Ruchit Patel

CSE13S, Spring 2021

Prof. Darrell Long

26 April 2021

Assignment 4: The Circumnavigations of Denver Long

Brief Description

- In this assignment, a Graph (directed or undirected) abstract data structure (ADT) is implemented.
- A depth-first search (DFS) algorithm is run on the given graph and a shortest path is returned from a given index (zero for this lab) to itself.
- The graphs are either inputted from a file or from stdin stream.
- Arguments: -h (prints the help message),
  -v (enables verbose printing),
  -u (to use undirected graphs),
  -i (input file),
  -o (output file).

Descriptions/Implementation

Argument Parsing

- Arguments are parsed using getopt GNU utility.
- If a valid argument is encountered, an integer is added to an abstract data type Set (a bit mask).
- For example, if -a is encountered, inside the case for -a, an integer value in an enum is added to the Set.

  Pseudocode:

  define flags

  while (getopt returns valid)

  switch to the argument case if valid

add the flag to Set

if invalid print usage message and return

Graph

- The graph is implemented using a matrix. Each index of the matrix represents a vertex and each entry at <row, col> of the matrix has a weight.
- The weight represents how "far" a vertex <col> is from the vertex <row>.
- The graph can be either directed or undirected depending on the constructor parameters.

- Structure/Members of Graph:
  a. Number of vertices : Total number of vertices.
  b. Boolean for undirected graph : True if undirected. False otherwise.
  c. Array for visited vertices : Indicates if vertex has been visited.
  d. Matrix/Graph : Representation of the map/graph.

- Methods associated with a graph:
  a. graph pointer graph_create(no. of vertices, boolean undirected)
     i. Set number of vertices to vertices
     ii. Set undirected graph boolean depending on argument -u
     iii. Set each element of array for visited vertices to false
     iv. Set each element of the matrix to zero

  b. graph_delete(pointer to a graph)
     i. Free the memory of stack (if present/possible).
     ii. Free the memory of the path (if present/possible).
     iii. Make the pointer pointing to path struct = null.

  c. integer graph_vertices(pointer to a graph)
     i. Return the element *number of vertices*

d. boolean graph_add_edge(pointer to a graph, row, column, weight)
    i. Make the matrix entry at [row][column] = [weight]
    ii. If the graph is undirected, make [column][row] = [weight] as well.
    iii. Mark the newly added entry or entries as unvisited.

e. boolean graph_has_edge(graph, row, column)
    i. Return true if the weight of matrix entry at [row][col] > 0.

f. integer graph_edge_weight(graph, row, column)
    i. Return value of matrix entry at [row][col].

g. boolean graph_visited(graph, vertex)
    i. Return boolean entry of graph's visited vertices array at index *vertex*.

h. graph_mark_visited(graph pointer, vertex)
    i. Set the entry of the graph's visited vertices array at index *vertex* to true.

i. graph_mark_unvisited(graph pointer, vertex)
    i. Set the entry of the graph's visited vertices array at index *vertex* to false.

j. graph_print(graph_pointer)
    i. Loop over the matrix and print each element. Print a newline after each row.

Depth First Search (DFS)

- DFS is used to find all Hamiltonian paths from a starting vertex to itself.
- By doing so, a shortest path is found and returned.
- If the verbose option is not used only the shortest path is printed. Otherwise, all paths are printed.

- Pseudocode [CREDIT: Modified version of DFS pseudocode from the documentation uploaded to GitLab]

```
dfs(vertex v):
    mark v as visited
    if all vertices have been visited:
        check is the last vertex connects to the starting vertex (Valid hamiltonian):
            update shortest path if possible:
                either shortest path has 0 length (does not exist) or
                length of current path < length of shortest path
            print current path
        if it does not connect (graph does not have edge current to start):
            return
    else if all the vertices have been visited and the last edge does not connect to first:
        return (no hamiltonian path)
    else:
        for all connected edges from v to w:
            if w has not been visited already:
                push vertex to current path
                recurse from w (call dfs(vertex w))
                pop vertex from the path
    mark v as unvisited
```

Path

- The path ADT has two members: A stack of vertices and total length of the path.
- Each path (vertex) taken is added to the stack and the length of the path is updated by the weight of each vertex in the path as it is encountered.

