```
======================================================================
            /local/submit/submit/comp10002/ass2.late/chanjieh/src/ass.c
======================================================================

/* Solution to comp10002 Assignment 2, 2018 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: [Chan Jie Ho - 961948]
 * Dated:     [25/9/18]

 */


/* ========================================================================== */

/* Libraries to include and hash-defined variables sorted alphabetically */
/* -------------------------------------------------------------------- */

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

#define COMPLETE            3
#define EMPTY               0
#define ERROR              -1
#define EVEN                2
#define FIRST               0       /* MAY BE CHARACTER OR STRING */
#define FIRST_TEN           10
#define MAX_VALUE           100
#define MIN_VALUE           1
#define MAX_VERTICES        52
#define MULTIPLE_OF_FIVE    5
#define NO                  0
#define NON_EMPTY           1
#define ODD                 1
#define PRINT_LIMIT         12
#define PRINT_LENGTH        6
#define SECOND              1       /* MAY BE CHARACTER OR STRING */
#define STAGE_ONE           1
#define STAGE_TWO           2
#define STAGE_ZERO          0
#define THIRD               2
#define TRAVERSABLE         2
#define YES                 1
#define ZERO_OFFSET         1
```

```
/* ========================================================================== */

/* Type Definitions */
/* --------------- */

typedef int data_t;

typedef struct node node_t;

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

typedef struct my_node my_node_t;

struct my_node {
    char vertex;
    data_t data;
    my_node_t *next;
};

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

typedef struct {
    my_node_t *head;
    my_node_t *foot;
} my_list_t;

typedef struct {
    char next;
    data_t data;
} edge_t;

typedef struct {
    char vertex;
    my_list_t *edges;
    int length;
    int skip;
} vertices_t;

/* ========================================================================== */

/* Function prototypes made by me */
/* ------------------------------ */

int stage0(vertices_t *array, char start_point);


int find_index(vertices_t* array, char vertex, int *found, int leftover);

void get_details(int *min, int *max, int *edges, int *total, int scenic_value);

void put_into_array(vertices_t *array, int *vertices, char start, char pointed,
int scenic_value);

void print_stage0(int vertices, int edges, int min, int max, int total, char
start_point, int odd, int even);

void print_output(my_list_t *new, char start_point, int *output, int final, int sta
ge);

void copy_vertices(vertices_t *new, vertices_t *old, int leftover);

void stage1_loop(vertices_t *array, my_list_t *list, int *leftover, char vertex, ch
ar start, int *output);

int stage1_leftover(vertices_t *new_vertices, my_list_t *new, char start_point, int
 vertices, int *output);

```

```
void add_skip(vertices_t *array, char vertex, int leftover) ;




my_list_t *insert(my_list_t *list, char dest, data_t value);

my_list_t *scan_remove(my_list_t *list, char dest, data_t value);

my_list_t *create_loop(vertices_t *all_vertices, int vertices, char start_point);

void my_free_list(my_list_t *list);

void print_list(my_list_t *list);

my_list_t *scan_insert(my_list_t *list, my_list_t *loop, char start_point, char ver
tex, int skip);

int get_count(my_list_t *list);

my_list_t *copy(my_list_t *list);

my_list_t *my_insert_at_head(my_list_t *list, char dest, data_t value);

my_list_t *my_insert_at_foot(my_list_t *list, char dest, data_t value);

my_list_t *my_make_empty_list(void);

int get_leftover(vertices_t *list, int vertices);

/* -------------------------------------------------------------------------- */

/* Function prototypes made by Alistair */
/* ------------------------------------ */

list_t *make_empty_list(void);

int    is_empty_list(list_t*);

void   free_list(list_t*);

list_t *insert_at_head(list_t*, data_t);

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);

/* ========================================================================== */

/* Main function */
/* ------------- */

int main(int argc, char *argv[]) {
    char start_point;
