asgn 4: DESIGN - Surfin' U.S.A.

Eric Wang

May 17, 2023

Purpose:

For assignment 4, we are to use graphs and depth first search to find the most optimal/cheapest way to go around the coastal cities of the United States. In hindsight, we are to use graph theory, stacks, as well as depth first search to create a solution to the Travelling Salesman Problem. Like the previous assignments, we are also required to create the main() function to test the graph, stacks, as well as the path files. Similar to the previous assignments, the main function is also required to have command line options. The command line options this time are -i -o -d -h. Where -i reads a file, -o outputs to a file, -d treats all graphs as directed graphs, and -h prints out a help message.

How to use the program:
Download the following required files from the git repository:
tsp.c
graph.c
stack.c
path.c
As well as the following required header files:
graph.h
stack.h
path.h
vertices.h
At last, download the Makefile as well as the premade .graph files
Once you have the files above, run "make" inside your terminal, you should be able to get the binary.
Syntax:
-i : Sets the file to read from (input file). Requires a filename as an argument. The default file to read from is stdin
-o : Sets the file to write to (output file). Requires a filename as an argument. The default file to write to is stdout
-d : Treats all graphs as directed. Remember that the default is to assume an undirected graph, which means that any edge (i, j) that is specified should be added as both (i, j) and (j, i). So if -d is specified, then (i, j) will be added, but (j, i) won't.
-h : Prints a help message to stdio

once you get the binary file, inside your terminal, type "**./tsp -i** (.graph file name) **-o** (outfile name, default is stdout) (*optional: do **-d** if you want to enable directed graph)".

Pseudocodes:

graph.c

```
define graph_create(vertices, directed):
        Allocate memory for graph, and initialize item to 0 (using calloc)
        Set vertices in struct as vertices
        Set the boolean value directed in struct as directed
        Return G

define graph_free(**gp):
        Deallocate memory of *gp (using free)
        Set gp pointer to NULL
        Return

Define graph_vertices(g):
        Return vertices

Define graph_add_edge(*g, start, end, weight)
        If start < vertices and start >= 0 and end < vertices and end >= 0:
        Weights[start][end] = weight
        If not directed:
                Weights[end][start] = weight
        Return

Define graph_get_weight(*g, start, end)
        If start < vertices and start >= 0 and end < vertices and end >= 0 and weights [start][end] > 0:
                Return weights[start][end]
        Return 0

Define graph_visit_vertex(*g, v)
        If v < vertices and v >= 0:
                Visited[v] = true
        Return

Define graph_unvisit_vertex(*g, v)
        If v< vertices and v >= 0:
                visited[v] = false
        Return

Define graph_visited(*g, v)
        If visited[v] == true:
                Return true
define graph_get_names(*g)
        Return names
```

```
Define graph_add_vertex(*g, v)
        if names[v]
                free names[v]
        names[v] = name

define graph_get_vertex_name(*g, v)
        return names[v]

define graph_print(*g)
        For i in range (0, vertices):
                For j in range (0, vertices)
                        Print the elements of weights[i][j]
                        If j == vertices - 1
                                Print newline
        Return
```

stack.c

```
define stack_create(capacity)
        Allocate memory for stack s
        Set stack capacity in struct to capacity
        Set top in struct to 0
        Allocate memory for items in stack
        Return stack s

Define stack_free(**sp)
        If *sp and *so->items
                Free


Define stack_push(*s, val)
        If stack is full
                Return false
        Items[top] = val
        Increment top by 1
        Return true

Define stack_pop*s, *val)
        If not stack empty
                Subtract top by 1
                *val = items[top]
                Return true
        Return false
```

```
Define stack_peek(*s, *val)
        If not stack empty
                *val = items[top - 1]
                Return true
        Return false

Define stack_empty(*s)
        If top is 0
                Return true
        Return false

Define stack_full(*s)
        If top is equal to capacity
                Return true
        Return false

Define stack_size(*s)
        Return top

Define stack_copy(*dst, *src)
        For i in rance (0, src->capacity):
                dst->items[i] = src->items[x]
        dst->top = src->top
        Return

Define stack_print(*s, *outfile, *cities[])
        For i in range(0, top):
                Print to outfile cities[i]
        return
```

