Jack McVeigh

CS 260

6 December 2021

Programming Assignment 5: Solving Sliding Puzzle with A* (A-Star)

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.

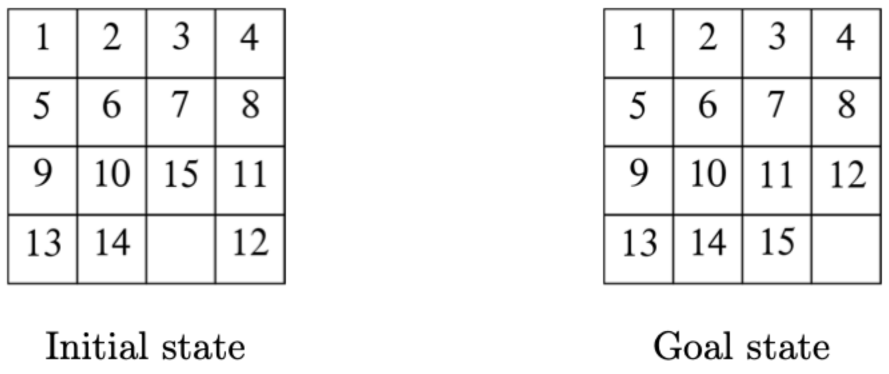
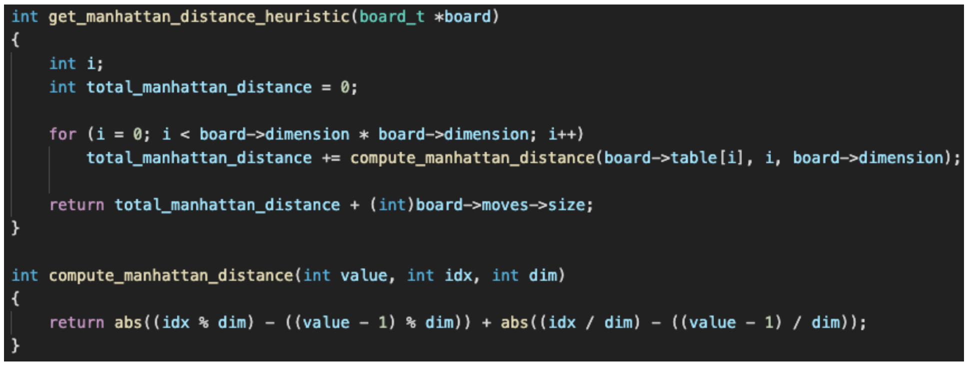


Figure 1. Example Sliding Puzzle.

Sliding puzzles, while a simple concept, can be difficult to solve in the fastest possible way. While algorithms like Breadth First Search (BFS) are quick to implement and will solve a sliding puzzle in the minimum number of steps, these simple algorithms are intensive in that they try every possible path until the solution is found. An algorithm such as A* (A-Star) uses heuristics to weigh each sub path, choosing to take the best scoring path first. This report will detail the implementation of solving sliding puzzles in the minimum number of steps using the A* search algorithm and graph like data structures. The heuristic used will be a combination of Manhattan distance and path cost. (Note: Much of the required overhead is the same as BFS.)