    int vertices;
    vertices_t all_vertices[MAX_VERTICES];
    my_list_t *base;

    int leftover, final_skip;
    int found, index, i, max, use, output = EMPTY, use_start=NO;
    vertices_t new_vertices[MAX_VERTICES];
    my_list_t *new=NULL, *loop, *temp;
    my_node_t *vert;

    start_point = *argv[SECOND];

    /* STAGE 0 */
    /* ------- */

    vertices = stage0(all_vertices, start_point);
```

```
/* -------------------------------------------------------------------------- */


    /* Create base circuit */

    base = create_loop(all_vertices, vertices, start_point);
    vertices = get_leftover(all_vertices, vertices);
    leftover = vertices;
    new = copy(base);


/* ========================================================================== */

    /* STAGE 1 */
    /* ------- */

    printf("\nStage 1 Output \n---------------\n");
    print_output(new, start_point, &output, NO, STAGE_ONE);

    copy_vertices(new_vertices, all_vertices, leftover);

    /* Check if we need to use the start */

    index = find_index(new_vertices, start_point, &use_start, leftover);


    if (use_start) {

        /* Create a loop using the start */
        stage1_loop(new_vertices, new, &leftover, start_point, start_point, &output
);

    }

    while (leftover) {
        assert(leftover);
        leftover = stage1_leftover(new_vertices, new, start_point, leftover, &outpu
t);
    }

    print_output(new, start_point, &output, YES, STAGE_ONE);


/* ========================================================================== */

    /* STAGE 2 */
    /* ------- */

    printf("\nStage 2 Output \n---------------\n");
    output = EMPTY;

    new = copy(base);
    print_output(new, start_point, &output, NO, STAGE_ONE);

    leftover = get_leftover(all_vertices, vertices);


    /* RECOPY VERTICES INTO NEW VERTICES */
    copy_vertices(new_vertices, all_vertices, leftover);




    while (leftover) {
        temp = copy(new);

        max = EMPTY;

        /* Make all skips zero again */

        for (i = FIRST; i < leftover; i++) {
            new_vertices[i].skip = NO;
            all_vertices[i].skip = NO;
        }

        /* CHECK FOR START */

        for (i = FIRST; i < leftover; i++) {

            if (new_vertices[i].vertex == start_point) {
                loop = create_loop(new_vertices, leftover, start_point);
                temp = scan_insert(temp, loop, start_point, start_point, NO);
                max = get_count(temp);
                copy_vertices(new_vertices, all_vertices, leftover);
                add_skip(all_vertices, start_point, leftover);

                use = start_point;
            }
        }

        vert = new -> head;


        /* CHECK OTHER VERTICES */

        while (vert) {

            leftover=get_leftover(new_vertices,vertices);

            copy_vertices(new_vertices, all_vertices, leftover);

            temp = copy(new);

            /* CHECK IF THE VERTEX HAS LEFTOVER EDGES */

            index = find_index(new_vertices, vert -> vertex, &found, leftover);



            if (found == YES) {

                leftover=get_leftover(new_vertices,vertices);

                loop = create_loop(new_vertices, leftover, vert->vertex);
                copy_vertices(new_vertices, all_vertices, leftover);
                add_skip(all_vertices, vert -> vertex, leftover);
                index = find_index(new_vertices, vert -> vertex, &found, leftover);
                temp = scan_insert(temp, loop, start_point, vert->vertex, new_verti
ces[index].skip);

                /* CHECK IF THE LOOP USING THAT VERTEX HAS A HIGHER SCENIC
                 * VALUE
                 */

                if (get_count(temp) > max) {
                    max = get_count(temp);
                    final_skip = new_vertices[index].skip;
                    use = vert -> vertex;
                }

            }

            vert = vert -> next;

        }

        /* CREATE FINAL LOOP */

        temp = create_loop(new_vertices, leftover, use);
        new = scan_insert(new, temp, start_point, use, final_skip);
        leftover = get_leftover(new_vertices, leftover);
        print_output(new, start_point, &output, NO, STAGE_TWO);
        copy_vertices(all_vertices, new_vertices, leftover);

    }
    print_output(new, start_point, &output, YES, STAGE_TWO);


