

Assignment 5 – Surfin’ U.S.A

Fiona Leung

CSE 13S – Fall 2023

Purpose

The purpose of this program is to calculate the shortest path of a given list of locations and connections with the conditions that the path begins and ends at the same location and all locations are visited only once.

How to Use the Program

To use the program, begin by compiling and running the program using the following commands:

```
$ make clean
$ make
$ make format
$ ./tsp [options]
```

`$ make clean` prepares the compiling process by ensuring that everything will be recompiled.

`$ make` compiles all executables and links headers and `.o/.c` files together. It only recompiles anything that has recently changed since the last `$ make` command.

`$ make format` can be used if you use clang-format and have a configuration file alongside your code. `$ make format` will add a target to run it with.

Note that there are 4 optional flags:

- `-i` needs to be followed by argument. this argument will tell the program where to direct the input. if the `-i` flag is not used, the input will be directed to stdin by default
- `-o` needs to be followed by argument. this argument will tell the program where to direct the output. if the `-o` flag is not used, the output will be directed to stdout by default
- `-d` sets the graph to be directed. if the `-d` flag is not used, the graph is defaulted to undirected
- `-h` prints the program usage message.

The input that `tsp.c` program takes in is in the form:

- an unsigned 32-bit integer that represents how many locations will be listed
- all the locations, each location separated by a newline. the number of locations equates to the first integer that is entered
- an unsigned 32-bit integer that represents how many edges will be listed
- all the edges, each edge separated by a newline. the number of edges equates to the second integer that is entered. additionally, each edge consists of 3 unsigned 32-bit integers delimited by a space character (start_vertex end_vertex weight)

Program Design

Data Structures

In this section, only the graph ADT, stack ADT, and path ADT structs will be explained. For explanations of other data structures, please refer to the **Function Descriptions** section.

Note: The graph, stack, and path ADT structs have been given by Jess Srinivas and Ben Grant

Graph ADT Struct

Graph ADTs consist of two main parts: vertices and edges. Sometimes, weights can be specified for each edge, and in this program, weights are used to represent distance. Graphs can also be directed or undirected. In this program, if a graph is directed, it means that there exists an edge from $A \rightarrow B$, but it is not guaranteed that there is an edge from $B \rightarrow A$. Similarly, if a graph is undirected then for every edge $A \rightarrow B$, there exists an edge from $B \rightarrow A$ of the same weight.

To represent the number of vertices in the graph, there is a vertices member with type unsigned 32-bit integer. To know if the graph is directed or undirected, there is a member named directed (Boolean). To keep track of the vertices that have been visited, an array named visited is a member in the Graph struct. The visited member will be helpful when doing a depth breath search. The next member in the struct is the names member. This is a double-character pointer which essentially means names is an array of "strings". This will be used to store the names of each vertex and will be needed when printing the stack and path. Lastly, there's a double unsigned 32-bit integer pointer named weights. This is an array of arrays. The first level of the weights array are the rows to the adjacency graph while the second level of the weights array are the columns to the adjacency graph.

```
typedef struct graph {
    uint32_t vertices;
    bool directed;
    bool *visited;
    char **names;
    uint32_t **weights;
} Graph;
```

Stack ADT Struct

Stack ADTs follow a first in last out kind of structure and they also cannot be randomly accessed. The capacity member is of type unsigned 32-bit integer and is used to create the stack and allocate enough memory when creating the array of items. The top member is of type unsigned 32-bit integer and allows us to keep track of where the next empty space on the stack. Top will be necessary for the stack ADT when implementing stack operations such as pop, peek, and push. Finally, the items member is an array of unsigned 32-bit integers. Elements within the items array represent the items in the Stack.

```
typedef struct stack {
    uint32_t capacity;
    uint32_t top;
    uint32_t *items;
} Stack;
```

Path ADT Struct Path ADTs are implemented using a stack and a variable to keep track of the current weight of the path. The member vertices is a Stack pointer to a stack. Vertices is how the path ADT will keep track of the order that the path has taken. The total_weight member is an unsigned 32-bit integer and keeps track of the total weight of the path given the vertices in the vertices stack.

```
typedef struct path {
    uint32_t total_weight;
    Stack *vertices;
} Path;
```

Algorithms

Depth First Search (DFS)

