```
======================================================================
            /local/submit/submit/comp10002/ass2/hjthorpe/src/ass2submission.c
======================================================================

/* Solution to comp10002 Assignment 2, 2019 semester 2.

   Authorship Declaration:

   (1) I certify that the program contained in this submission is completely
   my own individual work, except where explicitly noted by comments that
   provide details otherwise.  I understand that work that has been developed
   by another student, or by me in collaboration with other students,
   or by non-students as a result of request, solicitation, or payment,
   may not be submitted for assessment in this subject.  I understand that
   submitting for assessment work developed by or in collaboration with
   other students or non-students constitutes Academic Misconduct, and
   may be penalized by mark deductions, or by other penalties determined
   via the University of Melbourne Academic Honesty Policy, as described
   at https://academicintegrity.unimelb.edu.au.

   (2) I also certify that I have not provided a copy of this work in either
   softcopy or hardcopy or any other form to any other student, and nor will
   I do so until after the marks are released. I understand that providing
   my work to other students, regardless of my intention or any undertakings
   made to me by that other student, is also Academic Misconduct.

   (3) I further understand that providing a copy of the assignment
   specification to any form of code authoring or assignment tutoring
   service, or drawing the attention of others to such services and code
   that may have been made available via such a service, may be regarded
   as Student General Misconduct (interfering with the teaching activities
   of the University and/or inciting others to commit Academic Misconduct).
   I understand that an allegation of Student General Misconduct may arise
   regardless of whether or not I personally make use of such solutions
   or sought benefit from such actions.

   Signed by: Hunter James Thorpe 1079893
   Dated:     20/10/2019

*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>

#define COORD_LEN 2
#define NO_DIRECTIONS 4
#define CELLS_PER_LINE 5

#define ROW 0
#define COL 1
#define COUNT 2
#define SEED_STATUS 3
#define NOT_USED -1

#define YES 1
#define NO 0

#define UP 0
#define DOWN 1
#define LEFT 2
#define RIGHT 3

#define NO_OF_STATUS 5
#define STATUS_1 1
#define STATUS_2 2
#define STATUS_3 3
#define STATUS_4 4
#define STATUS_5 5


#define MAX_LINE_LEN 400
```

```c

#define EMPTY_CELL ' '
#define BLOCK '#'
#define I_CELL 'I'
#define G_CELL 'G'
#define ROUTE_CELL '*'
#define END_OF_BLOCKS '$'
/**************************************************************************/
/* typdefs */
typedef struct {
    int** route_coords;
    int route_end;
    int route_illegal;
    int no_coords;
} route_info_struct_t;

typedef struct {
    int row_;
    int col_;
} data_t;

typedef struct node node_t;

struct node {
    data_t data;
    node_t *next;
};

typedef struct {
    node_t *head;
    node_t *foot;
} list_t;

typedef struct {
    int coord[COORD_LEN];
    int block_end;
} line_info_struct_t;

typedef struct {
    int rows;
    int columns;
    int no_blocks;
    int initial_cell[COORD_LEN];
    int goal_cell[COORD_LEN];
    int path_status;
} grid_t;
/**************************************************************************/
/* functino prototypes */
int mygetchar();
void read_line(char *line);
int my_getnbr(char *str);
void nullify_line(char *line, int len);
void nullify_line_int(int *line, int len);
line_info_struct_t handle_line(void);
int scrape_coord(char *line, int *coord);
list_t *make_empty_list(void);
int is_empty_list(list_t *list);
void free_list(list_t *list);
list_t *insert_at_head(list_t *list, data_t value);
list_t *insert_at_foot(list_t *list, data_t value);
data_t get_head(list_t *list);
list_t *get_tail(list_t *list);
int print_status(int *status);
route_info_struct_t read_route(int old_row, int old_col);
int illegal_route(int **r_array, int old_row, int old_col, int array_len);
void print_stage0(int blocks, grid_t grid);
int create_route_list(list_t *list, int old_row, int old_col, grid_t *grid,
    route_info_struct_t *route_struct, int *status, data_t *route_coord);
void print_route(list_t *list, grid_t *grid, char **grid_array, int **old_route,
    route_info_struct_t *route_struct, int *status, data_t *route_coord);
int print_stage1(int col_no, int row_no, char **grid_array, int status_no,
    int no_blocks, int **old_route, int old_route_len, grid_t *grid);
void print_grid(int col_no, int row_no, char **grid_array);
void traversal(int base_row, int base_col, int **trav_array);
```

```
int verify_coord(int row_val, int col_val, char **grid_array, int **queue_array,
    int no_rows, int no_cols, int queue_size);
int repair_route(int **queue_array, int **old_route, int old_route_len,