    return 0;
}

/* ========================================================================== */

/* Helper functions created by me by order of use */
/* ---------------------------------------------- */

/* Stage 0 function */

int stage0(vertices_t *array, char start_point) {
    char start, pointed;
    int scenic_value, min = MAX_VALUE , max = MIN_VALUE, edges= EMPTY;
    int total = EMPTY, vertices = EMPTY, even = EMPTY, odd = EMPTY, i;

    /* Read input */

    while (scanf("%c %c %d\n", &start, &pointed, &scenic_value) == COMPLETE) {

        /* Edit the min, max, number of edges, and the total scenic value */

        get_details(&min, &max, &edges, &total, scenic_value);

        /* Add the edge going from the start to the destination and vice versa
         */

        put_into_array(array, &vertices, start, pointed, scenic_value);
        put_into_array(array, &vertices, pointed, start, scenic_value);

    }

    /* Check for the number of vertices with even/odd degrees */

    for (i = FIRST; i < vertices; i++) {

        if ((array[i].length) % EVEN == ODD) {

            odd += NON_EMPTY;
        }

        else {

            even += NON_EMPTY;
        }
    }

    /* Print out the results */

    print_stage0(vertices, edges, min, max, total, start_point, odd, even);

    return vertices;
}


/* -------------------------------------------------------------------------- */

/* Function to increment the number of edges, total scenic value, and to check
 * if there's a new min or max
 */

void get_details(int *min, int *max, int *edges, int *total, int scenic_value) {


    *edges += NON_EMPTY;
    *total += scenic_value;

    if (scenic_value < *min) {
        *min = scenic_value;
    }

    if (scenic_value > *max) {
        *max = scenic_value;
    }
}


/* -------------------------------------------------------------------------- */

/* Function to add the edge and the vertex to an already existing array of
 * vertices
 */

void put_into_array(vertices_t *array, int *vertices, char start, char pointed,
int scenic_value) {
    int found=NO, index;

    /* Check if the vertex is already within the array of vertices */

    index = find_index(array, start, &found, *vertices);

    if (found) {

        /* Add the edge into the list of edges */

        array[index].edges = insert(array[index].edges, pointed, scenic_value);
        array[index].length += NON_EMPTY;
    }

    else {

        /* Put the details of the vertex at the bottom of the array */

        array[*vertices].vertex = start;

        array[*vertices].edges = my_make_empty_list();
        array[*vertices].edges = my_insert_at_head(array[*vertices].edges,
        pointed, scenic_value);

        array[*vertices].length = NON_EMPTY;
        array[*vertices].skip = EMPTY;

        /* Increment the number of vertices */

        *vertices += NON_EMPTY;

    }
}

/* -------------------------------------------------------------------------- */

/* Return the index within the array of the vertex and make found to be true
 * or retrun the end of the array if not found
 */

int find_index(vertices_t* array, char vertex, int *found, int leftover) {
    int i;

    *found = NO;

    for (i = FIRST; i < leftover; i++) {

        if (array[i].vertex == vertex) {
            *found = YES;
            return i;
        }
    }
    return i;
}


/* -------------------------------------------------------------------------- */

/* Function to put edges into an array of vertices and store the edges from
 * lowest scenic value to highest and then alphabetically if multiple edges
 * with the same scenic value so that when we create the loop, we can just take
 * the top-most one
 */

my_list_t *insert(my_list_t *list, char dest, data_t value) {
    my_node_t *temp, *prev, *curr;

    temp = (my_node_t*)malloc(sizeof(*temp));
    prev = (my_node_t*)malloc(sizeof(*prev));
    curr = (my_node_t*)malloc(sizeof(*curr));

    assert(temp && prev && curr);

    /* Add the details into the temporary node */

    temp->data = value;
    temp->vertex = dest;
    temp->next = NULL;

    /* If the list is empty then just insert the node at the head */

    if(list == NULL) {
        list -> head = temp;
    }

    else {

        /* Have the current node be the head of the list */

        prev = NULL;