Depth first search (DFS) is a search algorithm that uses recursion and recursively searches one path down a tree until a leaf is hit. After that it searches the parent's other child and recursively searches down until a leaf is hit. Since we're looking for the most optimal path (in terms of distance), breadth first search would not be ideal. Here is the pseudocode implementation of the depth first search algorithm in this program.

```
in tsp.c
dfs(uint32_t node, Graph *g, Path *p, Path *best)
    visit node
    add node to path
    loop through all the possible vertices that we can travel to starting from node
        to determine whether or not we can go to node2, see if there exists an edge
        (weight is not 0) between node and node2. if weight between node and node2 is
        not 0 and node2 has not been visited yet
            run dfs now with node2 instead of node. this will allow us to continue
            searching down the search tree and branch off to different paths

    let's check if the path we found is valid, that is, every location has been
    visited and we end at the starting node
    if every location has been visited
        if the node we're at right now has a route back to the starting node
            add the starting node
            let's check if this is our best path yet
            if the current best path has a distance of 0, this means that this is
            the first time that we have a complete path
                copy the current path (p) onto the best path
                remove the starting node
            else, if the total distance of the current path (p) is less than the
            total distance of the current best path
                copy the current path (p) onto the best path
                remove the starting node
            else
                just remove the starting node. this path is not the best one we've
                found so far, so we can just trash this path and try looking for
                another one

    unvisit the node
    remove node from path
```

Pseudocode

```
in tsp.c
main()
    check for command line options
    in the case that -h is activated
```

```

    print the help message, then exit

in the case that -i is activated
    set the Boolean dash_in to true
    set the in "string" to the argument provided after -i

in the case that -o is activated
    set the Boolean dash_out to true
    set the out "string" to the argument provided after -o

in the case that -d is activated
    set the Boolean directed to true

if the option is not recognized as any of them or an argument is not provided for -i
or -o, default to just exiting the program

use fscanf to read an unsigned integer from the input file and put the value read
into the address that is pointing to num_vertices
if there is more than one number read from fscanf or value(s) read is not an
unsigned integer
    print an error message and exit the program

create a graph struct, let's name it adj_matrix. the arguments to create the graph
are num_vertices (unsigned 32-bit integer) and directed (Boolean)

loop from 0 to the num_vertices, using i to keep track of the current iteration
    use fgets to read a line from the input file and put the value read into the
    address that is pointing to the location variable
    if fgets() returns a NULL pointer, free the graph, print the error message, and
    exit the program
    otherwise, if fgets() did not return a NULL pointer, remove the extra
    new line character from location (fgets() returns an extra new line
    character by default). also, add this vertex (location) to the graph

use fscanf to read an unsigned integer from the input file and put the value read
into the address that is pointing to num_edges
if there is more than one number read from fscanf or value(s) read is not an
unsigned integer
    print an error message and exit the program

loop from 0 to the num_edges, using i to keep track of the current iteration
    use fscanf() to read 3 unsigned integers from the infile
    if fscanf() does not find three unsigned integers delimited by a space character
    or at least of the characters are not an unsigned integer, free the graph, print
    the error message, and exit the program
    otherwise, if fscanf() was able to read three unsigned integers, add this
    vertex (location) to the graph

close the infile if it isn't stdin

create two Paths, current_path and best_path, and set both their capacities to
num_vertices + 1
run dfs with the START_VERTEX, adj_matrix Graph, current_path Path, and best_path
Path to find the best path. the best path will be stored in best_path after

```

```

running dfs()

if the total_weight of best_path is not 0
    print the path and total_weight to the outfile
otherwise
    print that alissa is lost to the outfile

close the outfile if it isn't stdout

free the two Paths and the Graph
return 0 to indicate program success

```

Function Descriptions

graph.c functions

Graph *graph_create(uint32_t vertices, bool directed)

- Inputs: takes in an unsigned 32-bit integer (vertices) and a Boolean (directed)
- Outputs: returns a pointer to the created Graph
- Purpose: creates a Graph so we can use a graph ADT to store information (location names, nodes, connections between nodes). the graph is needed for the implementation of the path ADT
- Pseudocode (actual implementation given by Jess Srinivas and Ben Grant):

```

Graph *graph_create(uint32_t vertices, bool directed)
    allocate memory using calloc(1, sizeof(Graph)) for the Graph pointer (let's call
    it g)
    assign the Graph g's vertices unsigned integer to the vertices parameter that was
    passed in, this will be how we keep track of how many vertices are in the Graph
    assign the Graph g's directed Boolean to the directed parameter that was passed in,
    this is how we will be able to determine if Graph g is directed (true) or
    undirected (false)

    allocate memory using calloc(vertices, sizeof(bool)) so we can make an array that's
    initialized to all false for the Graph g's visited array. we want to ensure that
    all places are unvisited at the beginning, so we cannot allocate using
    malloc in this case

    similarly, allocate memory using calloc(vertices, sizeof(char*)) for the Graph g's
    name array. this is where the names of the locations will be stored as
    strings (char *'s)

    then, allocate memory using calloc(vertices, sizeof(g->weights[0])) for the
    Graph g's weights. this will be the rows to the Graph's adjacency matrix.
    lastly, for each row in the adjacency matrix (number of vertices), create another
    array within that row using calloc(vertices, g->weights[0][0]). this will be
    our columns in the Graph's adjacency matrix.

    return the pointer to the newly created Graph (g)

