Assignment 6 Design

Purpose

The purpose of this program is to find the quickest way to visit every point on a graph. These points are known as nodes, and the program starts at one node, visits all the other nodes only once each, and returns to the starting node. The user will give a text file to the program, which specifies the number of nodes, the names of the nodes, the number of connections between nodes (known as edges), and then finally what the connections between the nodes are. The program will then figure out the quickest way to go to all the nodes and print them out and what the total shortest distance is.

Questions

1. An adjacency list is better at not taking as much data since we only add to it when there is a connection. Adjacency matrices are much easier to work with because it takes less time to access data since we already know where it is in the matrix.
2. I will use an adjacency matrix to store the weights between nodes. I will use an adjacency list when taking input from the text file, which has the start and end nodes, then the weight. I will then translate that into the matrix.
3. Once we find a valid path, we must see if there is an edge back to the starting node. If there is not, then the path does not count. If there is, we check if the total weight of that

path is less than the current best weight, and if it is, we store that as the new current best. After that, we keep looking until we have gone through all the paths.

4. If there are two paths with the same weight, we choose whichever one we traversed first. A new best node must have a lower weight than the previous best one in order to replace it.
5. Yes, the path chosen is deterministic. We are not using anything random and we are traversing the graph in the same way each time, so we can safely say that the result is deterministic.
6. The graph is a struct in this assignment that has many different fields. It keeps track of whether the graph is directed or not, meaning that an edge only applies from the starting to ending node, while an undirected graph will have an edge from end to start as well. The graph will also keep track of the number of vertices. An array of strings will keep track of the names of each vertex, and an array of booleans will keep track of which ones have been visited already. Finally, a 2d array of ints will keep track of the weights between the edges, with the first index being the starting node and the second being the ending node. There will not always be an edge between two given vertices.
7. There must be one final edge weight between the last and the starting node. If you have already gone through all the nodes that have a connection to the starting node, then that path is invalid.

Testing

To test my code, I will first test the helper files individually. One file will test the graph.c file by making sure all the functions work correctly. It will create a graph, add some vertices, find out how many vertices there are, get the name of some vertices, add edges with certain weights, print the weights, visit and unvisit vertices, then print everything out. It will also free everything then will be checked through valgrind. Next, I must test stack. I will have another testing file to create a stack, push some elements on, and print it out. I will then peek the top element, pop a few, and print the new stack. I will check the stack size at some point too to make sure it is the proper length. I will try copying a stack then printing it out. Lastly for the stack, I must free it and check it through valgrind. Next up is the path. I will create a path that gives the stack the given capacity. I will add some vertices and make sure that path_vertices works correctly. I will also check to make sure that the distance is correct. Then I will remove some vertices and once again check the number of vertices and total distance covered. Then just like the others, I will print it out, free it, and check it with valgrind.

For my main, I will have a bunch of text files and will compare my results to the binary reference using bash scripting. I will test for invalid graph formats, graphs with no solution, and a variety of correct graphs. I will also check for text files that don't exist. I will also make sure that the flags work properly, making sure that -i and -o take input and output files respectively, -h prints the proper help message, and -d treats the graph as directed.

How to use the program

To compile this program, all you have to do is type make. When you run it, you will give the program a text file. This text file represents a graph and should have the following format: the first line is the number of vertices, the next few lines are the names of the vertices. So if there are x vertices, then there should be x lines each with the name of a vertex. Then, the next line will be the number of edges. Finally the last few lines will specify the edges between vertices. The format for the last set of lines is a b c where a is the index of the starting vertex, b is the index of the ending vertex, and c is the weight of the edge. To run the program, type ./tsp -<flag> where <flag> is either i, o, h, or d. The flags are optional and you can do however many you would like. If you do -i, follow it by the name of a text file which will be the graph. If you don't have -i as a flag, then the program will use stdin by default. Use -o followed by another file name to specify the file that you want the program to output to. If you don't have -o as a flag, then the program will use stdout by default. -h will print a help message. -d will specify that you want a directed graph. An undirected graph means that if there is an edge between vertices x and y, then there is also an edge between y and x with that same weight. A directed graph would mean that there is only an edge between x and y unless you specify an edge between y and x. The program will output the names of the vertices in order of the quickest path, then prints out the total distance, either to stdout or whichever output file you specify.

