Assignment 4: Perambulations of Denver Long

Aditya Mahajan

Professor Long

Purpose:

Create 3 different functions and files named Graph, Stacks, and Path. The Graph function will create the graph and allow the user to create a matrix with the sides and lengths of the functions. The Stacks function on the other hand will create a system to backtrack and remove steps of the path. The last function Path was made to create the path and follow it and alter it as you go.

Structures:

Graph: create and alter a matrix with the vertices and sides.
Stack: allows you to backtrack and remove steps for the path
Path: Create a path and alter it as one goes through the paths

Pseudo:

Graph:

Struct:

Define undirected
Define vertices
Define visited list
Define matrix for the graphs

Graph_create:

```
Graph *graph_create(uint32_t vertices, bool undirected) {
    Graph *G = (Graph *)calloc(1, sizeof(Graph));
    G->vertices = vertices;
    G->undirected = undirected;
    return G;
}
```

Graph_delete:

Free storage of G
G = null

Graph_vertices:

Return vertices

Graph_add_edge:

Graph G[i][j] = side length

```
If graph is undirected:
    Then the Graph G[j][i]  also equals side length
```

Graph_has_edge:

```
Check if i and j is the bounds:
If its true:
    If g[i][j] greater than 0
        Return true:
    Else
        Return false
Else:
    Return false
```

Graph_edge_weight:

```
Check if i and j is the bounds:
If its true:
    If g[i][j] greater than 0
        Return g[i][j]:
    Else
        Return 0
Else:
    Return 0
```

Graph_visited:

```
If g -> vertices[v]:
    Return = true;
else :
    Return = false;
```

Graph_mark_visited:

```
If v < vertices:
    G -> vertices[v] = true;
```

Graph_mark_unvisited:

```
If v < vertices:
    G -> vertices[v] = false;
```

Graph_print:

```
For i and j in g -> matrix:
    Print matrix[i][j]
```

Stacks:

Struct:

Define top

Define Capacity

Define *items

***Stack_create**

```c
Stack *stack_create(uint32_t capacity) {
    Stack *s = (Stack *) malloc(sizeof(Stack));
    if (s) {
        s->top = 0;
        s->capacity = capacity;
        s->items = (uint32_t *) calloc(capacity, sizeof(uint32_t));
        if (!s->items) {
            free(s);
            s = NULL;
        }
    }
    return s;
}
```

Stack_delete

```c
void stack_delete(Stack **s) {
    if (*s && (*s)->items) {
        free((*s)->items);
        free(*s);
        *s = NULL;
    }
    return;
}
```

Stack_empty

Return top==0

Stack_full

Return top == capacity

Stack_size

Return top

Stack_push

If top equal to capacity

Return false

Items[top]=1

top += 1

Return true

Stack_peek

*x = s->items[s->top];

Stack_pop

If top = 0

Return false

Top -= 1

*x=items[top]

Return true

Stack_copy

For item in items

stack_push(src items, item)

Dst top = src top

Stack_print:

```c
void stack_print(Stack *s, FILE *outfile, char *cities[]) {
    for (uint32_t i = 0; i < s->top; i += 1) {
        fprintf(outfile, "%s", cities[s->items[i]]);
        if (i + 1 != s->top) {
            fprintf(outfile, " -> ");
        }
    }
    fprintf(outfile, "\n");
}
```

Path

Struct

Define *vertices

Define length

Path_create

Path p = malloc(size p)

P length = 0

Stack vertices = stack create vertices

If memory not dedicated

Then free()

Path_delete

If p-> vertices exists

stack_delete(vertices)

free()

*p = null

Path_push_vertex

stack_push(p-> vertices, v)

stack_push(p-> vertices, x)

g-> length += graph_edge_weight(g,x,v)

Path_pop_vertex

popped = stack_pop(p-> vertices, v)
Peeked = stack_peek(p-> vertices, x)
g-> length += graph_edge_weight(g,x,v)
If both popped and peeked
    Then return true
Else return false

Path_vertices

Return stack size(vertices)

Path_length

Return p-> length

Path_copy

Stack copy(dst,src)
dst->length = src-> length

Path_print

stack_print(p, outfile, cities[])