Karthi Sankar

kasankar@ucsc.edu

10/25/21

CSE 13S Assignment 4 Design Document

Purpose:

The purpose of this program is to use different data structures in order to find the shortest Hamiltonian path when given a list of places, or “vertices” that can form a path. This program will use three different data structures or ADTs, to accomplish this task: a graph, a stack, and a path. This program will require the use of data structures, dynamic memory allocation, the use of file input/output, etc. The finding of a Hamiltonian path will be done using a depth-first search method.

Structure:

The structure of this program is that the three ADTs will build upon each other. Graph and stack will each have their own members and functions, and both of these ADTs will be used in the Path ADT and its functions. As before, the three programs will then be used and run using command line options within a test file named tsp.c. This file will contain the function dfs() or depth-first search, that will make use of all three ADTs, as well as the main function. The command line will be parsed for given options and run certain operations accordingly. The input graph that will be needed to run the program will be supplied either from an input file or stdin on the command line; the output will be printed either to an output file or stdout on the command line. Which occurs will be up to the user and which command line options they specify.

File Description/Pseudocode:

(pseudocode is simplified and in Python/English)

Note: in my pseudocode, I indicate what would be a “->” used when accessing members of a struct as a “.”

graph.c:

```
int graph_vertices (Graph g):
    return g.vertices

bool graph_add_edge (Graph g, int i, int j, int k):
    if 0 < i and j < 26:
        g.matrix[i][j] = k
        if g.undirected == true:
            g.matrix[j][i] = k
        #end
        return true
    #end
    return false

bool graph_has_edge (Graph g, int i, int j):
    if (g.matrix[i][j] > 0):
        return true
    #end
    else:
        return false

int graph_edge_weight (Graph g, int i, int j):
    return g.matrix[i][j]

bool graph_visited (Graph g, int v):
    return g.visited[v]

void mark_visited (Graph g, int v):
    g.visited[v] = true

void mark_unvisited (Graph g, int v):
    g.visited[v] = true
```

The graph data structure is a dynamically allocated structure type with a few different parts. When a graph is created, it has 4 parts: 32-bit integer representing the number of vertices it has, a boolean stating whether it is undirected or not, a 32-bit integer matrix

that will show the adjacency of its edges, and a boolean array visited that will show which vertix has been visited.

The first function graph_vertices returns an integer, the variable (or member) vertices, which is the number of vertices a graph has.

The second function, graph_add_edge, returns a bool, which states if the adding of the edge to the graph was successful. First, it will check if the parameter integers i and j are in bounds (between 0 and the variable vertices, the number of vertices G has). (Note: the max number of vertices a graph can have is 26, as specified by the header file, vertices.h.) If they are, a value (the parameter k) will be added into the graph's matrix (at [i][j]) to show their adjacency. If the graph is undirected, a value will be added to show adjacency another way (from [j][i]). If the adding of an "edge" was successful, true will be returned, if not, false will be returned.

The next function, graph_has_edge, returns a boolean that states if the graph has an edge between two of the given vertices, integers i and j. My pseudocode is simplified, but first, it will check if i and j are within the bounds and also if there exists a non-zero positive weight between the vertices. So, that entails checking the element at [i][j] in the graph's matrix to see if it is positive. If that edge exists, true is returned; if for whatever reason, the vertices are out of bounds, or an edge does not exist, false is returned.

The function graph_edge_weight, returns a 32-bit integer, the value at an edge between the vertices i and j, essentially a value from the matrix of the graph. In my actual code, I first check for if the values of i and j are within bounds and if the matrix at [i][j] is positive. If any of these conditions are false, 0 is returned, indicating no edge from i to j; if the conditions are true, the value at matrix[i][j] is returned.

