Assignment 5 – Surfin' U.S.A.

Sebastian Avila

CSE 13S – Fall 2023

Purpose

The purpose of this program is implement a solution to the traveling salesman problem. This means the program will attempt to find a route that visits each city using the smallest amount of gas for a given set of cities in a graph.

How to Use the Program

To use this program you will need to have several files in the same directory. The files are `tsp.c`, `graph.c`, `graph.h`, `stack.c`, `stack.h`, `path.c`, `path.h`, `vertices.h`, and `Makefile`. With these files in the same directory, use command `make clean` to remove any preexisting object files. Then, use command `make format` to format all the header and source files. Then the same directory use command `make` to compile the program. Then run the program using `./tsp` with any desired command line arguments. The possible command line arguments and their meanings are as follows:

- `-i` : Sets the input file. It requires a file name as an argument. The default file to read from is `stdin`.
- `-o` : Sets the output file. It requires a file name as an argument. The default out file is `stdout`.
- `-d` : Tells the program to treat all graphs as directed. The default rule is to assume an undirected graph.
- `-h` : Prints a help message to `stdio`.

The program uses several optional compiler flags:

- `-Wall`: This flag enables all warning messages.
- `-Werror`: This flag turns all warnings into errors.
- `-Wextra`: This flag enables extra warning flags that are not enabled by `-Wall`.
- `-Wstrict-prototypes`: This flag warns if a function is declared or defined without specifying the argument types.
- `-pedantic`: This flag issues all the warnings demanded by strict ISO C and ISO C++.
- `-lm`: This flag links the math.h library. This allows the program to access and use the functions from the math.h library.

Program Design

The program begins by creating boolean variables to represent whether the input file command line argument `-i` or the output file command line argument `-o` has been entered and for whether the graph is directed or not. It also initializes two character strings to store the input and output file names. It then checks the command line arguments using the `getopt` function. The program silences `getopt`'s automatic error messages by setting `opterr` to 0. Any error messages printed by the program are its own. If an incorrect argument is received or a required argument is missing, it prints an error message, the usage message, and then terminates the program. If `-i` or `-o` are received, the corresponding file name variable is set to the argument received after the `-i` or `-o` and the boolean variables representing whether these arguments are given is set to true.

It then designates a file pointer to `stdin`. It then checks the boolean variable for the input file command line argument; if it is true, the file name set earlier is opened and the file pointer is set to the file. The same happens with the out file and its boolean variable. Its default value it `stdout`. If an error occurs in opening either the input or output file, an error message is printed and the program terminates.

The program then reads an integer from the input file. This integer is used to create a graph data structure with a number of vertices equal to the integer received. If there is an error reading the integer, the program prints an error message and terminates the program. It then creates a character string to store the names of the vertices that will be parsed from the input file. The program loops for an amount equal to the number of vertices. Each loop, the program takes in a line from the input file, stores it in the name variable, removes the newline character from the end of the string, and then adds the name to the corresponding vertex in the graph using a function. It then reads another integer from the input file. This integer is used to determine how many edges will be given by the input file. If there is an error reading the integer, an error message is printed and the program terminates, if not then the program loops that many times. In each loop, the program reads in three variables: the start of the edge, the end of the edge, and its weight. The edge is added to the graph using a function. It then creates a two path data structures: one will be the current path being made and the other will store the shortest path found. It then sends these paths, the graph, and the start vertex to a depth first search function in order to find the shortest path.

After the function, the program checks whether the shortest path's distance is equal to 0 which is its default value meaning it was never overwritten. It finds the shortest path's distance using a function. If it is, then the program prints a message to the output file saying no path was found and terminates the program. If the shortest path's distance is not zero, the program prints the path found message to the output file. This message includes the names of the vertices along the path which are printed by a function and the path's distance which is found by a function. The program then frees the two paths and the graph. It also closes the input and output files. Then it terminates.

Data Structures

Graph

This program creates a type `Graph`. It is a struct with a 32 bit integer variable, a boolean variable, a pointer to a boolean array, a double pointer to an array of strings, and a double pointer to a 32 bit integer:

- `vertices` : This 32 bit integer represents how many vertices the graph has.
- `directed` : This boolean variable represents whether the edges are directed (true) or not (false).
- `*visited` : This is an array of boolean variables that represents whether the vertexes have been visited (true) or not (false).
- `**names` : This is an array of strings. They are the names of the vertices on the graph.
- `**weights` : This is an adjacency matrix used to represent the edges and their weights. If there is no edge, the weight is zero.