Program Design

This program is divided into a number of different files. graph.c, with graph.h as its header, stores the graph given by the inputted file. It has functions to create, free, add vertices

and edges, visit or unvisit vertices, and access certain information from the graph. We track the order that Alissa travels through the vertices using a stack. In stack.c, with stack.h being its header, we can create, free, push, pop, peek, print, copy, and access certain values from the stack. Now a stack alone is not enough to find the best path because it does not keep track of the total weight, which is why we use a path. In path.c, with path.h being its header, we can create, free, add, and subtract from a path, as well as print it out or copy it. We can also find the number of vertices and total distance covered by a path. The tsp.c contains the main function, and puts everything together. It handles the potential flags and has the actual DFS algorithm to go through the given vertices and find the best possible path.

Pseudocode/function descriptions

Let's start with graph.c:

graph_create() takes in a number of vertices and a bool of whether it is directed or not. It allocates memory for the int vertices, bool directed, list of visited vertices, array of vertex names, and weights for edges, and returns a pointer to that created graph. graph_free() takes in a double pointer to a Graph and frees all the field in the graph, starting with the weight, and lastly the graph itself. It also sets the pointer to the graph to NULL. graph_vertices() takes in a pointer to the graph and returns the number of vertices in the graph. graph_add_vertex() takes in a pointer to a graph, a string for the name of the vertex, and a uint32_t for the index of the vertex. It first sees if a vertex already exists, and if it does, then frees it. Then it sets names field with the given vertex to a copy of the passed in string, and returns void. graph_get_vertex_name() takes in a pointer to a graph and a uint32_t value, and returns the

name of the vertex of the given graph at the given index value. The return value is constant because we don't want to change it. graph_get_names() takes in a pointer to a graph and returns an array of strings. These strings are the names of the vertices. graph_add_edge() takes in a pointer to a graph, a starting index, an ending index, and a weight which is a uint32_t. It sets the 2d array field weights in the given graph at the starting and ending index to whatever the weight passed in is, and returns void. graph_get_weight() takes in a pointer to a graph, a starting index, and an ending index, and returns the uint32_t weight in the 2d array. graph_visit_vertex() takes in a pointer to a graph and a uint32_t val and turns the array of visited vertices at the index val to true. graph_unvisit_val() is similar except it turns the given value to false in that array. graph_visited() takes in a pointer to a graph and a uint32_t and returns whether the vertex at the given value has been visited or not. graph_print() prints a readable display of all the field of the graph.

Now the stack:

stack_create() takes in a uint32_t and dynamically allocates memory for a stack, sets the capacity to the given int, sets the top to 0, and allocates memory for the items field for the size of a uint32_t times the capacity. Stack_free() takes in a double pointer to a stack and frees the items, sets the items pointer to NULL, frees what is pointed to by the stack pointer, and sets that stack pointer to NULL. stack_push() takes in a pointer to a stack and a uint32_t val and sets the top to that val, increments top by 1, and returns true if successful. If the stack is already full, the function returns false. stack_pop() takes in a pointer to a stack and a pointer to a uint32_t to store the popped item in. It removes the top item, passes it into the memory address of the

uint32_t given, and decrements top by 1, then returns true if successful. stack_peek() is the same as stack_pop, only it does not remove the top element and keeps the same top value. stack_empty() takes in a pointer to a stack and returns whether or not it is empty, which can be done by checking if top is 0. stack_full() takes in a pointer to a stack and returns whether or not it is full, which is if the top is equal to the capacity. stack_size() takes in a pointer to a stack and returns the number of elements in the stack, which is just what the top value is. stack_copy() takes in pointers to two stacks, a source and a destination. It copies all of the elements in the source to the destination, and copies the value of top as well. It is important to make sure that the top of the source is not greater than the capacity of the destination which can be done with an assert statement. stack_print() takes ina pointer to a stack, a pointer to a file, and an array of strings. It prints out all the vertex names in order of where they are in the stack, starting at the bottom, to the given outfile.