    grid_t *grid, char **grid_array, int queue_size, int **new_route);
void trace_route(int **queue_array, int curr_row, int curr_col, int join_len,
    int queue_size,  int **repair_array);
/****************************************************************************/
/* main function */
int
main(int argc, char *argv[]) {
    char** grid_array;
    int row_iter;
    int col_iter;
    int malloc_iter;
    int block_counter = 0;
    int status[NO_OF_STATUS + 1] = {'\0', NO, NO, NO, NO, NO};
    int old_row;
    int old_col;
    int **old_route;
    int route_len;
    int status_no;
    grid_t grid;
    line_info_struct_t received_struct;
    route_info_struct_t route_struct;
    list_t route_list;
    data_t route_coord;

    /* reading first line, dimensions */
    received_struct = handle_line();
    grid.rows = received_struct.coord[ROW];
    grid.columns = received_struct.coord[COL];

    /* reading second line, initial cell */
    received_struct = handle_line();
    grid.initial_cell[ROW] = received_struct.coord[ROW];
    grid.initial_cell[COL] = received_struct.coord[COL];

    /* reading third line, goal cell */
    received_struct = handle_line();
    grid.goal_cell[ROW] = received_struct.coord[ROW];
    grid.goal_cell[COL] = received_struct.coord[COL];

    /* allocating memory to grid array and initialising cells to not blocked */
    grid_array = malloc(grid.rows * sizeof(char*));
    for (row_iter = 0; row_iter < grid.rows; row_iter++) {
        grid_array[row_iter] = malloc(grid.columns * sizeof(char));
        for (col_iter = 0; col_iter < grid.columns; col_iter++) {
            grid_array[row_iter][col_iter] = EMPTY_CELL;
        }
    }

    /* counting number of blocks and adding them to 2d array */
    while ((received_struct = handle_line()).block_end == NO) {
        block_counter = block_counter + 1;
        grid_array[received_struct.coord[ROW]][received_struct.coord[COL]] =
            BLOCK;
    }

    /* creating a linked list that stores the proposed route*/
    old_row = grid.initial_cell[ROW] + 1; /* +1 prevents step size being */
    old_col = grid.initial_cell[COL];     /* recognized as 0 */
    route_list = *make_empty_list();
    route_len = create_route_list(&route_list, old_row, old_col, &grid,
        &route_struct, status, &route_coord);

    /* allocating memory to array that stores old route path in array */
    old_route = malloc(route_len * sizeof(int*));
    for (malloc_iter = 0; malloc_iter < route_len; malloc_iter++) {
        old_route[malloc_iter] = malloc(COORD_LEN * sizeof(int));
    }

    /* printing stage0 output */
    print_stage0(block_counter, grid);
    /* printing route out */
    print_route(&route_list, &grid, grid_array, old_route, &route_struct,
        status, &route_coord);

    /* adding initial cell and goal cell to map */
    grid_array[grid.initial_cell[ROW]][grid.initial_cell[COL]] = I_CELL;
    grid_array[grid.goal_cell[ROW]][grid.goal_cell[COL]] = G_CELL;

    /* printing status */
    status_no = print_status(status);

    /* printing stage1 output */
    print_stage1(grid.columns, grid.rows, grid_array, status_no, block_counter,
        old_route, route_len, &grid);

    /* freeing allocated memory (route_list is freed as it is printed) */
    for (malloc_iter = 0; malloc_iter < grid.rows; malloc_iter++) {
        free(grid_array[malloc_iter]);
    }
    free(grid_array);
    for (malloc_iter = 0; malloc_iter < route_len; malloc_iter++) {
        free(old_route[malloc_iter]);
    }
    free(old_route);

    return 0;
}
/*****************************************************************************/
/* helper functions */
/*****************************************************************************/
/* takes flooded queue and position of join back with route, and uses this to
    establish an array that contains the repair */
void
trace_route(int **queue_array, int curr_row, int curr_col, int join_len,
    int queue_size,  int **repair_array) {
    int target_count;
    int **tie_break_array;
    int **trav_array;
    int trav_iter;
    int queue_pos;
    int null_iter;
    int malloc_iter;
    int tie_break_pos;

    /* allocating memory for tie_break_array and traversal array */
    tie_break_array = malloc(NO_DIRECTIONS * sizeof(int*));
    trav_array = malloc(NO_DIRECTIONS * sizeof(int*));
    for (malloc_iter = 0; malloc_iter < NO_DIRECTIONS; malloc_iter++) {
        tie_break_array[malloc_iter] = malloc(COORD_LEN * sizeof(int));
        trav_array[malloc_iter] = malloc(COORD_LEN * sizeof(int));
    }
    /* adding last cell to end of array */
    repair_array[join_len][ROW] = curr_row;
    repair_array[join_len][COL] = curr_col;

    target_count = join_len - 1;
    while (target_count != -1) {
        /* resetting tie break array */
        for (null_iter = 0; null_iter < NO_DIRECTIONS; null_iter++) {
            tie_break_array[null_iter][ROW] = NOT_USED;
            tie_break_array[null_iter][COL] = NOT_USED;
        }

        /* generating possible cells */
        traversal(curr_row, curr_col, trav_array);
