Perambulations of Denver Long - CSE13S Assignment 4

Nicholas Reis

October 18, 2021

Program Description

This program will implement three different abstract data types (ADTs) including Graph, Stack, and Path along with a test harness that reads in a file with the number of vertices, the names of the vertices, and the edges of a matrix. The program will then use recursive depth-first searching in order to find the shortest Hamiltonian path within the matrix. A Hamiltonian path will visit each vertex only once and end with an edge that leads back to the origin vertex. After the search is complete it will print of the length of the shortest path, the path itself, and the number of calls to dfs(). Command line options include: printing a help message, printing out all Hamiltonian paths, making the graph undirected (each edge between vertices points in both directions), specifying an input file path, and specifying an output file path.

Test Harness

Within tsp.c, I implemented code that supports command line options for this assignment. One of the options prints out a help message, one prints out all of the Hamiltonian paths found, one specifies that graph to be undirected, one specifies the input file, and one specifies the output file. The test harness utilizes the three ADTs that I created (Graph, Stack, and Path) in order to take in inputs that are given in a file and find the Hamiltonian paths within it. It does so by utilizing a function that implements depth-first search, which goes through each edge in the created graph and recursively calls itself if the next vertex has not been visited. It then determines whether the path is a Hamiltonian path, and subsequently finds the shortest Hamiltonian path as well.

Pseudocode

```
include graph.h, path.h, stack.h, vertices.h, inttypes.h, stdio.h,
        stdlib.h, string.h, unistd.h
define OPTIONS "hvui:o:"

create static variable to store recursive calls

void dfs():
    increment recursive calls
    if the current path is less than the shortest and the shortest is not 0:
        return

    if the number of vertices in the path equals the graph's total vertices - 1
    and there is an edge from current vertex to the origin:
        push the origin vertex to the path
        if verbose option enabled:
            print out the current path
        if the current path length is less than the shortest length and shortest
        is not 0:
            copy the current path to the shortest path
        pop the origin vertex from the path

    mark the current vertex as visited
    for loop from w = 0 to graph_vertices() - 1, increment by 1:
        if there is an edge from current vertex to w and w is not visited:
            mark w as visited
            push w to the path
            recursive call with w
            mark w as unvisited
            pop w from the path

void usage():
    print out the usage/help section
    exit the program

main:
    set the default input and output to be stdin and stdout
    create booleans for each option

    while loop that uses getopt to access the options:
        switch:
        case 'hvu':
            set these booleans as true
        case 'io':
            open the specified input or output file
        case '?' and default:
            set h boolean as true
```

```
if help option enabled:
    usage()

dynamically allocate memory for a string array of city names

scan the number of vertices from the first line of the file

for loop that iterates (number of vertices) times:
    use character buffers and fgets() to scan the city names
    change last character of buf to NULL instead of newline
    copy buf into names array using strdup()

create a Graph with given number of vertices and direction
for loop iterates until it reaches end of file
    scan each line for the two vertices and edge weight between them
    add an edge between them

create a current Path
create a shortest Path

call dfs()
if the shortest path is greater than 0:
    print out the path length, the path, and the recursive calls
else:
    print out "There's nowhere to go."

delete the graph
delete the current path
delete the shortest path
free each element of the names array
free the names array from memory
```

---

1. Need to include a lot of libraries since the test harness uses many different functions in each of them.
2. Increment the recursive calls at the beginning of the functon and then check whether the current path length is more than the shortest length (if it is, skip it). Check if a Hamiltonian path has been found, add the origin to the path, print the current path if verbose is enabled, otherwise check if the current path length is less than the shortest path (if so copy it over). Then pop the origin vertex from the path. Mark the current vertex as visited and enter the for loop. Iterate through each vertex to check if it has not been visited before and if there is an edge between it and the current vertex. If so, mark it as visited, push the vertex to the path, and call dfs recursively. Need to mark the vertex as unvisited and pop the vertex after

the dfs call to make sure it properly backs up when it needs to.