Now for the path:

path_create() takes in a capacity and dynamically allocates memory for a stack with the given capacity. It also sets the total weight to 0 and returns a pointer to that path. path_free() takes in a double pointer to a path, and calls the stack_free() function by dereferencing the double pointer once and accessing the stack in the path, then passing it in to stack_free(). It also frees what is pointed to by the pointer to the stack, then sets that pointer to NULL. path_vertices() takes in a pointer to a path and returns the number of vertices in the path. This can be attained by accessing the top value in the stack inside the path. path_distance() finds the total distance in the path and can be attained by returning the total weight field of the path.

path_add() takes in a pointer to a path, a uint32_t val, and a pointer to a graph. This function takes the vertex from the graph and adds it to the path, and increases the total weight of the path by accessing the 2d array of weights in the graph with the last vertex in the stack and the given one. path_remove() takes in a pointer to a path and a pointer to a graph, and pops the last element in the stack and decreases the distance of the path by the weight between the second to last element and the last one just popped. This function returns the index of the element most recently popped. path_copy() is essentially the same thing as stack_copy(), only it copies the total weight as well. path_print() takes in a pointer to a path, a pointer to a file, and a pointer to a graph. It prints the names of the vertices in the path from bottom to top.

The main function for tsp.c puts it all together starting with getopt() for the potential flags. It should create a graph with the given file, making each edge go both ways if -d was not given. I will use a depth first search which starts at a node, marks it as visited, adds it to the path, then visits each of its edges. If that edge has not been visited yet, it will recursively call itself again with that new edge. It will keep doing this until it reaches a node that has no more unvisited edges or way back to the start, in which case it will mark as unvisited, go back a node, and repeat. Once we find a path that has visited every node, we check if there is a way back to the starting node. If there is, we see if the total weight is less than the previous best path, and if it is, then that new path is the new best. It will go through all the paths and save the best possible path somewhere. If there is no possible path, it will inform the user of this. If there is one, then it will write the names of the vertices in the correct order, and the total distance, to either stdout or the output file given. If all works well, main() returns 0.

Update: Testing

My code works! Here, I will show how I tested it.

To test graph.c, I made a file called tg.c to test the graph. It created a graph and worked through the various functions.

```c
#include "graph.h"

#include <assert.h>
#include <fcntl.h>
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char **argv) {
    (void) argc;
    bool directed = false;
    FILE *infile = stdin;
    char line[PATH_MAX];
    char *token;
    uint32_t s;
    uint32_t e;
    uint32_t w;
    Graph *g;
    infile = fopen(argv[1], "r");
    if (infile == NULL) {
        return 1;
    }
    fgets(line, PATH_MAX, infile);
    uint32_t a = (uint32_t) atoi(line);
    g = graph_create(a, directed);
    for (uint32_t i = 0; i < a; i++) {
        fgets(line, PATH_MAX, infile);
        line[strlen(line) - 1] = 0;
        graph_add_vertex(g, line, i);
    }
```

```
    fgets(line, PATH_MAX, infile);
    uint32_t b = (uint32_t) atoi(line);
    for (uint32_t i = 0; i < b; i++) {
        fgets(line, PATH_MAX, infile);
        token = strtok(line, " ");
        s = (uint32_t) atoi(token);
        token = strtok(NULL, " ");
        e = (uint32_t) atoi(token);
        token = strtok(NULL, " ");
        w = (uint32_t) atoi(token);

        graph_add_edge(g, s, e, w);
        if (!directed) {
            graph_add_edge(g, e, s, w);
        }
    }

    assert(a == graph_vertices(g));

    graph_add_vertex(g, "Crown", 2);

    assert(strcmp("Big rock hole", graph_get_vertex_name(g, 1)) == 0);

    char **c = graph_get_names(g);

    for (uint32_t i = 0; i < a; i++) {
        printf("%s\n", c[i]);
    }

    assert(graph_get_weight(g, 2, 3) == 15);

    graph_visit_vertex(g, 1);
    graph_visit_vertex(g, 2);
```

```
    assert(strcmp("Big rock hole", graph_get_vertex_name(g, 1)) == 0);

    char **c = graph_get_names(g);

    for (uint32_t i = 0; i < a; i++) {
        printf("%s\n", c[i]);
    }

    assert(graph_get_weight(g, 2, 3) == 15);

    graph_visit_vertex(g, 1);
    graph_visit_vertex(g, 2);
    graph_visit_vertex(g, 3);

    graph_unvisit_vertex(g, 2);

    graph_print(g);

    graph_free(&g);

    fclose(infile);

    /*
        Graph *g = read_new_graph(argv[1], false)
        uint32_t v;
        v = graph
        */

    return 0;
}
```

