Bill Zhang
jzhan411@ucsc.edu
4/28/2021

CSE13S Spring 2021
Assignment 4: The Circumnavigations of Denver Long
Design Document

This code will find the optimal path to travel through a list of cities, without repeating cities.

PseudoCode:

Graph ADT:

Struct Graph
    Uint32 vertices
    Bool undirected
    Bool visited[Vertices]
    Uint32 matrix[Vertices][Vertices]

The constructor for a graph. It is through this constructor in which a graph can be specified to be undirected.
Graph *graph_create(uint vertices, bool undirected)
    Vertices = vertices
    Undirected = undiirected
    For each element in visited
        Element = false
    For each element1 in Matrix
        For each element2 in Matrix[Vertices]
            Element 2 = 0.

Deletes Graph
Void graph_delete(Graph **G)
    If (*g and matrix and visited)
        Free matrix
        Free visited
        Free *g
        *s = NULL

Checks if value is in bounds
Bool within_bounds(uint32_t x)
    If x > 0 and x < Vertices
        Return True
    Return false

Returns vertices
uint32_t graph_vertices(Graph *G)
```
Return vertices
```

Adds edge of weight k from vertex i to vertex j.
bool graph_add_edge(Graph *G, uint32_t i, uint32_t j, uint32_t k)
```
If within_bounds(i) and within_bounds(j)
    *G matrix[i][j] = k
    Return true
    If *g undirected
        *G matrix[j][i] = k
Return false
```

Return true if vertices i and j are within bounds and there exists an edge from i to j.
bool graph_has_edge(Graph *G, uint32_t i, uint32_t j)
```
If within_bounds(i) and within_bounds(j)
    If Matrix[i][j] > 0
        Return true
Return false
```

Return the weight of the edge from vertex i to vertex j.
uint32_t graph_edge_weight(Graph *G, uint32_t i, uint32_t j)
```
If within_bounds(i) and within_bounds(j)
    Return *G matrix[i][j]
Return 0
```

Return true if vertex v has been visited and false otherwise.
bool graph_visited(Graph *G, uint32_t v)
```
If *G visited[v] = true
    Return true
Return false
```

If vertex v is within bounds, mark v as visited.
void graph_mark_visited(Graph *G, uint32_t v)
```
If v > 0 and v < vertices
    *G visited[v] = true
```

If vertex v is within bounds, mark v as unvisited.
void graph_mark_unvisited(Graph *G, uint32_t v)
```
If v > 0 and v < vertices
    *G visited[v] = False
```

Debug Function
void graph_print(Graph *G)
For i in range Vertices
    For j in range vertices
        Print (Matric[i][j])
    Print new line

Depth First Search

Search function
DFS(Graph*G, Vertex V, Current path, shortest path, verbose, cities, outfile)
    Mark v visited
    For each vertex i in graph
        If graph has edge between v and i
            If vertices in current path is equal to vertices in graph and the i is = to start vertex
                Push i onto path
                If path length is 0 and path length of current is shorter than shortest path
                    Copy current path to shortest path

            Pop the vertex i from path
        If graph visited
            Path push vertex i
            dfs(g, i, current path, shortest path, berrbose, cities, outfile)
            Path pop vertex i
    Mark v unvisited
    return

Path ADT

Vertices is the stack of vertices in the path
Length is length of path
Struct Path
    Stack *vertices
    INt Length

The constructor for a path.
Path *path_create(void)
    Length of the path = 0
    Vertices = Stack_vcreate(Vertices)

The destructor for a path.

void path_delete(Path **p)
    If *Vertices and *P
        Free *Vertices
        Free *P
        *P = null

Pushes vertex v onto path p.
bool path_push_vertex(Path *p, uint32_t v, Graph *G)
    Return Stack_Push(*P Vertices, v)

