Assignment 4
The Perambulations of Denver Long
Design Document

Name: Arad Levin
CSE13S - Fall Quarter
Prof. Long
10/25/21

Description of Program:

The purpose of this program is to solve the Traveling Salesman Problem (TSP) using depth-first search (DFS). First, a graph (matrix, created using an array of arrays) is created with a certain amount of vertices, defined by the user (this number must be less than the predetermined maximum). Each vertex will have a name, generally the name of a city. The user also has the option of making the graph undirected using a command line argument. There will then be two paths that are created, with one being the current path that is being traveled and the other existing to hold the shortest path that has been found so far. DFS will then be used, started at the origin of the graph, to find the paths. Once the shortest path is found the length of it is printed as well as the path itself. Additionally, the number of calls to the DFS function will be printed for research purposes. The user has the option to print out all of the paths found as well using a command line option.

How Each Part of the Program Should Work:

The stack file will create a stack that can be used by the graph and path files. This stack will have three variables, ‘top’ (the index of the next empty slot), ‘capacity’ (the maximum size of the stack), and ‘items’ (an array of uint32_t’s). Within this file multiple functions will exist to serve as constructors, destructors, accessors, and more. The first function, the constructor, will allocate memory for the stack and set each of its variables to their beginning values. The second function will be a destructor, freeing the memory used by the stack and setting its pointer to null. The following functions will be accessor or manipulator functions that each change or return a value within the stack. There will also be a function to copy the stack from a source to a destination, and a function that will print the stack to an outfile.

The graph file will create a matrix of vertices, serving as the graph used for the TSP. Other than the matrix, it will have a uint32_t variable called ‘vertices’ (the amount of vertices), ‘undirected’ (a boolean whose truth is decided by the user), and ‘visited’ (another boolean, keeping track of whether a vertex has been visited). Similarly to the stack file, there are multiple functions that will exist to serve as constructors,

destructors, accessors, and more. The first function, the constructor, will allocate memory for the graph and set each of its variables to their beginning values. The second function will be a destructor, freeing the memory used by the graph and setting its pointer to null. The following functions will be accessor or manipulator functions that each change or return a value within the graph. There will also be a function that will print the graph.

The path file utilizes the previous files and functions to create paths along the vertices. It only has two variables, A stack 'vertices' (the vertices comprising the path), and a uint32_t length (the length of the path). Like its predecessors, this file has multiple functions that serve as constructors, destructors, accessors, and more. The first function, the constructor, will allocate memory for the path and set each of its variables to their beginning values. The second function will be a destructor, freeing the memory used by the path and setting its pointer to null. The following functions will be accessor or manipulator functions that each change or return a value within the path. There will also be a function that will print the path.

A file containing the main function named sorting.c will be used to create the executable used to run the program. It will be called with an argument (or multiple) corresponding to the sorts the user wants to run. There will also be arguments that can be used to show more specific statistics about each sort, one to run all sorts, and one to print a help page. The sorting options will be enumerated and then, when called, placed in a set from which they will be called and run. This will be done using the getopt function to separate the arguments and a while loop containing a switch statement that will parse the arguments provided and insert the corresponding sort option into the set. If the help option is chosen, only the help page will be printed.

Pseudocode/Explanation:

tsp.c file:

open DFS function, DFS will do the following:
This function uses Depth-First-Search to finds multiple paths (and save the shortest) through the graph that is given. The function takes in multiple parameters, including a Graph G, a vertex (uint32_t) v, two Path's: curr and shortest, a character array cities, and an outfile to print the results out to. The function is void and returns nothing.
close DFS

open help function
The help function is used to print out the help screen when
a user has not entered the correct arguments or has requested for
the help screen to appear. This function will also terminate the program.
The function takes no parameters and returns nothing as it exits the code
no matter what.

include relevant files/libraries
define options as a string of possible arguments for the main function

open help function with no parameters with type int

    print the ‘help page’ for the program
    exit the program

end help function

open main function with two parameters, one to count arguments and one to store them

    if there were no arguments, call the help function
    open a while loop that will iterate through the arguments until it has gone through all of them (using the getopt function from the unistd.h library)
        begin a switch statement taking in the variable holding the current argument
        create a case for each argument
        For v case
            enable verbose printing
        For u case
            enable undirected graph
        For i and o cases
            set input/output files
        For the help case
            call the help function
        For the default case
            call the help function
        each case will end with a break statement.

    take input for how many vertices and store in variable
    loop the amount of times specified right above and each time store input into array of cities
    create a graph using the inputs
    until end of file take inputs line by line of 3 numbers which will be vertex, vertex, weight
    use graph add edge function to add each one to the graph

    creates two paths, current and shortest
    call the DFS function and print the shortest path

    free everything there is to free

    return zero to indicate success

end main function

stack.c file:

create the Stack structure

    initialize three uint32_t’s: top, capacity, and array items

close structure

create a stack taking capacity as the parameter

    allocate memory for stack using malloc with the size of a stack

```
    check whether stack s exists
        initialize variables top (to 0), capacity (to capacity), and allocate memory for the
array items
        if items doesn't exist
            free the memory of the stack and set s to a null pointer

end stack creation

open stack_empty function as a boolean with parameter 's' which is a Stack

    check whether the stack 's' and the items array it consists of exist
        free them and set the s pointer to null
    return

close stack_empty function

open stack_full function as a boolean with parameter 's' which is a Stack

    check whether the value 'top' belonging to 's' is equal to the value of 'capacity'
belonging to 's'
        return the value true
    return the value false

close stack_full function

open stack_size function as a a uint32_t with parameter 's' which is a Stack

    return the value of variable 'top' belonging to 's'

close stack_size function

open stack_push function as a boolean with parameters 's' which is a Stack and a
uint32_t 'x'

    check whether the stack if full
        increase 'top' belonging to 's' by one
        set the 'top' item of 'items' belonging to 's' to 'x'
        return the value true
    return the value false

close stack_push function

open stack_pop function as a boolean with parameters 's' which is a Stack and a
uint32_t 'x' with a pointer

    check whether the stack is empty
        decrement 'top' belonging to 's' by one
        set the pointer to 'x' equal to the 'top' item in 'items' belonging to 's'
        return the value true
    return the value false

end stack_pop function

open stack_peek function as a boolean with parameters 's' which is a Stack and a
uint32_t 'x'

    check whether the stack is empty
        set the pointer to 'x' equal to the 'top' item in 'items' belonging to 's'
        return the value true
```

return the value false

close stack_peek function

open stack_copy function as void with parameters dst and src which are stacks with pointers

set ‘items’ belong to ‘dst’ equal to ‘items’ belonging to ‘src’
do the same for ‘top’

close stack_copy function

open stack_print function as void with parameters ’s’ which is a stack with a pointer, outfile which is a file with a pointer, and cities which as an array with a pointer

iterate from zero to ‘top’ belonging to ’s’
print to the outfile the current item belonging to ’s’ using the iterator variable as its index
check whether ‘top’ is next to be reached by the iterator
print an arrow ‘->’ into the outfile
print a newline to the outfile

end stack_print function

path.c file:

Make a strut that creates the ADT Path, which carries a stack of vertices and a length based on the total weight of the path.

Make a function that creates a path as empty and with no length. It takes in no parameters, but returns a path.

Make a function that deletes the path it takes in as a parameter, returning nothing but freeing the memory the path was taking.

Make a function that returns a boolean value indicating whether it successfully pushed a vertex onto the stack, using the parameters of a graph from which to take a vertex, the path onto which it will be pushed, and the vertex itself.

Make a function pops out a vertex, taking as parameters the graph to refer to for vertex name,

the path from which to pop, and a pointer to
a variable that will hold the popped value.
The function returns a boolean to indicate
success.

Make a function takes in a path
as input and returns the size of the
vertices stack it holds.

Make a function that will return the length of a path

Make a function takes in two paths, one
is a destination path and one is a source
path. The source is copied to the destination
and nothing is returned.

Make a function takes in as input a path, an outfile,
and the array of cities. It prints out the path.
It returns nothing.

graph.c

Make a struct was taken from the assignment doc
The struct initializes the ADT Graph, which has an amount
of vertices, could be undirected, and has an array that
keeps track of whether a city (vertex) has been visited
or not. Additionally, it has the matrix of positions.

Make a function was taken from the assignment doc
This function takes in as input the amount of vertices
to create the graph with, and a boolean value to
indicate whether or not the graph shall be undirected.
It returns a graph.

Create a function was taken from the assignment doc
The function deletes the graph it takes as input and
frees the memory that was used to hold it. Returns
nothing.

Make a function that returns the amount of vertices of a graph

Create a function adds an edge weight between two vertices
of the graph, the two vertices it takes as parameters and
the graph itself is also taken as a parameter. It returns
a boolean to indicate whether this was successful or not.

Make a function does almost the same thing as the previous

function but instead of changing the weight it simply
checks whether the weight exists. Same inputs and returns.

Make a function that returns the weight between the vertices of a graph.

Make a function that takes in a graph and a vertex,
and returns a value based on whether the
vertex has been visited.

Make one that marks the vertex as visited.

Files/Libraries/Binaries:

DESIGN.pdf (pushed to git)
- This file

README.md (pushed to git)
- Contains information about the program and how to run it.

graph.h (pushed to git) (provided)
- Contains the necessary function headers for set editing/checking/overall usage

path.h (pushed to git) (provided)
- Contains the necessary function headers for the graph.c file's functions to be used in the main file.

stack.h (pushed to git) (provided)
- Contains the necessary function headers for the path.c file's functions to be used in the main file.

shell.h (pushed to git) (provided)
- Contains the necessary function headers for the stack.c file's functions to be used in the main file.

vertices.h (pushed to git) (provided)
- Contains the necessary function definitions for vertices and indexes.

tsp.c (pushed to git)
- Contains the main function for this project in which all of the other files are linked and are called in order to run the TSP program.

tsp (not pushed to git)
- Executable file for program.

graph.c (pushed to git)
- Creates a graph based on the vertices and edges provided by the user

stack.c (pushed to git)

- Handles the creation/deletion/manipulation of stacks that are used by graph.c and path.c

path.c (pushed to git)
- Creates a path that consists of the vertices travelled so far (through DFS) and their total weight.

Makefile (pushed to git)
- Contains instructions used to create a binary that can be run in order to use the program, instructions for removing unnecessary files,
and instructions for formatting all of the files.