```
=================================================================
            /local/submit/submit/comp10002/ass2/xuliny/src/ass2sol10.c
=================================================================

/*comp10002 assignment2 by Xulin Yang 904904, October 2017*/

#include <stdio.h>
#include <stdlib.h>

#define STAGE_ONE "S1: " /*stage 1 output indication*/
#define STAGE_TWO "S2: " /*stage 2 output indication*/
#define STAGE_THREE "S3: " /*stage 3 output indication*/

#define TOWARDS_LEFT "<<<<" /*car go left*/
#define TOWARDS_RIGHT ">>>>" /*car go right*/
#define NO_LEFT_RIGHT "    " /*car go neither left or right*/
#define TOWARDS_UP '^' /*car go up*/
#define TOWARDS_DOWN 'v' /*car go down*/
#define NO_UP_DOWN ' ' /*car go neither up or down*/
#define SHORTER_SEPARATOR "----+" /*separator for first column*/
#define LONGER_SEPARATOR "--------+" /*separator for non first column*/

#define SMALLER 1 /*return value for a smaller than b*/
#define EQUAL 0 /*return value for a equals b*/
#define BIGGER -1 /*return value for a bigger than b*/

#define RIGHT 3 /*direction right*/
#define DOWN 2 /*direction down*/
#define UP 1 /*direction up*/
#define LEFT 0 /*direction left*/
#define ONE_CAR 1 /*number of the start grid in stage2*/
#define START_COST 0 /*the cost of grid to reach itself*/

/*max number of directions a grid can go to its adjacent grid*/
#define BLOCK_NUM 4

/*the cost indicate the grid can't go in this direction*/
#define INVALID_PATH 999

/*return alphabet from a-z to integer 0-25*/
#define CHAR_TO_INT(c) (c - 'a')

/*return interger from 0-25 to alphabet a-z*/
#define INT_TO_CHAR(n) (n + 'a')

/*return the column index of left grid*/
#define TO_LEFT(y) (y - 1)

/*return the row index of the upper grid*/
#define TO_UP(x) (x - 1)

/*return the row index of the under grid*/
#define TO_DOWN(x) (x + 1)

/*return the column index of the right grid*/
#define TO_RIGHT(y) (y + 1)

/***********************************************************/

typedef struct location location_t;
typedef struct city city_t;
typedef struct travel travel_t;
typedef struct grid grid_t;

/*structure to store the coordinates of the grid*/
struct location {
    int y; /*column index of the grid*/
    char x; /*row index of the grid*/
};

/*structure to store relevent information of the city*/
struct city {
    int num_row; /*number of west-east street in the city*/
    int num_column; /*number of the north-south street in the city*/
```

```
    int total_grid; /*total number of the grid in the city*/

    /*a row*column*block_num array to store the cost
      from one grid to its adjacent grid*/
    int ***adj_cost;

    /*number of path from one grid to adjacent grid has cost = 999*/
    int invalid_path;

    /*sum of path from one grid to adjacent grid has cost < 999*/
    int total_valid_path_cost;
};

/*structure to store the start grid(car) coordinates to travel in city*/
struct travel {
    location_t *location; /*coordinates of all start grids*/
    int total_travel; /*number of strat grids are given*/
};

/*structure of each grid in the city*/
struct grid {
    /*the coordinates of the previous grid
      where the current grid from to obtain this cost_used*/
    location_t pre;

    /*the coordinates of the current grid*/
    location_t cur;

    /*the cost already used to reach the current grid from the start grid*/
    int cost_used;
};

/**********************************************************/

void get_city_dimension(city_t *city);
void make_empty_grid(city_t *city);
void load_information(city_t *city);
void get_cost(city_t *city);
void path_census(city_t *city, int *adj_cost);
void load_car_grid(travel_t *travel, city_t city);
void print_stage1(travel_t travel, city_t city);

grid_t** make_empty_map(city_t city);
void load_car(grid_t ***map, travel_t travel, int car_num);
void cal_map(grid_t ***map, city_t city);
int has_road(int *adj_cost, int direction);
void adjacent_test(grid_t ***map, city_t city, int x, int y, int *changed);
void update_path(grid_t ***map, city_t city, int from_x, int from_y,
        int to_x, int to_y, int direction, int *flag);

void print_stage2(grid_t **map, travel_t travel, city_t city);