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

Stack

This program creates a type `Stack`. It is a struct with two 32 bit integer variables and a pointer array of 32 bit integers. These are:

- `capacity` : This is the maximum number of elements the stack can hold.
- `top` : This variable tracks where the top of the stack is.
- `*items` : This is a pointer to an array of elements that make up the stack.

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

Paths

This program creates a type `Path`. It is a struct with a 32 bit integer variable and a pointer to a stack:

- `total_weight` : This 32 bit integer represents the weight/distance of the path.
- `*vertices` : This is a pointer to a stack of vertices.

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

START_VERTEX

This is a constant global variable of type uint32_t (a 32 bit unsigned integer). It is equal to zero. This means all paths start at vertex 0.

Algorithms

Depth First Search

This algorithm travels all possible paths in a graph. It was given in the assignment instructions[1]. Its specific implementation for this program is seen in the function descriptions section.

```
def dfs(node n, graph g):
       mark n as visited
       for every one of n's edges:
           if (edge is not visited):
               dfs(edge, g)
       mark n as unvisited
```

Function Descriptions

TSP

- `void dfs(Graph *g, Path *curr_path, uint32_t curr_vertex, Path *short_path)` : This function takes in a pointer to a graph `g`, two pointers to paths `current_path` and `shortest_path`, and a 32 bit integer `shortest_distance` . It does not return anything. This function implements a depth first search which will traverse every possible path of a graph and stores the shortest path in `shortest_path`.

```
void dfs(Graph *g, Path *curr_path, uint32_t curr_vertex, Path *short_path)
    graph_visit_vertex(g, curr_vertex)
    path_add(curr_path, curr_vertex, g)

    if path_vertices(curr_path) = graph_vertices(g) then
        if graph_get_weight(g, curr_vertex, START_VERTEX) = 0 then
            path_remove(curr_path, g)
            graph_unvistit_vertex(g, curr_vertex)
            return

        path_add(curr_path, START_VERTEX, g)

        if path_distance(short_path) = 0 then
            path_copy(short_path, curr_path)

        if path_distance(curr_path) <= path_distance(short_path)
            path_copy(short_path, curr_path)

        path_remove(curr_path, g)
        path_remove(curr_path, g)
        graph_unvistit_vertex(g, curr_vertex)

        return

    for i, 0 to i < graph_vertices(g)
        if graph_get_weight(g, curr_vertex, i, g) != 0 then
            if graph_visited(g, i) not true then
                dfs(g, current_path, i, shortest_path)

    path_remove(current_path, g)
    graph_unvisit_vertex(g, i)
```

Graph

- `Graph *graph_create(uint32_t vertices, bool directed)` : This function takes in a 32 bit integer `vertices` and a boolean variable `directed`. It returns a pointer to a graph. Its purpuse is to create a new graph struct and return a pointer to it. It initializes all items in the `visited` array to `false`. This function was given in the assignment instructions[1].

```
Graph *graph_create(uint32_t vertices, bool directed)
    *g = memory for 1 item with size of graph
    g veritces = verices
    g directed = directed

    g visited = memory set to 0 for # of vertices with size of bool
    g names = memeory set to 0 for # of vertices with size of character pointer

    g weights = memory for # of vertices with size of g weights[0]

    for i, 0 to i < vertices
```

```
        g weights[i] = memory for # of vertices with size of g weights[0][0]

    return g
```

- `void graph_free(Graph **gp)` : This function takes in a double pointer to a graph `gp`. It does not return anything. Its purpose is to free all the memory used by the graph.

```
void graph_free(Graph **gp)
    if gp not NULL and *gp not NULL then
        if *gp visited then
            free(*gp visited)
            *gp visited = NULL

        if *gp names then
            for i, 0 to i < *gp vertices
                free(*gp *names[i])
                *gp names[i] = NULL
            free(*gp names)
            *gp names = NULL

        if *gp weights then
            for j, 0 to j < *gp vertices
                free(*gp *weights[j])
                *gp weights[j] = NULL
            free(*gp *weights)
            *gp weights = NULL

        free(*gp)

    if gp != NULL
        *gp = NULL
```