Solving Puzzle Via A*

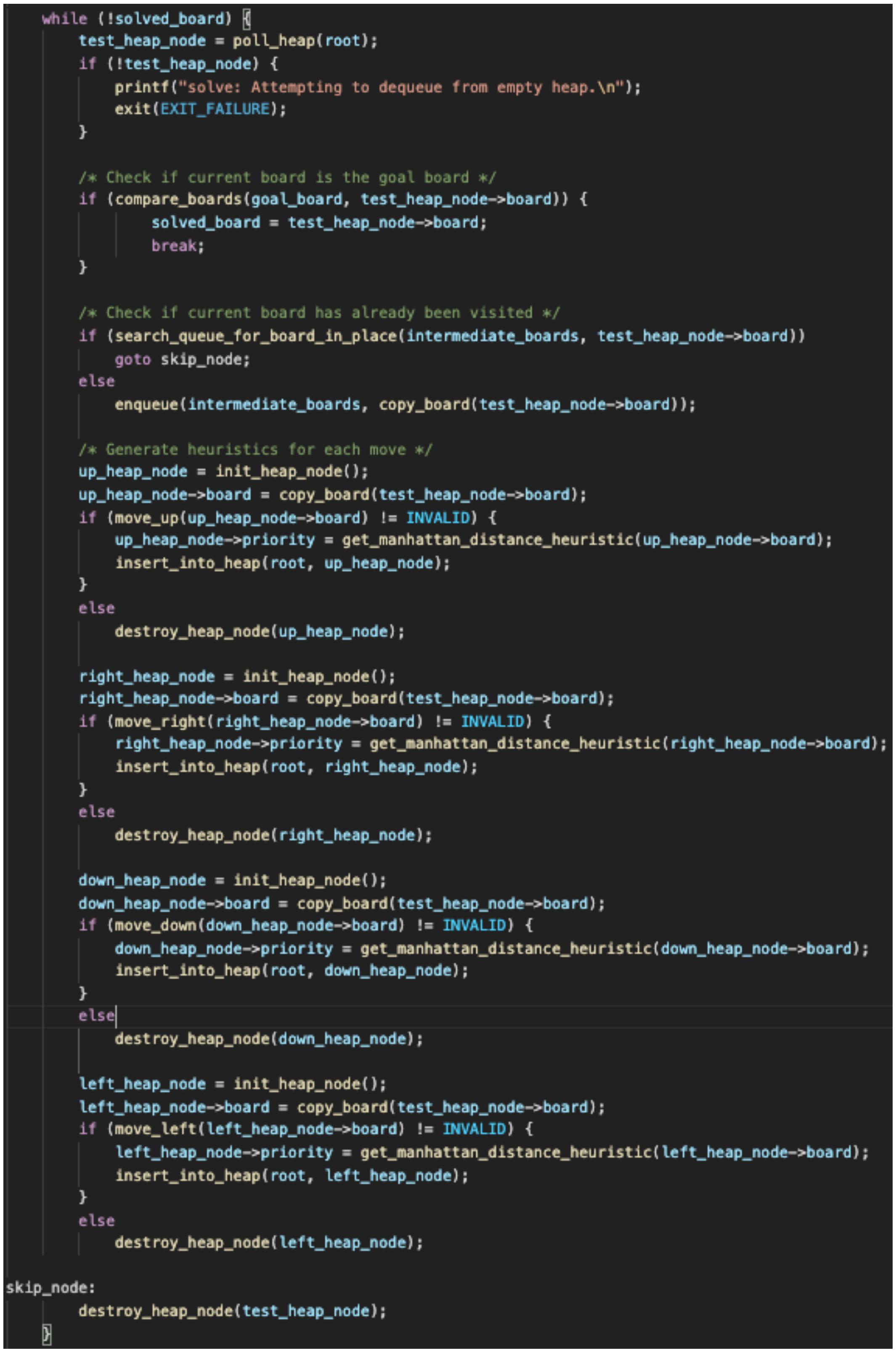
Unlike BFS, A* weighs each move differently using a heuristic. The heuristic implemented is the sum of the Manhattan distance and path cost for a potential move. Figure 2 shows the heuristic calculation implementation.



```
int get_manhattan_distance_heuristic(board_t *board)
{
    int i;
    int total_manhattan_distance = 0;

    for (i = 0; i < board->dimension * board->dimension; i++)
        total_manhattan_distance += compute_manhattan_distance(board->table[i], i, board->dimension);

    return total_manhattan_distance + (int)board->moves->size;
}

int compute_manhattan_distance(int value, int idx, int dim)
{
    return abs((idx % dim) - ((value - 1) % dim)) + abs((idx / dim) - ((value - 1) / dim));
}
```

Figure 2. Manhattan distance and Path Cost Calculation.

The moves are inserted into a priority queue based on the heuristic and polled as the algorithm progresses. It is important to insert all moves if the heuristic does not lead to the correct path on the first attempt. Figure 2 contains the implementation of A* for solving a sliding puzzle. Not that only unique boards are considered (boards that have not already been processed) and that invalid moves are skipped (moves outside of the board).



```c
    while (!solved_board) {
        test_heap_node = poll_heap(root);
        if (!test_heap_node) {
            printf("solve: Attempting to dequeue from empty heap.\n");
            exit(EXIT_FAILURE);
        }

        /* Check if current board is the goal board */
        if (compare_boards(goal_board, test_heap_node->board)) {
                solved_board = test_heap_node->board;
                break;
        }

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

        /* Generate heuristics for each move */
        up_heap_node = init_heap_node();
        up_heap_node->board = copy_board(test_heap_node->board);
        if (move_up(up_heap_node->board) != INVALID) {
            up_heap_node->priority = get_manhattan_distance_heuristic(up_heap_node->board);
            insert_into_heap(root, up_heap_node);
        }
        else
            destroy_heap_node(up_heap_node);

        right_heap_node = init_heap_node();
        right_heap_node->board = copy_board(test_heap_node->board);
        if (move_right(right_heap_node->board) != INVALID) {
            right_heap_node->priority = get_manhattan_distance_heuristic(right_heap_node->board);
            insert_into_heap(root, right_heap_node);
        }
        else
            destroy_heap_node(right_heap_node);

        down_heap_node = init_heap_node();
        down_heap_node->board = copy_board(test_heap_node->board);
        if (move_down(down_heap_node->board) != INVALID) {
            down_heap_node->priority = get_manhattan_distance_heuristic(down_heap_node->board);
            insert_into_heap(root, down_heap_node);
        }
        else
            destroy_heap_node(down_heap_node);

        left_heap_node = init_heap_node();
        left_heap_node->board = copy_board(test_heap_node->board);
        if (move_left(left_heap_node->board) != INVALID) {
            left_heap_node->priority = get_manhattan_distance_heuristic(left_heap_node->board);
            insert_into_heap(root, left_heap_node);
        }
        else
            destroy_heap_node(left_heap_node);

skip_node:
        destroy_heap_node(test_heap_node);
    }
```

Figure 2. A Implementation.*

Conclusion

A* is a much smarter algorithm than BFS and did not require many changes to implement in the sliding puzzle. The bulk of the overhead for the A* implementation is the same as the BFS implementation. The only additional overhead required was a priority queue implementation, heuristic calculation, and the A* algorithm itself.

A* can limit the computation overhead by approaching a problem in a smarter way, and in this project, was able to dramatically decrease the number of puzzle boards traversed through.