Assignment 4: The Perambulations of Denver Long Design Document

Thadeus Ballmer

October 2021

1 Introduction

Assignment 4 is a variation on the classic Travelling Salesman Problem. In the travelling salesman problem, the best possible path needs to calculated from given vertices (locations) and edges (roads) which connect the vertices. Each edge has a *weight* value which represents the distance between the two vertices. Each edge may be directed or undirected (a one way road vs a two way road). In assignment 4, the desired path needs to travel through all vertices exactly once and end at the origin matrix—what is called a *Hamiltonian Path*. Given that there could be several *Hamiltonian Paths* in a problem set, the final path solution should be the shortest. A path's distance is calculated through summation of all edge *weights* traveled through.

2 Graphs

The graph data structure will be used to hold the set of vertices, the set of edges, and each edge's weight. This data structure will also hold the number of vertices, which vertices have been visited, and if the graph is undirected.

```
struct Graph {
    uint32_t vertices;
    bool undirected;
    bool visited[VERTICES];
    uint32_t matrix[VERTICES][VERTICES];
}
```

VERTICES is a macro defined in **vertices.h**:

```
#pragma once

#define START_VERTEX 0
#define VERTICES 26
```

The macro vertices defines the maximum number of vertices in graph as 26. This means that matrix[VERTICES][VERTICES] is defined with a size of 26x26. The set of edges and weights are stored by inputting a value into the index of matrix. For example: if the edge (3,4) has a weight of 7, matrix[3][4] = 7. If the graph is defined as undirected, the index should be reflected by setting matrix[4][3] = 7 as well.

2.1 Graph Interface

The Graph struct is an abstract data type. Abstract data types need an interface comprised of constructor, destructor, accessor, and manipulator functions. The interface is specified in **graph.h** and implemented in **graph.c**.

2.2 Constructor

A constructor allocates memory for the type. This constructor also sets the number of vertices in the graph and indicates if the graph is undirected.[1]

```
Graph *graph_create(uint32_t vertices , bool undirected) {
    Graph *G = (Graph *)calloc(1, sizeof(Graph));
    G->verices = vertices;
    G->undirected = undirected;
    return G;
}
```

2.3 Destructor

Destructor functions free allocated data and should set the pointer to the previously allocated memory to NULL.

[1] For the grader: I only use C for the code blocks if the code was given in the assignment document (This includes headers).

```
void graph_delete(Graph **G) {
    free(*G);
    *G = Null;
    return;
}
```

2.4 Accessor Functions

Accessor functions access the data stored in graph and may interpret said data. graph_vertices() accesses the number of vertices in graph.

```
uint32_t graph_vertices(Graph *G)
    return vertices
```

graph_has_edge() checks if the given vertices have an edge connecting them.

```
bool graph_has_edge(Graph *G, uint32_t i, uint32_t j)
    if (i < 26 and j < 26 and matrix[i][j] > 0)
        return true
    return false
```

graph_edge_weight() returns the weight of the edge from vertex i to vertex j. Returns 0 if either vertex is out of bounds, or if an edge does not exist.

```
uint32_t graph_edge_weight(Graph *G, uint32_t i, uint32_t j)
    if (i < 26 and j < 26 and matrix[i][j] > 0)
        return matrix[i][j]
    return 0
```

graph_visited() checks if vertex v has been visited.

```
bool graph_visited(Graph *G, uint32_t v)
    if (v < 26)
        return vertices[v]
    return false
```

2.5 Manipulator Functions

graph_add_edge() adds an edge to the graph

```
bool graph_add_edge(Graph *G, uint32_t i, uint32_t j, uint32_t k)
    if (i < 26 and j < 26)
        if (graph is undirected)
            matrix[j][i] = k
        matrix[i][j] = k
        return true
    return false
```

graph_mark_visited() marks v as visited

```
void graph_mark_visited(Graph *G, uint32_t v)
    if (v < 26)
        visited[v] = true
```

graph_mark_unvisited() marks v as unvisited

```
void graph_mark_unvisited(Graph *G, uint32_t v)
    if (v < 26)
        visited[v] = false
```

2.6 Debug Function

graph_print() prints the graph to make sure the interface works properly

```
void graph_print(Graph *G)
    for (int row = 0; row < VERTICES; row++)
        for (int col = 0; col < VERTICES; col++)
            print(matrix[row][col])
            if (col = VERTICES - 1)
                print(Newline)
```

3 Depth-first Search

Depth-first search is the algorithm that will try out every possible path.

```
procedure DFS(G, v):
    label v as visited
    for all edges from v to w in G.adjacentEdges(v) do
        if vertex w is not labeled as visited then
            recursively call DFS(G, w)
    label v as unvisited
```

DFS can be used to find the shortest *Hamiltonian Paths* when applied to the graph input. The DFS algorithm in C will necessarily apply the ADTs created into its implementation in **tsp.c**.

4 Stacks

In order track the current path taken by depth-first search, a *stack* must be implemented with its own interface. The stack is declared in **stack.h** and defined in **stack.c**.

```
struct Stack {
    uint32_t top;
    uint32_t capacity;
    uint32_t *items;
}
```

4.1 Constructor

stack_create() dynamically allocates memory for the stack and for the items array included in the stack. This constructor also sets **top** to zero which serves as the index of the next empty value in the stack. The stack size is based on the capacity input.

```
Stack *stack_create(uint32_t capacity) {
    Stack *s = (Stack *) malloc(sizeof(Stack));
    if (s) {
        s->top = 0;
        s->capacity = capacity;
        s->items = (uint32_t *) calloc(capacity, sizeof(uint32_t));
        if (!s->items) {
            free(s);
            s = Null;
        }
    }
    return s;
}
```

4.2 Destructor

stack_delete() frees the memory allocated in the constructor and sets the pointer to the previously allocated memory to NULL.

```
void stack_delete(Stack **s) {
    if (*s && (*s)->items) {
        free((*s)->items);
        free(*s);
        *s = NULL;
    }
    return;
}
```

4.3 Accessor Functions

stack_empty() checks if the stack is empty.

```
bool stack_empty(Stack *s)
    if (top > 0)
        return false
    return true
```

stack_full checks if the stack is full.