void recursive_back_trace(grid_t **map, location_t cur, city_t city);
int grid_location_cmp(location_t cur, location_t pre);

void print_stage3(city_t city, grid_t **map);
void print_head(city_t city);
void print_cost_row(city_t city, grid_t **map, int row);
void print_left_right(city_t city, grid_t **map, int x, int y);
void print_direction_row(city_t city, grid_t **map, int row);
void print_up_down(city_t city, grid_t **map, int x, int y);

grid_t** free_map(city_t city, grid_t **map);
int*** free_adj_cost(city_t *city);
location_t *free_travel(travel_t *travel);

/**********************************************************/

int main() {
    city_t city;
    travel_t travel;

    /*stage1*/
    get_city_dimension(&city);
    make_empty_grid(&city);
```

```c
    load_information(&city);
    load_car_grid(&travel, city);
    print_stage1(travel, city);
    printf("\n");

    /*stage2*/

    /*a map of city that contains the path from one grid to another grid*/
    grid_t **map = make_empty_map(city);
    load_car(&map, travel, ONE_CAR);
    cal_map(&map, city);
    print_stage2(map, travel, city);
    putchar('\n');

    /*stage3*/
    map = free_map(city, map);
    map = make_empty_map(city);
    load_car(&map, travel, travel.total_travel);
    cal_map(&map, city);
    print_stage3(city, map);
    putchar('\n');

    /*free everything*/
    map = free_map(city, map);
    city.adj_cost = free_adj_cost(&city);
    travel.location = free_travel(&travel);
    return 0;
}

/*********************************************************/

/*get the number of columns and rows of the city from file*/
void get_city_dimension(city_t *city) {
    scanf("%d %d", &(city->num_column), &(city->num_row));
    city->total_grid = city->num_row * city->num_column;
    return;
}

/*malloc space to store the cost from one grid to adjacent grid*/
void make_empty_grid(city_t *city) {
    int i, j;
    city->adj_cost = (int***)malloc(sizeof(int**) * city->num_row);
    for (i = 0; i < city->num_row; i++) {
        city->adj_cost[i] = (int**)malloc(sizeof(int*) * city->num_column);
        for (j = 0; j < city->num_column; j++) {
            city->adj_cost[i][j] = (int*)malloc(sizeof(int) * BLOCK_NUM);
        }
    }
    return;
}

/*load grid information from file*/
void load_information(city_t *city) {
    /*initialize value*/
    city->invalid_path = 0;
    city->total_valid_path_cost = 0;

    get_cost(city);
    return;
}

/*get cost from one grid to its adjacent grid*/
void get_cost(city_t *city) {
    int i, j, column;
    char row;

    /*read row with number of total grid times from formatted input*/
    for (i = 0; i < city->num_row; i++) {
        for (j = 0; j < city->num_column; j++) {

            /*cost are read in right, up, left and down order but stored in
              left, up, down and right order in order to compute cost for
              adjacent grid in stage2 and stage3 in lexicographical order*/
            scanf("%d%c %d %d %d %d", &column, &row,
```

```
                &(city->adj_cost[i][j][RIGHT]), &(city->adj_cost[i][j][UP]),
                &(city->adj_cost[i][j][LEFT]), &(city->adj_cost[i][j][DOWN]));

                /*census for current grid*/
                path_census(city, city->adj_cost[i][j]);
            }
        }

        return;
    }

    /*census number of cost = 999 and sum of all cost < 999*/
    void path_census(city_t *city, int *adj_cost) {
        int i;
        for (i = 0; i < BLOCK_NUM; i++) {
            if (adj_cost[i] == INVALID_PATH) {
                city->invalid_path++;
            } else {
                city->total_valid_path_cost += adj_cost[i];
            }
        }
        return;
    }

    /*load grid locations of cars which are needed to find path*/
    void load_car_grid(travel_t *travel, city_t city) {
        char tmp_row;
        int tmp_column;

        /*maximum number of car is the number of total grid in city*/
        travel->location = (location_t*)malloc(sizeof(location_t) *
            city.total_grid);

        /*keep reading locations of cars until no more input*/
        for (travel->total_travel = 0;
