Fabrice Kurmann
fkurmann@ucsc.edu
25 April, 2021

CSE 13S Spring 2021
Assignment 4: The Circumnavigations of Denver Long
Design Document

Program Flow:

Begin by scanning command line arguments. (Command line arts will be in the form of formatted files)

1st line: Number of cities - will be represented by vertices in the graph
Check that vertices is wishing specified range

2nd line block: Names of cities - each name is saved to its own array.

3rd line block: Scan edges from the input line and add them to the newly created graph, G.

Begin the search process for the shortest path.

Create 2 paths, one to store shortest yet, one to store current.

Perform your depth first scan on graph G.

Report results.

Print out the shortest path and it's length. Also print how many depth first calls were needed.

Verbose option means print all paths you find, not just shortest.

Command line Options:

-h for help message
-v for verbose printing (print all paths, not just the shortest)
-u for undirected graph
-i (INPUT) specifies on of the graphs as an input file
-o (OUTPUT) specifies an output

The Stack:

Functions to implement:

stack_create
stack_delete
stack_empty - returns Boolean of emptiness status
stack_full - returns Boolean of fullness status
stack_size - returns stack size
stack_push - if stack full return false
stack_pop - if stack empty return false
stack_print - print stack to outfile

stack_peek - observe top element - return false if empty
stack_copy - make a stack with the same contents and top value - the destination stack must already be initialized before this is run

Representing the map with a Graph ADT:

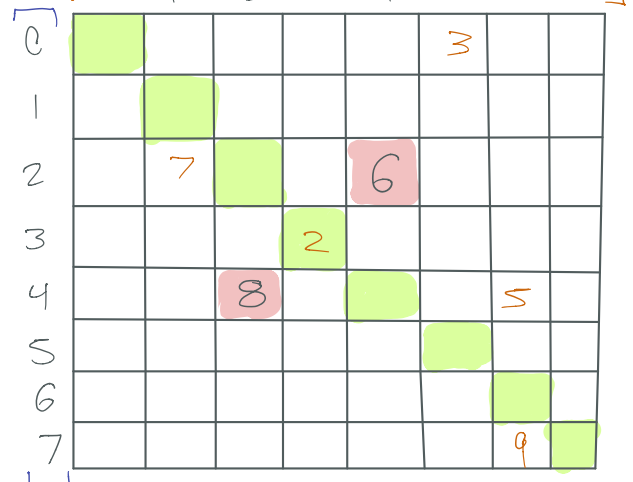
Functions to implement:

- graph_create
- graph_delete
- graph_vertices - return number of vertices in graph
- graph_add_edge - adds an edge (or edge pair if undirected) to graph
- graph_has_edge - check if edge exists (has positive weight value)
- graph_edge_weight - return edge weight - if edge exists
- graph_visited - returns Boolean of visited status of particular vertex
- graph_mark_visited - mark vertex as visited
- graph_mark unvisited - mark vertex as unvisited
- graph_print

(Matrix Examples)

Directed

i: indicates departing vertex

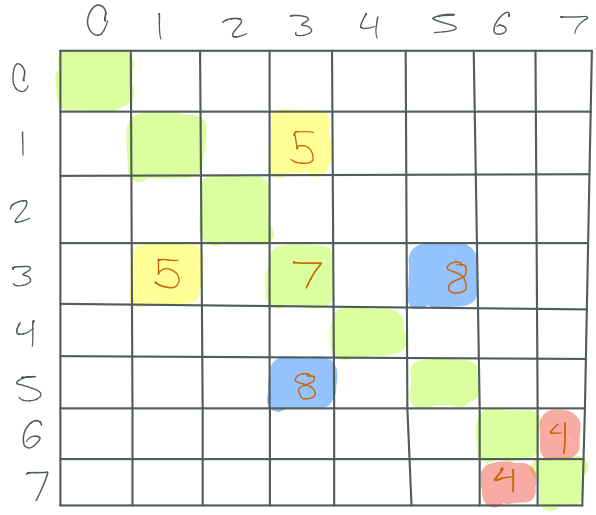


j: indicates arrival vertex

(Blank = 0)

- unequal weight for opp directions is legal

Undirected



(Blank = 0)

Based on example situations make sure to implement highlighted checks

add edge (6, 7, 4)

(check if 6,7 + 7,6 blank

add 6,7 + 7,6

add edge (3, 3, 7)

(check if 3 = 3 (i = j)

(is so add only once

Visited array.