- `uint32_t graph_vertices(const Graph *g)` : This function takes in a pointer to a constant graph `g`. It returns a 32 bit integer. Its purpose is to return the number of vertices in the graph.

```
uint32_t graph_vertices(const Graph *g)
    v = g vertices
    return v
```

- `void graph_add_vertex(Graph *g, const char *name, uint32_t v)` : This function takes in a graph pointer `g`, a constant character string `name`, and a 32 bit integer `v`. It does not return anything. Its purpose is to give the vertex `v` the name passed in. This function is given in the assignment instructions[1].

```
void graph_add_vertex(Graph *g, const char *name, uint32_t v)
    if g names[v] then free(g names[v])
    g names[v] = strdup(name)
```

- `const char* graph_get_vertex_name(const Graph *g, uint32_t v)` : This function takes in a pointer to a constant graph `g` and a 32 bit integer `v`. it returns a constant character string. It purpose is to get the name of the city with vertex `v` and return it.

```
const char* graph_get_vertex_name(const Graph *g, uint32_t v)
    return g names[v]
```

- `char **graph_get_names(const Graph *g)` : This function takes in a pointer to a constant graph `g`. It returns a double pointer - an array of strings.

```
char **graph_get_names(const Graph *g)
    return g names
```

- `void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight)` : This function takes in a pointer to a graph `g`, and three 32 bit integers: `start`, `end`, and `weight`. It does not return anything. Its purpose is to add an edge between `start` and `end` with a weight `weight` to the adjacency matrix of the graph.

```
void graph_add_edge(Graph *g, uint32_t start, uint32_t end, uint32_t weight)
    if g directed false then
        g weights[end][start] = weight
    g weights[start][end] = weight
```

- `uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end)` : This function takes in a pointer to a constant graph `g` and two 32 bit integers: `start` and `end`. It returns a 32 bit integer. Its purpose is to return the weight of the edge between `start` and `end`.

```
uint32_t graph_get_weight(const Graph *g, uint32_t start, uint32_t end)
    return g weights[start][end]
```

- `void graph_visit_vertex(Graph *g, uint32_t v)` : This function takes in a pointer to a graph `g` and a 32 bit integer `v`. It does not return anything. Its purpose is to set the index of the vertex `v` in the array of visited vertices to true.

```
void graph_visit_vertex(Graph *g, uint32_t v)
    g visited[v] = true
```

- `void graph_unvisit_vertex(Graph *g, uint32_t v)` : This function takes in a pointer to a graph `g` and a 32 bit integer `v`. It does not return anything. Its purpose is to set the index of vertex `v` in the array of visited vertices to false.

```
void graph_unvisit_vertex(Graph *g, uint32_t v)
    g visited[v] = false
```

- `bool graph_visited(Graph *g, uint32_t v)` : This function takes in a pointer to a graph `g` and a 32 bit integer `v`. It returns either true or false. Its purpose is to return true if vertex `v` is visited in graph `g` and return false otherwise.

```
bool graph_visited(Graph *g, uint32_t v)
    return g visited[v]
```

- `void graph_print(const Graph *g)` : This function takes in a pointer to a constant graph `g`. It does not return anything. Its purpose is to print a human-readable representation of a graph.

```
void graph_print(const Graph *g)
    print(num of vertices : g vertices)
    print(Directed: g directed)

    for i, 0 to i < g vertices
        print(g names[i] | Visited?: g visited[i])

    print space
    for l, 0 to l < g vertices
        print(   l)

    print newline
```

```
    for j, 0 to < g vertices
        print(j)
        for k, 0 to k < g vertices
            print(   g weights[j][k])
        print newline
```

Stack

- `Stack *stack_create(uint32_t capacity)` : This function takes in a 32 bit integer `capacity`. It returns a pointer to a stack that it creates. It purpose is to create a stack, dynamically allocate space for it, then return a pointer to it. This function was given in the assignment instructions[1].

```
Stack *stack_create(uint32_t capacity)
    stack pointer s = memory of size(sizeof(Stack)) cast to type Stack
        s capcacity = capacity
        s stop = 0

        s items = memory for s capacity number of 32 bit integers

        return s
```