                (scanf("%d%c", &tmp_column, &tmp_row) == 2);
                travel->total_travel++) {

            travel->location[travel->total_travel].y = tmp_column;
            travel->location[travel->total_travel].x = tmp_row;
        }

        /*realloc space to store locations of cars based on number of cars are
          read*/
        travel->location = realloc(travel->location,
            sizeof(location_t) * travel->total_travel);
        return;
    }

    /*print the output of stage one*/
    void print_stage1(travel_t travel, city_t city) {
        printf("%sgrid is %d x %d, and has %d intersections\n", STAGE_ONE,
            city.num_column, city.num_row, city.total_grid);
        printf("%sof %d possibilities, %d of them cannot be used\n",
            STAGE_ONE, city.total_grid * BLOCK_NUM, city.invalid_path);
        printf("%stotal cost of remaining possibilities is %d seconds\n",
            STAGE_ONE, city.total_valid_path_cost);
        printf("%s%d grid locations supplied, first one is %d%c,\
        last one is %d%c\n",
            STAGE_ONE, travel.total_travel,
            travel.location[0].y, travel.location[0].x,
            travel.location[travel.total_travel - 1].y,
            travel.location[travel.total_travel - 1].x);

        return;
    }

    /*********************************************************/

    /*malloc space for map of city*/
    grid_t** make_empty_map(city_t city) {
        int i, j;

        /*malloc map with the same number of grid in city*/
```

```
    grid_t **map = (grid_t**)malloc(sizeof(grid_t*) * city.num_row);

    /*initialize every grid's initial cost and its coordinates*/
    for (i = 0; i < city.num_row; i++) {
        map[i] = (grid_t*)malloc(sizeof(grid_t) * city.num_column);
        for (j = 0; j < city.num_column; j++) {
            map[i][j].cost_used = INVALID_PATH;
            map[i][j].cur.y = j;
            map[i][j].cur.x = INT_TO_CHAR(i);
        }
    }

    return map;
}

/*set specified number(car_num) of start grids with cost_used = 0 in the map*/
void load_car(grid_t ***map, travel_t travel, int car_num) {
    int k, row, column;

    /*set start grid(s) according to number of car
      with cost = 0 and its previous grid is itself*/
    for (k = 0; k < car_num; k++) {
        row = CHAR_TO_INT(travel.location[k].x);
        column = travel.location[k].y;

        (*map)[row][column].cost_used = START_COST;
        (*map)[row][column].pre = (*map)[row][column].cur;
    }
    return;
}

/*calculate the minimun cost from start grid to any grid in city and find the
  previous grid to obtain this cost to this grid*/
void cal_map(grid_t ***map, city_t city) {
    int i, j, changed = 1, have_valid = 0;

    /*iterate unitil no change for each grid which means
      the costs have stabilized and reached their final minimum values and
      have found their previous grid*/
    while (changed) {
        changed = 0;
        have_valid = 0;

        /*traverse every grid*/
        for (i = 0; i < city.num_row; i++) {
            for (j = 0; j < city.num_column; j++) {
                /*when found first grid's cost != 999,
                  dierction test can be applied to all grids remained*/
                if (((*map)[i][j].cost_used != INVALID_PATH) || have_valid) {

                    /*have found first non-999 cost grid*/
                    have_valid = 1;

                    /*test whether there's a less cost way to reach adjacent
                      grid from the current grid*/
                    adjacent_test(map, city, i, j, &changed);
                }
            }
        }
    }
    return;
}

/*return whether the currecnt grid has a valid path in specified direction*/
int has_road(int *adj_cost, int direction) {
    return adj_cost[direction] != INVALID_PATH;
}

/*check whether there is a cheaper way to reach the adjacent grid from current
  grid with coordinates(x, y)*/
void adjacent_test(grid_t ***map, city_t city, int x, int y, int *changed) {
    if (has_road(city.adj_cost[x][y], LEFT)) {
        update_path(map, city, x, y, x, TO_LEFT(y), LEFT, changed);
    }
```

```c