        for (queue_pos = 0; queue_pos < queue_size; queue_pos++) {
            if (queue_array[queue_pos][COUNT] == target_count) {
                /* iterating through up, down, left and right to see if
                potential cell is in one of these locations, then adding it to
                tie_break_array in appropraite position */
                for (trav_iter = 0; trav_iter < NO_DIRECTIONS; trav_iter++) {
                    if (trav_array[trav_iter][ROW] ==
                        queue_array[queue_pos][ROW] &&
                        trav_array[trav_iter][COL] ==
                        queue_array[queue_pos][COL]) {
                        tie_break_array[trav_iter][ROW] =
                        queue_array[queue_pos][ROW];
                        tie_break_array[trav_iter][COL] =
                        queue_array[queue_pos][COL];

                    }
                }
            }
        }
        /* iterating through tie break array to add correct cell to repair */
        for (tie_break_pos = 0; tie_break_pos < NO_DIRECTIONS;
            tie_break_pos++) {
            if (tie_break_array[tie_break_pos][ROW] != NOT_USED) {
                repair_array[target_count][ROW] = (curr_row =
                    tie_break_array[tie_break_pos][ROW]);
                repair_array[target_count][COL] = (curr_col =
                    tie_break_array[tie_break_pos][COL]);
                break;
            }
        }
        target_count = target_count - 1;
    }

    /* freeing allocated memory */
    tie_break_array = malloc(NO_DIRECTIONS * sizeof(int*));
    trav_array = malloc(NO_DIRECTIONS * sizeof(int*));
    for (malloc_iter = 0; malloc_iter < NO_DIRECTIONS; malloc_iter++) {
        free(tie_break_array[malloc_iter]);
        free(trav_array[malloc_iter]);
    }
    free(tie_break_array);
    free(trav_array);
}
/***************************************************************************/
/* checks if coordinates are out of bounds, already in queue or is blocked */
int
verify_coord(int row_val, int col_val, char **grid_array, int **queue_array,
    int no_rows, int no_cols, int queue_size) {
    int queue_iter = 0;

    if (row_val >= no_rows || col_val >= no_cols) {
        return NO;
    }
    if (row_val < 0 || col_val < 0) {
        return NO;
    }
    for (queue_iter = 0; queue_iter < queue_size; queue_iter++) {
        if (queue_array[queue_iter][COL] == col_val &&
            queue_array[queue_iter][ROW] == row_val) {
            return NO;
        }
    }
    if (grid_array[row_val][col_val] == BLOCK) {
        return NO;
    }
    return YES;
}
/***************************************************************************/
/* fills array with coordinates of cells above, below, left and right (in
    that order) based on given coordinates */
void
traversal(int base_row, int base_col, int **trav_array) {

    /* up one coordinate */
    trav_array[0][ROW] = base_row - 1;
    trav_array[0][COL] = base_col;

    /* down one coordinate */
    trav_array[1][ROW] = base_row + 1;
    trav_array[1][COL] = base_col;

    /* left one coordinate */
    trav_array[2][ROW] = base_row;
```

```c
    trav_array[2][COL] = base_col - 1;

    /* right one coordinate */
    trav_array[3][ROW] = base_row;
    trav_array[3][COL] = base_col + 1;
}
/****************************************************************************/
/* floods grid to find a repair, updates grid and route and prints them */
int
repair_route(int **queue_array, int **old_route, int old_route_len,
    grid_t *grid, char **grid_array, int queue_size, int **new_route) {
    int join_row;
    int join_col;
    int join_len;
    int rejoin;
    int **repair_array;
    int queue_pos = 0;
    int break_pos = 0;
    int post_break_pos = 0;
    int coords_in_queue = 0;
    int break_loop = NO;
    int new_route_len;
    int format_counter;
    int block_status = NO;
    int row_iter;
    int malloc_iter;
    int col_iter;
    int pot_cell;
    int row_no;
    int col_no;
    int **trav_array;
    int route_iter;
    int new_route_pos;
    int rep_array_iter;

    /* finding position of first instance of block on route using block_iter */
    for (route_iter = 0; route_iter < old_route_len; route_iter++) {
        if (grid_array[old_route[route_iter][ROW]][old_route[route_iter][COL]]
            == BLOCK) {
            break_pos = route_iter - 1;
            /* adding the first cell before break to queue at position 0 */
            queue_array[0][ROW] = old_route[break_pos][ROW];
            queue_array[0][COL] = old_route[break_pos][COL];
            queue_array[0][COUNT] = 0;
            queue_array[0][SEED_STATUS] = NO;
            break;
        }
    }

    /* finding position where blocked segment of route finishes */
    for (post_break_pos = break_pos + 1; post_break_pos < old_route_len;
        post_break_pos++) {
        if (grid_array[old_route[post_break_pos][ROW]]
            [old_route[post_break_pos][COL]] != BLOCK) {
            break;
