Jack McVeigh

CS 260

30 November 2021

Programming Assignment 4: Solving Sliding Puzzle with BFS

Introduction

A Sliding (k^2 – 1) Puzzle is a puzzle where a N x N grid is filled with pieces N * N – 1 pieces with the goal of moving, or "sliding," the pieces into a specific order. Figure 1 contains an example puzzle.

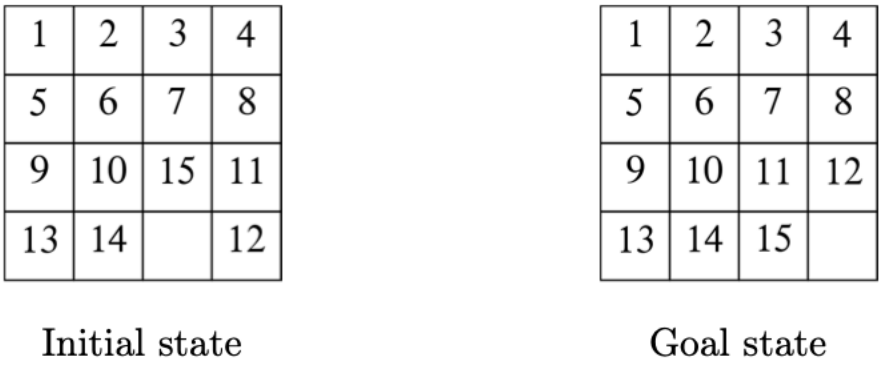

Figure 1. Example Sliding Puzzle.

Sliding puzzles, while a simple concept, can be difficult to solve in the fastest possible way. This report will detail the implementation of solving sliding puzzles in the minimum number of steps using Breadth First Search (BFS) and graph like data structures.

Parsing

Parsing the provided board file simply checks each line against the known header tags, parsing the subsequent line as the referenced field. The implementation of the board parser is seen in Figure 2.

```c
while (getline(&line, &len, fp) != -1) {
    line[strlen(line) - 1] = '\0'; /* replace \n with \0. */

    if (!strcmp(line, "")) /* Blank line */
        continue;
    else if (!strcmp(line, "#k")) { /* Dimension header */
        if (getline(&line, &len, fp) == -1)
            break;
        line[strlen(line) - 1] = '\0'; /* replace \n with \0. */

        board->dimension =  atoi(line);
        board->table = malloc(board->dimension * board->dimension * sizeof(*board->table));
        check_malloc_error(board->table, "solve: Failed to malloc table");
    }
    else if (!strcmp(line, "#initial state")) { /* Initial state header */
        if (getline(&line, &len, fp) == -1) {
            printf("solve: Initial state header read but missing data\n");
            break;
        }
        line[strlen(line) - 1] = '\0'; /* replace \n with \0. */
        piece = strdup(line);
        check_malloc_error(piece, "solve: Failed to strdup piece");

        board->table[0] = atoi(strtok(piece, " "));
        if (!board->table[0])
            board->position = 0;
        for (i = 1; i < board->dimension * board->dimension; i++) {
            board->table[i] = atoi(strtok(NULL, " "));
            if (!board->table[i])
                board->position = i;
        }

        free(piece);
    }
    else { /* Unknown line in board file */
```

Figure 2. Implementation of the board parser.

Solving Puzzle Via BFS

BFS requires a queue to be implemented to hold board states pending consumption. When a board is consumed, it is first checked to see if it has already been used before. If it has already been used, it is skipped. When a new board state is consumed, the board if moved in each of the state's applicable directions (moves must be within the board bounds and previous moves are not undone, ex: move left followed by move right). After a move is completed, the resulting board is compared against the goal board. If the board equivalent to the goal board, the process stops. If the board is not equivalent to the goal board, the board is queued for consumption.

