Shreyas Telkar
05/03/2021
CSE13S
Professor Long

CSE 13s Assignment 4: TSP (Traveling Salesman Problem)

Description:
This assignment finds the shortest hamiltonian path given vertices that are kept track graph structure and a weight also saved in the n by n matrix which has indices for the path between vertices. To go through all the vertices once in the hamiltonian path and return back to the origin, dfs (depth first search) to go to unvisited vertices and back track if the origin is not reached. Directed paths go one way and have 1 weight between two vertices. Undirected paths can go the other way and the weight can be different.

For undirected paths, (From vertex 4 to 5 there is an edge of length 2)

If matrix[4][5] = 2

Then there should be matrix[5][4] = 2

In the dfs, *vertices* will be popped or pushed to the stack to compare edge pairs to get the shortest pair. This is discussed in the doc later.

The path file will be used above the stack and will be similar to copy the path between vertices if it is a hamiltonian path.

Graph.c

Struct:

Takes in parameters- vertices, and undirected specified by file and optarg options
Defines vertices, bool undirected, visited[], and matrix[][]

Visited keeps track if the vertex is visited by storing a true or false at that matrix's vertex.
Vertices is the number of vertices and undirected is whether the graph is directed or undirected.
Matrix keeps track of the weight given two vertices.

Functions:

Visited[] is how graph keeps track of visited functions.

For edge functions, it uses matrix[][] to store the weight at the vertices.
Valid edges are if matrix[i][j] > 0 since all matrix values are initialized to 0.

If undirected is passed as true (if received from OPTIONS), then it will add the edge at the diagonal: matrix[j][i] = k. and where specified at matrix[i][j] = k. This will result in different output in path because vertices are connected differently.

Tip: edges can be overwritten when vertices are repeated and have different weights.

Visited[i] for i in range(vertices) is set as false.

Usage:
To mark a vertex visited or unvisited, use visited[v] = true or false.

Get vertices and edges from the input file, graph_add_edge(i, j, k) to the matrix.

These will be used to determine if there is a path from one vertex to another. The path structure will increment the weights along paths if it is added. Vice Versa when removed.

Stack.c
Struct:
Same structure as assignment 3. Used primarily in path.c for keeping track of vertices on the path.

Stack peek - Gets the top of the stack without popping it. This is done by decrementing the stack, getting the value, and then incrementing the stack.

Uses the top of the stack to store vertices. Make sure when popping or pushing the capacity bounds are not reached. Path then uses these values to make sure to get the shortest path. Do not need to push origin to stack when dfs.

Path.c
The length of the path is increased when the vertex is pushed onto the stack. This is done by peeking the stack and then adding to the current length if it is unvisited.

```
Path_add_vert(curr):
        stack_peek(&last)
        Length += weight(last, curr)
        Stack push
```

If the stack is empty, then use the macro START_VERTEX and use that as the last value.

```
Path_rm_vert(curr):
        Stack pop()
        Stack peek()
        Decrement length using values from pop and peek
```

**This is used when backtracking in dfs when a vertex cannot find a pair.

path_copy():

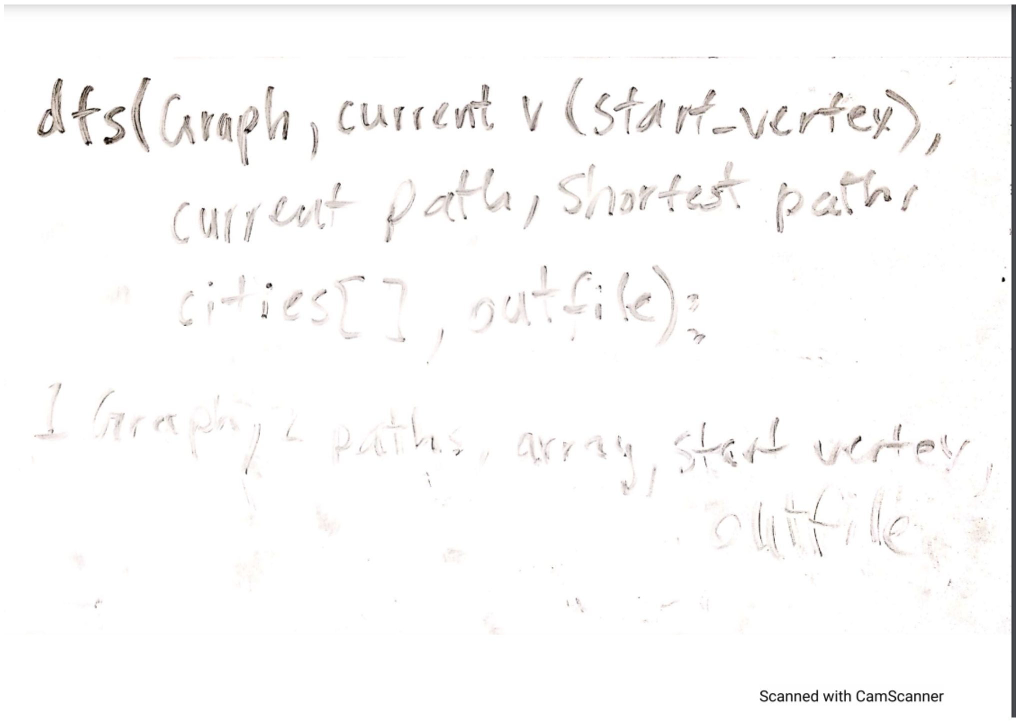
    Sets contents of one stack to another
    Set path length to dst path.