        curr = list -> head;

        /* Go through the list until you find the edge (curr) with an equal or
         * higher scenic value
         */

        while (curr && (curr->data < value)) {
            prev = curr;
            curr = curr->next;
        }

        /* If the edges are equal in value then continue until you find the
         * vertex with a higher ASCII value
         */

        if (curr && curr-> data == value) {
            if (curr -> vertex < dest) {
                prev = curr;
                curr = curr->next;
            }
        }

        /* If we reach the end, it means this has the highest value and must be
         * added to the tail
         */

        if (!curr) {
            prev -> next = temp;

        }

        else {

            /* If there is an edge before the current one then have insert the
             * new edge in between those the current and the previous ones or
             * as the new head if not
             */

            if(prev) {
                temp -> next = curr;
                prev -> next = temp;
            }

            else {
                temp -> next = list -> head;
                list -> head = temp;
            }
        }
    }
    return list;
}


/* ------------------------------------------------------------------------ */

/* Printing the big block of text for Stage 0 */

void print_stage0(int vertices, int edges, int min, int max, int total, char
start_point, int odd, int even) {

    printf("\nStage 0 Output \n--------------\n");
    printf("S0: Map is composed of %d vertices and %d edges\n", vertices, edges)
    ;
    printf("S0: Min. edge value: %d\n", min);
    printf("S0: Max. edge value: %d\n", max);
    printf("S0: Total value of edges: %d\n", total);
    printf("S0: Route starts at \"%c\"\n", start_point);
    printf("S0: Number of vertices with odd degree: %d\n", odd);
    printf("S0: Number of vertices with even degree: %d\n", even);

    /* If there are vertices with odd degrees then exit the program but also
     * print that it’s traversable if there’s only 2 vertices
     */

    if (odd != EMPTY) {

        if (odd == TRAVERSABLE) {
            printf("S0: Multigraph is traversable\n");
        }

        exit(EXIT_FAILURE);
    }

    printf("S0: Multigraph is Eulerian\n");

}


/* ------------------------------------------------------------------------ */

/* Function to create a loop from the vertex */

my_list_t *create_loop(vertices_t *list, int vertices, char start_point) {
    my_list_t *loop;
    my_node_t *new_head;
    int i, index, use_start = NO;
    char prev;

    new_head = (my_node_t*)malloc(sizeof(*new_head));

    loop = my_make_empty_list();

    /* Check if the starting point is within the array */

    index = find_index(list, start_point, &use_start, vertices);


    if (use_start) {

        loop = my_insert_at_foot(loop, list[index].edges -> head -> vertex,
        list[index].edges -> head -> data);

        /* Check if that vertex still has leftover edges */

        if (list[index].edges -> head -> next != NULL) {

            /* Remove that edge from the list of edges */

            new_head = list[index].edges -> head -> next;
            free(list[index].edges -> head);
            list[index].edges -> head = new_head;
        }

        else {

            /* Free the whole list */

            my_free_list(list[index].edges);
            for (i=index; i < vertices; i++) {
                list[i] = list[i+NON_EMPTY];
            }

            vertices -= YES;
        }
    }


    /* Remove the edge going the opposite way as well */
    for (i = FIRST; i < vertices; i++) {
        if (list[i].vertex == loop -> foot -> vertex) {
            index = i;
        }
    }

    list[index].edges = scan_remove(list[index].edges,
    start_point, loop -> foot -> data);


    if (list[index].edges -> head == NULL) {
        my_free_list(list[index].edges);
        for (i=index; i < vertices; i++) {
            list[i] = list[i+NON_EMPTY];
        }
        vertices -= YES;
    }

    /* Keep adding until we added an edge that points to the starting point */

    while (loop -> foot -> vertex != start_point) {

        for (i = FIRST; i < vertices; i++) {
            if (list[i].vertex == loop -> foot -> vertex) {
