Assignment 5 – Surfin' USA

Johann Lee

CSE 13S – Fall 2023

Purpose

The purpose of this program is to demonstrate the usage of depth-first search algorithms in order to find the shortest Hamiltonian cycle within a tree. This is done in conjunction with reading and writing to files outside of the executable allowing for outputs to be either written in stdout or in a file. The other way is also through for reading where the program can either read through a file to create a graph or read stdin. Based on the weight between vertices in the graph, the algorithm will search and find the most efficient path from point A to every vertex on the graph and back to point A.

How to Use the Program

This program will use the command line in order to execute. First run "make" in order to create the executable which will be named "tsp". For help on to running "tsp" run "./tsp -h" which will print a help sheet in order to help figure out the command line options. The command line is "-d" which will define whether the created graph is directed or not. This means whether the graph has identical weights between vertices or if the weight changes based on the direction between vertices. Next is the "-i" command which will by default accept a graph through stdin, however, it would be much easier to write a file for the graph. To read a file use "-i ¡filename.graph¿". This will prompt the program to read and create a graph from the file given. The read works by requiring the graph to be written in this order. First is the number of vertices and a newline. After the name of each vertex followed by a newline. Note that the first vertex will be the starting point and end point. Then print the number of edges and a newline. Then the adjacency matrix. It must be given in this order for the graph to be properly created. Next is the "-o" command line which will direct the program to print the output of the shortest path. By default if no "-o" is present it will print it in stdout, else, use "-o ¡filename¿" in order to direct the location of printing the results. Regarding reading the output, it will show Alissa's starting point and then each of the places that they visit. The destination should end at the same place as the starting point. It will then end with a total distance traveled showing how much was traveled during the cycle.
Regarding the makefile there are 2 commands to know. "make" and "make clean". Make as stated before will create the executable for tsp and make clean will remove all object and executables. Know that "make" can be substituted by "make all" and "make tsp" for this program.
This program had no reports of valgrind memory leaks or scan-build errors. Regarding results, there is a bug regarding the wrong output with clique graphs. However, disconnected graphs work as required.

Program Design

Functions

This section will explore the different functions that are utilized throughout the program.

- Graph functions
  - This is the graph struct that is used throughout the graph.c file

```
given by Jesse Srinivas asgn5.pdf page 4

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

- The first function is the graph create function. This takes parameters uint32_t vertices and boolean directed. First the graph struct must be initialized with dynamic memory. Then initialize the vertices and directed portions of the graph struct to their respective parameters. Use dynamic memory again to create the visited, names and weights of the graph struct. The next portion does the same within a for loop to create dynamic memory for 0 through vertices in weights. Finally return the created graph.
- The next function is the freeing the graph which only has the parameter of a double pointer to a graph. First use an if statement to verify that a pointer to graph is not null. Then use a for loop from 0 through the graph vertices in order to free each dynamic memory within weights. Then free weights itself. Next use the same for loop in order to free each of the names of the graph. Then free names itself and the visited dynamic memory as well. Finally free the graph and if the graph is not null set it to null.
- The graph vertices function needs a graph parameter and just returns the graph vertices
- Graph add vertex was given and adds a vertex to the graph using parameters for a graph, a const char, and a 32 bit unsigned integer.

```
given by Jesse Srinivas asgn5.pdf page 5

if (g->names[v]) free(g->names[v]); g->names[v] = strdup(name);
```

- Next is a function to get the vertex name of a graph. It requires parameters for a graph and a 32 bit unsigned integer. All that is needed is to return the graph names at the position of the integer.
- The next function is a function to the return all the names which uses the parameter for a graph. This returns all the names within a graph and requires no position.
- next is the add an edge to the graph. This requires a parameter for a graph, and three 32 bit unsigned integers named start, end, and weight. Set variable weight at position start and end of the graph weights. If the graph is not directed do the same for the reverse with position end to start.
- Next is a function to return the weight of the a graph weight at position start and end. This uses the parameters for a graph and two 32 bit integers named start and end. Just return the g weight at position start and end.
- Next is a method to set a graph vertex to visited. It is a void so all that is required is to set graph visited at position v to true.
- The reverse is also needed and sets a graph visited at position v to false.
- Finally, a check to see if the graph visited at position v is true or false. It is as said with an if statement checking if graph visited at position v is true. If it is then return true else return false.
- This is the section for the stack functions.
  - This is the given struct for stack and how it creates and frees itself.