- Structure/Members of Graph:
   a. Stack : Stack to store new path
   b. Length : Total length of the path so far

- Methods associated with a path:
    a. path path_create()
        i. Initialize *length* to zero.
        ii. Create a stack of size VERTICES using stack_create().
        iii. Return pointer to newly created path.

    b. path_delete(path)
        i. Delete the stack using stack_delete() [if possible].
        ii. Free the memory of the current path. [if possible]
        iii. Set the pointer to path to null.

    c. boolean path_push_vertex(path, vertex, graph)
        i. If the stack is empty already, increase the length of path by weight from origin vertex to current vertex.
        ii. Else increase the length by weight of the last element on the stack to the current vertex.
        iii. Add the vertex onto the path's stack using stack_push.
        iv. Return true if it can be pushed (stack not full). False otherwise.

    d. boolean path_pop_vertex(path, vertex, graph)
        i. Pop the vertex off the top of the stack.
        ii. If the stack is empty already, decrease the length of path by weight from origin vertex to popped vertex.
        iii. Else decrease the length by the weight from the current element on the stack to the popped vertex.
        iv. Add the vertex onto the path's stack using stack_push.
        v. Return true if it can be pushed (stack not full). False otherwise.

    e. integer path_vertices(path)

i. Return stack size of path's vertices stack.

f. integer path_length(path)

i. Return *length* element of path structure.

g. path_copy(destination path, source path)

i. Assign the destination path the same length as the source path.

ii. Copy the source's stack to destination's stack using stack_copy().

h. path_print(path, output file, cities array)

i. Print the stack of the path to the output file using stack_print().

ii. Print the length of the path. (Optional)

Stack

- The stack ADT is implemented using a structure whose elements are: capacity, top, and

- Structure/Members of Stack:

a. Top	:	Index of next empty element
b. Capacity	:	Total elements a stack can hold
c. Items array	:	Array that actually stores the elements

- Stack methods implementation:

  a. stack_create(capacity)

  i. Allocate memory (dynamically) for stack struct.

  ii. Create a pointer to stack structure.

  iii. Initialize top element to 0, capacity to capacity.

  iv. Allocate memory (dynamically) for items array of size capacity.

  v. Return pointer to newly created stack.

  b. stack_delete(pointer to stack pointer)

    i. Free the memory of items array (if present/possible).
    ii. Free the memory of stack struct (if present/possible).
    iii. Make the pointer pointing to stack struct null.

c. stack_empty(stack pointer)
    i. Return true if the top element is zero, otherwise false.

d. stack_full(stack pointer)
    i. Return true if the top element is greater than the capacity of stack.

e. stack_size(stack pointer)
    i. If the stack is full then return top element value - 1, else return top.

f. stack_push(stack pointer, integer)
    i. If the stack is not full, assign the element in the items array at index top x's value.

g. stack_pop(stack pointer, integer pointer)
    i. If the stack is not empty, decrement the value of top and assign x (after dereferencing) value of element in the array at index top.

h. stack_peep(stack pointer, integer pointer)
    i. If the stack is not empty, assign x (after dereferencing) the value of the element in the array at index top - 1. [No change in value of top].

i. stack_copy(destination, source)
    i. Since we have to copy not only the elements but also the elements pointed by the *items* array, memcpy() is used to copy the memory held by the source's structure and its *items* array.

j. stack_print(stack pointer) [CREDITS: from the lab documentation]
    i. Start from the tail, loop until top+1 and print the contents [array[i]] out to a specified file(decrementing temporary each time).

Set

- A set in this assignment is just a value (bit mask) that is used to keep track of which flags are passed.
- Each bit corresponds to a flag. Upon encountering a flag, that bit is set to one to indicate the flag has been passed.
- The element of the set structure is a mask (8 bit integer).
- Set methods implementation:
    a. set_create()
        i. Allocate memory (dynamically) for set struct.
        ii. Create a pointer to set structure.
        iii. Initialize mask to 0.
        iv. Return pointer to newly created set (if possible).

    b. set_delete(pointer to set pointer)
        i. Free the memory of set structure (if present/possible).
        ii. Set the set pointer to null.

    c. is_member(set pointer, element)
        i. Return bitwise AND of the mask element of set struct with hex 1 left shifted by element (if possible).

    d. add_member(set pointer, element)
        i. Bitwise OR the mask element of the set struct with hex 1 left shifted by element (if possible).

Input/Graph Parsing

- The graphs are parsed from a stream (either a file or standard input).
- The format of the graph file/stream should be a number *n* followed by *n* city names and edges.

- Getting city names:
    1. City names are stored in an array of *n* size.
    2. As each city name is encountered, memory is allocated for each string to be saved in an array and the city name (one line or upto *N* bytes where $N < 1024$) is copied into the array at corresponding index.

- Adding edges:
    1. After each city name gets added, edges are set for a newly made graph with *n* vertices.
    2. Three characters--excluding spaces--are continuously read until the end of file. Those three characters are integers that correspond to matrix row, column, and weight at [row][column] respectively.
    3. If at the last read less than 3 characters are read then the edges are malformed and an error is printed and the program exits.

- Error Handling:
    1. If the first line does not have a valid input (integer > -1), an error is printed and the program exists.
    2. If the edges have invalid format (not a triplet consisting i, j, k), an error is printed and the program exists.

High-level Program Flow (error checking is done throughout the program)

parse arguments

read in graph file (make modifications according to flags)

setup for DFS (make a city array, initialize graphs with edges, etc.)

call DFS

clean up (free memory)

return