        }
    }

    /* allocating memory to trav_array that stores potential flood cells
        (coordinates of cells above, below, to the left and right, in order) */
    trav_array = malloc(NO_DIRECTIONS * sizeof(int*));
    for (malloc_iter = 0; malloc_iter < NO_DIRECTIONS; malloc_iter++) {
        trav_array[malloc_iter] = malloc(COORD_LEN * sizeof(int));
    }

    /* iterating through the queue */
    while (break_loop == NO && queue_pos < queue_size) {
        /* if the cell hasnt been used as a seed */
        if (queue_array[queue_pos][SEED_STATUS] == NO) {
            /* creating possible next steps based on seed */
            traversal(queue_array[queue_pos][ROW], queue_array[queue_pos][COL],
                trav_array);
            /* pot_cell iterates through the up, down, left, right potential
                cells in traversal array */
            for (pot_cell = 0; pot_cell < NO_DIRECTIONS; pot_cell++) {
                /* checking the coordinate is legitamate */
                if (verify_coord(trav_array[pot_cell][ROW],
                    trav_array[pot_cell][COL], grid_array, queue_array,
                    grid->rows, grid->columns, queue_size) == YES) {
                    /* adding cell to queue and updating queue counter */
                    coords_in_queue = coords_in_queue + 1;
                    queue_array[coords_in_queue][ROW] =
                        trav_array[pot_cell][ROW];
                    queue_array[coords_in_queue][COL] =
                        trav_array[pot_cell][COL];
                    queue_array[coords_in_queue][COUNT] =
                        queue_array[queue_pos][COUNT] + 1;
                    queue_array[coords_in_queue][SEED_STATUS] = NO;
                    /* checking old route to see if a join has been made,
                        route_iter iterates through old route */
                    for (route_iter = post_break_pos;
                        route_iter < old_route_len; route_iter++) {
                        if (old_route[route_iter][ROW] ==
                            trav_array[pot_cell][ROW] &&
                            old_route[route_iter][COL] ==
                            trav_array[pot_cell][COL]) { /* join made */
                            join_row = old_route[route_iter][ROW];
                            join_col = old_route[route_iter][COL];
                            join_len = queue_array[coords_in_queue][COUNT];
                            /* to break out of loops */
                            pot_cell = NO_DIRECTIONS;
                            break_loop = YES;
                            break;
                        }
                    }
                }
            }
        }
        queue_pos =queue_pos + 1; /* updating position in queue */
    }

    /* end of queue reached with no join found = route cannot be repaired */
    if (break_loop == NO) {
        print_grid((row_no = grid->rows), (col_no = grid->columns), grid_array);
        printf("--------------------------------------------------\n");
        printf("The route cannot be repaired!\n");
        return 0;
    }

    /* allocating memory to repair array that stores new section of route */
    repair_array = malloc((join_len + 1) * sizeof(int*));
    for (malloc_iter = 0; malloc_iter < join_len + 1; malloc_iter++) {
        repair_array[malloc_iter] = malloc(COORD_LEN * sizeof(int));
    }

    /* fills repair array with new section of route */
    trace_route(queue_array, join_row, join_col, join_len, queue_size,
        repair_array);

    /* allocating memory to array that will store complete repaired route */
    new_route = malloc((new_route_len = old_route_len + join_len - 2)
        * sizeof(int*));
    for (malloc_iter = 0; malloc_iter < new_route_len; malloc_iter++) {
        new_route[malloc_iter] = malloc(COORD_LEN * sizeof(int));
    }

    /* filling new_route with old_route up until first break */
    for (route_iter = 0; route_iter < break_pos; route_iter++) {
        new_route[route_iter][ROW] = old_route[route_iter][ROW];
        new_route[route_iter][COL] = old_route[route_iter][COL];
    }

    /* iterating with rejoin to find at what position the repaired section
        rejoins the old route */
    for (rejoin = break_pos; rejoin < old_route_len; rejoin++) {
        if (old_route[rejoin][ROW] == repair_array[join_len][ROW] &&
            old_route[rejoin][COL] == repair_array[join_len][COL]) {
            break;
        }
    }

    /* adding repaird section into new route */
    new_route_pos = break_pos;
    for (rep_array_iter = 0; rep_array_iter < join_len + 1; rep_array_iter++) {
        new_route[new_route_pos][ROW] = repair_array[rep_array_iter][ROW];
        new_route[new_route_pos][COL] = repair_array[rep_array_iter][COL];
        new_route_pos = new_route_pos + 1;
    }

    /* adding rest of old route to new route */
    for (route_iter = new_route_pos; route_iter < new_route_len; route_iter++) {
        rejoin = rejoin + 1;
