# Assignment 4 - The Perambulations of Denver Long

Teresa Joseph

CSE 13S13S - Professor Long

Fall 2021 - RD October 21/ FD October 24

## Purpose

Assignment 4 is the implementation of Hamiltonian paths, where two vertices are connected by an edge, each vertex is only visited once, and the path starts and ends with the same vertex. These paths can be directed or undirected (symmetric about a matrix's diagonal or not). Three files called graph.c, path.c, and stack.c will be used to create and manipulate the paths. tsp.c will use these three files and take on command line arguments h, v, u, i, and o for testing purposes.

## Breakdown of Files

- graph.h specifies the interface to the graph ADT
- graph.c implements the graph ADT
- path.h specifies the interface to the path ADT
- path.c implements the path ADT
- stack.h specifies the interface to the stack ADT
- stack.c implements the stack ADT
- tsp.c contains main() and may contain any other necessary functions
    - Takes command line options h (help message), v (verbose printing), u (specifies graph is undirected), i infile (specifies input file), o outfile (specifies output file)
- vertices.h defines macros regarding vertices
- Makefile contains commands for executing, cleaning, and formatting
- README.md describes program and Makefile
- DESIGN.pdf includes design/design process, pseudocode, and explanations
- diego.graph, mythical.graph, short.graph, solarsystem.graph, and ucsc.graph are example graph files of predetermined locations and edge weights

## Command Line Arguments

- h: prints help message describing the purpose of the graph and the command-line options it accepts, exiting the program afterwards

- v: enables verbose printing (prints all Hamiltonian paths found as well as the total number of recursive calls to dfs)
- u: specifies that the graph should be undirected
- i (infile): specifies the input file path that has locations and edges of a graph (default is set as stdin)
- o (outfile): specifies the output file path that the program prints to (default output is set as stdout)

## Pseudocode

### graph.c

```
define Graph structure as stated in header file
*graph_create
    initialize graph memory using pointer G with 0's
    let G's vertices variable be vertices
    let G's undirected variable be undirected
    return G
graph_delete
    free memory using pointer G
    set G to null/none
    return nothing (graph_delete is void type)
graph_vertices
    return G's variable vertices
graph_add_edge
    if i or j is out of bounds (greater than G's vertices):
        return false
    set matrix of row i and column j be = to variable k (represents edge weight)
    if G is undirected:
        set matrix of row j and column i also = to k
    assuming above requirements are met:
        return true
graph_has_edge
```

```
        if i and j are both less than the max possible vertex value, and the edge weight k
            between i and j is greater than zero (meaning it has a valid edge):
                return true
        if none of the above conditions are met, return false
    graph_edge_weight
        if i and j are both less than the max possible vertex value, and the edge weight k
            between i and j is greater than zero (meaning it has a valid edge):
                return the edge k
        if none of the above conditions are met, return 0
    graph_visited
        if v is in G's array visited:
            return true
        if v is not in visited, return false
    graph_mark_vistited
        if vertices.h's START_VERTEX <= v <= vertices.h's VERTICES:
            mark v as visited with visited[v] = true
    graph_mark_unvisited
        if START_VERTEX <= v <= VERTICES:
            mark v as unvisited with visited[v] = false
    graph_print
        check that i and j are in bounds
        check that k exists for i and j (0 or not)
        for every i, then for every j (both until G's vertices value):
            print the corresponding k edge value in matrix of row i and column j
```

**path.c**

```
    define Path structure as stated in header file
    *path_create
        initialize path memory using pointer p (initialize to size of path)
        call stack_create and set it = to vertices
        set path length variable = 0
    path_delete
```

```
        free pointer p’s vertices
        free pointer p
        set p to null/none
path_push_vertex
        keep track of return status with a boolean
        if vertices stack is empty:
                call stack_push on p’s vertices with v and return it
        else:
                keep track of current stack top
                let the return status be what stack_peek on p’s vertices and current top is
                call stack_push on p’s vertices with v
                if the return status is true:
                        increase p’s length by graph_edge_weight of current top and v
                        return true
                if above conditions aren’t met, return false
        if main conditions aren’t met, return false here too
path_pop_vertex
        keep track of return status with a boolean
        call stack_pop on p’s vertices with x and if this value is false:
                return false
        keep track of current stack top
        let the return status be what stack_peek on p’s vertices and current top is
        call stack_pop on p’s vertices with x
        if the return status is false:
                return false
        decrease p’s length by graph_edge_weight of current top and x
        if above conditions aren’t met, return
path_vertices
        return stack_size of p’s vertices
path_length
        return p’s length
```

```
path_copy
    set the length of the destination to the length of the source
    call stack_copy on the destination’s vertices and the source’s vertices
path_print
    print path length using p’s length to outfile
    call stack_print on p’s vertices with cities and print to outfile as well
```