path.c

Create edge variable

```
Define path_create(capacity)
        Allocate memory for path p elements
        Vertices = stack_create(capacity);
        Total_weight = 0;
        Return p

Define path_free(**pp)
        If *pp and (*pp)->vertices:
                Free pp->vertices stack
                Free (*p)->vertice
                Free *p
```

```
            Set *p to NULL
        Return
Define path_vertices(*p)
        Return stack_size(vertices)

Define path_distance(*p)
        Return total_weight

Define path_add(*p, val, *g)
        If stack is full
            Return
        Else
            If stack is empty
                Edge = graph_get_weight(g, 0, val);
            Else
                stack_peek(vertices, cood_start)
                Edhe = graph_get_weight(g, coord_start, val);
            stack_push(vertices, val);
            Total_weight += edhe
            Return

Define path_remove(*p, *g)
        If stack is empty
            Return 0
        Else
            Int peek_v, pop_v
            stack_pop(vertices, pop_v);
            If the size of the path is 0;
                Find edge from 0 to y
            Else
                Peek on stack, and store in peek_v
                Find edge from peek to pop
            Length -= edge
            Store y in v*
            Return true

Define path_copy(*dst, *src):
        stack_copy (dst, src)
        dst->length = src->length
        Return

Define path_print(*p, *f, *g)
        Print path length title to f
        Print origin/home path to f
```

stack_print(p, f, g->names)
Return

Depth-first search

Define DFS(*g, v, current, shortest, cities[])
Mark v as visited
If length of current path >= length of shortest path and the length of the shortest path is not 0:
Skip finding current path
If you find a valid hamiltonian path:
Push origin to path stack
If length of current is less than length of shortest
Copy current to shortest
Pop origin from path stack
For i in range(0, vertices)
If an edge exists and i is not visited
Mark i as visited
Push i to path stack
Call DFS on i
Mark i as unvisited
Pop i from path stack

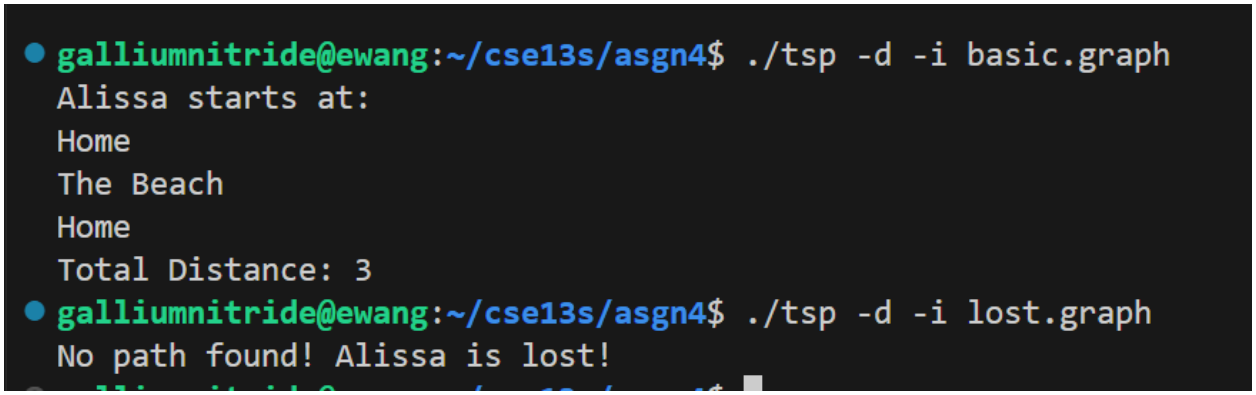
tsp.c

Get opt options here

While loop for getopt using the options
Switch case:
Case h: set boolean header to true
Case i: specify infile to read
Case o: specify outfile to write
Case d: set directed to true

Results:

Here are the results screenshot



Error Handling:
Since the one of the most important parts of this assignment is to allocate memories for the path, stack, as well as graphs needed. It is important to free the memories after doing operations with the variables. We have to keep on freeing the memory after printing. We also needed to print out an error message when the command line option inputted does not match the given options, instead of printing a help message. By default, getopt will print its own error message. But we wanted to customize it. How? First to delete the message we have to set operr to 0 in order to disable the getopt default invalid message. To print out the customized character. We have to use create a char variable and store optopt in it, thus printing out the character we inputted.

Credit:
Dr. Kerry Veenstra's asgn4 document.