```
This was given by Jesse Srinivas asgn5.pdf page 7 8

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

Stack *stack_create(uint32_t capacity) {
    //Attempt to allocate memory for a stack
    //Cast it to a stack pointer too
    Stack *s = (Stack *) malloc(sizeof(Stack));
    s->capacity = capacity;
    s->top = 0;
    //We ened enough memory for <capacity> numbers
    s->items = calloc(s->capacity, sizeof(uint32_t));
    //we created our stack, return it
    return s;
}

void stack_free(Stack **sp) {
    if (sp != NULL && *sp != NULL) {

        if ((*sp)->items) {
            free((*sp)->items);
            (*sp)->items = NULL;
        }

        free(*sp);
    }
    if (sp != NULL) {
        *sp = NULL;
    }
}
```

– Next is the stack push function which pushes an element onto the given stack.

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

    check if the stack is full
        if it is then return false

    s->items at position s->top should be set to val;
    increase top of the stack

    return true
```

– Since we have a push we also need a pop and a peek

```
bool stack_pop(Stack *s, uint32_t *val)
and bool stack_peek(const Stack *s, uint32_t *val)

    for both check if the stack is empty
        if its true then return false
```

```
                set val to s->items and position s->top -1

                for stack pop decrease top of the stack
                for stack peek do nothing

                return true
```

  - Next we need a stack empty function since we use them in prior functions

```
            bool stack_empty(const Stack *s)

                if the top of the stack is 0
                    return true
                else return false
```

```
            bool stack_full(const Stack *s)

                if the top of the stack is at the capacity of the stack
                    return true
                else return false
```

  - The next is a stack size function which just returns the top of the stack
  - We now need a way to copy a stack for later usage

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

                assert that the capacity of the destination
                is more than the size of the source

                for 0 through stack size of source
                    set destination stack item to source stack item

                set top of destination to top of stack
```

  - Our final function for stack is a way to print a stack from bottom to top. This is given.

```
                given by Jesse Srinivas asgn5.pdf page 9

                void stack_print(const Stack *s, FILE *f, char *vals[]) {
                    for (uint32_t i = 0; i < s->top; i += 1) {
                        fprintf(f, "%s\n", vals[s->items[i]]);
                    }
                }
```

- Our final helper function file is path which will use both previous function files.
  - The path uses the given struct throughout the file.

```
given by Jesse Srinivas asgn5.pdf page 10

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

  - The first two functions are the same as before a creation and a free function

```
Path *path_create(uint32_t capacity)

    create a path with dynamic memory
    set the path total weight to 0
    set path vertices to a new stack using stack_create
    return the created path

void path_free(Path **pp)

    if the path is not NULL
        free the vertices using stack_free
        free the path itself
        set path to NULL
```

- We need both a way to return the number of vertices and the total weight of the path. This is done by creating two functions. One that returns the stack size of the path stack. Another function that returns the total weight of the path.
- Now we need a method to add and remove things within our path structure.

