

DESIGN.pdf  
 Asgn4\_ Circumnavigations of Denver Long

The Matrix:

This is an  $n \times n$  adjacency matrix  $M$ .  
 The matrix will only be positive numbers. Within this matrix there will be Edges( $K$ ) shown highlighted in yellow.

If  $M_{x,y} = K$  and  $1 \leq x \leq y \leq n$   
 Then that's a **directed edge**.

	0	1	2	3	4	5	...	25
0	0	10	0	0	0	0	0	0
1	0	0	2	5	0	0	0	0
2	0	0	0	0	0	3	0	5
3	0	0	0	0	21	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
$\vdots$	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	0	0

Edges will be represented as a triplet.  $\langle x, y, n \rangle$

$$E = \{\langle 0, 1, 10 \rangle, \langle 1, 2, 2 \rangle, \langle 1, 3, 5 \rangle, \langle 2, 5, 3 \rangle, \langle 2, 25, 5 \rangle, \langle 3, 4, 21 \rangle\}.$$

In C thinking, this will be a struct. In C we can't explicitly make a tuple inside of an array so we create a struct.

The Graph Struct and helper functions has a "Graph add Edge" function that remembers the  $x, y$  and  $k$ . But I will keep this here as it's still my thinking process.

Pseudocode for the Struct:

```
{
  Int x; //This holds x value
  Int y; //This holds the y value
  Int k; //This holds the edge
} points;
```

Definitions:

EDGE =  $\langle \text{Row, Column, Weight} \rangle$

Vertex = Cities

## Creating the Graph

Now we have this stack, we're able to store the Graph and remember the data of all the edges and coordinates. Now we have to think of these values as a graph, which means making another stack.

This is Dr.

Long's code >

```
1 struct Graph {  
2     uint32_t vertices;           // Number of vertices  
3     bool undirected;            // Undirected graph?  
4     bool visited[VERTICES];     // Where have we gone  
5     uint32_t matrix[VERTICES][VERTICES]; // Adjacency matrix.  
6 };
```

As we have this graph, we need several helper functions to allow us to remember things like whether or not we've checked a point, whether or not it has an edge and so on. The following list are helper functions for graph. I did not invent these functions, they are based off of the Asgn4 Documentation.

- Graph Create
  - Create a Graph, and set all of the x,y axis to false, Make sure Visited? Is false, and make sure # of array is equal to amount of elements.
- Graph Delete
  - Destroy Graph. No memory leaks here
- Graph Vertices
  - Return the # of vertices in the graph
- Graph add Edge
  - Add an edge with Graph, x,y,k as a parameter. Return True if it's possible, False otherwise.
- Graph has edge?
  - Return true if x,y exists and there exists an edge at that x,y spot.
- Graph edge weight
  - Return the weight of the edge from X to Y.
  - So just get the weight of x,y
- Graph visited?
  - Return true if a point has been visited
- Graph mark visited
  - Mark a point as visited
  - To get a point that wasn't visited, just add "!" when using it
- Graph Mark unvisited
  - Mark as unvisited
  - This will be necessary in the recursive function to test all the values.
- Graph print
  - Tester function to make sure Graphs work as expected

## DFS RECURSION

Cool, We have a structure for the graph. Now what?

We need a way to search through the graph and find the best and cheapest way to get to where we need to go. We're going to use "Depth-first search" (DFS) This marks the first vertex as visited, then recursively works through all the non visited edges.

```
1 procedure DFS(G,v):
2   label v as visited
3   for all edges from v to w in G.adjacentEdges(v) do
4     if vertex w is not labeled as visited then
5       recursively call DFS(G,w)
6   label v as unvisited
```

Finding a Hamiltonian path then reduces to:

1. Using DFS to find paths that pass through all vertices, and
2. There is an edge from the last vertex to the first. **The solutions to the Traveling Salesman Problem are then the shortest found Hamiltonian paths.**

*Professor Long's Code*

What's going on in this pseudocode?

This is a recursive algorithm that FIRST labels the arguments as visited, and then iterates through all of the reachable and non-visited vertices. Eventually this function will come out with SEVERAL paths.