        new_route[route_iter][ROW] = old_route[rejoin][ROW];
        new_route[route_iter][COL] = old_route[rejoin][COL];
    }

    /* resetting 2d grid_array */
    for (row_iter = 0; row_iter < grid->rows; row_iter++) {
        for (col_iter = 0; col_iter < grid->columns; col_iter++) {
            if (grid_array[row_iter][col_iter] == ROUTE_CELL) {
                grid_array[row_iter][col_iter] = EMPTY_CELL;
            }
        }
    }

    /* putting new route into 2d grid_array and printing it */
    for (route_iter = 0; route_iter  < new_route_len; route_iter ++) {
        if (grid_array[new_route[route_iter][ROW]][new_route[route_iter][COL]]
            == EMPTY_CELL) {
            grid_array[new_route[route_iter][ROW]][new_route[route_iter][COL]]
            = ROUTE_CELL;
        }
    }

    print_grid((col_no = grid->columns), (row_no = grid->rows), grid_array);
    printf("----------------------------------------------------------\n");

    /* printing first cell of route */
    printf("[%d,%d]", new_route[0][ROW], new_route[0][COL]);
    format_counter = 1;

    /* printing rest of route */
    for (route_iter = 1; route_iter < new_route_len; route_iter ++) {
        if (format_counter == CELLS_PER_LINE) {
            printf("->\n");
            printf("[%d,%d]", new_route[route_iter][ROW],
                new_route[route_iter][COL]);
            format_counter = 0;
        } else {
            printf("->[%d,%d]", new_route[route_iter][ROW],
                new_route[route_iter][COL]);
        }
        format_counter = format_counter + 1;
        /* checking if the cell is blocked */
        if (grid_array[new_route[route_iter][ROW]][new_route[route_iter][COL]]
            == BLOCK) {
            block_status = YES;
        }
    }
    printf(".\n");

    if (block_status == YES) {
        printf("There is a block on this route!\n");
    } else {
        printf("The route is valid!\n");
    }

    /* freeing allocated memory */
    for (malloc_iter = 0; malloc_iter < NO_DIRECTIONS; malloc_iter++) {
        free(trav_array[malloc_iter]);
    }
    free(trav_array);
    for (malloc_iter = 0; malloc_iter < join_len + 1; malloc_iter++) {
        free(repair_array[malloc_iter]);
    }
    free(repair_array);
    for (malloc_iter = 0; malloc_iter < new_route_len; malloc_iter++) {
        free(new_route[malloc_iter]);
    }
    free(new_route);

    return new_route_len;
}
/****************************************************************************/
/* for printing appropriate stage 1 output */
int
print_stage1(int col_no, int row_no, char **grid_array, int status_no,
    int no_blocks, int **old_route, int old_route_len, grid_t *grid) {
    int **queue_array;
    int space_in_grid;
    int malloc_iter;
    int **new_route = NULL;
    int new_route_len;

    /* printing intial grid */
    printf("==STAGE 1=========================================\n");
    print_grid(col_no, row_no, grid_array);

    /* terminating stage 1 if status is not 4 */
    if (status_no != 4) {
        printf("==================================================\n");
        return 0;
    }

    printf("--------------------------------------------------\n");

    /* declaring memory required for queue array, 4 ints in each internal array
    for 4 data points: Row, Col, Count and Seed_status */
    space_in_grid = ((col_no * row_no) - no_blocks) + 1;
    queue_array = malloc(space_in_grid * sizeof(int*));
    for (malloc_iter = 0; malloc_iter < space_in_grid; malloc_iter++) {
        queue_array[malloc_iter] = malloc(4 * sizeof(int));
        nullify_line_int(queue_array[malloc_iter], 4);
    }

    new_route_len = repair_route(queue_array, old_route, old_route_len, grid,
        grid_array, space_in_grid, new_route);
    printf("==================================================\n");

    /* freeing allocated memory */
    for (malloc_iter = 0; malloc_iter < space_in_grid; malloc_iter++) {
        free(queue_array[malloc_iter]);
    }
    free(queue_array);

    return 0;
}
/****************************************************************************/
/* prints a grid based off given inputs */
void
print_grid(int col_no, int row_no, char **grid_array) {
    int a_count1;
    int a_count2;
    int print_count = 0;

    /* printing top coordinates */
    printf(" ");
    for (a_count1 = 0; print_count < col_no; a_count1++) {
        if (a_count1 == 10) {
            a_count1 = 0;
        }
        printf("%d", a_count1);
        print_count = print_count + 1;
    }
    printf("\n");
    /* printing lines of grid */
    for (a_count1 = (print_count = 0); print_count < row_no; a_count1++) {
        if (a_count1 == 10) {
            a_count1 = 0;
        }
        printf("%d", a_count1);
        for (a_count2 = 0; a_count2 < col_no; a_count2++) {
            printf("%c", grid_array[a_count1][a_count2]);
        }
        print_count = print_count + 1;
        printf("\n");
    }
}
/****************************************************************************/
