asgn4: The Perambulations of Denver Long

Student name: *Serjo Barron*

Course: *CSE13S* – Professor: *Prof. Long*
Due date: *10/21/21*

1 Program

This program implements a shortest Hamiltonian path search algorithm. What is a Hamiltonian path? To put it briefly, it is a path which visits all locations/places once and returns back to the origin when finished. An example of this would be going from Earth to Mars, Mars to Venus, Venus to Jupiter, and then Jupiter to Earth. The program's task though is to determine the shortest path, and will do so by using 3 ADT's (abstract data structures).

The first of the ADT's is a graph ADT, which represents our travels in the form of an n $\times$ n matrix. Each of the matrix points (such as i,j) will have a weight specified as k, which is a form of representing distance between each of the vertex's (representing locations). The weight (k) must be a non 0 positive integer, so any other points with a weight of 0 on the matrix are not considered as places.

The second ADT is the stack ADT, which will be our way of storing data. This will be useful for inserting and removing certain pieces of data which is perfect for backtracking on a calculating path if needed.

The last ADT is the path ADT, which will make use of the stack ADT and graph ADT, but will be used to create our path (in the form of a stack). An example of a path using the previous planet example above in a stack would be [Mars, Venus, Jupiter, Earth]. The path does not store the first instance of the origin (Earth in this case), but will include it as the last element (meaning the path has returned back to the origin).

Together, these 3 ADT's will make the graph, initialize all the vertices (places) on the graph, and calculate the shortest path using the DFS algorithm.

2 Required Files

1. graph.h
    - This header file includes the function prototypes of graph.c, and is used for linking purposes with other ".c" files.
2. graph.c

- This file includes the implementation of the Graph ADT.

3. path.h
    - This header file includes the function prototypes of path.c, and is used for linking purposes with other ".c" files.
4. path.c
    - This file includes the implementation of the Path ADT.
5. stack.h
    - This header file includes the function prototypes of stack.c, and is used for linking purposes with other ".c" files.
6. stack.c
    - This file includes the implementation of the Stack ADT.
7. tsp.c
    - This file will serve as the Test Harness and will be the executable run to begin the program.
8. vertices.h
    - This header file specifics macros for the start vertex and maximum number of vertices of the graph.
9. Makefile
    - This file is used to link all the required files to the Test Harness "tsp.c" and create an executable to run as the program.
10. README.md
    - A file in markdown syntax containing the program description and instructions on how to run the program
11. DESIGN.pdf
    - The design process and structure of the program (this document).

3 Psuedocode and Function Descriptions

3.1 Structs and Pointers

Before discussing the structure and the functionality of the functions, it should be noted that most if not all functions will deal with a struct, and will use the "->" arrow operator to point to a specific struct element to modify or use the value stored in that element.

3.2 graph.c

The first function of graph.c takes an integer number of vertices and boolean of undirected (true/false) as parameters. This function serves as a **constructor** of our graph, and initializes the values (elements) of the graph to 0.

```
define graph_create(vertices, undirected):
        allocate memory for graph, and initialize them to 0 (calloc)
        set vertices in struct as vertices
        set bool undirected in struct as undirected
        return G
```

This function serves as a **destructor** for a graph, and does so by taking a double pointer to a graph as a parameter. The function will deallocate the memory associated with the graph and then set the pointer to the graph as NULL.

```
define graph_delete(**G):
        deallocate memory of *G (free)
        set G pointer to null
        return
```

This function takes a pointer to a graph as a parameter and takes that graph to point to the struct element vertices, to then return the number of vertices.

```
define graph_vertices(G):
        return vertices
```

This function takes a pointer to a graph, the value of vertex i and j, and the weight of k as parameters. The function itself adds an edge of weight k from vertex i to vertex j if directed, and if undirected it adds the weight k from vertex j to vertex i as well. It will also return a boolean true if both vertices (i, j) are within bounds of the matrix and the edge (weight k) is added, and return false otherwise.

```
define graph_add_edge(*G,i,j,k):
        if i>=0 and i<vertices and j>=0 and j<vertices:
                matrix[i][j]=k
                if undirected:
                        matrix[j][i]=k
                return true
        return false
```