                index = i;
            }
        }
        prev = loop -> foot -> vertex;

        loop = my_insert_at_foot(loop, list[index].edges -> head -> vertex,
        list[index].edges -> head -> data);

        if (list[index].edges -> head -> next != NULL) {
            new_head = list[index].edges -> head -> next;
            free(list[index].edges -> head);
            list[index].edges -> head = new_head;
        }

        else {
            my_free_list(list[index].edges);
            for (i=index; i < vertices; i++) {
                list[i] = list[i+NON_EMPTY];
            }

            vertices -= YES;

        }

        for (i = FIRST; i < vertices; i++) {
            if (list[i].vertex == loop -> foot -> vertex) {
                index = i;
            }
        }

        list[index].edges = scan_remove(list[index].edges, prev,
        loop -> foot -> data);

        if (list[index].edges -> head == NULL) {
            my_free_list(list[index].edges);
            for (i=index; i < vertices; i++) {
                list[i] = list[i+NON_EMPTY];
            }
            vertices -=YES;
        }
    }

    return loop;
}


/* -------------------------------------------------------------------------- */

/* Function to remove the edge from the linked list upon using it */

my_list_t *scan_remove(my_list_t *list, char dest, data_t value) {
    my_node_t *prev, *curr;

    prev = (my_node_t*)malloc(sizeof(*prev));
    curr = (my_node_t*)malloc(sizeof(*curr));

    assert(curr && prev);

    /* Set current as the head of the list */
    prev = NULL;
    curr = list -> head;

    /* Go through the list of edges until you find the exact node we used */

    while(!(curr -> vertex == dest && curr -> data == value)) {

        prev = curr;
        curr = curr -> next;
    }

    /* If the edge we want is the head itself then just make the node the head
     * pointed to be the new head
     */

    if (!(prev)) {

        list -> head = list -> head -> next;
    }

    else {

        prev->next = curr->next;
    }

    return list;
}


/* -------------------------------------------------------------------------- */

/* Function to get the number of vertices leftover */

int get_leftover(vertices_t *list, int vertices) {
    int length = EMPTY, i;

    /* Check if the list of edges for that vertice is null and increment length
     * if not
     */

    for (i=FIRST; i< vertices; i++) {
        if (list[i].edges != NULL) {
            length++;
        }
    }
    return length;
}


/* -------------------------------------------------------------------------- */

/* Function to copy everything in a list to a new list that is independent of
 * the original list
 */

my_list_t *copy(my_list_t *list) {

    my_list_t*  new;
    my_node_t* temp;


    new = my_make_empty_list();
    temp = (my_node_t*)malloc(sizeof(*temp));

    assert(new && temp);

    /* Have temp be the head of the list and while it is not null we insert the
     * data from temp into the foot of the new list then go to the next node
     */

    temp = list -> head;

    while(temp) {
                new = my_insert_at_foot(new, temp-> vertex, temp->data);
        temp = temp->next;

    }

    return new;
}


/* -------------------------------------------------------------------------- */

/* Function to copy a list of vertices to a new one */

void copy_vertices(vertices_t *new, vertices_t *old, int leftover) {
    int i, count = FIRST;

    for (i = FIRST; i < leftover; i++) {
        new[count].vertex = old[i].vertex;
        new[count].edges = copy(old[i].edges);
        new[count].skip = old[i].skip;
        count++;
    }
}


/* -------------------------------------------------------------------------- */

/* Function to create the loop in stage 1 */

void stage1_loop(vertices_t *array, my_list_t *list, int *leftover, char vertex,
char start, int *output) {
    my_list_t *loop = NULL;

    loop = create_loop(array, *leftover, vertex);
    *leftover = get_leftover(array, *leftover);
    list = scan_insert(list, loop, start, vertex, NO);
    print_output(list, start, &*output, NO, STAGE_ONE);

}


/* -------------------------------------------------------------------------- */

/* Function to scan the existing list (circuit) and insert the loop at the