/* prints out propsed route, checks status and fills old_route array */
void
print_route(list_t *list, grid_t *grid, char **grid_array, int **old_route,
    route_info_struct_t *route_struct, int *status, data_t *route_coord) {
    int format_counter = 1;
    int old_route_count = 1;

    /* checking if first route cell is same as initial cell and printing it */
    *route_coord = get_head(list);
    if ((route_coord->row_ != grid->initial_cell[ROW]) ||
        (route_coord->col_ != grid->initial_cell[COL])) {
        status[STATUS_1] = YES;
    }
    old_route[0][ROW] = route_coord->row_;
    old_route[0][COL] = route_coord->col_;

    printf("[%d,%d]", route_coord->row_, route_coord->col_);
    /* checking coord is in grid bounds */
    if (route_coord->row_ >= 0 && route_coord->col_ >= 0 &&
        route_coord->row_ < grid->rows && route_coord->col_ < grid->columns) {
        /* changing map of grid to show route */
        if (grid_array[route_coord->row_][route_coord->col_] == ’ ’) {
                grid_array[route_coord->row_][route_coord->col_] = ’*’;
        }
    }
    list = get_tail(list);

    /* printing rest of route */
    while(!is_empty_list(list)) {
        *route_coord = get_head(list);
        old_route[old_route_count][ROW] = route_coord->row_;
        old_route[old_route_count][COL] = route_coord->col_;
        if (format_counter == CELLS_PER_LINE) {
            printf("->\n");
            printf("[%d,%d]", route_coord->row_, route_coord->col_);
            format_counter = 0;
        } else {
            printf("->[%d,%d]", route_coord->row_, route_coord->col_);
        }
        if (grid_array[route_coord->row_][route_coord->col_] == ’ ’) {
            grid_array[route_coord->row_][route_coord->col_] = ’*’;
        }
        list = get_tail(list);
        format_counter = format_counter + 1;
        old_route_count = old_route_count + 1;
        /* checking if the cell is blocked */
        if (grid_array[route_coord->row_][route_coord->col_] == BLOCK) {
            status[STATUS_4] = YES;
        }
    }
    printf(".\n");

    /* checking if last cell is same as goal cell */
    if ((route_coord->row_ != grid->goal_cell[ROW]) ||
        (route_coord->col_ != grid->goal_cell[COL])) {
        status[STATUS_2] = YES;
    }
}
/****************************************************************************/
/* creating a linked list that stores the proposed route*/
int
```

```
create_route_list(list_t *list, int old_row, int old_col, grid_t *grid,
    route_info_struct_t *route_struct, int *status, data_t *route_coord) {
    int route_cont = YES;
    int route_len = 0;
    int a_count1;

    while (route_cont == YES) {
        *route_struct = read_route(old_row, old_col);
        /* adding cells to linked list that stores route */
        for (a_count1 = 0; a_count1 < route_struct->no_coords; a_count1++) {
            route_coord->row_ = (old_row =
                route_struct->route_coords[a_count1][ROW]);
            route_coord->col_ = (old_col =
                route_struct->route_coords[a_count1][COL]);
            list = insert_at_foot(list, *route_coord);
            route_len = route_len + 1;
            /* checking route coords are out of bounds */
            if (old_row < 0 || old_col < 0 || old_row >= grid->rows ||
                old_col >= grid->columns) {
                status[STATUS_3] = YES;
            }
        }
        /* checking if it is end of route */
        if (route_struct->route_end == YES) {
            route_cont = NO;
        }
        /* checking if it is a valid route */
        if (route_struct->route_illegal == YES) {
            status[STATUS_3] = YES;
        }
    }
    return route_len;
}
/****************************************************************************/
/* prints relevant stage0 output */
void
print_stage0(int blocks, grid_t grid) {
    printf("==STAGE 0=======================================\n");
    printf("The grid has %d rows and %d columns.\n", grid.rows, grid.columns);
    printf("The grid has %d block(s).\n", blocks);
    printf("The initial cell in the grid is [%d,%d].\n", grid.initial_cell[0],
        grid.initial_cell[1]);
    printf("The goal cell in the grid is [%d,%d].\n", grid.goal_cell[0],
        grid.goal_cell[1]);
    printf("The proposed route in the grid is:\n");
}
/****************************************************************************/
/* reads a line of route coords, returns struct with array of coords, 3 ints
 that show if: route is legal, if its the last route line, and no. coords */
route_info_struct_t
read_route(int old_row, int old_col) {
    char line[MAX_LINE_LEN];
    char copy_str[MAX_LINE_LEN];
    char last_char;
    int chars_1coord;
    int chars_total = 0;
    int copy_coord[COORD_LEN];
    int coord_count = 0;
    int a_iter;
    int a_iter2;
    route_info_struct_t return_struct;

    nullify_line(line, MAX_LINE_LEN);
    nullify_line(copy_str, MAX_LINE_LEN);
    read_line(line);

    /* checking if it is the last line of the route */