This function takes a pointer to a graph, value of i and j as parameters. The function checks whether vertices i and j are within bounds of the n $\times$ n matrix and returns true if they are within bounds. Return false otheriwse.

```
define graph_has_edge(*G,i,j):
        if i>=0 and i<vertices and j>=0 and j<vertices:
                if matrix[i][j]!=0:
```

```
                return true
        return false
```

This function takes a pointer to a graph, value of i and j as parameters. The function finds the weight k in vertex i, j and returns the weight. If either i or j are out of bounds of the matrix, return 0 (end program).

```
define graph_edge_weight(*G,i,j):
        if graph_has_edge(G,i,j):
                return matrix[i][j]
        return 0 (terminates program)
```

This function takes a pointer to a graph and vertex v as parameters. The function checks whether vertex v (a place) has been visited and returns true if so. Return false otherwise.

```
define graph_visited(*G,v):
        if visited[v]==true:
                return true
        return false
```

This function takes a pointer to a graph and vertex v as parameters. The function will check if vertex v is within bounds of the matrix, and mark v as **visited**.

```
define graph_mark_visited(*G,v):
        if v>=0 and v<vertices:
                visited[v]=true
                return
```

This function takes a pointer to a graph, and vertex v as parameters. It will check whether vertex v is within bounds, if so mark it as **unvisited**.

```
define graph_mark_unvisited(*G,v):
        if v>=0 and v<vertices:
                visited[v]=false
                return
```

The last function part of graph.c takes a pointer to a graph as a parameter, and will print a visual representation of the graph. This is for debugging purposes to make sure the graph ADT (abstract data type) works as intended.

```
define graph_print(*G):
        for x in range (0,vertices):
                for y in range (0,vertices):
                        print element value of matrix[x][y]
                        if y == (vertices) - 1:
                                print newline
        return
```

3.3 stack.c

The first function of stack.c serves as a **constructor**, which initializes an array which we will treat as a stack, and sets the elements of that array to 0. It takes a integer capacity as a parameter, which defines how much memory needs to be allocated for the stack.

```
define stack_create(capacity):
        (allocate memory for the stack s elements)
        if s:
                set top in struct as 0
                set capacity in struct as capacity
                items=(allocate memory for an array of size capacity)
                if not s->items:
                        free s (stack)
                        set stack pointer to NULL
        return s
```

This function serves as a **destructor** of the stack, and takes a double pointer to stack s as a parameter.

```
define stack_delete(**s):
        if *s and (*s)->items:
                free
```

This function takes a pointer to stack s as a parameter and returns true if the stack is empty. Return false otherwise.

```
define stack_empty(*s):
        if top is 0:
                return true
        else:
                return false
```

This function takes a pointer to stack s as a parameter and returns true if the stack is full. Return false otherwise.

```
define stack_full(*s):
        if top equals capacity:
                return true
        else:
                return false
```

This function takes a pointer to stack s as a parameter and returns the number (unsigned 32-bit integer) of items in the stack.

```
define stack_size(*s):
        return top (simply return the current top)
```

This function takes a pointer to stack s and an item x as parameters. The function pushes an item x on to the stack and returns true if that item successfully gets pushed to the stack. But, if the stack is full prior to pushing the item return false.

```
define stack_push(*s,x):
        if the stack is not full
                items[top]=x
                increment top by 1
                return true
        else:
                return false
```

This function takes a pointer to stack s and a pointer x as parameters. The function will pop an item off the stack (LIFO) and then pass that value to the pointer x (by dereferencing the pointer x). And then proceeds to make the index of the popped element as empty (or reset to 0). If the stack is empty prior to popping the item return false, otherwise return true.

```
define stack_pop(*s,*x):
        if stack is not empty:
                decrement top by 1
                *x=items[top] (dereference x)
                return true
        else:
                return false
```

This function takes a pointer to stack s and a pointer x as parameters. This function behaves similarly to stack_ pop() (since it dereferences the pointer x to store it in the address of x) but does not modify the stack itself. It will return false is the stack is empty.

```
define stack_peek(*s,*x):
        if stack is empty:
                return false
        else:
                *x=items[(top)-1] (dereference x)
                return true
```