```

void graph_free(Graph **gp)

- Inputs: a pointer to (a pointer to a Graph), this is necessary since we want to point Graph pointer to NULL, so we can't reference it again after it's been freed
- Outputs: `graph_free()` does not return or print anything
- Purpose: the only purpose of `graph_free()` is to free every single memory allocation so we can ensure that there are no memory leaks
- Pseudocode:

```
void graph_free(Graph **gp)
    check if the pointer to the Graph pointer is NULL and also check if the Graph
    pointer is NULL. we're doing this so that way we don't reference or free
    any NULL pointers
        if the Graph has a visited array (memory was allocated), then free that visited
        array and point the visited pointer to NULL, so we can no longer use it

        if the Graph has a name array (memory was allocated)
            free every name in the names array and point it to NULL
            free the names array pointer
            point the names array pointer to NULL

        if the Graph has a weights array (memory was allocated)
            for every row in the weights array, free every column associated with it
            free the weights array pointer
            point the weights array pointer to NULL

        free the Graph pointer
    if the pointer to the Graph pointer is not NULL then point the Graph pointer to
    NULL so we can no longer access memory associated with this Graph
```

`uint32_t graph_vertices(const Graph *g)`

- Inputs: a pointer to a Graph (`g`)
- Outputs: returns the number of vertices in the Graph `g` (unsigned 32-bit integer)
- Purpose: `graph_vertices()` is used to retrieve the number of vertices in the Graph `g`
- Pseudocode:

`void graph_add_vertex(Graph *g, const char *name, uint32_t v)`

- Inputs: a pointer to a Graph (`g`), `string/char*` (`name`), unsigned 32-bit integer (`v`)
- Outputs: returns and prints nothing, however it does modify the Graph's names array
- Purpose: copies `name` and adds it into the `v`-th index in the Graph's name array. this is helpful for when we need to print the path and need to know the names of the locations
- Pseudocode (actual implementation given by Jess Srinivas and Ben Grant):

```
void graph_add_vertex(Graph *g, const char *name, uint32_t v)
    if there's an item at the v-th index of the Graph's name array, then free it
    set the v-th index of the Graph's name array to a copy of the name (string/char*
    parameter that was passed in)
```

`const char* graph_get_vertex_name(const Graph *g, uint32_t v)`

- Inputs: a pointer to a Graph (g), unsigned 32-bit integer (v)
- Outputs: returns the name of the vertex at index v of the Graph g's name array
- Purpose: used for retrieving the name of the vertex at index v. this could be used when printing the path

char **graph_get_names(const Graph *g)

- Inputs: a pointer to a Graph (g)
- Outputs: returns an array of character arrays (strings). this returned array is the Graph g's array of location names
- Purpose: used for retrieving Graph g's array of names. this could be used when printing the path

void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight)

- Inputs: a pointer to a Graph (g), 3 unsigned 32-bit integers (start, end, and weight)
- Outputs: doesn't return or print anything, however, graph_add_edge modifies Graph g's weights array and adds an edge between start and end and sets the weight to the unsigned integer weight. also, if Graph g is an undirected graph, it will add an edge between the end and start with the same unsigned integer weight.
- Purpose: this is necessary for storing and keeping track of the edges and weights between vertices. graph_add_edge() is used in tsp() when parsing through input.
- Pseudocode:

```
void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight)
    set the edge weight between the start and end vertices to weight
    if Graph g is undirected then also set edge weight between the end and
    start vertices to weight
```

uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end)

- Inputs: a pointer to a Graph (g), 2 unsigned 32-bit integers (start and end)
- Outputs: returns the weight of the edge between the start vertex and end vertex. this return value is an unsigned 32-bit integer.
- Purpose: this function is necessary for depth first search because we want to increase the total_weight of the path by the weight between two vertices.

void graph_visit_vertex(Graph *g, uint32_t v)

- Inputs: a pointer to a Graph (g), unsigned 32-bit integer (v)
- Outputs: graph_visit_vertex() doesn't return or print anything, however, it modifies Graph g's visited array
- Purpose: graph_visit_vertex() marks the v-th index in Graph g's visited array as true. this is helpful because it will help us keep track of where we've already visited when doing depth first search.

void graph_unvisit_vertex(Graph *g, uint32_t v)

- Inputs: a pointer to a Graph (g), unsigned 32-bit integer (v)
- Outputs: graph_unvisit_vertex() doesn't return or print anything, however, it modifies Graph g's visited array

- Purpose: `graph_unvisit_vertex()` marks the `v`-th index in Graph `g`'s visited array as false. this is helpful because it will help us keep track of where we've already visited when doing depth first search.