When the goal board is found, the summary is printed to the output file as per the project specifications. Figure 3 displays the BFS process.

```c
    /* BFS Queue */
    bfs_queue_t *queue = init_queue();
    enqueue(queue, initial_board);

    /* Intermediate Board Storage (Queue DS used but operated on like a linked list) */
    bfs_queue_t *intermediate_boards = init_queue();

    solved_board = NULL;
    while (!solved_board) {
        bfs_node_t *test_node = dequeue(queue);
        if (!test_node) {
            printf("solve: Attempting to dequeue from empty queue.\n");
            exit(EXIT_FAILURE);
        }

        /* Check if current board has already been visited */
        if (search_queue_for_board_in_place(intermediate_boards, test_node->board))
            goto skip_node;
        else
            enqueue(intermediate_boards, copy_board(test_node->board));

        /* Do applicable moves and queue mutated boards if valid */

        if (test_node->board->last_move != DOWN) {
            temp_board = copy_board(test_node->board);
            if (move_up(temp_board) != INVALID) {
                if (compare_boards(goal_board, temp_board)) {
                    solved_board = temp_board;
                    break;
                }

                enqueue(queue, temp_board);
            }
            else
                destroy_board(temp_board);
        }

        if (test_node->board->last_move != LEFT) {
            temp_board = copy_board(test_node->board);
            if (move_right(temp_board) != INVALID) {
                if (compare_boards(goal_board, temp_board)) {
                    solved_board = temp_board;
                    break;
                }

                enqueue(queue, temp_board);
            }
            else
                destroy_board(temp_board);
        }

        if (test_node->board->last_move != UP) {
            temp_board = copy_board(test_node->board);
            if (move_down(temp_board) != INVALID) {
                if (compare_boards(goal_board, temp_board)) {
                    solved_board = temp_board;
                    break;
                }

                enqueue(queue, temp_board);
            }
            else
                destroy_board(temp_board);
        }

        if (test_node->board->last_move != RIGHT) {
            temp_board = copy_board(test_node->board);
            if (move_left(temp_board) != INVALID) {
                if (compare_boards(goal_board, temp_board)) {
                    solved_board = temp_board;
                    break;
                }

                enqueue(queue, temp_board);
            }
            else
                destroy_board(temp_board);
        }

skip_node:
        destroy_node(test_node);
    }

    destroy_queue(queue);
    destroy_queue(intermediate_boards);
    return solved_board;
}
```

Figure 3. Implementation of BFS with board graph.

Determining if Given Sliding Puzzle has a Solution

The process for determining if a board has a solution is dependent on its dimensions. If a board's dimension is odd, it must have an even number of inversions. If a board's dimension is even, it must either have its empty piece located on an even row (from the bottom) and an odd number of inversions, or its empty piece located on an odd row (from the bottom) and an even number of inversions. An inversion is when a given piece is in a position that comes before a piece of lower value. For example, if piece 2 is before piece 1, there is an inversion.

```
/* Check following cases:
    1. if dimension is odd
        a. number of inversions in even
    2. if dimension is even
        a. position is on even row from bottom and number of inversions is odd
        b. position is on odd row from bottom and number of inversions is even
 */

size_t i, j, pos, level;
int inversions = 0;
/* Calculate number of inversions */
for (i = 0; i < board->dimension * board->dimension - 1; i++) {
    if (i == board->position)
        continue;
    for (j = i + 1; j < board->dimension * board->dimension; j++) {
        if (j == board->position)
            continue;
        if (board->table[i] > board->table[j])
            inversions++;
    }
}

if (board->dimension % 2) { // Dimension is odd
    if (!(inversions % 2)) // Inversions in even
        return 1;
}
else { // Dimension is even
    pos = board->position;
    level = 0;
    while (pos < board->dimension * board->dimension) {
        pos += board->dimension;
        level++;
    }

    if ((level % 2) && !(inversions % 2)) // odd and even
        return 1;
    else if (!(level % 2) && (inversions % 2)) // even and odd
        return 1;
}

return 0; // no condition is true
```

Figure 4. Determining if a board has a solution.

Conclusion

BFS guarantees that the determined solution path is the minimum path, however it requires a lot of computation to do so. BFS searches all directions at the same rate meaning that despite having the knowledge of what the goal state looks like, the search will still compute states that do not move any closer to the goal state. This is the main issue with using a search algorithm like BFS without heuristics.

The chosen implementation efficiently leverages the BFS algorithm to solve any sliding puzzle in the minimum steps. This implementation also ensures only useful moves are computed for a given board state by checking if the move is counter intuitive (undoes the most recent move) or if the board state has already been handled previously. In addition, the implementation checks if a board is even possible to solve given its input state using the commonly known rules for sliding puzzles.

The runtime of the BFS algorithm used is O(V+E) where V is the number of board states and E is the number of moves. The space complexity of the selected algorithm is O(V+E) as the graph grows with board states and moves.

A better implementation of this project would use a better search algorithm with heuristics to limit the number of necessary computations and lower the program runtime.