**stack.c**

```
define Stack structure as stated in header file
*stack_create
    initialize stack memory (calling it s, a pointer)
    if s:
        initialize s’s top variable = 0
        let s’s capacity variable = capacity
        initialize the array items to the size of s’s capacity and make everything 0
        (this allocates enough memory for items given s’s capacity)
        if not s’s items:
            free pointer s
            set pointer s to null/none
    return s
stack_delete
    to delete the stack, look at pointer s and s’s items:
        free pointer s’s items
        free pointer s itself
        set pointer s to null/none
    return nothing (stack_delete is void type)
stack_empty
    if the top of s is 0 (meaning there is nothing in the stack):
        return true
    if the above condition isn’t met, then return false
stack_full
    if the top of the stack is (greater than or) equal to its capacity:
        return true
    if the above condition isn't met, return false
stack_size
    the top of the stack is the greatest it can be, so return s's top
stack_push
    if stack is full:
        return false
    otherwise, set x to be the stack top's content
    increment top by one so that the new top is the next empty slot
    once all actions are complete, return true
stack_pop
    if stack is empty:
        return false
    otherwise, decrement the top of the stack (since we lost one slot)
    set what x points to as the stack's second-to-top content
    return true
stack_peek
    if stack is empty:
        return false
    otherwise, set what x points to as the stack's second-to-top content (like what we
    did in stack_pop but without changing the top-most element)
    return true
stack_copy
    set the capacity of the destination to the capacity of the source
    set the top of the destination to the top of the source
    for every iter until the top of the source:
        set the items[iter] of the destination to the items[iter] of the source
stack_print
    for every iter until the top of the source:
        print outfile and cities[stack items[iter]]
        if iter+1 isn't the top of the stack:
            print outfile with “->”
        print outfile and new line
```

**tsp.c**

```
    define the command line options h, v, u, i, and o
    create variable to keep track of recursion calls
    dfs function
        mark v as visited
        push vertex v onto current path
        for w starting from 0 until graph_vertices:
            use variable weigh to track graph_edge_weight of v and w
            if weight is 0, continue searching
            if w was visited on the graph, continue searching
        increment recursion call variable by 1
        call dfs again with same parameters except w instead of v
        mark v as visited
    own strdup function(char destination pointer, char source pointer)
        initialize a character type variable
        while variable = source pointer:
            if variable is a newline or carriage return:
                set destination pointer to 0
                break out of for loop
            set the destination pointer to the source pointre
            increment both destination and source by 1
        set destination pointer to 0
        return 0
    main function
        set file types to stdin and stdout by default
        create a graph pointer (calling it gptr)
        create a path pointer (calling it current_path)
        create a path pointerer (calling it shortest_path)
        use getopt to parse through the options
```

```
use switch and cases for each of the five options
    if h, print help message
    if v, print all Hamiltonian paths found and total number of calls to dfs
    if  u, make graph undirected (set a false boolean to true)
    if i, specify input file path containing cities and edges of graph
        use optarg set to read
        check that the file is not null (print error if so)
    if o, specify output file path to print to
        use optarg set to write
        check that the file is not null (print error if so)

use sscanf to read the number of cities (first number in infile)
create enough memory to look at the names of cities based on number of cities
    check that the number is valid as given in vertices.h (print error and close
    file if not valid)

create enough memory to look at the actual cities
    check again that there are no errors aka that the lines are not malformed (if
    malformed are present, again, print error and close file)
use the previous allocated memory to create an array of cities

create graph with graph_create
    if -u is an inputted option, make graph undirected (use a boolean)
while true:
    use sscanf to parse through matrix row, column, and weight
    check that row and column values are within bounds
        if not, print error and close file
    add row value, column value, and edge to graph using graph_add_edge

use current_path and shortest_path to call DFS function
push start vertex (0 at first) to current_path using path_push_vertex
```

```
        print the path using current_path, the provided outfile, and given cities
        print the total number of recursive calls + 1 from the DFS function

        free both path memory as well as graph memory (use _delete functions)
```

## Overall Description

graph.c creates the needed functions for initializing the graph. path.c does the same for creating the path, and stack.c does the same for creating the stack. Each file makes use of a structure that defines variables and pointers needed to connect the functions. All files are of type uint32_t, bool, or void, which will return a 32-bit integers, true/false, or nothing respectively. All functions in each of these files are used to implement the main function in tsp.c. This is the main test harness, the file that will take on command line arguments and simulate Hamiltonian paths based on combinations of arguments. tsp.c contains the depth-first search (which finds the shortest path), has a function that removes extraneous newlines/carriage return characters, and the main function (which parses through command line arguments and displays the output).

malloc() and calloc() are used as necessary in all of the files. These commands create dynamic memory that can be manipulated by various sizes, especially if they are unknown at the time of creation (which they are in this assignment). This is really useful because we make use of different sizes of pointers and variables based on different inputs.

## Notes

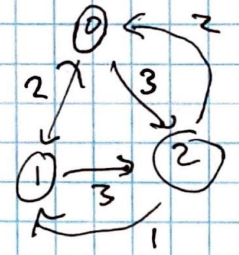
- The pseudocodes provided in the assignment document was used for the most part
- Header files were included as necessary (vertices.h in graph.c, all headers in tsp.c, etc.)
    - Header files are taken from the resources repository
- Example graph files were taken from the resources repository as well
- The command line argument “-v” was not fully implemented as I ran into trouble providing all paths and ultimately ran out of time to figure this out

## Other (visualized graph pattern)

$\boxed{\text{ucsc.graph}}$

graph_print

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & 0 & 2 & 3 \\ 1 & 2 & 0 & 3 \\ 2 & 2 & 1 & 0 \end{array}$$



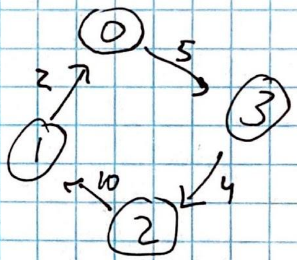
3 locations

3 recur calls cow → stere → mem → cow

$\boxed{\text{mythical}}$

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 & 5 \\ 1 & 2 & 0 & 0 & 0 \\ 2 & 0 & 10 & 0 & 0 \\ 3 & 0 & 0 & 4 & 0 \end{array}$$

assume rest are 0s?



must be 12+9=21 ✓
0 to 3 to 2 to 1
(only path)

exe: 21 len
asg → shang → oly → oly → asg   4 calls

need
gptr → graph
current } paths
shortest }