This function takes a pointer to dst (destination stack) and a pointer to src (source stack). It will copy the contents (items) of the source stack over to the destination stack.

```
define stack_copy(*dst,*src):
        for x in range(0,src->capacity):
                dst->items[x]=src->items[x]
        dst->top=src->top
        return
```

This function takes a pointer to stack s, a pointer to an outfile, and a pointer to the cities array as parameters.

```
define stack_print(*s,*outfile,*cities[]):
        for x in range(0,top):
                print to outfile cities[x]
                if x+1 != top
                        print arrow (used to visually link paths)
        print newline
        return
```

3.4 path.c

This function takes no parameters and is the **constructor** for a path. A path essentially creates a stack (like in stack.c) but makes one for the struct element vertices which instead has a capacity of VERTICES which is a macro set to 26.

```
define path_create():
        (allocate memory for path p elements)
        if (p):
                vertices=stack_create(VERTICES) (VERTICES is 26)
                if (!p->vertices):
                        free path pointer
                        set path pointer to NULL
        return p
```

This function takes no parameters and is the **destructor** for the path. It sets the pointer which is pointed to the path as NULL.

```
define path_delete(**p):
        if *p and (*p)->vertices:
                delete p->vertices stack
                free((*p)->vertices)
                free(*p)
                set *p to NULL
        return
```

This function takes a pointer to p, vertex v, and a pointer to a graph as parameters. The function pushes vertex v onto path p and increments the length (defined in Path struct) by the weight connecting the vertex at the top of the stack and vertex v. It will return true if the vertex was pushed and false otherwise.

```
define path_push_vertex(*p,v,*G):
        if the stack is full:
                return false
        else:
```

```
        if the path is empty:
                find edge from 0 to v
        else:
                peek on top of stack (store in x)
                find edge from x to v
        push v onto the path stack
        length += edge
        return true
```

This function takes a pointer to p, a pointer to vertex v, and a pointer to a graph as parameters. The function pops from the vertices stack and passes the popped value to pointer v. It will also decrement the length by the edge weight connecting the vertex at the top of the stack and the popped vertex. It returns true if the vertex was popped and false otherwise.

```
define path_pop_vertex(*p, *v, *G):
        if stack is empty:
                return false
        else:
                x (peeked vertex)
                y (popped vertex)
                pop from stack, and store in y
                if the size of the path is 0:
                        find edge from 0 to y
                else:
                        peek on stack, and store in x
                        find edge from x to y
                length -= edge
                store y in v*
                return true
```

This function takes a pointer to p as a parameter and returns the current number of vertices in the path.

```
define path_vertices(*p):
        return stack_size of vertices
```

This function takes a pointer p as a parameter and returns the current length (of the path).

```
define path_length(*p):
        return length
```

This function takes a pointer to dst (destination) and a pointer to src (source) as parameters. The function will copy the contents of the dst path over to the src. The copying process requires making a copy of the vertices stack and the length of source path.

```
define path_copy(*dst,*src):
        stack_copy(dst,src)
        dst->length=src->length
        return
```

This function takes a pointer p, a pointer to an outfile, and a pointer to an array of cities as parameters.

```
define path_print(*p,*outfile,*cities[]):
        print path length title to outfile
        print origin/home path to outfile
        stack_print(p,outfile,cities)
        return
```

3.5 DFS: Depth-first search

This function will be defined inside the Test Harness "tsp.c", and is an recursive algorithm which goes through all the vertices to create a path, and of its solved paths, it wil return the shortest Hamiltonian path.

