Derick Pan

DESIGN.pdf
Asgn4_ Circumnavigations of Denver Long

The Matrix:

This is an n*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≤ x ≤ y ≤ n
Then that's a directed edge.

$$\begin{array}{c|cccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & \cdots & 25 \\
\hline
0 & 0 & 10 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 2 & 5 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 5 \\
3 & 0 & 0 & 0 & 0 & 21 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\vdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
25 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Edges will be represented as a triplet. <x,y,n>

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

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.

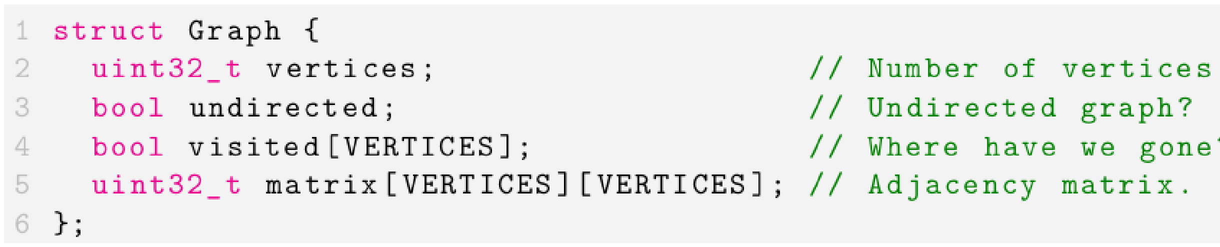
The Graph Struct and helper functions has a "Graph add Edge" function that remembers the x,y and k. But I will keep this here as it's still my thinking process.

Pseudocode for the Struct:

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

Creating the Graph

Now we have this stack, we’re able to store aCreating the GraphCreating the GraphCreating the Graphnd 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.



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

This is Dr. Long’s code >

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.

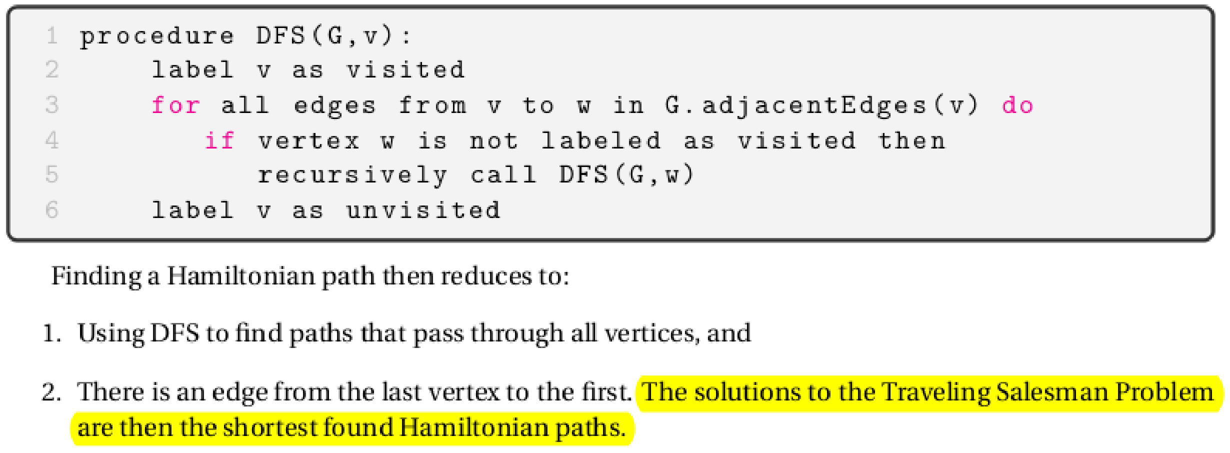
- Graph Create
  - Create a Graph, and set all of the x,y axis to false, Make sure Visited? Is false, and make sure # of array is equal to amount of elements.
- Graph Delete
  - Destroy Graph. No memory leaks here
- Graph Vertices
  - Return the # of vertices in the graph
- Graph add Edge
  - Add an 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.
  - What is “weight”??
  -
- Graph visited?
  - Return true if a point has been visited
- Graph mark visited
  - Mark a point as visited
  -

- Graph Mark unvisited
    - Mark as unvisited
- Graph print
    - Tester function to make sure thinks work as expected.

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.



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

Finding a Hamiltonian path then reduces to:

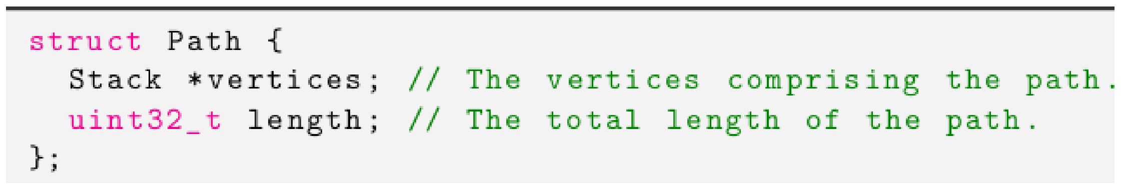
1. Using DFS to find paths that pass through all vertices, and
2. There is an edge from the last vertex to the first. The solutions to the Traveling Salesman Problem are then the shortest found Hamiltonian paths.

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. 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
  - Initalize 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
- Pop vertex

    - The exact opposite of Push Vertex. This will remove a vertex/pop then pass it back through the pointer argument.
    - Stack format
- # 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.
- 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.

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.

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 :

Helper functions are the usual

- Create
- Delete
- Stack empty
- Stack full
- Stack size

- Stack push
- Stack peek
- Stack pop
- Stack copy
- Stack print

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

Command line arguments;

-h

- `-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

`-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.

-u

-i

`-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

`-o outfile`: Specify the output file path to print to. If not specified, the default output should be set as `stdout`.