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

stack_size returns the number of items in the stack

```
uint32_t stack_size(Stack *s)
    return top
```

stack_peek() sets the value of the input to the value of the element at the top of the stack.

```
bool stack_peek(Stack *s, uint32_t *x)
    if (stack_empty())
        return false
    x = items[top - 1]
    return true
```

stack_print() prints out the contents of stack to a file.

```
void stack_print(Stack *s, FILE *outfile, char *cities[]) {
    for(uint32_t i = 0; i < s->top; i += 1) {
        fprintf(outfile, "%s", cities[s->items[i]]);
        if (i + 1 != s->top) {
            fprintf(outfile, " -> ");
        }
    }
    fprintf(outfile, "\n");
}
```

4.4 Manipulator Functions

stack_push() adds the input value to the stack and increments top.

```
bool stack_push(Stack *s, uint32_t x)
    if (stack_full())
        return false
    items[top] = x
    top += 1
    return true
```

stack_pop() pops an item off of the stack and decrements top.

```
bool stack_pop(Stack *s, uint32_t *x)
    if (stack_empty())
        return false
    x = items[top - 1]
    top -= 1
    return true
```

stack_copy() makes one stack a copy of the source stack

```
void stack_copy(Stack *dst, Stack *src)
    for(int x = 0; x < top; x++)
        dst_items[x] = src_items[x]
    dst_top = src_top
```

5 Paths

A path ADT will help in the arrangement of vertices from the traveled path.

```
struct Path {
    Stack *vertices;
    uint32_t length;
};
```

The interface is specified in **path.h** and implemented in **path.c**.

5.1 Constructor

path_create() constructs Path using the Stack ADT.

```
Path *path_create(void)
    Path *p = (Path *) malloc(sizeof(Path))
    Stack *vertices = stack_create(VERTICES)
    length = 0
```

5.2 Destructor

path_delete() frees memory allocated using path_create().

```
void path_delete(Path **p)
    stack_delete(Stack p->vertices)
    free(*p)
    *p = NULL;
```

5.3 Accessor Functions

path_vertices returns the number of vertices in the path

```
uint32_t path_vertices(Path *p)
    return stack_size(Stack p->vertices)
```

path_length() returns the length of the path

```
uint32_t path_length(Path *p)
    return length
```

path_print() prints out path to a specified outfile

```
void path_print(Path *p, FILE *outfile, char *cities[])
    fprintf(outfile, "Contents of stack print %", stack_print())
```

5.4 Manipulator Functions

path_push_vertex() pushes a vertex onto path and increases length

```
bool path_push_vertex(Path *p, uint32_t v, Graph *G)
    if (stack_peek(Stack p->vertices, x) == false or
        stack_push(Stack p->vertices, v) == false)
            return false
    length += graph_edge_weight(Graph G, x, v)
    return true
```

path_pop_vertex() pops from vertices stack and decreases length

```
bool path_pop_vertex(Path *p, uint32_t *v, Graph *G)
    if (stack_pop(Stack p->vertices, v) == false or
        stack_peek(Stack p->vertices, x))
            return false
    length -= graph_edge_weight(Graph G, x, v)
    return true
```

path_copy() copies the source path to the properly initialized path

```
void path_copy(Path *dst, Path *src)
    for(int x = 0; x < stack_size(Stack src->vertices); x++)
        dst->vertices[x] = src->vertices[x]
    dst->length = src->length
```

6 tsp.c - It all comes together

tsp.c contains all the implemented header files. This file brings together all the ADTs and the DFS program in order to solve the assignment problem. **tsp.c** takes the following command-line options:

- -h: Prints out a manual on how to use the program, then exits the program.
- -v: The program will now print out all *Hamiltonian paths* discovered.
- -u: Specifies the graph to be undirected
- -i infile: Specifies the graph input file path. If not specified, the default input is set to stdin.
- -o outfile: Specifies the output file. If not specified, the default output is set to stdout.

After taking command-line inputs, main() computes the shortest *Hamiltonian Path*.

```
main()
    while (Loop through getopt options)
        switch(opt)
        case 'i'
            infile = fopen(optarg, "r")
        case 'o'
            outfile = fopen(optarg, "w")
        case 'h'
            help = true
        case 'v'
            verbose = true
        case 'u'
            undirected = true

    if (h)
        print(Program usage manual)
        exit()
    if ((n = scanf(First line from input)) > VERTICES)
        print(Error)
        exit()
    if (fgets(Next n lines) fails)
        print(Error)
        exit()
    G = graph_create(size, undirected)
    while ((x[] = scanf()) != EOF)
        if (x malformed)
```

```
        print(Error)
        exit()
    graph_add_edge(Graph G, x[0], x[1], x[2])
    if (undirected)
        graph_add_edge(Graph G, x[1], x[0], x[2])

cur = path_create()
shortest = path_create()

dfs(Graph G, START_VERTEX, Path cur, Path shortest, cities[], outfile)

print(Shortest Path)

graph_delete(Graph G)
path_delete(Path cur)
path_delete(Path short)
```