The final three functions will deal with the boolean array named visited. The visited essentially says if position v in visited is true, then the vertex v has been visited, and if it is false, it has not. The first function, graph_visited, returns a boolean stating if a vertex in the graph has been visited or not. As such, it returns the element at the given parameter v in the array vertices, which indicates whether this condition is true or false. The next two, graph_mark_visited and graph_mark_unvisited have a return type of void. Each function checks if the parameter v is within bounds, or less than the number of vertices that the graph g has. mark_visited will change the element at position v in vertices to true, and mark_unvisited will change the element at position v to false.

stack.c:

The stack data structure is a dynamically allocated type. It has 3 members or variables: an 32-bit integer variable that represents the top of the stack, an 32-bit integer variable that states the capacity of the stack, and an 32-bit integer array named items, which will contain the items of the stack. The array items will be dynamically allocated to be the size of the variable capacity.

```
int stack_size (Stack s):
   return s.top

bool stack_empty (Stack s):
   if s.top == 0:
      return true
   return false

bool stack_full (Stack s):
   if s.top == s.capacity:
      return true
   return false
```

The function stack_size returns a 32-bit integer that is the size of the stack. The variable top will indicate how big the stack is, (or how there elements there are in it) because it always is the value of the position one space above the top element in the stack. So, this function will return the variable top. The next two functions check if the stack is empty or full. stack_empty returns a boolean indicating whether or not the stack is empty. It checks if the top is 0 (indicating nothing is in the stack) and returns true if that fact is true, false if otherwise. stack_full returns a boolean indicating whether or not the stack is full. It checks if the stack's top is equal to the stack's capacity (indicating the stack is full and no more elements can be pushed) and returns true if that fact is true, false if otherwise.

```
bool stack_push (Stack s, int x):
    if stack_full(s):
        return false
    s.items[s.top] = x
    s.top += 1
    return true

bool stack_pop (Stack s, int pointer x):
    if stack_empty(s):
        return false
    s.top -= 1
    pointer x = s.items[s.top]
    return true
```

The function stack_push returns a boolean indicating whether the pushing of an element onto the stack was successful or not. takes in the stack and the 32-bit integer parameter x. If the stack is full (the value of this function call is true), therefore nothing can be pushed, then false is returned. If the stack isn't full, the index at the top of the array items is set to the integer value x that was passed, placing the value at the top of the stack. Then, the stack top is incremented, indicating that the stack is one element larger. Finally, true is returned to indicate a successful push.

The function stack_pop returns a boolean indicating whether the popping of an element onto the stack was successful or not. The function stack_pop takes in the stack and the integer pointer x as parameters. If the stack is empty (the value of this function call is true), therefore nothing can be popped off it, then false is returned. If the stack isn't empty, the stack top is decremented, indicating the stack has had one element taken off it. Then the value that was popped off the stack, which would be at the current top of the stack (within the array items), is passed to the pointer x (and saved), and true is returned, indicating a successful pop.

```
bool stack_peek (Stack s, int x):
    if stack_empty(s):
       return false
   pointer x = s.items[s.top-1]
   return true

void stack_copy(dst, src):
   if dst.capacity == src.capacity:
      for i in range (0, src.top, 1):
         dst.items[i] = src.items[i]
      #end
      dst.top = src.top
```

The function stack_peek returns a boolean indicating whether the peeking at an element on top of the stack was successful or not. The goal of the function is to save the value that is at the top of the stack. If the stack is empty (the value of this function call is true), therefore nothing can be peeked at, then false is returned. If the stack isn't empty, the value at the top-1 index (because the top indicates the next empty slot in the stack) of the stack (within the items array) is passed to the integer pointer x. This effectively saves the value that was peeked at into the pointer that was passed as a parameter, allowing for the value to be accessed later.

Finally, the function stack_copy returns nothing so it is void. Its goal is to make a given destination stack a copy of a given source stack. The two parameters are pointers to the stack designated as the dst (destination) and src (source). If the two stack parameters' capacities are equal (indicating that they can hold the same number of elements), then all of the values from the source's items are copied to the destination, until the top of the source stack is reached. This is achieved using a for loop, assigned each element of the items array of drc to the element in the items array of src at the same position. Finally, the destination's top variable is set to the top of the source stack, effectively making the two stacks the same size.

path.c:

The path data structure is a dynamically allocated type. It has two members (or variables): a stack named vertices and a 32-bit integer representing the length of the path.

```
Path *path_create(void):
    dynamically allocate (using malloc) path pointer
    if pointer p exists:
        p.vertices = stack_create(VERTICES)
        p.length = 0
    #end
    return p

void path_delete (Path (double pointer) **p):
    if p and p.vertices exist:
        stack_delete(p.vertices)
        free pointer p
        set pointer p to NULL
    #end
    return
```