`bool graph_visited(Graph *g, uint32_t v)`

- Inputs: a pointer to a Graph (`g`), unsigned 32-bit integer (`v`)
- Outputs: `graph_visited()` returns true or false depending on whether or not the vertex `v` has been visited. if the `v`-th index in Graph `g`'s visited array is true, then that means the `v`-th location has been visited and thus this function will return true. otherwise, it will return false signifying that the `v`-th location has not yet been visited.
- Purpose: `graph_visited()` is helpful because it will allow us to determine if a vertex has already been visited when doing depth first search.

`void graph_print(const Graph *g)`

- Inputs: a pointer to a Graph (`g`)
- Outputs: prints the graph to stdout.
- Purpose: `graph_print()` is used to help visualize the graph
- Pseudocode:

```
void graph_print(const Graph *g)
    loop to print the top row of indices from 0 to the number of vertices.
    this will help visualize which vertex we are going to
    loop to print each row of the weights array
    loop to print each weight
```

the output of `graph_print()` will look like:

going to city #	0	1	2	3	4	5	6	7	8	9
starting at city #0	0	48	0	0	0	0	0	0	0	32
starting at city #1	48	0	12	20	0	0	0	0	0	0
starting at city #2	0	12	0	17	0	0	0	0	17	31
starting at city #3	0	20	17	0	10	0	0	0	0	0
starting at city #4	0	0	0	10	0	12	0	0	0	0
starting at city #5	0	0	0	0	12	0	16	0	15	0
starting at city #6	0	0	0	0	0	16	0	17	0	0
starting at city #7	0	0	0	0	0	0	17	0	11	32
starting at city #8	0	0	17	0	0	15	0	11	0	28
starting at city #9	32	0	31	0	0	0	0	32	28	0

`stack.c` functions

`Stack *stack_create(uint32_t capacity)`

- Inputs: an unsigned 32-bit integer (`capacity`)
- Outputs: returns a pointer to a Stack (`s`)
- Purpose: creates a stack ADT. the path will use the stack to store the current path.
- Pseudocode (actual implementation provided by Jess Srinivas and Ben Grant):

```

Stack *stack_create(uint32_t capacity)
    allocate memory using malloc(sizeof(Stack)) for the Stack pointer (let's call
    it s) and to ensure that the pointer is pointing something that is of the Stack type,
    type cast the pointer using (Stack *).

    set the Stack s's capacity to the unsigned 32-bit integer capacity that was passed
    through
    set the Stack s's top to 0 since we're creating an empty stack
    to allocate enough memory to store the vertices in the path, use malloc(capacity,
    sizeof(uint32_t)). this will allocate enough memory for capacity amount of items
    with a size of sizeof(uint32_t) each.

    then return the Stack pointer (s)

```

```
void stack_free(Stack **sp)
```

- Inputs: a pointer to (a pointer to a Stack), this is necessary since we want to point Stack pointer to NULL, so we can't reference it again after it's been freed
- Outputs: stack_free() does not return or print anything
- Purpose: the only purpose of stack_free() is to free every single memory allocation so we can ensure that there are no memory leaks
- Pseudocode (actual implementation provided by Jess Srinivas and Ben Grant):

```

void stack_free(Stack **sp)
    check if the pointer to the Stack pointer equals NULL or if the Stack pointer equals
    NULL. we want to do this to ensure that we don't try to free a NULL pointer later.
    if both aren't NULL pointers, then:
        if Stack s's item array pointer does not equal NULL
            free Stack s's item array
            point Stack s's item array pointer to NULL
        free the Stack pointer

    if the pointer to the Stack pointer does not equal NULL, then
        point the Stack pointer to NULL <-- so that way we can no longer accidentally
        use the freed Stack pointer

```

```
bool stack_push(Stack *s, uint32_t val)
```

- Inputs: a pointer to a Stack (s), an unsigned 32-bit integer (val)
- Outputs: returns Boolean value depending on whether or not the push operation was successful (true) or not (false)
- Purpose: stack_push() is used when we want to add a vertex(val) to the path in the path ADT
- Pseudocode:

```

bool stack_push(Stack *s, uint32_t val)
    if the stack is full then return false, a push operation cannot be done if the
    stack is at its capacity
    using s->top to find the next empty index, set the next empty index of the
    Stack s's items array to val
    increment to the Stack s's top by one
    return true to indicate that the push was successful

```

bool stack_pop(Stack *s, uint32_t *val)

