Alex Asch
Assignment 4 Design
Description of Program
This program will begin by taking command line inputs for names of cities and number of specified vertices. It then will parse the command line inputs and make a graph, G . It then takes this graph, and finds the shortest Hamiltonian path of the given graph. It then prints out the length and route of the shortest Hamiltonian path.
Deliverables

- tsp.c
  - This contains the main function and is the testing harness for this assignment.
- graph.h
  - This specifies the interface to graph ADT.
  - This is given by Professor Long in the resources repository.
- graph.c
  - This implements the ADT for graph
- path.h
  - This specifies the interface to the path ADT.
  - This is given by Professor Long in the resources repository.
- path.c
  - Implements the path ADT.
- stack.h
  - This specifies the interface to the path ADT.
  - This is given in the resources repository by Professor Long.
- stack.c
  - This implements the stack ADT
- vertices.h
  - Defines macros regarding vertices.
  - Given in the resources repository by Professor Long.
- Makefile
  - Makes the above program.
  - Links the above files
- README.md
  - A document in markdown syntax on the usage of the program.
- DESIGN.pdf
  - The pdf being viewed at the moment (this document).

Notes

- Step 1: How we interact with the Graph struct
  - The graph struct is a struct with an integer number of vertices, a boolean for directed or undirected, a boolean array for if vertices are visited, and a matrix (2d array) which stores the vertices

- The graph itself is an i by j array, and if there is a nonzero value at (i,j) this means that there is a path of length m(i,j) from i to j.
    - If the graph is undirected, that means for every path from i to j there is a path of the same length from j to i.
- We will write creation and deletion functions for the graph struct.
    - The creation struct, called a constructor, takes two arguments, the number of vertices and the boolean if the graph is directed or undirected.
    - This allocates the appropriate amount of memory for a graph struct using calloc to make sure the memory is zeroed, it then initializes the number of vertices and the directedness create the graph
    - It returns a pointer value to the graph struct.
    - The deletion function.
        - The deletion function takes a pointer to a pointer of G, and frees the memory at point g, it also sets the pointer to null to prevent use after free errors.
- The graph_vertices function.
    - This function will simply return the number of vertices a graph has. This is because we are creating opaque data types, which cannot be accessed outside of their implementation.
    - This function ensures there is a way for the user to access the values and data, but only when we want them to be able to.
- The graph add edge function.
    - This adds an edge at (i,j) of weight k, taking all above values as integer values.
    - This will mirror this point across the value for undirected graphs.
    - For built-in error checking, this function checks if the vertices are in bounds and the edges are non negative, and returns a boolean value, which is based on successful creation of edges.
- The graph has edge function.
    - This returns true if theres is an edge between the values i and j that are in the bounds of the matrix.
- The graph_edge_weight function
    - This returns the weight of the graph at the vertices, if it is out of bounds or doesn't exist return zero.
- The graph_visited function.
    - Return true if the vertex v has been visited false otherwise.
- The graph_mark_visited function
    - If vertex v is within bounds, mark v as visited.
- The graph_mark_unvisited function
    - Unvisit the vertex v.

    - graph_print
        - This is a debug function for making sure our implementation of the graph ADT is sound.
- Depth First search
    - How do we search over the graph?
    - We will be using a depth first search implemented recursively. This means that it marks the first vertex v as having been visited, then iterates through all of the edges (v.w) recursively if w is not visited.
    - In essence, we will use a depth first search to find paths that pass thru all vertices and if there is an edge from last to first this is a Hamiltonian path
    - Find the shortest
- Stack ADT implementation
    - The stack adt in this assignment is for tracking the path that Denver traveled.
    - This section defines the interface for a stack. A stack is LIFO. This is a struct called stack, and is an opaque data type just like the graph ADT.
    - *stack_create
        - The constructor function for a stack, the top of the stack is 0. The capacity of a stack is set to the specified capacity.
        - This also indicates the number of items to allocate memory for.
        - Memory is allocated twice, once for the struct, and once for the list
    - This returns the pointer to the struct.
    - stack_delete
        - Deletes the stack and frees the memory.
        - Takes the address of the address of struct (**), this allows us to set the address to NULL to prevent use after free.
    - stack_empty
        - Returns true if the stack is empty
        - Returns false if the stack is not empty
    - stack_size
        - returns the number of items in the stack.
    - stack_push
        - This function pushes something to the top of the stack.
        - Return false for error checking, return false when stack is full and can't be added to top
    - stack_pop
        - This function pops an item off of the specified stack, if the stack is empty, return false.
        - Takes a pointer sets a pointer
    - Stack_peek
        - Peeks into the top of the stack, basically pop without removing.