The function path_create is responsible for constructing the path. The path pointer p is first dynamically allocated using malloc. Then if the pointer was created, then the stack

vertices is created using stack_create. The capacity of this stack is set to be VERTICES, or 26, a constant defined in the header file vertices.h. Then the variable length is set to 0. Finally the pointer p is returned, as path_create's return type is a path. Path_delete's return type is void, and it is responsible for destructing a path after it is created and used. If the path and its vertices exist, they must be deleted as they were created using dynamically allocated memory. So, the stack vertices is deleted using stack_delete and the path pointer p is freed and set to NULL. This effectively frees all memory taken up by a path when it's created. Finally, I used a return with nothing after to indicate the end of the function.

```
bool path_push_vertex (Path p, int v, Graph g):
    if stack_full(p.vertices) != true:
        int val;
        stack_peek(p.vertices, (address of) val);
        p.length += graph_edge_weight(g, val, v)
    #end
    return stack_push(p.vertices, v)

bool path_pop_vertex (Path p, int pointer v, Graph g):
    if stack_empty(p.vertices):
        return false
    stack_pop(p.vertices)
    if stack_empty(p.vertices):
        return true
    int val;
    stack_peek(p.vertices, (address of) val);
    p.length -= graph_edge_weight(g, val, pointer v)
    return true
```

The function path_push_vertex returns a variable indicating if a push to the stack vertices was successful or not. First, if vertices is full, then false is returned because nothing can be added to the stack. Then I use an integer named val to peek at the element at the top of the stack using stack_peek. This is needed in order to add to the length of the path. I use stack_peek, saving the element peeked at within the integer val. Then, the length of the path must be incremented, indicating that another vertex has

been visited. I add to the length the graph edge weight between val and the parameter because that is the “length” between the two vertices that have been visited. Finally, I return the boolean result of stack_push of v onto the stack vertices, indicating if a push was successful or not.

The function path_pop_vertex returns a variable indicating if a pop off the stack vertices was successful or not. First, if vertices is empty, then false is returned because nothing can be added to the stack. Then, the vertex v is popped off the stack vertices using stack_pop. After that, if the stack is empty, true is returned and nothing can be decremented from the path length because the end has been reached. If the stack isn’t empty then I use an integer named val to peek at the element at the top of the stack using stack_peek, saving the value of the peeked element into val. Then, the length of the path must be decremented, indicating that another vertex has popped off the “path.” I subtract from the length the graph edge weight between val and the parameter (pointer) v because that is the “length” between the two vertices that has to be “taken off” the path. Then, true is returned to indicate a successful pop.

```
int path_vertices (Path p):
    return stack_size(p.vertices)

int path_length:
    return p.length

void path_copy (Path dst, Path src):
    stack_copy(dst.vertices, src.vertices)
    dst.length = src.length

void path_print (Path p, FILE outfile, char cities[]):
    print the path length
    print the first element of the path
    stack_print(p.vertices, outfile, cities)
```

path_vertices returns a 32-bit integer which is the size of stack vertices. So the one line of the function is to return the value of the function call stack_size(p.vertices).

path_length returns a 32-bit integer that is the length of the path. So the one line of the function is to return the variable p.length, a member of the Path struct.

The function path_copy's return type is void. It takes in two parameters, a dst (destination) path, and a src (source) path. The two members of the source path must be copied into the destination path. So first, I use stack_copy to copy the vertices stack of src to be the vertices stack of Path dst. Then, I set the path length of dst to be the path length of src, essentially making dst a copy of src.