- Inputs: a pointer to a Stack (s), a pointer to an unsigned 32-bit integer (val)
- Outputs: returns Boolean value depending on whether or not the pop operation was successful (true) or not (false). also puts a copy of the item at the top of Stack into the address that val is pointing to.
- Purpose: stack_pop() is used when we want to remove a vertex from the path in the path ADT. stack_pop() will also change the value at the address that val is pointing to to the item that was popped.
- Pseudocode:

```
bool stack_pop(Stack *s, uint32_t *val)
    if the stack is empty then return false, a pop operation cannot be done if the
    stack is empty
    set the value that the val pointer is pointing to to the item at the top of the
    stack (this would be top - 1)
    decrement to the Stack s's top by one
    return true to indicate that the pop was successful
```

bool stack_peek(Stack *s, uint32_t *val)

- Inputs: a pointer to a Stack (s), a pointer to an unsigned 32-bit integer (val)
- Outputs: returns Boolean value depending on whether or not the peek operation was successful (true) or not (false). also puts a copy of the item at the top of Stack into the address that val is pointing to.
- Purpose: stack_peek() is used when we want to look at the vertex at the top of the Stack from the path in the path ADT. stack_peek() will also change the value at the address that val is pointing to to the item that was peeked at.
- Pseudocode:

```
bool stack_peek(Stack *s, uint32_t *val)
    if the stack is empty then return false, a pop operation cannot be done if the stack
    is empty
    set the value that the val pointer is pointing to to the item at the top of the
    stack (this would be top - 1)
    return true to indicate that the pop was successful
```

bool stack_empty(const Stack *s)

- Inputs: a pointer to a Stack (s). this pointer is initialized as a constant because we don't want to accidentally modify the pointer.
- Outputs: returns Boolean value depending on whether or not the stack is empty (true is empty and false otherwise).
- Purpose: stack_empty() is used to determine if a pop or peek operation is possible.

bool stack_full(const Stack *s)

- Inputs: a pointer to a Stack (s). this pointer is initialized as a constant because we don't want to accidentally modify the pointer.
- Outputs: returns Boolean value depending on whether or not the stack is full (true is full and false otherwise).

-
- Purpose: `stack_full()` is used to determine if a push operation is possible.

`uint32_t stack_size(const Stack *s)`

- Inputs: a pointer to a Stack (`s`). this pointer is initialized as a constant because we don't want to accidentally modify the pointer.
- Outputs: returns the size of the stack which is equivalent to the Stack's top member (an unsigned 32-bit integer).
- Purpose: `stack_size()` will be helpful when trying to find out how many vertices are in a path.

`void stack_copy(Stack *dst, const Stack *src)`

- Inputs: two pointers to Stacks. one is a pointer to a Stack called `dst` and the other is a pointer to a Stack (`s`). this second pointer is initialized as a constant because we don't want to accidentally modify the pointer.
- Outputs: doesn't return or print anything, but modifies the Stack that the `dst` pointer is pointing to.
- Purpose: `stack_copy()` will be used when copying the contents of one path to another.
- Pseudocode:

```
void stack_copy(Stack *dst, const Stack *src)
    set the top of the dst Stack to 0 to prepare for overwriting
    loop through src Stack's items and push the i-th item in the dst Stack
```

`void stack_print(const Stack *s, FILE *outfile, char *cities[])`

- Inputs: a pointer to a Stack (`s`) (the pointer is initialized as a constant because we don't want to accidentally modify the pointer), file to print the Stack to (`outfile`), a double pointer that's essentially an array of "strings" (`cities`)
- Outputs: prints the Stack to `outfile`. `stack_print()` prints the vertices according to their location names.
- Purpose: helpful for visualizing the Stack
- Pseudocode (actual implementation given by Jess Srinivas and Ben Grant):

```
void stack_print(const Stack *s, FILE *outfile, char *cities[])
    loop the Stack's item array from 0 to the top of the Stack
        print the (current vertex's index)-th item from cities array to outfile
```

`path.c` functions

`Path *path_create(uint32_t capacity)`

- Inputs: an unsigned 32-bit integer that represents the capacity of the path (`capacity`)
- Outputs: returns a pointer to a new, empty Path
- Purpose: creates a Path that will be used in the depth first search
- Pseudocode:

```
Path *path_create(uint32_t capacity)
    allocate memory for the Path pointer using calloc (let's call this new Path p)
    set the Path p's vertices member to a new Stack with a Stack capacity of capacity
    set the Path p's total_weight member to 0 since the Path is empty upon creation
    return the Path pointer (p)
```

void path_free(Path **pp)