    for (a_iter = 0; a_iter < MAX_LINE_LEN; a_iter++) {
        if (line[a_iter] != ’\0’ && line[a_iter] != ’\n’) {
            last_char = line[a_iter];
        }
        copy_str[a_iter] = line[a_iter];
    }
    if (last_char == ’]’) {
```

```c
        return_struct.route_end = YES;
    } else {
        return_struct.route_end = NO;
    }

    /* counting the number of coordinates in line */
    for (a_iter = 0; a_iter < MAX_LINE_LEN; a_iter++) {
        if (line[a_iter] == '[') {
            coord_count = coord_count + 1;
        }
    }
    return_struct.no_coords = coord_count;

    /* creating appropraite array to store coords */
    return_struct.route_coords = malloc(coord_count * sizeof(int*));
    for (a_iter = 0; a_iter < coord_count; a_iter++) {
        return_struct.route_coords[a_iter] = malloc(COORD_LEN * sizeof(int));
    }

    /* reading coords into 2d array */
    for (a_iter = 0; a_iter < coord_count; a_iter++) {
        chars_1coord = scrape_coord(copy_str, copy_coord);
        return_struct.route_coords[a_iter][ROW] = copy_coord[ROW];
        return_struct.route_coords[a_iter][COL] = copy_coord[COL];

        chars_total = chars_total + chars_1coord + 2;
        /* updating copy_str, the +2's are to skip over the -> in input */
        for (a_iter2 = 0; a_iter2 + chars_total + 2 < MAX_LINE_LEN;
            a_iter2++) {
                copy_str[a_iter2] = line[a_iter2 + chars_total];
        }
    }

    /* checking if route is legal */
    return_struct.route_illegal = illegal_route(return_struct.route_coords,
        old_row, old_col, coord_count);

    return return_struct;
}
/****************************************************************************/
/* checks step sizes between coords to see if route is legal or not */
int
illegal_route(int **r_array, int old_row, int old_col, int array_len) {
    int a_iter;
    int step_size1;
    int step_size2;

    /* checking step from last of last line to first of this line */
    step_size1 = old_row - r_array[0][ROW];
    step_size2 = old_col - r_array[0][COL];

    if (((step_size1 > 1 || step_size1 < -1) || (step_size2 > 1 ||
        step_size2 < -1) || (step_size1 != 0 && step_size2 != 0)) ||
        (step_size1 == 0 && step_size2 == 0)){

        return YES;
    }
    /* checking step sizes in route array */
    for (a_iter = 0; a_iter + 1 < array_len; a_iter++) {
        step_size1 = r_array[a_iter][ROW] - r_array[a_iter + 1][ROW];
        step_size2 = r_array[a_iter][COL] - r_array[a_iter + 1][COL];
        if (((step_size1 > 1 || step_size1 < -1) || (step_size2 > 1 ||
            step_size2 < -1) || (step_size1 != 0 && step_size2 != 0)) ||
            (step_size1 == 0 && step_size2 == 0)) {
            return YES;
        }
    }
    return NO;
}
/****************************************************************************/
/* prints status based on info in status array */
int
print_status(int *status) {
    int status_no;
    for (status_no = 0; status_no < NO_OF_STATUS; status_no++) {
        if (status[status_no] == YES) {
            break;
        }
    }
    if (status_no == STATUS_1) {
        printf("Initial cell in the route is wrong!\n");
        return status_no;
    }
    if (status_no == STATUS_2) {
        printf("Goal cell in the route is wrong!\n");
        return status_no;
    }
    if (status_no == STATUS_3) {
        printf("There is an illegal move in this route!\n");
        return status_no;
    }
    if (status_no == STATUS_4) {
        printf("There is a block on this route!\n");
        return status_no;
    }
    printf("The route is valid!\n");
    return status_no;
}
/****************************************************************************/
/* interprets none route coordinate lines */
line_info_struct_t
handle_line(void) {
    line_info_struct_t return_struct;
    int coordinate[COORD_LEN];
    char line[MAX_LINE_LEN];

    nullify_line(line, MAX_LINE_LEN);
    read_line(line);
    scrape_coord(line, coordinate);
    return_struct.coord[ROW] = coordinate[ROW];
    return_struct.coord[COL] = coordinate[COL];
    /* end of block sequence */
    if (line[0] == END_OF_BLOCKS) {
        return_struct.block_end = YES;
        return return_struct;
    } else {
        return_struct.block_end = NO;
    }

    return return_struct;
}
/****************************************************************************/
/* reads the value of the first coords in line, adds them to coords as ints */
int
scrape_coord(char *line, int *coord) {
    char string[MAX_LINE_LEN];
    int r_dig_count;
    int c_dig_count;
    int copy_count;
    int chars_proccessed = 0;
    int offset;

    for (offset = 0; isdigit(line[offset]) == 0; offset++) {
    }

    nullify_line(string, MAX_LINE_LEN);

    /* counting digits in row value in cell */
    for (r_dig_count = 0; isdigit(line[r_dig_count + offset]) > 0;
        r_dig_count++) {
    }

    chars_proccessed = chars_proccessed + r_dig_count;

    /* copying row value to string array */