3. The usage function simply is used later on in case the usage/help section needs to be printed. exit(0) needs to be used to end the program after it is called.
4. Used booleans to keep track of options, can use sets here but this works just fine.
5. Create an array to store city (vertex) names that will be read from a file. Need to dynamically allocate memory for this.
6. If user selects option for input or output file, read or write to this file instead of stdin and stdout.
7. Use fscanf() to scan number of vertices.
8. Scan the next (number of vertices) lines in order to get the city names. Store each city name into the string array.
9. Create a Graph and scan the rest of the lines in the file to get the edges.
10. Create a shortest and current Path and call dfs(). Then print out the shortest Hamiltonian Path normally or all the Hamiltonian paths if the verbose option is enabled. Print out "There's nowhere to go." if there are no Hamiltonian paths.
11. Delete the graph and anything that was allocated to the stack, such as the path vertices and the names array.

Graph ADT

Within graph.c, I implemented code to create an ADT called Graph. It provides functions that allow me to create an adjacency matrix that stores values of a given file input with a number of vertices, edges, and edge weights. The graph.c file includes functions that create a Graph, delete a Graph, return the number of vertices, add edges, check if there exists an edge, return the edge weight, check if a vertex has been visited, mark a vertex as visited, and mark a vertex as unvisited.

Pseudocode

```
include graph.h, vertices.h, stdbool.h, stdint.h, stdlib.h

struct Graph:
    uint32_t vertices                   // Number of vertices
    boolean undirected                  // Is the graph undirected?
    boolean visited[VERTICES]           // Where have we visited?
    uint32_t matrix[VERTICES][VERTICES] // Adjacency matrix
```

```
Graph *graph_create(int vertices, bool undirected):
    use calloc to dynamically allocate memory for a Graph *G
    set the Graph's number of vertices to vertices
    set the Graph's underictted boolean to undirected
    return the Graph

void graph_delete(Graph **G):
    free() the Graph
    set the Graph to NULL
    return

uint32_t graph_vertices(Graph *G):
    return the Graph's number of vertices

bool graph_add_edge(Graph *G, int i, int j, int k):
    if i and j are both less than VERTICES:
        location [i][j] in the Graph's matrix = k
        if the Graph is undirected:
            location [j][i] in the Graph's matrix = k
        return true
    else:
        return false

bool graph_has_edge(Graph *G, int i, int j):
    if i and j are both less than VERTICES and an edge exists between them:
        return true
    else:
        return false

uint32_t graph_edge_weight(Graph *G, int i, int j):
    if i and j are both less than VERTICES:
        if graph_has_edge(G, i, j) is false:
            return 0
        else:
            return the weight of the edge at [i][j] in the matrix
    return 0

bool graph_visited(Graph *G, int v):
    if the Graph has visited v:
        return true
    else:
        return false

void graph_mark_visited(Graph *G, int v):
    if v is less than VERTICES:
        mark the Graph to have visited v
    return

void graph_mark_unvisited(Graph *G, int v):
    if v is less than VERTICES:
        mark the Graph to have unvisited v
    return

void graph_print(Graph *G):
    write any code to debug
    return
```

---

1. struct Graph defines the structure we will be using in the rest of these functions. It creates a uint32_t variable vertices, a boolean variable undirected, a boolean variable visited[VERTICES], and a uint32_t variable matrix[VERTICES][VERTICES].
2. Graph *graph_create makes a graph with dynamically allocated memory and a specified number of vertices and direction, and then returns the graph.
3. void graph_delete frees the memory allocated for the graph, and then dereferences the pointer to NULL and returns.
4. uint32_t graph_vertices returns the given graph's vertices.
5. bool graph_add_edge sets the value of matrix[i][j] to the weight, k, if both vertices are less than VERTICES (26). It also sets the value of matrix[j][i] to k if the graph is undirected. If the conditions are met and the edges are added, the function returns true, otherwise it returns false.
6. bool graph_has_edge returns true if the vertices are within bounds (less than VERTICES) and if there is an edge between the vertices (checks that the value in matrix[i][j] is greater than 0).
7. uint32_t graph_edge_weight returns false if there is not an edge between the two vertices or if the vertices are not within bounds. It returns true otherwise.
8. bool graph_visited returns true if the graph has visited the vertex v, and false otherwise.
9. void graph_mark_visited marks the given vertex as visited if it is less than VERTICES and then returns.
10. void graph_mark_unvisited marks the given vertex as unvisited if it is less than VERTICES and then returns.

Stack ADT

Within stack.c, I implemented code to create an ADT called Stack. It provides functions that allow me to use a stack to deal with adding and removing vertices from the traveled path. These functions include creating a stack, deleting a stack, checking if a stack is full or empty, returning the number of items in the stack, pushing a vertex, popping a vertex, looking at the vertex on top of the stack, copying the stack items to another stack, and printing the contents of the stack to an output file.