- Inputs: a pointer to a Path pointer. this is necessary because we want to point the Path pointer to NULL later.
- Outputs: doesn't return or print anything.
- Purpose: frees all memory allocated for the Path to ensure that there are no memory leaks.
- Pseudocode:

```
void path_free(Path **pp)
    check if the pointer to the Path pointer equals NULL or if the Path pointer equals
    NULL. we want to do this to ensure that we don't try to free a NULL pointer later.
    if both aren't NULL pointers, then:
        if Path pp's vertices Stack does not equal NULL
            use stack_free() to free the Path's vertices Stack
        free the Path pointer

    if the pointer to the Path pointer does not equal NULL, then
        point the Path pointer to NULL <-- so that way we can no longer accidentally
        use the freed Path pointer
```

uint32_t path_vertices(const Path *p)

- Inputs: a Path pointer (p). the pointer is initialized as a constant because we don't want to accidentally modify the pointer.
- Outputs: returns an unsigned 32-bit integer that represents the number of vertices in the path
- Purpose: used to find out how many items are in the Stack (p's vertices array). helpful for printing the Path list.

uint32_t path_distance(const Path *p)

- Inputs: a Path pointer (p). the pointer is initialized as a constant because we don't want to accidentally modify the pointer.
- Outputs: returns an unsigned 32-bit integer that represents the total distance traveled in the Path.
- Purpose: used to find the total distance traveled and is helpful in depth first search when we want to determine if this Path is our best Path yet.

void path_add(Path *p, uint32_t val, const Graph *g)

- Inputs: a Path pointer (p), an unsigned 32-bit integer (val), a pointer to a Graph (g)
- Outputs: doesn't return or print anything, however, path_add() modifies the Path p
- Purpose: path_add() is helpful in depth first search when we want to add a vertex to the Path and calculate the new total distance
- Pseudocode:

```
void path_add(Path *p, uint32_t val, const Graph *g)
    check if Path is empty first using stack_empty(), if it is not empty,
        peek at the item at the top of the Path and store that value in a
        variable (start)
        increase the Path's total weight by the weight between the start and val
    check if Path is full, if it is not full,
        push the val into the Path
```

uint32_t path_remove(Path *p, const Graph *g)

- Inputs: a Path pointer (p), a pointer to a Graph (g)
- Outputs: doesn't return or print anything, however, path_add() modifies the Path p
- Purpose: path_add() is helpful in depth first search when we want to remove a vertex from the Path and calculate the new total distance
- Pseudocode:

```
uint32_t path_remove(Path *p, const Graph *g)
    initialize two unsigned 32-bit integers (first_popped and second_popped)
    if there are at least two vertices in the Path
        pop a vertex and send a copy to first_popped
        pop a vertex and send a copy to second_popped
        decrement the Path's total weight by the weight between the first_popped vertex
        and the second_popped vertex
    else, if there's only one vertex in the Path
        pop a vertex and send a copy to the first_popped
        then set the Path's total weight to 0
    return the first_popped vertex
```

void path_copy(Path *dst, const Path *src)

- Inputs: two pointers to Paths. the first Path pointer (dst) is where we want to copy the second Path pointer's (src) contents to. the Path src pointer is initialized as a constant because we don't want to accidentally modify the pointer.
- Outputs: doesn't return or print anything, however, path_copy() modifies the Path dst
- Purpose: path_copy() is helpful in depth first search because it allows us to copy the current Path (src) to the best Path (dst) if a better Path has been found
- Pseudocode:

```
void path_copy(Path *dst, const Path *src)
    copies Path src's vertices array into Path dst's vertices array using stack_copy()
    sets the value of Path dst's total weight to Path src's total weight
```

void path_print(const Path *p, FILE *outfile, const Graph *g)

- Inputs: a Path pointer (p), file to print output to (outfile), a Graph pointer (g). the Path and Graph pointers are initialized as constants because we don't want to accidentally modify the pointers
- Outputs: prints the Path by printing the names of the vertices. path_print() prints to the specified output file (outfile)
- Purpose: path_print() is used for printing best Path after running depth first search in tsp.c
- Pseudocode:

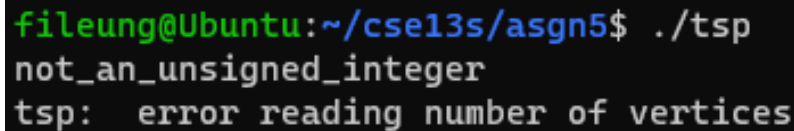
```
void path_print(const Path *p, FILE *outfile, const Graph *g)
    initialize two unsigned 32-bit integers (top_index and size)
    create a new Stack pointer (backwards)
    ! the start location is at the bottom of the Stack, so to print the first item
    first, we can reverse the Stack and then pop and push items one at a time.
    loop from 0 to the size of the Stack
```

```
pop a vertex from the Stack and then push it into the backwards Stack.
when popping items from the Stack, send a copy of the item to top_index,
so we can push it into the backwards Stack.
loop from 0 to the size of the Stack
pop a vertex from the backwards Stack and send a copy of the item to top_index
print to the outfile the (top_index)-th location of location array (accessed via
graph_get_names())
free the backwards Stack
```