        - Return false if it’s empty
    - Stack_copy
        - Copies the contents of contents (not addresses) of the items from one to another.
        - The top should also match.
        - Copy the values not the addresses.
    - Stack_print.
        - Prints the city name the vertex corresponds to working from bottom up
- Paths
    - The paths ADT is a struct of vertices comprising the path, and the length of the path.
    - path_create function
        - The constructor for a path. Set the vertices as a freshly created stack that can hold up to VERTICES number of vertices, Initialize the length to be 0.
    - path_delete
        - Deletes the given path.
    - path_push_vertex
        - Pushes a vertex v onto the given path. The length of the path is increased by the edge weight connecting the vertex at the top of stack and v. Return true is success, false if not.
    - path_pop vertex
        - pops the vertices stack, passing the popped vertex back through the pointer v.
        - The length of the path is decreased by the edge weight connecting the vertex at the top of the stack and the popped vertex
        - Return true for successful pop.
    - path_vertices.
        - An accessor function for the number of vertices in the path.
    - path_length
        - An accessor function for the length of the path
    - path_copy
        - Makes a copy of the path, this will copy the stack and length as well
    - path_print
        - Prints out the path.

Psuedocode

graph.c

- Step 1.
    - Define the struct graph
        - This struct has an integer number of vertices
        - A boolean for directedness

        - An array of boolean of length VERTICES(26)
        - A 26x26 matrix of VERTICES
- Graph creation
    - write function Graph *graph_create(uint32_t vertices, bool undirected)
    - Allocate memory for the graph struct.
    - Set the vertices count to the number of vertices defined.
    - Set the bool undirected to undirected or not as specified as user input
    - return the pointer to the graph
- Graph delete
    - Takes one argument, the pointer to the pointer of G.
    - This deletes the graph
    - Takes pointer to pointer to the graph
    - First free the memory at the pointer given
    - Then set the pointer to NULL.
    - This prevents use after free
- Graph vertices.
    - Takes one argument, the pointer to graph G.
    - Returns the number of vertices in the graph.
- Graph adding edge
    - Take the pointer to the graph, and three integers. One for the x axis of graph, one for y axis, and for the magnitude
    - Check if the point i,j are within bounds
        - If they are.
            - Add a vertex of weight k at (i,j) in the matrix, by just adding the element to the matrix.
            - Check if the graph is undirected
                - If the graph is undirected
                    - Add a vertex of (j,i) as well, in the same method as above. Since graphs are square, we do not need to recheck bounds
                    - Return true for successful add.
                - If it is not.
                    - Return true for successful addition.
        - If they are not.
            - Return false for failed add operation.
- Graph has edge.
    - Takes the pointer to the graph, and two ints, i and j for x and y.
    - Check if the vertices i and j are within bounds
        - Return false if they are not
        - If they are.