- length = matrix dimension

↳ VERTICES constant

Observation:

** If no edges Enter/Exit a vertex in your path → can return false

Representing paths with a Path ADT:

Functions to implement:

- path_create - vertices will be a STACK of size VERTICES - represents essential stops
- path_delete
- path_push_vertex - push vertex onto path, increase length of path be edge weight*
- path_pop_vertex - pop vertex, decrease the path length accordingly by edge weight*

] found on Graph Matrix

- path_vertices - return number of vertices
- path_length - return length (THIS CONSIDERS EDGE WEIGHT)
- path_copy - make a copy assuming destination is initialized
- path_print - print to outfile

Implementing the Depth First Search:

Basic process:

Mark vertex v as having been visited — vertex is a stack element of path; visited is stored in graph's visited array

Iterate through all edges departing from vertex v

If an edge destination, vertex w, has not been visited, recursively call depth first search on that.

Finishing steps:

Once a final vertex has been found (recursive step no longer happens since all vertices have been visited), path is complete.

Narrow down paths to those that have an edge from the final vertex back to the start

Lastly, given the remaining paths, *pick the shortest*** - that's the answer.

***Shortest path is an argument to the recursive function, it must therefore be determined inside the function.*

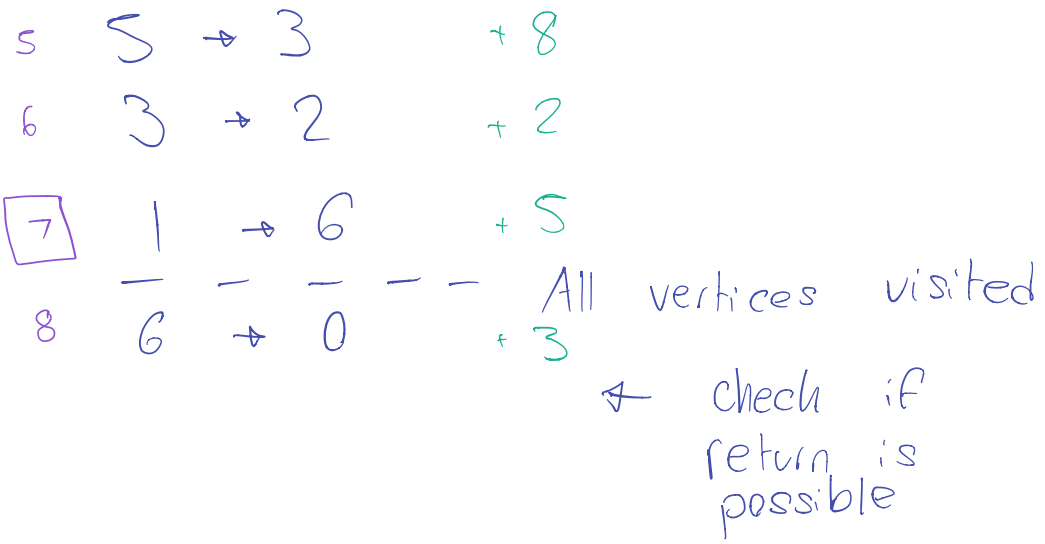
Pseudocode:

Begin with default starting vertex (0,0)

```
dfs(Graph, vertex*, current_path, shortest_path, outfile):
                                  (path objects)
    if visited +1 = vertices
        if current stack is shorter than shortest stack
            shortest stack = current stack
            if verbose printing
                print shortest stack to outfile

    visited[vertex] = true——Consider how to quantify vertices for the visited array
    current path.add(vertex)

    if currentStack is shorter than shortest stack**

    for i in range 0-VERTICES
        if matrix [departing vertex = v] [i] != 0 ——Then there is a path that departs from here
            if vertices[matrix[departing vertex][i] == false: ——Destination has not been visited
                dfs(Graph, matrix [departing vertex = v] [i], current path, shortest path)

    //End of recursive section

    visited[vertex] = false

    return
```

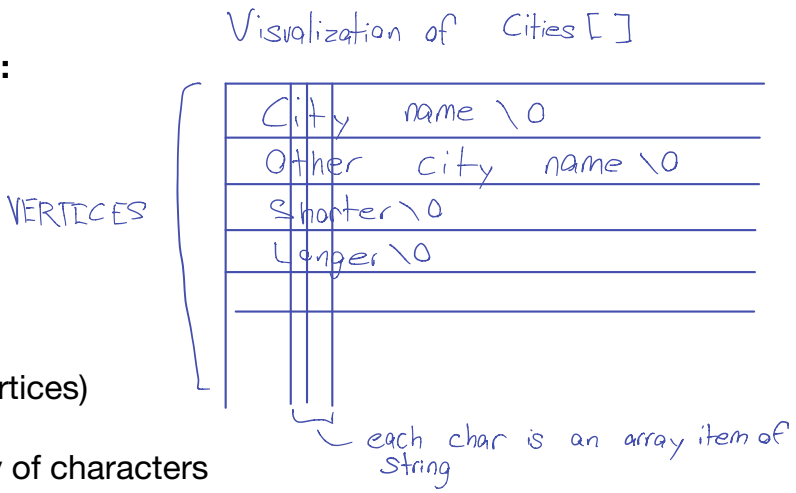
Example: Solar System has 1 path:

path verts:		path length
1	0 → 7	+1
2	7 → 2	+3
3	2 → 4	+2
4	4 → 5	+6



When the number of vertices visited = graph(vertices -1) then check the final vertex.

**Use this short circuit logic to reduce the number of useless recursive calls.



Implementing the Command Line/File reading:

Information to store:
- undirected (default = false)
- verbose printing (default = false)
- outfile (default = stdout)
- infile (default = stdin)

- Vertices (integer to store number of vertices)

- Cities (array of strings) = Array of array of characters
- Coordinates (array of integers)
  - coordinate index (integer to track position in coordinate array)

Pseudocode:

Switch :
- H - print help
- u - undirected = true
- v verbose printing = true

- o
  - outfile = optarg
  - **check if there is an output file of that name, report error otherwise

- i
  - infile = optarg
  - read infile:

  - line 1 = vertices

  - lines 2 -> 2+vertices
    - each line is a city
    - cities[i] = line [i]

**Dynamically allocate secondary arrays in cities, remove end of line character*
lines 2+vertices -> end
read individual integer values into Coordinates

End Switch.
***Loop through matrix of graph, setting all edges to 0*
for i in coordinates
Graph_add_edge(i, i +1, i+2)
if undireced
Graph_add_edge(i+i, i, i+2)

* Coordinats:
multiples of 0 + 3n = i
multiples of 1 + 3n = j
multiples of 2 + 3n = K

CLOSE ALL FILES (read and write files)
Clear all ADTs (graph, current path, shortest path)

Unexpected items in main method implementation:

Larger Matrices (such as that of Texas) contained garbage values in certain indexes. To avoid this, I had to integrate the loop before adding edges to clear these values to 0.

The fgets method I used to read a file included an unnecessary \n at the end of strings. To remove this I simply truncated every string by one before adding it the the cities array.

Arrays inside the city arrray are best allocated dynamically so the declaration in the main method must not contain them. If this were to be the case, they would have to be passed as declared, causing a conflict with the header files.