Error Handling

Potential errors and solutions:

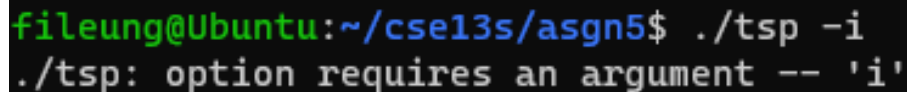
- Invalid inputs: If the user enters any invalid inputs such as an invalid number of edges, number of vertices, EOF, vertex, edge, the program prints an error message to stderr, freeing any initialized ADTs, and exits the program.



```
fileung@Ubuntu:~/cse13s/asgn5$ ./tsp
not_an_unsigned_integer
tsp: error reading number of vertices
```

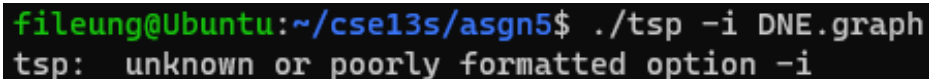
Figure 1: Error message when unable to read number of vertices

- Unrecognized file name or no file name specified: When the user uses the -i or -o command line options, they are required to follow the option with a valid file name. In the case that this option argument is not given or does not exist, the program prints an error message and exits the program.



```
fileung@Ubuntu:~/cse13s/asgn5$ ./tsp -i
./tsp: option requires an argument -- 'i'
```

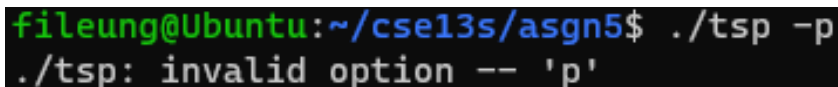
Figure 2: Error message when no argument is specified for -i option



```
fileung@Ubuntu:~/cse13s/asgn5$ ./tsp -i DNE.graph
tsp: unknown or poorly formatted option -i
```

Figure 3: Error message when given file argument cannot be found

- Unrecognized option: if the option isn't one of the valid options, the program will let the user know that the option they entered is invalid and exit the program.



```
fileung@Ubuntu:~/cse13s/asgn5$ ./tsp -p
./tsp: invalid option -- 'p'
```

Figure 4: Error message when command line option is not recognized

Testing

To test my program, I've created a tests.c file to test my ADTs and ensure that they work. Within the tests.c file I've used a combination of print statements and assert statements to help me visualize the ADT (e.g. graph_print()) and ensure that the functions return the expected return value.

```
beginning of graph tests
number of vertices: 5
test_graph city #0: san francisco
test_graph city #1: new york
test_graph city #2: shanghai
test_graph city #3: seoul
test_graph city #4: tokyo
testing if edges are added in both directions for an undirected graph
going to city #      0      1      2      3      4

starting at city #0   0      0      0      0      0
starting at city #1   0      0 888      0      0
starting at city #2   0 888      0      0      0
starting at city #3   0      0      0      0      0
starting at city #4   0      0      0      0      0

assigning random weights (1-10) to adjacency matrix
going to city #      0      1      2      3      4

starting at city #0   0      7      9      8      3
starting at city #1   7      7      9      2      2
starting at city #2   9      9      2      1      5
starting at city #3   8      2      1      4      6
starting at city #4   3      2      5      6      9
graph_get_weight of start = 2 and end = 4 --> 5

city #3 visited: 0
visiting city #3
city #3 visited: 1
unvisiting city #3
city #3 visited: 0

beginning of path tests
path_distance from city 1, 2, and 3: 10
printing test_path
san francisco
new york
printing destination_path
san francisco
new york

test_path before vertex removal
san francisco
new york
after removing the most recently added vertex (city 3), the new distance/total_weight is: 9
test_path after 1 vertex removal
```

Figure 5: Graph and path ADT tests in tests.c

I've also ran Valgrind to ensure that my program has no memory leaks and scan-build to ensure that my program is bug-free when compiling it using my Makefile.

When running Valgrind, I tested that there were no memory leaks when the program ran successfully without errors and also when the program encountered errors to make sure that my program was memory leak-free in different scenarios. Here are the outcomes of both Valgrind tests. You can see that in both scenarios, no memory leaks are possible.