    if (has_road(city.adj_cost[x][y], UP)) {
        update_path(map, city, x, y, TO_UP(x), y, UP, changed);
    }
    if (has_road(city.adj_cost[x][y], DOWN)) {
        update_path(map, city, x, y, TO_DOWN(x), y, DOWN, changed);
    }
    if (has_road(city.adj_cost[x][y], RIGHT)) {
        update_path(map, city, x, y, x, TO_RIGHT(y), RIGHT, changed);
    }
    return;
}

/*change adjacent grid's previous grid to current grid(from_x, from_y
  as well as the cost to reach adjacent grid(to_x, to_y) from the start grid*/
void update_path(grid_t ***map, city_t city, int from_x, int from_y,
        int to_x, int to_y, int direction, int *changed) {
    /*when a less cost way or
      an equal cost and a lexicographically smaller path is found,
      update adjacent grid's previous grid to the current grid and
      its cost used from start point*/
    if (((*map)[to_x][to_y].cost_used > (*map)[from_x][from_y].cost_used +
            city.adj_cost[from_x][from_y][direction]) ||
        (((*map)[to_x][to_y].cost_used == (*map)[from_x][from_y].cost_used +
            city.adj_cost[from_x][from_y][direction]) &&
            (grid_location_cmp((*map)[from_x][from_y].cur,
            (*map)[to_x][to_y].pre) == SMALLER))) {

        /*update the cost can be used to reach the adjacent grid from
          currecnt grid*/
        (*map)[to_x][to_y].cost_used = (*map)[from_x][from_y].cost_used +
            city.adj_cost[from_x][from_y][direction];

        /*update current grid as the previous grid of the adjacent grid*/
        (*map)[to_x][to_y].pre = (*map)[from_x][from_y].cur;

        /*has updated path*/
        *changed = 1;
    }
    return;
}

/*print the output of stage two*/
void print_stage2(grid_t **map, travel_t travel, city_t city) {
    int i;

    /*trace out the path from start grid to all destination
      and first locaton in travel is the start grid, so
      no need to trace back from that*/
    for (i = 1; i < travel.total_travel; i++) {
        recursive_back_trace(map, travel.location[i], city);
    }

    return;
}

/*back trace the path from end to start and print out with cost to
  reach it recursively*/
void recursive_back_trace(grid_t **map, location_t cur, city_t city) {
    int x = CHAR_TO_INT(cur.x), y = cur.y;

    /*if the current grid's previous grid is not itself, then it hasn't
      reached the start grid from the end*/
    if (grid_location_cmp(cur, map[x][y].pre)) {
        recursive_back_trace(map, map[x][y].pre, city);
    }

    /*print out the path with cost already used*/
    if (!grid_location_cmp(cur, map[x][y].pre)) {
        /*start grid's previous grid is itself*/
        printf("%sstart at grid ", STAGE_TWO);
    } else {
        printf("%s    then to ", STAGE_TWO);
    }
    printf("%d%c, cost of %d\n", map[x][y].cur.y, map[x][y].cur.x,
```

```
            map[x][y].cost_used);

        return;
    }

    /*compare two grids' coordinates(x, y) according to lexicographical order*/
    int grid_location_cmp(location_t first, location_t second) {
        /*if first grid's y is smaller then second grid's y or
          a tie in y but first grid's x is smaller then second grid's x,
          then first grid is lexicographically smaller than second grid*/
        if ((first.y < second.y) ||
            ((first.y == second.y) && (first.x < first.x))) {
            return SMALLER;

        /*if two grid have the same coordinates,
          then first grid is lexicographically equals second grid*/
        } else if ((first.x == second.x) && (first.y == second.y)) {
            return EQUAL;
        } else {
            return BIGGER;
        }
    }

    /*******************************************************/

    /*print the output of stage three*/
    void print_stage3(city_t city, grid_t **map) {
        int i;
        print_head(city);

        for (i = 0; i < city.num_row; i++) {
            print_cost_row(city, map, i);

            /*last row doesn't need to consider have route toward below or
              route from below*/
            if (i < city.num_row - 1) {
                print_direction_row(city, map, i);
                print_direction_row(city, map, i);
            }