**My implementation** of this algorithm in more words:

- Procedure DFS(Graph \*G, uint32\_t V, Path \*c, Path \*s)
  - The arguments are as follows: Graph, vertex, Current path and short path. The current path holds the current path we're working on, and the short path holds the SHORTEST path we've encountered. This will help us remember which path is shortest.
  - Graph mark visited
  - If the current path is < shortest path:
    - Copy current to shortest.
  - For All edges (w=0 ;w < graph\_vertices(G); w++)
    - If graph has edge and isn't visited then:
      - Push w onto the path
      - Recursive call DFS
      - Pop the vertex
  - Mark point V as unvisited.

## PATH

The usage of pushing, recurs, then popping tests all of the values from one point, then removes it. While doing this, if the shortest path is updated then great. Otherwise, it wasn't the shortest path.

Where will these paths go? How do we mark a "path" as searched or the total length of the path? We will need another ADT!

Following Psuedo and idea for functions are from Asgn4 doc

```
struct Path {  
    Stack *vertices; // The vertices comprising the path.  
    uint32_t length; // The total length of the path.  
};
```

- Path create
  - Constructor function for path that allows vertices to hold X amount of vertices
  - Initialize length to 0
  - Length field (Length of paths)
- Path Destroy
  - Destructor function to make sure we have no memory leaks
- Push Vertex
  - Pushes a vertex onto a path. The length of the path changes and goes up with a vertex's edge weight.
  - This will be a STACK so therefore push
  - We receive as parameter an V. We peek into the stack, grab that value and the Length is increased by the Peeked value and the V.
  -
- Pop vertex
  - The exact opposite of Push Vertex. This will remove a vertex/pop then pass it back through the pointer argument.
  - Stack format
  - We pop from the stack, remember that value, then peek at the next value. We get the edge weight of that then remove that value from the length.
- # of Path's vertices
  - Returns # of vertices in the path. This function allows us to run recursive calls and check amount of vertices
- Length of path
  - Returns the length of the path. This function will let us know which path has the shortest length.
- Path copy
  - Why would a path need a copy function? I thought about this for a while (trying to not read the documentation for an answer). We need a path

copy function to copy the path from one stack to another. Specifically, used in the recursive function.

- Path print
  - Of course, we'll need a debug function to print out a path. Either we can debug or directly print whatever we need to from here.

If we imagine how these Paths will recursively work. We can imagine a STACK of paper.

### STACK

I place down one paper (signaling visited), then another. Now I have two papers I could put on at the same time. I'll just place down the first piece of paper I saw. Unknowingly, this piece of paper is bad, so I take it off the TOP, and place the other paper down and continue doing my business. Keeping what's updated, which is the length.

This type of example I made ^, works like how the recursive function will. It'll keep testing a path, adding items onto the stack. If it's a bad path then it takes itself off the stack one by one.

So we're going to need a... STTACCCKKK struct and helper functions. **This will be similar to asgn3**, in terms of helper functions, and different in usage. So I won't be going too much into detail.

The Stack Struct will look like, DIFFERENCE only being stack peek, copy and print.

- Create
- Delete
- Stack empty
- Stack full
- Stack size
- Stack push
- Stack peek
  - Stack peek is similar to push, except we won't be removing any values. We will only be looking at the top of the stack, specifically one non-zero variable.
- Stack pop
- Stack copy
  - We need to loop over the amount of vertices, and copy each index of src to dst.
  - Then we update the top of the stack value.
- Stack print

This is from the asgn4 Documentation, and is fairly clear looping and printing.

```
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, " -> ");
        }
    }
}
```

## TESTING HARNESS

Implementing all of these things is one thing, but let's setup a TESTING HARNESSSS.

Command line arguments;

- **-h**: Prints out a help message describing the purpose of the graph and the command-line options it accepts, exiting the program afterwards. Refer to the reference program in the resources repo for an idea of what to print.
- **-v**: Enables verbose printing. If enabled, the program prints out *all* Hamiltonian paths found as well as the total number of recursive calls to `dfs()`.
- **-u**: Specifies the graph to be undirected.
- i infile**: Specify the input file path containing the cities and edges of a graph. If not specified, the default input should be set as `stdin`.
- o outfile**: Specify the output file path to print to. If not specified, the default output should be set as `stdout`.

Above are all of the command line arguments.

- h prints usage
- v allows for verbose printing that prints out all hamiltonian paths.
- u sets for undirected or directed
- i so the tsp executable knows which file to read the graph from.
- o if we want to print the output in a file rather than stdout.

The DFS recursive function will be placed in the testing harness for simplicity of separating .c files to do one specific thing. We don't want to add the recursive function into the path/ stack when those files only have one specific purpose.