**This is used in dfs to copy the current path to the shortest one when it is found.

Path_print():
Calls stack print to print the contents of the path of vertices.

Path uses the stack a lot to keep track of the vertices on the path. While they are on the path, it keeps track of the total distance using the graph weight between two vertices. Then it compares these lengths once a hamiltonian path is reached. If it is the shortest path, i.e. the total weight is shorter than the previous shortest path, then store it and print it if necessary. Dfs will use path functions to print out the path.

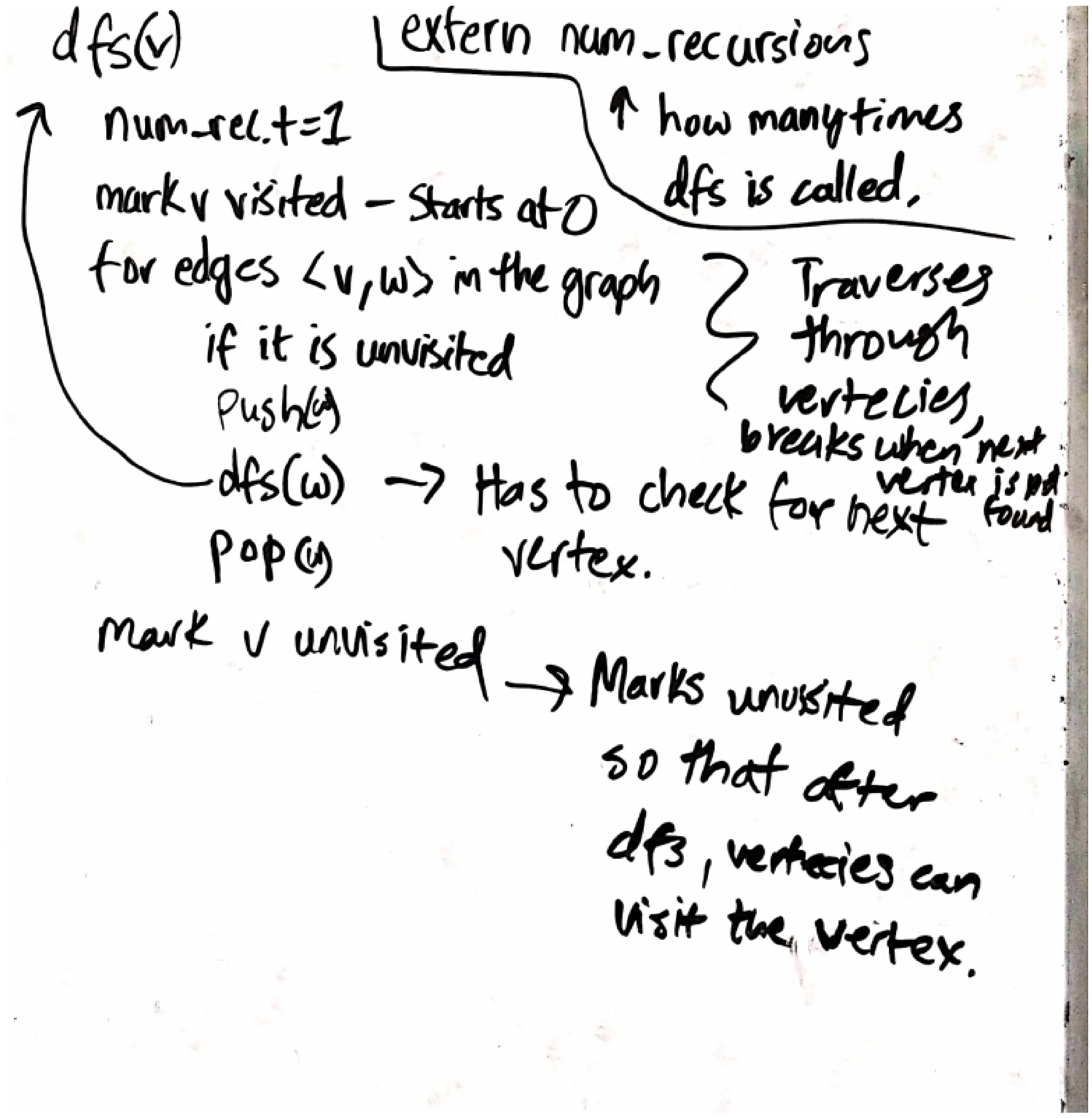
Dfs
First push origin vertex onto path.
If it reaches an end after the loop, pop off the last incremented path value and decrease the length when it back-tracks.
Definition:



```
dfs(Graph, current v (start_vertex),
    current path, Shortest path,
    cities[], outfile):

1 Graph, 2 paths, array, start vertex,
                          outfile
```