- `void stack_free(Stack **sp)` : This function takes in a Stack double pointer `sp`. It does not return anything. Its purpose is to free all space used by the stack pointed to by `sp` and to set the pointer to NULL. This function was given in the assignment instructions[1].

```
void stack_free(Stack **sp)
    if sp not NULL and sp pointer not NULL then
        if sp pointer items not NULL then
            free(sp pointer items)
            sp pointer items = NULL

        free(sp pointer)

    if sp not NULL
        sp pointer = NULL
```

- `bool stack_push(Stack *s, uint32_t val)` : This function takes in a pointer to a stack `s` and a 32 bit integer `val`. It returns either true or false. Its purpose is to add `val` to the top of the stack and increment the top of the stack. If this operation is successful, it returns true. If not, it returns false. This function was given in the assignment instructions[1].

```
bool stack_push(Stack *s, uint32_t val)
    if stack is full then
        return false

    s items[s top] = val
    s top = s top + 1

    return true
```

- `bool stack_pop(Stack *s, uint32_t *val)` : This function takes in a pointer to a stack `s` and a pointer to a 32 bit integer `val`. It returns either true or false. Its purpose is to set the integer pointed to by `val` to the item on the top of the stack and remove that item from the stack. If this happens successfully, it returns true. If not, it returns false.

```
bool stack_pop(Stack *s, uint32_t *val)
    if stack is empty then
```

```
        return false

    *val = s items[s top - 1]
    s top = s top - 1

    return true
```

- `bool stack_peek(const Stack *s, uint32_t *val)` : This function takes in a pointer to a constant stack `s` and a pointer to a 32 bit integer `val`. It returns either true or false. Its purpose is to set the integer pointed to by `val` to the value at the top of the stack. If this operation succeeds, true is returned. Otherwise, its returns false.

```
bool stack_peek(const Stack *s, uint32_t *val)
    if stack is empty then
        return false

    *val = s items[s top - 1]
    return true
```

- `bool stack_empty(const Stack *s)` : This function takes in a pointer to a constant stack `s`. It returns either true or false. Its purpose is to check whether the stack `s` is empty. If it is it returns true; it returns false otherwise.

```
bool stack_empty(const Stack *s)
    if s top = 0 then
        return true
    return false
```

- `bool stack_full(const Stat *s)` : This function takes in a pointer to a constant stack `s`. It returns either true or false. Its purpose is to check whether the stack `s` is full. If it is it returns true; it returns false otherwise.

```
bool stack_full(const Stack *s)
    if s top >= s capacity then
        return true
    return false
```

- `uint32_t stack_size(const Stack *s)` : This function takes in a pointer to a constant stack `s`. It returns a 32 bit integer. Its purpose is to return the number of elements in the stack `s`.

```
uint32_t stack_size(const Stack *s)
    return s top
```

- `void stack_copy(Stack *dst, const Stack *src)` : This function takes in a stack pointers, `dst` and a constant stack pointer, `src`. It does not return anything. Its purpose is to copy the stack `src` to the stack `dst`.

```
void stack_copy(Stack *dst, const Stack *src)
    assert dst capacity <= src capacity
    dst top = s top
    memcopy(to dst items, from src items, for dst capacity * sizeof(uint32_t) bits)
```

- `void stack_print(const Stack *s, FILE *outfile, char *cities[])` : This function takes in three parameters: a pointer to a constant stack `s`, a pointer to a file `outfile`, and a pointer to a character string `cities[]`. It does not return anything. Its purpose is to print out the stack as a list of elements, given a list of vertex names, starting with the bottom of the stack. This function was given in the assignment instructions[1].

```
void stack_print(const Stack *s, FILE *outfile, char *cities[])
    for i, 0 to i < s top
        print cities[s items[i]]\newline to outfile
```

Paths

- `Path *path_create(uint32_t capacity)` : This function takes in a 32 bit integer `capacity`. It returns a pointer to a Path data structure containing a stack and a weight of zero. Its purpose is to create the Path.

```
Path *path_create(uint32_t capacity)
    path pointer p = memory of size(sizeof(Path)) cast to type Path
    p vertices = create stack(capacity)
    p total_weight = 0
    return p
```