Here is the code for tg.c. When I run it, this is the output to stdout:

```
ehoner@13s:~/ehoner/asgn6$ ./tg eg.graph
Arbys
Big rock hole
Crown
deli
elephants bar
The graph contains 5 verticces
The graph is undirected
The vertices with the following indices are visited: 1 3
Here are the names of the vertices:
Arbys
Big rock hole
Crown
deli
elephants bar
Now the weights
Arbys(0) to Big rock hole(1): 16
Arbys(0) to Crown(2): 18
Arbys(0) to deli(3): 32
Arbys(0) to elephants bar(4): 17
Big rock hole(1) to Arbys(0): 16
Big rock hole(1) to Crown(2): 14
Big rock hole(1) to elephants bar(4): 5
Crown(2) to Arbys(0): 18
Crown(2) to Big rock hole(1): 14
Crown(2) to deli(3): 15
Crown(2) to elephants bar(4): 18
deli(3) to Arbys(0): 32
deli(3) to Crown(2): 15
deli(3) to elephants bar(4): 22
elephants bar(4) to Arbys(0): 17
elephants bar(4) to Big rock hole(1): 5
elephants bar(4) to Crown(2): 18
elephants bar(4) to deli(3): 22
ehoner@13s:~/ehoner/asgn6$
```

Now to test the stack, I created a testing file called ts.c. Here is the code:

```c
int main(void) {

    Stack *s;
    Stack *s2;
    uint32_t v;
    uint32_t p;
    uint32_t p2;
    //char *restaurants[] = {"Arbys", "Burger King", "Chilis", "Dennys", "Eriks deli cafe"};

    s = stack_create(5);
    s2 = stack_create(5);
    assert(stack_empty(s));
    assert(stack_push(s, 0));
    assert(stack_push(s, 1));
    assert(stack_push(s, 2));
    assert(stack_push(s, 3));
    assert(stack_push(s, 4));
    assert(stack_push(s, 5) == false);
    assert(stack_full(s));
    assert(stack_pop(s, &v));
    assert(v == 4);
    assert(stack_peek(s, &p));
    assert(p == 3);
    assert(stack_size(s) == 4);
    stack_copy(s2, s);
    assert(stack_size(s2) == 4);
    assert(stack_pop(s2, &p2));
    assert(p2 == 3);

    stack_free(&s);
    stack_free(&s2);

    return 0;
}
```

It runs successfully.