        }

        return;
    }

    /*print the column axis and the separator of the output*/
    void print_head(city_t city) {
        int i;

        /*print the column axis*/
        printf("%s  ", STAGE_THREE);
        for (i = 0; i < city.num_column; i++) {
            if (i == 0) {
                printf("%5d", i);
            } else {
                printf("%9d", i);
            }
        }
        putchar('\n');

        /*print the separator line*/
        printf("%s +", STAGE_THREE);
        for (i = 0; i < city.num_column; i++) {
            if (i == 0) {
                printf("%s", SHORTER_SEPARATOR);
            } else {
                printf("%s", LONGER_SEPARATOR);
            }
        }
        putchar('\n');

        return;
    }

    /*print the row with cost to reach the grid and the travel direction
```

```
  left and right*/
void print_cost_row(city_t city, grid_t **map, int x) {
    int i;

    for (i = 0; i < city.num_column; i++) {
        /*print the row axis with first column*/
        if (i == 0) {
            printf("%s%c |%5d", STAGE_THREE, INT_TO_CHAR(x),
                map[x][i].cost_used);

        /*print the column with its left, right or neither direction and
          the cost to reach here from start*/
        } else {
            printf(" ");
            print_left_right(city, map, x, i);
            printf("%4d", map[x][i].cost_used);
        }
    }

    putchar('\n');

    return;
}

/*print direcion left, right or neither*/
void print_left_right(city_t city, grid_t **map, int x, int y) {
    /*if the left grid's previous grid is the current grid,
      then the direction is towards left*/
    if (!grid_location_cmp(map[x][y].cur, map[x][TO_LEFT(y)].pre)) {
        printf("%s", TOWARDS_LEFT);

    /*if the current grid's previous grid is the left grid,
      then the direction is towards right*/
    } else if (!grid_location_cmp(map[x][y].pre, map[x][TO_LEFT(y)].cur)) {
        printf("%s", TOWARDS_RIGHT);
    } else {
        printf("%s", NO_LEFT_RIGHT);
    }
    return;
}

/*print the row indicates the travel direction up or down or neither*/
void print_direction_row(city_t city, grid_t **map, int row) {
    int i;

    for (i = 0; i < city.num_column; i++) {
        /*print the indention for first column*/
        if (i == 0) {
            printf("%s |  ", STAGE_THREE);

        /*print the indention for the rest column*/
        } else {
            printf("    ");
        }

        /*print up, down or neither direction for the column*/
        print_up_down(city, map, row, i);
    }

    putchar('\n');

    return;
}

/*print direcion up or down or neither*/
void print_up_down(city_t city, grid_t **map, int x, int y) {
    /*if the under grid's previous grid is the current grid,
      then the direction is towards down*/
    if (!grid_location_cmp(map[x][y].cur, map[TO_DOWN(x)][y].pre)) {
        putchar(TOWARDS_DOWN);

    /*if the current grid's previous grid is the under grid,
      then the direction is towards up*/
    } else if (!grid_location_cmp(map[x][y].pre, map[TO_DOWN(x)][y].cur)) {
```

```
        putchar(TOWARDS_UP);
    } else {
        putchar(NO_UP_DOWN);
    }
    return;
}

/*free the space malloced to map*/
grid_t** free_map(city_t city, grid_t **map) {
    int i;
    for (i = 0; i < city.num_row; i++) {
        free(map[i]);
        map[i] = NULL;
    }

    free(map);
    map = NULL;
    return map;
}

/*free memory malloced for city's cost information*/
int*** free_adj_cost(city_t *city) {
    int i, j;

    for (i = 0; i < city->num_row; i++) {
        for (j = 0; j < city->num_column; j++) {
            free(city->adj_cost[i][j]);
            city->adj_cost[i][j] = NULL;
        }

        free(city->adj_cost[i]);
        city->adj_cost[i] = NULL;
    }

    free(city->adj_cost);
    city->adj_cost = NULL;

    return city->adj_cost;
}

/*free memory malloced for travel*/
location_t *free_travel(travel_t *travel) {
    int i;

    for (i = 0; i < travel->total_travel; i++) {
        free(travel->location);
        travel->location = NULL;
    }

    return travel->location;
}

/*algorithms are fun*/
```