Pseudocode

```
include stack.h, stdbool.h, stdint.h, stdio.h, stdlib.h

struct Stack:
    uint32_t top          // Index of next empty spot
    uint32_t capacity     // Number of items that can be pushed to Stack
    uint32_t *items       // Array of items of type uint32_t

Stack *stack_create(capacity):
    use malloc to dynamically allocate memory for a Stack *s
    if s:
        set the top of the stack to 0
        set the stack's capacity to capacity
        use calloc to dynamically allocate memory for stack items
        if there are no items:
            free() the stack
            set the stack to NULL
    return the stack

void stack_delete(Stack **s):
    if the stack exists and has items:
        free() the memory allocated to the items
        free() the memory allocated to the stack
        set the pointer to the stack to NULL
    return

bool stack_empty(Stack *s):
    if the top of the stack is 0:
        return true
    else:
        return false

bool stack_full(Stack *s):
    if the top of the stack is at capacity:
        return true
    else:
        return false

uint32_t stack_size(Stack *s):
    return the top of the stack

bool stack_push(Stack *s, int x):
    if the top is at capacity:
        return false
    set the value in the items array at [top] to x
    increment the top
    return true

bool stack_pop(Stack *s, int *x):
    if the top is at 0:
        return false
    decrement the top
    dereference x to change the value it points to as the popped item
    return true

bool stack_peek(Stack *s, int *x):
    if the top is at 0:
        return false
    dereference x to change the value it points to as the top item - 1
    return true

void stack_copy(Stack *dst, Stack *src):
    for loop begins with i = 0, iterates while i < capacity, increment i:
        set value in dst items[i] = src items[i]
        set the top of dst to equal the top of src
    return

void stack_print(Stack *s, FILE *outfile, char *cities[]):
    for loop begins with i = 0, iterates while i < top of s, increment i:
        print the city name to outfile
        if the next i is not the top:
            print an -> to outfile
    print a newline to outfile
    return
```

---

1. struct Stack defines the structure we will be using in the rest of these functions. It creates uint32_t variables top, capacity, and an array of items.
2. Stack *stack_create makes a stack with dynamically allocated memory, a

top that is set to 0, a specified capacity, an array of items with dynamically allocated memory, frees the stack if there are no items, and returns the stack.

3. void stack_delete frees the memory allocated for the stack items and the stack, and then dereferences the pointer to NULL and returns.
4. bool stack_empty returns true if the top of the stack is at 0 and false otherwise.
5. bool stack_full returns true if the top of the stack is at capacity and false otherwise.
6. bool stack_size returns the top of the stack (which indicates how many items are in it).
7. bool stack_push returns false if the stack is full, otherwise increments the top of the stack to move on to the next index of the array and sets the value of the stack's items[top] to x. Returns true.
8. bool stack_pop returns false if the stack is empty, otherwise dereferences x to change the value it points to as the stack's items[top] and then decrements the top of the stack to move on to the previous index of the array. Returns true.
9. bool stack_peek returns false if the stack is empty, otherwise does the same as stack pop except does not decrement the top of the stack.
10. void stack_copy sets the top of the destination stack equal to the top of the source stack. The for loop variable begins at the value of the top of source and iterates while it is greater than 0, incrementing each loop. Inside the loop, the destination array item at the current index is set equal to the source array item at the current index. This makes each value of the items arrays the same. The function then returns.
11. void stack_print simply defines the formatting for printing a stack to the outfile.

Path ADT

Within path.c, I implemented code to create an ADT called Path. It provides functions that allow me to create a path that I will eventually use to find Hamiltonian paths. These functions include creating a path, deleting a path, pushing a vertex to a stack, popping a vertex from a stack, returning the number of vertices in a path, returning the length of a path, copying a path to another path, and printing out a path to an output file.