The final function path_print has a return type of void, and it is used to output the path and its length when necessary in the main function. All printing will be done using fprintf given that paths must be printed to a certain file (or stdout). The path will be printed to a specified outfile and the stack in this case is an array of cities that will serve as the vertices of the path. First, I print out the path length. Then I print the first element of cities, indicating that you can return to the beginning of the path from its end. Finally, I use stack_print, using as parameters the path's vertices, outfile, and the cities array, in order to print out the full path.

tsp.c:

DFS:

```
void dfs(Graph g, int v, Path curr, Path shortest, char cities[], FILE outfile, bool verbose, int rec_count):
    if path_vertices(curr) + 1 == graph_vertices(g) and g has an edge between START VERTEX and v:
        if path_length(shortest) == 0 or length of curr < length of shortest:
            push START_VERTEX onto path curr
            if (verbose):
                print path curr to outfile
            #end if
            copy curr path into shortest path
        #end if
    else:
        for i in range(1, graph_vertices(g), 1):
            if g has an edge between v and i and i has not been visited:
                push i onto path curr
                mark i visited (on g)
                rec_count += 1
                dfs (g, i, curr, shortest, cities, outfile, verbose, rec_count)
            #end if
        #end for
        int x;
        mark v visited (on g)
        pop vertex v off path curr (using address of x)
    #end if-else
```

Dfs stands for depth-first search. It is the first function in the file tsp.c. Its purpose is to find the shortest Hamiltonian path through the vertices when given an input graph. This Hamiltonian path must pass through all vertices and connect from the last vertex to the first. Its return type is void. Its parameters are Graph (pointer) g, 32-bit integer v, Path (pointer) curr, Path (pointer) shortest, character array (pointer) cities, file (pointer) outfile, boolean verbose, and integer rec_count. The basis of the function is a large if-else statement. On the first call, dfs will go to the else. The else is as follows. It uses an for loop to iterate through the vertices of graph g. Within the for loop, there is a check if the graph has an edge between the parameter v and loop counter i (two vertices of the graph) and position i has not been visited in graph g. If this is true, then the vertex i will be pushed on the path curr and position i will be marked visited in graph g. Then the variable rec_count will be incremented because in the next line, dfs is called recursively

with all of the required parameters. If the check within the for loop is false, then the graph is marked unvisited at position v and a vertex is popped off the path curr. An integer x is created for the sake of the integer pointer parameter required for path_pop_vertex. The contents of dfs beginning with the first if statement are as follows. There is an if statement to check if the vertices of path curr + 1 is equal to the number of vertices of graph g, meaning that the path is one vertex from ending; the other check is if the graph has an edge between the START_VERTEX and v, meaning if the path connects back to the starting vertex. If this is true, then another if is checked. This one checks if the shortest path length is 0 (contains nothing) or if the curr path length is shorter than the shortest path length (the shortest path can be updated). If this is true, then the START_VERTEX is pushed on the curr path. For the sake of the -v (verbose) command line option, the curr path may be printed. Finally, the current path is copied to the shortest path. This function serves to keep building paths using the parameters given. The else of the large if-else essentially builds the current path, pushing more and more vertices onto the curr path until the path is one away from being finished. When the path is one vertex away from being finished, if possible the START_VERTEX is pushed onto the path, connecting it back to the beginning. Then, there is a check to see if the current path made is shorter than the last; if so, that path is saved. DFS is called recursively until the shortest Hamiltonian path is found.

Main function:

```
OPTIONS are set as "vui:o:h"

int main():
    bool undirect = false;
    bool verbose = false;
    bool help = false;

    file pointer infile = stdin
    file pointer outfile = stdout
```

The main function of my tsp.c starts by declaring all variables I will need initially. First, the command line options are defined. The options are -v to enable verbose printing, -u to use an undirected graph, -h for the program usage and help, -i for taking input containing a graph, and -o for the output of the computed path. -i and -o have a colon after because a filename or path is to be typed into the command line after they are put. So, initially I declare 3 booleans to be used within my switch statement (initialized to false), undirect (for -u), verbose (for -v), and help (for -h). Then, I declare the file pointers infile and outfile, which will be used for the taking of an input graph and printing the path. They are initially set to stdin and stdout because that is where I will be taking input and printing if -i and -o are not used.

```
   while (options are being given to the command line);
      switch(opt):
      case 'h':
         help = true;
         break;
      case 'v':
         verbose = true;
         break;
      case 'u':
         undirect = true;
          break;
      case 'i':
         set infile to file given to command line
         if (file is invalid):
            print error message
            quit
         break;
      case 'o':
         set outfile to file given to command line
         if (file is invalid):
            print error message
            quit
         break;
      default:
         break;
      #end switch
#end while
```

Then I use while and switch loops using get opt to check for command line options given by the user. If the user puts in -u, -v, or -h, their corresponding boolean is set to true. In the case of -i and -o, I get the filename that the user specifies using optarg. The input file is set to read mode, while the output file is set to write mode. If either of the files given are invalid, then an error message is printed to stderr and the program is exited (or quit). The default case of the switch statement just contains a break.