- `void path_free(Path **pp)` : This function takes in a double pointer to a path `pp`. It does not return anything. Its purpose is to free the path and all its allocated memory.

```
void path_free(Path **pp)
    if pp not NULL and pp pointer not NULL then
        if pp vertices not NULL then
            stack_free(pp vertices)
        free(pp pointer)

    if pp not NULL then
        pp pointer = NULL
```

- `uint32_t path_vertices(const Path *p)` : This function takes in a pointer to a constant Path `p`. It returns a 32 bit integer. Its purpose is to find the number of vertices in the path and return that value.

```
uint32_t path_vertices(const Path *p)
    v = stack_size(p vertices)
    return v
```

- `uint32_t path_distance(const Path *p)` : This function takes in a pointer to a constant Path `p`. It returns a 32 bit integer. Its purpose is to return the distance of the path.

```
uint32_t path_distance(const Path *p)
    d = p total_weight
    return d
```

- `void path_add(Path *p, uint32_t val, const Graph *g)` : This function takes in three parameters: a pointer to a constant Path `p`, a 32 bit integer `val`, and a pointer to a constant Graph `g`. It does not return anything. Its purpose is to add vertex `val` from graph `g` to the path. It also updates the distance and length of the path.

```
void path_add(Path *p, uint32_t val, const Graph *g)
    if stack_empty(p vertices) then
        assert(stack_push(p vertices, val))
        return
    uint32_t start
    stack_peek(p vertices, start)
    assert(stack_push(p vertices, val))
    p total_weight += graph_get_weight(g, start, val)
```

- `uint32_t path_remove(Path *p, const Graph *g)` : This function takes in two parameters: a pointer to a constant path `p` and a pointer to a constant graph `g`. It returns a 32 bit integer. Its purpose is to remove the most recently added vertex from the path. It also updates the distance and length of the path. It returns the removed vertex.

```
uint32_t path_remove(Path *p, const Graph *g)
    uint32_t end
    assert(stack_pop(p vertices, &end) = true)
    if stack_size(p vertices) <= 1 then
        p total_weight = 0
        return end
    uint32_t start
    stack_peek(p vertices, start)
    total_weight -= graph_get_weight(g, start, end)
    return end
```

- `void path_copy(Path *dst, const Path *src)` : The function takes in a pointer to a path `dst` and a pointer to a constant path `src`. It does not return anything. Its purpose is to copy the path `src` to the path `dst`.

```
void path_copy(Path *dst, const Path *src)
    dst total_weight = src total_weight
    stack_copy(dst vertices, src vertices)
```

- `void path_print(const Path *p, FILE *f, const Graph *g)` : This function takes in three parameters: a pointer to a constant Path `p`, a pointer to a file `f`, and a pointer to a constant graph `g`. It does not return anything. Its purpose is to print the path `p`, using the vertex names from `g` to the file `f`.

```
void path_print(const Path *p, FILE *f, const Graph *g)
    **names = graph_get_names(g)
    uint32_t v
    l = stack-size(p vertices)
    reverse = calloc(l elements with size of(uint32_t)
    i = l - 1
    while stack_size(p vertices) > 0
        assert(stack_pop(p vertices, &v) = true)
        reverse[i] = v
        i = i - 1

    for j, 0 to j < l
        print names[reverse[j]]\newline to f

    free(reverse)
```