Additionally, when running Valgrind for the first time with my tsp executable, I did not pass the Valgrind test on the pipeline despite Valgrind telling me that there were no memory leaks possible. I learned that I need to test all/as many possible combinations of inputs. This includes different combinations of command line inputs and error-inducing inputs.

```

fileung@Ubuntu:~/cse13s/asgn5$ valgrind ./tsp -i maps/clique12.graph
==2177== Memcheck, a memory error detector
==2177== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2177== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2177== Command: ./tsp -i maps/clique12.graph
==2177==
Alissa starts at:
v1
v7
v2
v8
v3
v9
v4
v10
v5
v11
v6
v12
v1
Total Distance: 362
==2177==
==2177== HEAP SUMMARY:
==2177==   in use at exit: 0 bytes in 0 blocks
==2177==   total heap usage: 39 allocs, 39 frees, 6,679 bytes allocated
==2177==
==2177== All heap blocks were freed -- no leaks are possible
==2177==
==2177== For lists of detected and suppressed errors, rerun with: -s
==2177== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
fileung@Ubuntu:~/cse13s/asgn5$ |

```

Figure 6: Running Valgrind when the program encounters no errors

```

fileung@Ubuntu:~/cse13s/asgn5$ valgrind ./tsp
==2009== Memcheck, a memory error detector
==2009== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2009== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2009== Command: ./tsp
==2009==
2
hello
world
hahahhahahahhahahhahahhahahhahah
tsp: error reading number of edges
==2009==
==2009== HEAP SUMMARY:
==2009==   in use at exit: 0 bytes in 0 blocks
==2009==   total heap usage: 9 allocs, 9 frees, 1,118 bytes allocated
==2009==
==2009== All heap blocks were freed -- no leaks are possible
==2009==
==2009== For lists of detected and suppressed errors, rerun with: -s
==2009== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Figure 7: Running Valgrind when the program encounters an error (invalid input error)


```
fileung@Ubuntu:~/cse13s/asgn5$ scan-build make
scan-build: Using '/usr/lib/llvm-14/bin/clang' for static analysis
/usr/share/clang/scan-build-14/bin/../libexec/clang-analyzer -Werror -Wall -Wextra -Wconversion -Wdouble-promotion -Wstrict-prototypes -pedantic -gdwarf-4 -g -c tsp.c
/usr/share/clang/scan-build-14/bin/../libexec/clang-analyzer -Werror -Wall -Wextra -Wconversion -Wdouble-promotion -Wstrict-prototypes -pedantic -gdwarf-4 -g -c graph.c
/usr/share/clang/scan-build-14/bin/../libexec/clang-analyzer -Werror -Wall -Wextra -Wconversion -Wdouble-promotion -Wstrict-prototypes -pedantic -gdwarf-4 -g -c path.c
/usr/share/clang/scan-build-14/bin/../libexec/clang-analyzer -Werror -Wall -Wextra -Wconversion -Wdouble-promotion -Wstrict-prototypes -pedantic -gdwarf-4 -g -c stack.c
scan-build: Analysis run complete.
scan-build: Removing directory '/tmp/scan-build-2023-11-11-202436-1939-1' because it contains no reports.
scan-build: No bugs found.
```

Figure 8: Results of running scan-build

Here are the results of running scan-build. As you can see there are no issues with compilation and no bugs were found

Results

Here are some of the results from running my program:

In this first image of my program output, notice that a path is printed along with the total distance that it would take to travel the path. The path that is output is the most optimal path (in terms of distance) given the vertices/locations and edges/connections. Also, in every path that is printed by my program, every location is visited once (excluding the start/end location) and Alissa starts and ends her journey at the same place.

```
fileung@Ubuntu:~/cse13s/asgn5$ ./tsp -i maps/surfin.graph
Alissa starts at:
Santa Cruz
Ventura County Line
Pacific Palisades
Manhattan
Redondo Beach, L.A.
Haggerty's
Sunset
Doheny
Trestles
San Onofre
Swami's
Del Mar
La Jolla
Santa Cruz
Total Distance: 965
```

Figure 9: Output of running tsp with surfin.graph when graph is undirected

In the second image of my program output, notice that there is no path printed. Instead, my tsp.c program tells the user that Alissa is lost. This is because there was no path found that visits every location once and begins and ends at the starting location.

```
fileung@Ubuntu:~/cse13s/asgn5$ ./tsp -di maps/lost.graph
No path found! Alissa is lost!
```

Figure 10: Output of running tsp with lost.graph when graph is directed

When testing my tsp executable on the different .graph files provided, I noticed one thing: the runtime of different files varied drastically. For an input of fewer edges such as basic.graph, bayarea.graph, or

surfin.graph, my program was able to print an output also instantly. However, when running .graph files with a large amount of edges such as clique11.graph or clique12.graph, it took about 25 minutes to print an output. I think that the reason why the runtimes of these .graph files differed so drastically is that as the number of edges increased, there were more possible paths to search.

References