Scanned with CamScanner



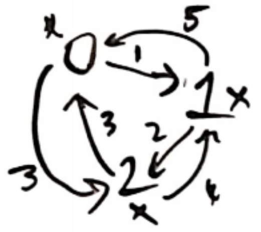
```
dfs(v)                              extern num_recursions
                                    ↑ how many times
                                      dfs is called.
  num_rec+=1
  mark v visited - Starts at 0
  for edges <v,w> in the graph      } Traverses
      if it is unvisited            } through
      Push(w)                       } vertecies,
                                      breaks when next
      dfs(w) → Has to check for next   vertex is not
  pop(w)          vertex.              found
  mark v unvisited → Marks unvisited
                     so that after
                     dfs, vertecies can
                     visit the vertex.
```

If hamiltonian path is found, then copy it.

If another hamiltonian path is found, then compare length to see if it is shorter then copy it.

Dfs example:

Ex) UCSC graph



Start at 0, mak visited

go to 1, push onto stack

mark visi.

$\boxed{1}$

go to 2

$\boxed{\begin{matrix}1\\2\end{matrix}}$

Cant find next vert, check if hamil path,

yes, path_shortest = 0 → 1 → 2

backtrack,

go from 0 → 2

$\boxed{2}$

go to 1

hamil. ? = Yes

Not the shortest!

∴ shortest

Get-opt and reading input files

Use get-opt to parse between arguments.

-h prints the help message, need to return 1 after print.

-v prints out the shortest path every time it is copied in dfs. The bool parameter is passed to dfs and prints if the option is specified.

-u, sets undirected as true, used in graph when adding edges to the matrix.

-i specifies the input file:

Initialize as FILE *input = stdin - default stdin
In getopt set input as fileopen(optarg), optarg is the file typed in.

-o specifies the output file:
Use the same method as input but set default to stdout.

**The output will only be used as a parameter to the dfs because dfs does the printing. Otherwise we need to specify fprintf(output).

To read ints:
Num_vertices:

```
Use while (fscanf does not reach the end of file):
    If the output of fscanf is 1:
        Save it to vertices variable to use in loops and graph parameter.
```

To read cities:
Use a buffer value when reading city string arguments. fgets is used to store the strings in the buffer. strdup from the buffer, where the strings are stored and loop through and set them to the contents of an array of cities. Make sure to get rid of the newline.

To read graph:
Use fscanf similar to the vertex, but graph_add_edge() when the output of fscanf is 3 until the end of file. Fscanf should expect 3 arguments because of how it is written in the input file.

**For all of these specify the input variable input_file because that is where it is reading the values from.

Make sure to create a graph and two paths after getting values for the vertices. Before dfs, add the edges.

Make sure print format is correct, if needed print the shortest path again after dfs.

If there are errors in the input file format, print malformed line. If there is no where to traverse through the graph, print there is no where to go.

Errors and memory:
I was stuck for a while because I did not free the array for cities after dfs. This results in segment errors. If run into this, use gdb to find where it went wrong. Also use valgrind to check if there are memory leaks. Different graphs take longer or shorter based on the number of inputs, it will take a long time if there are many inputs and may even timeout. My implementation will probably timeout for high number of inputs because my recursive count is not perfect and it will continue to look for a shorter path even if it encounters a larger path.