Psuedocode

```
main
    directed = false
    dash_i = false
    dash_o = false
    char *infile_name
    char *outfile_name

    int opt
    opterr = 0
    while (opt = getopt) != -1
```

```
    switch opt
        case 'h':
            print usageg message to stdio
            return 0
            break
        case 'd':
            directed = true
            break
        case 'i':
            infile_name = optarg
            dash_i = true
            break
        case 'o':
            outfile_name = optarg
            dash_o = true
            break
        default:
            print error and usage message to stderr
            return 1

FILE *infile = stdin;
if (dash_i) then
    infile = fopen(infile_name, read)
    if infile == NULL then
        print error and usage menu to stderr
        return 1

FILE *outfile = stdout;
if (dash_o) then
    outfile = fopen(outfile_name, write)
            if outfile == NULL then
                print error and usage menu to stderr
                return 1

uint32_t num_vertices
if (fscanf(infile, "% uint32_t" , &num_vertices) != 1) then
    print error message
    return 1

Graph *g = create_graph(num_vertices, directed)

char name[100]
for i, 0 to i < num_vertices
    fgets(name, 100, infile)
    if name[0] = newline then
        fgets(name, 100, infile)

    ln = strlen(name) - 1
    if name[ln] = newline then
        name[ln] = terminating character

    graph_add_vertex(g, name, i)

uint32_t num_edges
if fscanf(infile, "%" PRId32, &num_edges) != 1 then
    print error message
    return 1

int start
```

```
int end
int weight
for j, 0 to j < num_edges
    if fscanf(infile, "%d %d %d", &start, &end, &weight) != 3 then
        print error message
        return 1

    graph_add_edge(g, start, end, weight)
}

current_path = path_create(num_vertices + 1)
shortest_path = path_create(num_vertices + 1)

dfs(g, current_path, START_VERTEX, shortest_path)

if path_distance(shortest_path) = 0 then
    print "No path found! Alissa is lost"
    return 0

print "Alissa starts at:" to outfile
path_print(shortest_path, outfile, g)
print "Total Distance: " to outfile
print(path_distance(shortest_path)) to outfile
print newline to outfile

path_free(shortest_path)
path_free(current_path)
graph_free(g)

fclose(infile)
fclose(outfile)

return 0
```

Error Handling

- Invalid command line arguments : If an invalid command line argument is given, the usage message will be printed and the program will be terminated.
- Invalid input file: The program will print an error message and the usage message to stderr and terminate the program.
- Invalid input in input file : For any invalid input, a corresponding error message will be printed and the program will terminate. Some invalid inputs are as follows:
  - Number of edges not provided
  - Invalid number of vertices
  - Invalid number of edges
  - Invalid edge input
- Invalid out file : If the output file cannot be opened, an error message will be printed and the program will terminate.
- Invalid Stack Pop : If a pop is attempted on an empty stack, the program will terminate.
- Invalid Stack Push : If a push is attempted on a full stack, the program will terminate.

Testing

To ensure no memory leaks, the program was run with valgrind using the commands `make debug` and `valgrind ./tsp -i maps/some-graph.graph`. This is shown in Fig. 2. The program was also tested using `scan-build make` as seen in Fig. 1.

To confirm the output of the program is correct, the `diff` command is utilized. To do this, the binary given in the resources repository will be run and the output will be directed into a file. This is done using commands such as `./tsp_ref -i maps/some-graph.graph -o expect1.txt` or `./tsp_ref -i invalidfile 2> expect_err1.txt`. The same commands are used with this programs executable. The files generated can then be compared using the `diff` command. The results of some of this testing is seen in Fig. 3. The differences shown are because the path is different, however the distance is the same so the program works correctly.

I also tested the different functions as they were implemented. I used assert statements and print statements to ensure the functions were working correctly. Examples of this are shown in Figures 6 and 7. I also used valgrind to ensure the free functions worked correctly.

Results

The program runs as expected. It finds the shortest path for a correctly inputted graph or it tells the user there is no path. If there are any errors, the program prints the corresponding error message. When writing this program, I had to use more debugging techniques then I have before. I usually can get away with debugging using print statements to print various variables and there values, or to signify where in the code an error is occurring. With this program, however, because the dfs function is recursive, trying to check variable values by printing them to the terminal would print way to many values and would not be useful. So, for this program, I used gdb to debug. This allowed me to set a breakpoint on the dfs function. This stopped the program each time the dfs function was called, allowing me to check the variable and argument values. This debugging allowed me to find the reason why with some graphs, the path would contain the same vertex multiple times. This bug would've been a lot more difficult to find without using gdb. I've also become more comfortable with using valgrind through this program. I now understand the difference between still reachable memory leaks and definitely lost memory leaks. I was able to pinpoint where memory leaks were occurring and how to fix them with valgrind.

