

4. Complexity Analysis

The performance of this solver is governed by the principles of the A* search algorithm.

- **Time Complexity:** The complexity is exponential. In the worst-case scenario, the algorithm may need to explore a vast number of permutations before finding the goal. The heuristic function helps reduce this by pruning the search space, but the underlying nature remains exponential.
- **Space Complexity:** The space complexity is also exponential, . This is because the A* algorithm must store every visited state in the Set and the path history in the Map to ensure optimality and prevent cycles. As the puzzle grows larger or more complex, the memory usage increases significantly to track these states. The hashing mechanism is a critical optimization here, reducing the memory footprint per state, but the sheer volume of states remains the limiting factor.

5. Github