Pops the vertices stack, passing the popped vertex back through the pointer v.
bool path_pop_vertex(Path *p, uint32_t *v, Graph *G)
    Return Stack_Push(*P Vertices, *v)

Returns the number of vertices in the path.
uint32_t path_vertices(Path *p)
    Return *p vertices

Returns the length of the path.
uint32_t path_length(Path *p)
    Return *p length

Assuming that the destination path dst is properly initialized, makes dst a copy of the source path src.
void path_copy(Path *dst, Path *src)
    *dst vertices = *src vertices
    *dst length = *Src length

Prints out a path to out file using fprintf().
void path_print(Path *p, FILE *outfile, char *cities[])

Stack ADT

Struct Stack
    Int top
    Int capacity int *items

The constructor function for aStack.
Stack *stack_create(uint32_t capacity)
  Stack *s = (Stack *) malloc(sizeof(Stack))
  if (s)
    s->maxSize = 0
    s->top = 0

```
    s->capacity = capacity
    s->items = (int64_t *) calloc(capacity, sizeof(int64_t))
    if (!s->items)
        free(s)
        s = NULL
  return s
```

The destructor function for a stack.

void stack_delete(Stack **s)

```
  if (*s && (*s)->items)
    free((*s)->items)
    free(*s)
    *s = NULL
```

Return True If the stack is empty and false otherwise.

bool stack_empty(Stack *s)

```
  if (s->top == 0)
    return true

  return false
```

Return true if the stack is full and false otherwise.

bool stack_full(Stack *s)

```
  if (s->top == s->capacity)
    return true

  return false
```

Return the number of items in the stack.

uint32_t stack_size(Stack *s)

```
  return (s->top)
```

If the stack is full prior to pushing the itemx, return false to indicate failure. Otherwise, push the item and return true to indicate success.

bool stack_push(Stack *s, uint32_t x)

```
  if (stack_full(s))
    return false

  s->items[s->top] = x
  s->top++
  return true
```

Peeking into a stack is synonymous with querying a stack about the element at the top of the stack. If the stack is empty prior to peeking into it, return false to indicate failure.

Bool stack_peak(Stack*s, uint32_t *x)

```
if (stack_empty(s))
        return false
*x = s->items[s->top - 1]
```

If the stack is empty prior to popping it, return false to indicate failure. Otherwise, pop the item, set the value in the memory x is pointing to as the popped item, and return true to indicate success.

bool stack_pop(Stack *s, uint32_t *x)

```
if (stack_empty(s))
    return false

s->top--
*x = s->items[s->top]
return true
```

Assuming that the destination stack dst is properly initialized, makedsta copy of the source stack src.

void stack_copy(Stack *dst, Stack *src)

```
 *Dst top = *src top
*Dst items = *Src items
```

Prints out the contents of the stack to out file using fprintf().

void stack_print(Stack *s)

```
for(uint32_ti= 0;i<s->top;i+= 1) {
        fprintf(outfile,"%s",cities[s->items[i]]);
        if(i+ 1 !=s->top) {
                fprintf(outfile,"->");
fprintf(outfile,"\n");
```

Tsp.c

Main

```
Loop through get opt
        switch(opt)
                Case h, print out help options
                Case V, Enables or disables verbose printing
                Case u, specifies if directed or not.
                Case i, specifies in file
                Case o, specifies outfile

        Scan first line, error if number greater than vertices. Set vertices = number
        Read next vertices lines from file
```

Save name of cities to array
Create graph g, directed or not depending on U
Scan input and save edges to graph g, if input malformed, report error
Create 2 paths, one for current, and one for shortest
Start from origin vertex, perform depth first search
After search, print out shortest path found

Design history:

- 4/28 first design made
- 4/29 slight changes made to design
- 5/2 Updated pseudo code

The main thing I learned in this lab was how Depth First search worked. Working with a recursive function was also very eye opening. If I had to say, the hardest part of this lab was coding the DFS function. Everything else was similar to assgn3 in code structure.