Design Document Assignment 4

Abhinav Prasanna

October 21, 2021

Introduction

The core concept of this lab is to perform depth first search using a stack data structure and a path structure to save all the paths for our graph. The stack consists of top, capacity, and the array of items. The top variable has the top variable in the stack and the capacity variable has the the capacity of the stack. The path consists of two variables which is the vertices using our Stack variable to all find the possible vertices for each path and the size of the path. The graph consists of 4 variables which is the vertices saved, determining if the direction is directed or undirected, a boolean array checking if the vertix is visited and a 2D matrix for our graph. Our goal of this lab is to conduct a depth first search to find the most efficient path for our graph.

Graph

The graph data structure consists of the vertices that are saved, the direction if it is directed or undirected, a boolean array checking if the vertix is visited and the 2D matrix for the graph. We must code the methods specified in the PDF which are graph create, graph delete, graph vertices, graph add edge, graph has edge, graph edge weight, graph visited, graph mark visited, and graph print.

Pseudocode

```
graph_create(vertices,undirected){
G = calloc(1,sizeof(data type))
 if G exists{
 G.vertices = vertices;
 G.undirected = undirected;
 for(int row; row<vertices;row++){
     indexvisited from index row is false
     for(int col; col<vertices; col++){
     2Dmatrix from index row and index column equals 0
     }
  }
 }
}
graph_delete(Graph G){
free(G)
}
graph_vertices(Graph G){
return vertices
}
graph_add_edge(Graph G, int row, int col, int value){
 if(G.row < G.vertices && G.col < G.vertices){
    2DMatrix from row and col  = value;
 }
}
graph_has_edge(Graph G, int row, int col){
 if(int row and col are in bounds with Graph G){
    if(Matrix from int row and int col are greater than 0){
      return true;
    }
 }
 return false;
}
graph_edge_weight(Graph G, int row, int col){
val;
 if(row and col are in bounds){
  val = G from row and col index;
 }
 return val;
}
```

Path

The Path data structure consists of vertices and the integer length which is the size of the path. We need to have accessor methods for our path to manipulate the path data structure and a constructor.

Pseudocode

```
Path pathcreate(void){
path = calloc(size)
 if(path){
 path.Vertices = stack_create(VERTICES+1)
 path.size = 0
 }
 return path
}
void path_delete(Path p){
 if(p){
 stack_delete(&p.Vertices)
 free(p)
 p=NULL
 }
}
bool path_push_vertex(Path[] p, int v, Graph g){
 int top = 0
 stack_peek(p.Vertices, top);
 if(stack_push(p.Vertices,v)){
  if(top!=v){
   p.size += graph_edge_weight(G,top,v)
  }
  return true
 }
 return false
}
bool path_pop_vertex(Path[] p, int[] v,Graph g){
 if(stack_pop(p.Vertices,v)){
  int top = 0
  stack_peek(p.Vertices,top)
  if(top!=v){
   p.size -= graph_edge_weight(G,top,v);
  }
  return true
 }
 return false
}
int path_vertices(Path[] p){
 return stack_size(p.Vertices)
}
```

Stack

Stack data structure consists of the top integer on the stack, the capacity of the stack, and the array of items in the stack. We need to have accessor methods for our path to manipulate the stack data structure and a constructor.

Pseudocode

```
Stack stack_create(int capacity){
Stack stack = calloc(size)
 if(stack){
 stack.top = 0
 stack.capacity = capacity
 stack.items = new items[capacity]
 if(!stack.items){
  free(stack)
  stack = NULL
 }
 }
 return stack;
}
void stack_delete(Stack[][] s){
 if(s[] != NULL && s[].items == NULL){
  free(s[].items);
  free(S[])
  s[] = NULL;
 }
}
void stack_size(Stack[] s){
 return s.top;
}
void stack_empty(Stack[] s){
 if(s.top == 0){
  return true;
 }
 return false;
}
void stack_full(Stack[] s){
 if(s.top == s.capacity){
  return true;
 }
 return false;
}
bool stack_push(Stack[] s, int x){
 if(stack_full(s)==false){
  s.items[s.top]=x
  s.top++
  return true
 }
 return false
}
bool stack_pop(Stack[] s, int x){
 if(stack_empty(s) == false){
  s.top -= 1
  *x = s.items[s.top]
  s.items[s.top]=0
  return true
 }
 return false
}
bool stack_peek(Stack[] s, int x){
 if(stack_empty(s) == false){
  *x = s.items[s.top - 1]
   return true
 }
```

Testing Harness

We need to implement a testing harness with the different flag lines to test out different parts of the files. We must test the graphs with directions, stacks, and paths. We need to also implement DFS in our testing harness.

Pseudocode

```
dfs(){
stack = new Stack()
      stack.push( s )            //Push s to stack
      mark s as visited
      while ( stack is not empty):
            //Pop node from stack and start to visit its children
            v  =  stack.pop()

            if(v == key) return true //We found the key

            //Push all the unvisited neighbours of v to stack
            for all neighbours w of v in Graph G:
                //unvisited neighbors
                if w is not visited :
                     stack.push( w )
                     mark w as visited
      return false     // If it reaches here, then all nodes have been explored
                       //and we still havent found the key.
}
```