 * right vertex, skipping the first n times the vertex appears in the circuit
 */
```

```
my_list_t *scan_insert(my_list_t *list, my_list_t *loop, char start_point,
char vertex, int skip) {
    my_node_t *prev, *curr, *new_head;
    int occur = EMPTY;

    prev = (my_node_t*)malloc(sizeof(*prev));
    curr = (my_node_t*)malloc(sizeof(*curr));

    assert(prev && curr);

    prev = NULL;
    curr = list -> head;

    /* Check if we need to skip or not */

    if (skip == NO) {

        /* Keep going through the list until we find the vertex we want and
         * then insert the loop at that point
         */

        if (vertex != start_point) {

            prev = curr;
            curr = prev -> next;

            while(prev->vertex != vertex){

                prev = curr;
                curr = curr->next;
            }

            prev -> next = loop -> head;
            loop -> foot -> next = curr;
        }

        /* Add it to the head if it is starting from the starting point */

        else {

            new_head = list -> head;
            loop -> foot -> next = new_head;
            list -> head = loop-> head;

        }
    }

    else {

        /* Add to the occurrence at the beginning if we want to add it to the
         * same vertex as the start since the start is not within the list
         */

        if (start_point == vertex) {
            occur++;
        }

        prev = curr;
        curr = curr -> next;

        /* Keep going through the list until we find the vertex we want,
         * increment the occurrence and then continue until we skipped enough
         */

        while(curr && (prev->vertex != vertex || occur <= skip)) {
            prev = curr;
            curr = curr->next;
            if (prev->vertex == vertex) {
                occur ++;
            }
        }

        /* Add the loop at that point that we stopped */
        prev -> next = loop -> head;
        loop -> foot -> next = curr;

    }
    return list;
}

/* -------------------------------------------------------------------------- */

/* Function to print the output line and the list following the output number
 * requirement
 */

void print_output(my_list_t *new, char start_point, int *output, int final,
int stage) {

    /* Check if it is the final output line */

    if (final) {

        /* Print the output line again if it has not yet been printed */

        if (*output > FIRST_TEN && *output % MULTIPLE_OF_FIVE > NON_EMPTY) {
            printf("S%d: %c", stage, start_point);
            print_list(new);

        }

        /* Print the scenic route then free the list */

        printf("S%d: Scenic route value is %d\n", stage, get_count(new));
        my_free_list(new);
        new = NULL;

    }

    else {

        /* Print the output line following the output number requirements */

        if (*output <= FIRST_TEN || *output % MULTIPLE_OF_FIVE == EMPTY) {
            printf("S%d: %c", stage, start_point);
            print_list(new);
        }

        /* Increment the output number */

        *output+= YES;
    }
}


/* -------------------------------------------------------------------------- */

/* Function to print the list following the edge number requirement */

void print_list(my_list_t *list) {
    my_node_t *curr;
    int count = NON_EMPTY, length = EMPTY;

    curr = (my_node_t*)malloc(sizeof(*curr));

    assert(curr);

    /* Get the number of edges the loop has first */

    curr = list -> head;
    while (curr) {
        curr = curr -> next;
        length++;
    }

    curr = list -> head;
    while (curr) {

        /* If the length is more than 12 edges, then print only the first and
         * last 6 edges
         */

        if (length > PRINT_LIMIT) {

            if (count <= PRINT_LENGTH || count > (length - PRINT_LENGTH)) {
                printf("-%d->%c", curr -> data, curr -> vertex);
            }

            if (count == (length - PRINT_LENGTH)) {
                printf("...%c", curr -> vertex);
            }
        }

        else {

            printf("-%d->%c", curr -> data, curr -> vertex);
        }

        curr = curr -> next;
        count++;
    }

    printf("\n");

}


/* -------------------------------------------------------------------------- */

/* Function to get the scenic route value of the circuit */