    for (copy_count = 0; copy_count <= r_dig_count - 1; copy_count++) {
        string[copy_count] = line[copy_count + offset];
    }

    coord[ROW] = my_getnbr(string);
    nullify_line(string, MAX_LINE_LEN);

    /* counting number of digits in col value, (+1 is there to factor for
        the ',' or 'x' in the cordinate */
    for (c_dig_count = 0; isdigit(line[c_dig_count + offset +
        r_dig_count + 1]); c_dig_count++) {
    }

    chars_proccessed = chars_proccessed + c_dig_count;

    /* adding col digits to string array */
    for (copy_count = 0; copy_count < c_dig_count ; copy_count++) {
        string[copy_count] = line[copy_count + r_dig_count + offset + 1];
    }

    coord[COL] = my_getnbr(string);
    chars_proccessed = chars_proccessed + 3;
    return chars_proccessed;
}
/****************************************************************************/
/* fills char array with null bytes */
void
nullify_line(char *line, int len) {
    int k;

    for (k=0; k < len; k++) {
        line[k] = '\0';
    }
}
/****************************************************************************/
/* fills int array with null bytes */
void
nullify_line_int(int *line, int len) {
    int k;

    for (k=0; k < len; k++) {
        line[k] = '\0';
    }
}
/****************************************************************************/
/* mygetchar function obtained from assignment 1 FAQ page */
int
mygetchar() {
    int c;
    while ((c=getchar())=='\r') {
    }
    return c;
}
/****************************************************************************/
/* read_line takes one line from stdin and places it in line array */
void
read_line(char *line) {
    int line_iter = 0;
    char pot_char;

    while ((pot_char = mygetchar()) != '\n') {
        line[line_iter] = pot_char;
        line_iter = line_iter + 1;
    }

    line[line_iter] = '\0'; /* adding sentinel */
}
/****************************************************************************/
/* my_getnbr function takes a string and returns an integer based on string
    function obtained from:
    https://stackoverflow.com/questions/7021725/how-to-convert-a-
    string-to-integer-in-c . no changes made (i dont trust atoi) */
int
my_getnbr(char *str) {
    int result;
    int puiss;
    result = 0;
    puiss = 1;
    while ((’−’ == (*str)) || ((*str) == ’+’))
    {
        if (*str == ’−’)
            puiss = puiss * −1;
        str++;
    }
    while ((*str >= ’0’) && (*str <= ’9’))
    {
        result = (result * 10) + ((*str) − ’0’);
        str++;
    }
    return (result * puiss);
}
/***************************************************************************/
/* remaining functions taken from pgs 172−173 of alistair’s book, no changes */
/***************************************************************************/
list_t
*make_empty_list(void) {
    list_t *list;
    list = (list_t*)malloc(sizeof(*list));
    assert(list!=NULL);
    list−>head = list−>foot = NULL;
    return list;
}
/***************************************************************************/
int
is_empty_list(list_t *list) {
    assert(list!=NULL);
    return list−>head==NULL;
}
/***************************************************************************/
void
free_list(list_t *list) {
    node_t *curr, *prev;
    assert(list!=NULL);
    curr = list−>head;
    while (curr) {
        prev = curr;
        curr = curr−>next;
        free(prev);
    }
    free(list);
}
/***************************************************************************/
list_t
*insert_at_head(list_t *list, data_t value) {
    node_t *new;
    new = (node_t*)malloc(sizeof(*new));
    assert(list!=NULL && new!=NULL);
    new−>data = value;
    new−>next = list−>head;
    list−>head = new;
    if (list−>foot==NULL) {
        /* this is the first insertion into the list */
        list−>foot = new;
    }
    return list;
}
/***************************************************************************/
list_t
*insert_at_foot(list_t *list, data_t value) {
    node_t *new;
    new = (node_t*)malloc(sizeof(*new));
    assert(list!=NULL && new!=NULL);
    new−>data = value;
    new−>next = NULL;
    if (list−>foot==NULL) {
        /* this is the first insertion into the list */
        list−>head = list−>foot = new;
    } else {
        list−>foot−>next = new;
        list->foot = new;
    }
    return list;
}
/****************************************************************************/
data_t
get_head(list_t *list) {
    assert(list!=NULL && list->head!=NULL);
    return list->head->data;
}
/****************************************************************************/
list_t
*get_tail(list_t *list) {
    node_t *oldhead;
    assert(list!=NULL && list->head!=NULL);
    oldhead = list->head;
    list->head = list->head->next;
    if (list->head==NULL) {
        /* the only list node just got deleted */
        list->foot = NULL;
    }
    free(oldhead);
    return list;
}
/****************************************************************************/
/* algorithms are fun */
```