The function works by taking a vertex, marking that vertex as visited, and then searches for childs of that vertex to branch off to based on the number of vertices of the Graph. The vertex visited by v (which we will call w) will then recursively call DFS on itself until a complete Hamiltonian path is solved.

```
define DFS(*G,v,*current,*short,*cities[],*outfile):
        mark v as visited
        increment recursive call

        if length of current path >= length of shortest path
        and the length of shortest is not 0:
                skip finding current path

        if you find a valid hamiltonian path:
                push origin (0) to path stack
                if length of current < length of shortest:
                        copy current to shortest

        if verbose:
                print current path to outfile
        pop origin (0) from path stack

        for w in range(0,vertices in graph):
                if an edge exists and w is not visited:
                mark w as visited
```

```
push w to path stack
call DFS on w (recursive call)
mark w as unvisited
pop w from path stack
```

3.6 tsp.c "Test Harness"

This file will be the main source of user interaction and serve as the "Test Harness" for the program. The program works by using CLI options as a means of taking user input, and will take a file (a graph) to be read, and an outfile to print the results of the program.

```
define static bool to indicate if a graph is undirected or not
define OPTIONS: "hvui:o:"

static bools:
undirected (is graph undirected)
header (used to print synopsis message)
verbose (used to print verbose statistics)

static uint32's:
recursive (keeps track of DFS recursive calls)
vertices (the vertices in the graph file read)

insert DFS() function (defined here)

begin main function
        set infile to stdin
        set outfile to stdout

        while loop for getopt using OPTIONS:
                (switch cases):
                        case 'h':
                                set bool header to true
                        case 'v':
                                set bool verbose to true
                        case 'u':
                                set undirected to true
                        case 'i':
                                specify infile to read
                        case 'o':
                                specify outfile to write

if (header):
        print synopsis page
```

```
        return 0 (end program)

specify buffer (we will use 1024 char's)

if the infile is NULL:
        print error message, end program

file handling:
fscanf to read first line (the vertices)
store the number of vertices in vertices

if the vertices>26 or vertices<=0:
        print error message, end program

if vertices is 1:
        print error message indicating theres nowhere to go
        end program

allocate memory for city_names array:
use fgets to get the next vertices lines of names
delete the null char at the end of the buffer
store the city name in the city_name string array

create graph with specified number of vertices and bool for undirected

while the file being read hasnt hit EOF:
        use fscanf to read next lines of (i,j,k) formatted numbers
        if the line being read is another city name:
                print error message, end program
        graph_add_edge(G,i,j,k) (add edges)

create current path
create shortest path

Call DFS starting at v=0 (the origin), then the function
will recursively call itself until it finds all the paths.
If the verbose option is set to true, print the Hamiltonian
paths as they are solved.

print the shortest path found in DFS

print "total recursive calls" title to outfile

deallocate memory by iterating over the city names array
free each of those elements
free the array
```

```
delete the graph
delete the current path
delete the shortest path
return 0 (end program)
```

---

4 Error Handling

As seen in the pseudocode of the tsp "Traveling Sales Person" file, which serves as the main Test Harness of the program, it is important to deal with potential errors that could cause the program to rise a segmentation fault or any other errors. Those errors in particular are connected to opening files to be read, and considering things beforehand such as whether the file being opened even exists. To which if the file being opened has a value of NULL, it means opening the file was unsuccessful, and that the program will print an error message and terminate prematurely.

Another thing to consider is the graphs file being read (typically named name.graph). Where if the specified number of vertices (the first line read) is greater than 26 or less than or equal to 0, which is the maximum n value for the n $\times$ n matrix allowed for creating a graph (and that the graph must at least be a 1 by 1 matrix. Then the program will generate an error message and terminate prematurely since it does not fit within the specified graph size range.

Another variation of an improper amount of vertices is 1, since there would be nowhere to go. In other words, the only place that exists is home, and it would make no sense to go from home to home. Then the last potential error to account for is if the graph file were to have more city names listed than allowed by the number of vertices.

5 Credit

1. I attended Eugene's sections on Tuesday (10/19/21) and in-person (10/21/21), and took inspiration on which functions to use in my test harness design for file handling. Specifically things such as allocating memory for storing city names.
2. Certain functions such as ones pertaining to constructors, destructors, and printing have been provided in the corresponding assignment specifications (document). Also the DFS (depth first search) algorithm has been provided in brief pseudocode. All courtesy to Prof Long.
3. I attended Brian Mak's tutoring session on 10/22/21 to help debug my DFS function, to which he suggested to reformat my previous DFS implementation to mess around with where I mark, push, pop vertex's and to place my condition to stop searching a path if the current path being searched has a larger length than the shortest stored path in the beginning of the function.