            - Check if the matrix has a nonzero weight at i,j.
                - If it does, return true
- Graph edge weight
    - Takes a returns the weight of the edge from vertex i to vertex k.
    - CHeck if they aren't within bounds or if they don't exist, if that is the case return zero. (can use the has edge function to check this)
    - Else return matrix[i][j].
- graph_visited
    - Takes a pointer to a graph and an int for a vertex.
    - Check if the vertex is within the set bounds of the graph.
    - If it is, return the value of the visited array at vertex v.
- Mark visited
    - Check if the given vertex v is within bounds.
    - If it is within bounds.
        - Change the value at visited[v] to true.
- Mark unvisited
    - Check if the given vertex v is within bounds.
    - If it is within bounds.
        - Change the value at visited[v] to false.
- void graph_print
    - Take pointer to graph.
    - Print the graph.
    - Do this by iterating over matrix and printing weights.

DFS

- This recursively searches thru the graph, and takes a vertex, two paths, one for current and one for shortest, a character array of city names, and outfile, which is a file to print output to.
- Label the given vertex as visited.
- If all vertices visited, and the last connect to origin it a path

Stacks

- This stack ADT is for tracking the path traveled
- First declare the struct stack, which has an integer, top, which is the next empty slot, an integer of capacity, and an array of items
- Constructor for stack
    - Takes one integer for the capacity of the stack
    - Allocate memory for the stack struct
    - Set the top variable 0
    - Set the capacity variable to the given capacity
    - Allocate memory for the array of items
    - If the above allocation didn't properly allocate, make the pointer null

    - return the pointer to the stack struct
- Destructor for stat
    - Takes a pointer to the pointer of the stack struct
    - Free the memory that the struct pointer points to, but add a conditional as well, to check that both the pointer to stack and the pointer to stack array exist before beginning so we free memory that actually exists
- stack empty
    - Takes a pointer to a stack/
    - Check if the stack is empty, which is if top is 0. (could go for a "safer" implementation by checking through the items array as well).
    - Return a boolean value corresponding to the emptiness of the stack
- stack_full
    - Takes a pointer to a stack
    - Returns true if the stack is full, this is if the top value is equal to the capacity value, due to zero based indexing, it is this way instead of top = capacity + 1.
    - Return a boolean value for the fullness of the stack
- stack_push
    - Takes a pointer to a stack and an integer to push
    - This pushed value of given int to stack.
    - If the stack is full, return false, if successful return true.
- stack_pop
    - Tales a stack and THE POINTER to an int x.
    - PUTS THE POPPED VALUE IN THE POINTER, DO NOT RETURN THE VALUE POPPED.
    - Returns true on success, aka stack is not empty and x is correctly pointed.
- stack_peek
    - Takes a pointer to the stack and the pointer to an int x.
    - Put the value at top in the pointer x.
    - Return true if successful
- stack_copy
    - Takes two pointers to two stacks.
    - copy the CONTENTS of the items list and the top
    - this means copy contents not pointers
    - Iterate through items array
- stack_print
    - print the stack.
    - This is given as this needs to be very specific.

PATHS

- This is an ADT that uses the stack ADT to track the traveled path
- Define path

    - Has a stack called vertices and an integer for the length of the path
- Create path
    - The constructor for a path, set vertices as a new stack that can hold up to VERTICES number of vertices.
    - Initialize the length as zero.
    - Why can't we pass an int? It seems much smarter as we know the length of the path, so we can create the stack also as the correct length, and thus utilize the functions for fullness of stack?
- Destructor for path
    - Same as above destructors
- Path push vertex
    - Pushes vertex v on path p.
    - The length is increased by the weight of the connection between the current top of stack and the vertex.
    - use stack push as defined above to add the vertex to the stack.
    - Return true of false based on successful push of vertex.
- Path pop vertex.
    - Pops a vertex off the stack using the above stack function.
    - Passes the popped vertex out as a the given pointer
    - Decreases length of weight between the vertex at top of stack and the popped vertex
    - Return a boolean value for successful push or pop.
- path_vertices
    - Return the number of vertices in the path
    - just use stack size
- path_length
    - return the length of the path
- path_copy
    - Copy the current path to an already initialized path
- path_print
    - print the path to an outfile, requires a call to stack print as defined above to print out contents of vertices stack.
-