From this assignment, I've learned that using a depth first search to find the shortest Hamiltonian path is fast for small graphs, but quickly begins to take much longer. My implementation is slower than the binary file's. In order to determine that it is the depth first search that takes long rather than the way I read input from the input file, I changed all the edges of 0 of one of the graphs with a long wait time to have a weight of 0. This makes it so that there are no valid edges from 0 meaning the depth first search will finish as fast as possible for that number of vertices. However, the input file contains the same amount of input, so the input reading time should be the same. When running my program with the altered graph, it ran much faster. This is shown in Fig. 4. I also came to the conclusion that the amount of edges causes the program to take longer rather than the amount of vertices. I found this out by making a copy of a clique file and changing the copy so that it has more than double the amount of vertices, but the same amount of edges. When running the program with both files, the one with more vertices was much faster. This is shown in Fig. 5. I think that a large role in the time being so much longer, in the one with less vertices, is that there are many more valid paths. From my observations, I conclude that the number of valid paths plays the largest role in how long the program will take.

```
savila35@cse13s-vm:~/cse13s/asgn5$ make scan-build
rm -f tsp tsp.o stack.o path.o graph.o
scan-build --use-cc=clang make
scan-build: Using '/usr/lib/llvm-14/bin/clang' for static analysis
make[1]: Entering directory '/home/savila35/cse13s/asgn5'
/usr/share/clang/scan-build-14/bin/../libexec/ccc-analyzer -Wall -Wextra -Wstrict-prototypes -Werror -pe
dantic -gdwarf-4 -c tsp.c
/usr/share/clang/scan-build-14/bin/../libexec/ccc-analyzer -Wall -Wextra -Wstrict-prototypes -Werror -pe
dantic -gdwarf-4 -c stack.c
/usr/share/clang/scan-build-14/bin/../libexec/ccc-analyzer -Wall -Wextra -Wstrict-prototypes -Werror -pe
dantic -gdwarf-4 -c path.c
/usr/share/clang/scan-build-14/bin/../libexec/ccc-analyzer -Wall -Wextra -Wstrict-prototypes -Werror -pe
dantic -gdwarf-4 -c graph.c
/usr/share/clang/scan-build-14/bin/../libexec/ccc-analyzer -lm -o tsp tsp.o stack.o path.o graph.o
make[1]: Leaving directory '/home/savila35/cse13s/asgn5'
scan-build: Analysis run complete.
scan-build: Removing directory '/tmp/scan-build-2023-11-13-050055-9059-1' because it contains no reports
.
scan-build: No bugs found.
savila35@cse13s-vm:~/cse13s/asgn5$
```

Figure 1: Scan-build testing

```
savila35@cse13s-vm:~/cse13s/asgn5$ make debug
clang -Wall -Wextra -Wstrict-prototypes -Werror -pedantic -gdwarf-4 -c tsp.c
clang -Wall -Wextra -Wstrict-prototypes -Werror -pedantic -gdwarf-4 -c stack.c
clang -Wall -Wextra -Wstrict-prototypes -Werror -pedantic -gdwarf-4 -c path.c
clang -Wall -Wextra -Wstrict-prototypes -Werror -pedantic -gdwarf-4 -c graph.c
clang -lm -o tsp tsp.o stack.o path.o graph.o
savila35@cse13s-vm:~/cse13s/asgn5$ valgrind ./tsp -i basic.graph
==9047== Memcheck, a memory error detector
==9047== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==9047== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==9047== Command: ./tsp -i basic.graph
==9047==
Alissa starts at:
Home
The Beach
Home
Total Distance: 4
==9047==
==9047== HEAP SUMMARY:
==9047==     in use at exit: 0 bytes in 0 blocks
==9047==   total heap usage: 18 allocs, 18 frees, 5,789 bytes allocated
==9047==
==9047== All heap blocks were freed -- no leaks are possible
==9047==
==9047== For lists of detected and suppressed errors, rerun with: -s
==9047== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
savila35@cse13s-vm:~/cse13s/asgn5$
```

Figure 2: Valgrind testing

```
savila35@cse13s-vm:~/cse13s/asgn5$ ./tsp_ref -i invalidfile 2> expect_err1.txt
savila35@cse13s-vm:~/cse13s/asgn5$ ./tsp -i invalidfile 2> err1.txt
savila35@cse13s-vm:~/cse13s/asgn5$ diff expect_err1.txt err1.txt
savila35@cse13s-vm:~/cse13s/asgn5$ ./tsp -i maps/bayarea.graph -o output1.txt
savila35@cse13s-vm:~/cse13s/asgn5$ ./tsp_ref -i maps/bayarea.graph -o expect1.txt
savila35@cse13s-vm:~/cse13s/asgn5$ diff output1.txt expect1.txt
3,10d2
< San Jose
< Hayward
< Dublin
< Walnut Creek
< Oakland
< San Francisco
< Daly City
< San Mateo
11a4,11
> San Mateo
> Daly City
> San Francisco
> Oakland
> Walnut Creek
> Dublin
> Hayward
> San Jose
savila35@cse13s-vm:~/cse13s/asgn5$
```