int get_count(my_list_t *list) {
    my_node_t *curr;
    int value= EMPTY, count = NON_EMPTY;

    curr = (my_node_t*)malloc(sizeof(*curr));
    assert(curr);

    curr = list -> head;

    while(curr){
        value += count * curr -> data;
        curr = curr->next;
        count++;
    }

    free(curr);

    return value;

}


/* -------------------------------------------------------------------------- */

/* Function that will continuously test the start and every other vertex after
 * if it has any leftover edges until it finds the one with leftover edges
 */

int stage1_leftover(vertices_t *new_vertices, my_list_t *new, char start_point,
int vertices, int *output) {
    int use_start = NO, found = NO, leftover;
    my_node_t *curr;

    curr = (my_node_t*)malloc(sizeof(*curr));
    assert(curr);

    assert(vertices);
    leftover = vertices;

    /* Check if we have to use the start */

    find_index(new_vertices, start_point, &use_start, leftover);

    if (use_start) {

        stage1_loop(new_vertices, new, &leftover, start_point, start_point,
        &*output);
    }

    else {

        /* Iterate through the loop until you find the first vertex with a
         * leftover edge
         */

        curr = new -> head;

        while (!(found)) {

            find_index(new_vertices, curr->vertex, &found, leftover);

            if (found == YES) {

                /* Use this vertex to create the next loop */

                stage1_loop(new_vertices, new, &leftover, curr -> vertex,
                start_point, &*output);
                curr = new -> head;
            }

            else {

                curr = curr -> next;
            }
        }
    }
    return leftover;
}


/* --------------------------------------------------------------------------- */

/* Function to find the vertex within the array and increment the skip */

void add_skip(vertices_t *array, char vertex, int leftover) {
    int i;

    for (i = FIRST; i < leftover; i++) {
        if (array[i].vertex == vertex) {
            array[i].skip += YES;
        }
    }
}


/* =========================================================================== */

/* Helper functions created by Alistair "Algorithms Are Fun" Moffat */
/* ----------------------------------------------------------------- */

/*-------------------------------------------------------------------------------
   Code that follows is written by Alistair Moffat, as an example for the book
   "Programming, Problem Solving, and Abstraction with C", Pearson
   Custom Books, Sydney, Australia, 2002; revised edition 2012,
   ISBN 9781486010974.

   See http://people.eng.unimelb.edu.au/ammoffat/ppsaa/ for further
   information.

   Prepared December 2012 for the Revised Edition.
-------------------------------------------------------------------------------*/
```

```
my_list_t *my_make_empty_list(void) {
    my_list_t *list;

    list = (my_list_t*)malloc(sizeof(*list));
    assert(list);
    list -> head = list -> foot = NULL;

    return list;
}


/* --------------------------------------------------------------------------- */


void my_free_list(my_list_t *list) {
    my_node_t *curr, *prev;

    assert(list);
    curr = list -> head;

    while (curr) {
        prev = curr;
        curr = curr -> next;
        free(prev);
    }

    free(list);
}


/* --------------------------------------------------------------------------- */


my_list_t *my_insert_at_head(my_list_t *list, char dest, data_t value) {
    my_node_t *new;

    new = (my_node_t*)malloc(sizeof(*new));
    assert(list && new);
    new -> data = value;
    new -> vertex = dest;
    new -> next = list -> head;
    list -> head = new;

    if (list -> foot == NULL) {
        /* this is the first insertion into the list */
        list -> foot = new;
    }

    return list;
}

/* --------------------------------------------------------------------------- */


my_list_t *my_insert_at_foot(my_list_t *list, char dest, data_t value) {
    my_node_t *new;

    new = (my_node_t*)malloc(sizeof(*new));
    assert(list && new);
    new -> data = value;
    new -> vertex = dest;
    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;
    }


    /* ========================================================================== */


    /* AlGoRiThMs ArE fUn */
```