Finally, to test the path, I created a file called tp.c which just like the others goes through and tests all the functions in the path.

```c
#include "graph.h"
#include "path.h"
#include "stack.h"

#include <assert.h>
#include <inttypes.h>
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char **argv) {
    (void) argc;
    bool directed = false;
    FILE *infile = stdin;
    char line[PATH_MAX];
    char *token;
    uint32_t s;
    uint32_t e;
    uint32_t w;
    Graph *g;
    infile = fopen(argv[1], "r");
    if (infile == NULL) {
        return 1;
    }
    fgets(line, PATH_MAX, infile);
    uint32_t a = (uint32_t) atoi(line);
    g = graph_create(a, directed);
    for (uint32_t i = 0; i < a; i++) {
        fgets(line, PATH_MAX, infile);
        line[strlen(line) - 1] = 0;
        graph_add_vertex(g, line, i);
```

```
    }

    fgets(line, PATH_MAX, infile);
    uint32_t b = (uint32_t) atoi(line);
    for (uint32_t i = 0; i < b; i++) {
        fgets(line, PATH_MAX, infile);
        token = strtok(line, " ");
        s = (uint32_t) atoi(token);
        token = strtok(NULL, " ");
        e = (uint32_t) atoi(token);
        token = strtok(NULL, " ");
        w = (uint32_t) atoi(token);

        graph_add_edge(g, s, e, w);
        if (!directed) {
            graph_add_edge(g, e, s, w);
        }
    }

    fclose(infile);

    assert(a == graph_vertices(g));

    Path *p;
    p = path_create(graph_vertices(g));

    path_add(p, 2, g);
    //if (graph_get_weight(g, 2, 4
    path_add(p, 4, g);
    path_add(p, 1, g);
    path_add(p, 0, g);
    path_add(p, 3, g);

    //path_print(p, stdout, g);
```

```
    assert(path_vertices(p) == a);
    path_remove(p, g);
    assert(path_vertices(p) == a - 1);

    printf("%u\n", path_distance(p));
    path_print(p, stdout, g);

    assert(path_vertices(p) == 4);
    assert(path_distance(p) == 39);

    Path *pd;

    pd = path_create(576);

    path_copy(pd, p);

    path_print(pd, stdout, g);

    assert(path_vertices(p) == path_vertices(pd));
    assert(path_distance(p) == path_distance(pd));

    path_free(&p);
    path_free(&pd);
    graph_free(&g);

    return 0;
}
```

And finally, to test the actual program: here are the differences between my code and the binary reference (there are no differences):

```
ehoner@13s:~/ehoner/asgn6$ ./tsp -i maps/basic.graph -o o.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp_x86 -i maps/basic.graph -o o2.txt
ehoner@13s:~/ehoner/asgn6$ diff o2.txt o.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp -i maps/bayarea.graph -o o.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp_x86 -i maps/bayarea.graph -o o2.txt
ehoner@13s:~/ehoner/asgn6$ diff o2.txt o.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp -i maps/clique10.graph -o o.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp_x86 -i maps/clique10.graph -o o2.txt
ehoner@13s:~/ehoner/asgn6$ diff o2.txt o.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp -i maps/clique10.graph -o o.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp -i maps/clique11.graph -o o.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp_x86 -i maps/clique11.graph -o o2.txt
ehoner@13s:~/ehoner/asgn6$ diff o.txt o2.txt
ehoner@13s:~/ehoner/asgn6$ ../tsp -i maps/clique12.graph -o o.txt
-bash: ../tsp: No such file or directory
ehoner@13s:~/ehoner/asgn6$ ./tsp -i maps/clique12.graph -o o.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp_x86 -i maps/clique12.graph -o o2.txt
ehoner@13s:~/ehoner/asgn6$ diff o.txt o2.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp -i maps/clique13.graph -o o.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp_x86 -i maps/clique13.graph -o o2.txt
ehoner@13s:~/ehoner/asgn6$ diff o2.txt o.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp -i maps/clique9.graph -o o.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp_x86 -i maps/clique9.graph -o o2.txt
ehoner@13s:~/ehoner/asgn6$ diff o.txt o2.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp -i maps/lost.graph -o o.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp_x86 -i maps/lost.graph -o o2.txt
ehoner@13s:~/ehoner/asgn6$ diff o.txt o2.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp -i maps/surfin.graph -o o.txt
ehoner@13s:~/ehoner/asgn6$ ./tsp_x86 -i maps/surfin.graph -o o2.txt
ehoner@13s:~/ehoner/asgn6$ diff o.txt o2.txt
ehoner@13s:~/ehoner/asgn6$
```

```
ehoner@13s:~/ehoner/asgn6$ ./tsp -i eg.graph
Alissa starts at:
Arbys
Big rock hole
elephants bar
deli
Cats cradle
Arbys
Total Distance: 76
ehoner@13s:~/ehoner/asgn6$ ./tsp_x86 -i eg.graph
Alissa starts at:
Arbys
Big rock hole
elephants bar
deli
Cats cradle
Arbys
Total Distance: 76
ehoner@13s:~/ehoner/asgn6$
```

The code works!