Pseudocode

```
include path.h, graph.h, stack.h, vertices.h, stdbool.h, stdio.h, stdlib.h

struct Path:
    Stack *vertices; // Vertices of the path
    uint32_t length; // Length of the path

Stack *path_create(void):
    use malloc to dynamically allocate memory for a Path *p
    set vertices as a new stack with VERTICES capcity
    set the length of the path to 0
    return the path

void path_delete(Path **p):
    stack_delete() the vertices stack
    free() the Path
    set the Path to NULL
    return

bool path_push_vertex(Path *p, int v, Graph *G):
    create a variable to store the vertex on top of the stack
    if there is a vertex on top of the stack (call stack_peek):
        increase path length by edge weight between top of stack and v
    else:
        increase path length by edge weight between origin and v
    if the vertex is successfully pushed (call stack_push):
        return true
    else:
        return false

bool path_pop_vertex(Path *p, int *v, Graph *G):
    if the vertex is successfully popped:
        create a variable to store the vertex on top of the stack
        if there is a vertex on top of the stack:
            decrease the path length by edge weight between top of stack and v
        else:
            decrease the path length by edge weight between origin and v
        return true
    else:
        return false

uint32_t path_vertices(Path *p):
    return the stack_size() of the path's vertices stack

uint32_t path_length(Path *p):
    return the length of the path

void path_copy(Path *dst, Path *src):
    set the length of dst equal to the length of src
    stack_copy() the vertices stack of src to the vertices stack of dst
    return

void path_print(Path *p, FILE *outfile, char *cities[]):
    print the first city to outfile
    stack_print() the vertices stack
    return
```

---

1. struct Path defines the structure we will be using in the rest of these functions. It creates a Stack of vertices and a uint32_t variable that stores the length of the path.
2. Path *path_create makes a path with dynamically allocated memory, creates the stack of vertices with a capacity of VERTICES, and sets the length of the path to 0 before returning the path.
3. void path_delete frees the memory allocated for the path and then dereferences the pointer to NULL and returns.
4. bool path_push_vertex returns false if the vertex is not successfully pushed. Otherwise, it first checks if there is a vertex on top of the stack in order to increase the length of the path by the edge weight between the top of the stack and the vertex argument given. It then pushes the vertex v to the top of the stack and returns true.
5. bool path_pop_vertex returns false if the vertex is not successfully popped. Otherwise, it pops the vertex from the top of the stack, checks if there is a vertex at the new top of the stack, and if there is it decrements the length of the path by the edge weight between the two vertices. It will then return true.
6. uint32_t path_vertices uses the stack_size() function to check the size of the vertices stack and return that value.
7. uint32_t path_length simply returns the length of the path
8. void path_copy first sets the length of the dst path equal to the length of the src path. It then utilizes the stack_copy() function to copy the values from the src vertices stack over to the dst vertices stack before returning.
9. void path_print uses the stack_print() function to print out the vertices stack to an output file.

Further Notes

- I noticed that the graph_has_edge and graph_add_edge are implemented incorrecly in the binary, as it accounts for edges of weight 0 when the assignment pdf says not to. I made those functions work exactly like the binary, but they are technically not correct.
- I also noticed that the binary prints out some paths that are longer than previous ones (I made mine like this), however if it actually skipped paths that are longer than the current shortest path, it should only print out graphs with progressively shorter lengths. This can be corrected by moving the condition that skips paths to a different section in dfs.
- I recognize that although these two issues are different from the assignment pdf, I still made my program have the same output as the binary so that my pipeline passed with full credit.

Design Process

- Began by working on graph.c, implemented all of the functions using the instructions on the assignment 4 pdf
- Went to Eugene's section on Tuesday, made a few changes to graph.c when I noticed some errors with the code
- Moved on to stack.c, now that I had a better understanding of it because of the section
- Implemented all the functions of stack.c, still have yet to test them
- Moved on to path.c, noticing that many of the functions utilize functions from stack.c and graph.c
- Implemented all the functions of path.c, need to code the test harness before I do any serious testing for it
- Began working on tsp.c, pipelines seem to tell me that the ADTs work aside from one thing in graph.c
- Decided it was easier to use booleans rather than sets for options
- Went to another Eugene section, learned how to read and write files in order to complete most of tsp.c
- Wrote all of tsp.c, still have to work on figuring out dfs()
- Spent a lot of time learning how dfs() works and it took me many tries to set everything up in the correct places
- A few of my ADT functions worked incorrectly even though I passed the pipeline with them, had to look very thoroughly to find the issues
- Had to make sure I marked/unmarked vertices, pushed/popped vertices, and had the if conditions in the right locations
- Push and pop the origin before and after a Hamiltonian path is found. Copy the current path to shortest if the length is shorter
- Compared with binary and made sure all of my outputs were the same
- Ran valgrind and made sure I freed all the allocated memory so that it passed smoothly

- Made final adjustments with my code, including comments and formatting