Next, if the boolean help is true, I print the help message for the program.

```
if (help):
   print help message

int num_vertices;
scan infile for num_vertices
if (num_vertices > VERTICES):
   print error message
   quit
#end if
```

Then, I use scanf to take in the first number from the input graph from infile, whether it be stdin or a file. That number is set to the value of 32-bit integer num_vertices, which will be used in order to create an array later on. If this num_vertices taken from the input is greater than VERTICES (a constant with a value of 26 defined in vertices.h), an error message is printed and the program is quit because 26 is the max number of vertices allowed in this program.

```
char cities [num_vertices]

for i in range(i, num_vertices, i):
    get vertex name from infile
    if (vertex given is invalid):
        print error message
        quit
    #end if
    set last char of vertex name to "\0"
    cities[i] = vertex name;
#end for
```

Then, I create an array of vertices needed to call DFS. In my program, I use a buffer, which I have excluded from my pseudocode. The array I create is named cities, and it is a pointer to an array of characters, as the vertice names will be characters. The size of cities is num_vertices. In order to get input from the graph, I use a for loop to iterate through the cities array, filling each element with the name gotten from the input file using fgets. If fgets returns NULL at any point, indicating failure, an error message is printed and the program is exited. Another note is all strings gotten using fgets and the buffer must be changed to be null-terminated instead of with a newline, and that the strings are copied into the array cities using the function strdup().

```
Graph (pointer) g = graph_create(num_vertices, undirect)

int i = 0;
int j = 0;
int k = 0;
int suc;
while ((suc = 3 scanned values from infile) != EOF):
   if (suc != 3):
      print error message
      quit
   #end if
   graph_add_edge(g, i, j, k)
#end while
```

The graph to be used in DFS is now being constructed and filled. I create a new graph called g using graph_create, the constructor. Its size is num_vertices, and its boolean undirected to set to the value of the boolean undirect (which will be true or false depending if the user used -u). Then, I use 3 32-bit integers, i, j and k to represent the i and j vertices of the graph, and k being the edge weight between them. I also create an integer suc that will used to determine if the scanning of the edges from the input file was successful or not. To scan the edges, I use a while loop that will continue until fscanf (the result of which is set to suc) does not equal EOF, or hasn't reached the end of the input file. If any more than 3 integers are given as an edge, an error message is printed and the program is quit. Finally, the last thing done in the while loop is to add the edge using the i, j, and k to the graph g created earlier.

```
Path (pointer) current;
Path (pointer) shortest;
current = path_create();
shortest = path_create();

int recur_count = 0;
dfs(g, START_VERTEX, current, shortest, cities, outfile, verbose, (address of) recur_count);
path_print(shortest, outfile, cities)
print total number of recursive calls to outfile
```

Next, I create the paths to be used in the DFS function. I create path pointers current and shortest, and set them both to be constructed using the function path_create. Finally, I call DFS. First, I initialize an integer variable recur_count that will be used in DFS to count the number of recursive calls it takes to get the shortest path. I then call DFS with all the specified parameters it needs, including the graph g, the paths I created, the cities array, outfile, and the rest. After the call to DFS, I print the total number of recursive calls to outfile, whether it be an output file or stdin.

```
close infile and outfile
delete graph g
delete path current
delete path shortest
for n in range(0, num_vertices, 1):
   free(cities[n])
#end for
free cities array

return 0;
```

The final thing I do in my main function is to free all the memory that has been dynamically allocated. First, I close the file pointers infile and outfile using fclose. Then, I delete the paths shortest and current as well as the graph g. I free the cities array, iterating through all its elements and freeing them (using the free() function). Finally, I return 0, indicating the success of the program.