Figure 3: diff testing

```
savila35@cse13s-vm:~/cse13s/asgn5$ time ./tsp -i maps/clique11.graph
Alissa starts at:
v1
v11
v6
v5
v10
v4
v9
v3
v8
v2
v7
v1
Total Distance: 280

real    0m15.444s
user    0m15.391s
sys     0m0.020s
savila35@cse13s-vm:~/cse13s/asgn5$ time ./tsp -i maps/clique14.graph
No path found! Alissa is lost!

real    0m0.034s
user    0m0.006s
sys     0m0.024s
savila35@cse13s-vm:~/cse13s/asgn5$
```

Figure 4: Time: Amount of Input vs Depth First Search

```
savila35@cse13s-vm:~/cse13s/asgn5$ time ./tsp -i maps/clique11.graph
Alissa starts at:
v1
v11
v6
v5
v10
v4
v9
v3
v8
v2
v7
v1
Total Distance: 280

real    0m15.080s
user    0m15.023s
sys     0m0.021s
savila35@cse13s-vm:~/cse13s/asgn5$ time ./tsp -i maps/clique16.graph
Alissa starts at:
v1
v5
v6
v7
v8
v10
v9
v20
v19
v4
v18
v17
v16
v15
v14
v13
v12
v11
v2
v30
v29
v28
v27
v26
v25
v24
v23
v22
v21
v3
v1
Total Distance: 277

real    0m2.602s
user    0m2.554s
sys     0m0.026s
savila35@cse13s-vm:~/cse13s/asgn5$
```

Figure 5: More vertices, same amount of edges, less valid paths

```c
int main(void) {
    Stack *s = stack_create(2);
    assert(stack_empty(s));
    assert(stack_push(s, 5));
    assert(stack_push(s, 6));
    assert(!stack_push(s, 7));
    printf("size: %d\n", stack_size(s));
    uint32_t v = 10;
    assert(stack_full(s));
    assert(stack_peek(s, &v));
    printf("peek: %d\n", v);
    assert(stack_pop(s, &v));
    printf("pop: %d\n", v);
    assert(!stack_full(s));
    Stack *t = stack_create(2);
    stack_copy(t, s);
    assert(stack_pop(t, &v));
    printf("pop: %d\n", v);
    printf("s: %p\nt: %p\n", s, t);
    stack_copy(t, s);
    printf("%d\n", s == t);
    stack_free(&t);
    stack_free(&s);
    printf("s: %p\nt: %p\n", s, t);
```

Figure 6: Examples of testing

```c
    Graph *g = graph_create(3, false);
    graph_add_vertex(g, "home", 0);
    graph_add_vertex(g, "park", 1);
    graph_add_vertex(g, "zoo", 2);
    graph_add_edge(g, 0, 1, 2);
    graph_add_edge(g, 0, 2, 1);
    graph_add_edge(g, 1, 2, 3);
    Path *p = path_create(3);
    path_add(p, 0, g);
    printf("distance: %" PRId32 "\n", path_distance(p));

    path_add(p, 1, g);

    printf("distance: %" PRId32 "\n", path_distance(p));
    path_remove(p, g);
    printf("distance: %" PRId32 "\n", path_distance(p));

    path_add(p, 1, g);
    printf("distance: %" PRId32 "\n", path_distance(p));
    path_add(p, 2, g);
    printf("distance: %" PRId32 "\n", path_distance(p));
    path_add(p, 1, g);
    printf("distance: %" PRId32 "\n", path_distance(p));
    path_remove(p, g);
    printf("distance: %" PRId32 "\n", path_distance(p));
    path_free(&p);
    graph_free(&g);
```

Figure 7: More examples of testing

References

[1] Dr. Keery Veenestra and TAs. Assignment 5: Surfin' u.s.a. `https://git.ucsc.edu/cse13s/fall-2023-section-01/resources`, Fall 2023.