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

    if the path stack is empty
        set total weight to 0
        push the val onto the path stack
    else
        create a uint32 variable
        peek into the path stack and save
        that value to the uint32

        push the val onto path stack

        add the weight of the edge to the total weight
        of the path from the variable to the val using
        graph get weight

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

    create a uint32 variable and set it to 0
    use stack peek and set the variable to
    top of the path stack

    if the path stack size is 2 or less
        pop the path stack
        set the total weight to 0
    else
        create another uint32 variable at 0
        pop and peek the path stack onto the second variable

        adjust the total weight by subtracting the weight of the
        graph weight from the first and second variables
```

- Just as before we also need a method to copy paths. This will be just like the stack copy like before where there is a destination path and a source path. The total weights of source need to be copied over to total weight of destination. Then use stack copy on the source path stack onto

  the destination path stack.
  - We also need path print function. This uses stack print that we created earlier onto path stack our outfile and the graph get names function.

Main and DFS

This section will explore the main function and the depth first search helper function that is located in the main tsp.c file.

- This is the DFS function.

```
void dfs(Graph *g, Path *p, Path *f, uint32_t vertex,
uint32_t starter, uint32_t *min_dist)
To clarify the variables, vertex is the current node
that is being used. Starter is the the starting
node. min_dis the current most optimal path.

    use graph_visit_vertex on vertex to mark it true
    path_add the vertex to the current path

    if the path is full
        if the path can return home
            path add the starter
            create uint32 curr distance and set it
            to current path distance

            if the curr_dist is less than the min_dist
                set the curr_dist to min_dist
                path_copy p onto f

            remove the path to starter

    for i=0 through graph vertices
        if the weight of vertex to current i is not 0 and
        it is not visited
            recurse dfs

    path remove and unvisit vertex
```

- This is the main function

```
int main(int argc, char *argv[])

    create integer opt, boolean directed = false
    infile defaulted at stdin
    outfile defaulted at stout
    create graph

    while opt = getopt for letter "dhi:o:"
        switch case opt
            case h print help message
            case d directed is true
            case i
                infile is set to optarg on read
                    check infile is not null
                uint32t vertex, edge
                fscanf for vertex
                set graph to graph create using vertex

                for i=0 to vertex
                    create char name with large number buffer

                    fget file onto name
                    remove newline

                    graph add vertex

                fscanf for edge
                for i=0 to edge
                    uint32 start, end, weight
                    fscanf start, end, weight
                    graph add edge using start end weight

                if infile is not stdin
                    fclose infile

            case o
                outfile is set to optarg on write

    create temp path
    create final path

    uint32 min_dist = VERY LARGE NUMBER
    dfs(graph, temp path, final path, START_VERTEX, START_VERTEX, min_dist)

    if final path distance is not 0
        print path and distance
    else
        print lost message
    if outfile is not stdout
        fclose outfile

    pathfree temp path
    pathfree final path
    graph free graph

    return 0
```

Results

The code was tested using the pipeline and the binary. It was found that my code could not find the shortest path of any clique graphs, however, it was able to find the shortest path in disconnected graphs like bayarea.graph and surfin.graph. There was also no memory leaks for error messages from scan-build. The reason for my inability to solve the clique graph was, to be frank completely unknown. I tried numerous hours to attempt to solve my it would not work for clique graphs but I could not find the error to my dismay. Regarding what I learned I learned a massive amount during this project. One thing is how complicated DFS can be when including it with Hamiltonian cycles. Trying to create the recursive was something that I really struggled with as it was very easy to get my program stuck into a loop. I also noticed that TSP took noticeably longer to work compared to previous projects. I think the reason for this was the sheer amount of loops dfs has to go through in order to search through every cycle of a graph. I found that the total amount of cycles in a clique graph is a factorial meaning that the O notation of this program is a factorial scaling which very quickly increased. This time was nearly as long on non-clique graphs like surfin and bayarea. Regarding debugging, I have decided that I most definitely need sanity checks on my next assignment. I also need to learn how to test each of my files because I think that has taken up a lot of time for me. I think this project while not completely successful in terms of my grade, helped me to learn my work etiquette and how I code.

References

Jesse Srinivas, asgn5.pdf pages 4,5 ,7, 8, 9, 10