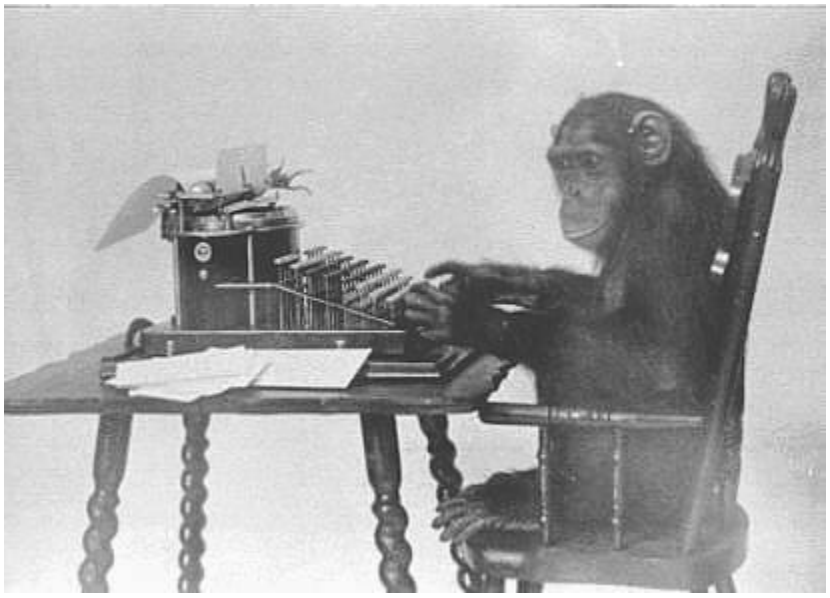
Asgn4 Design

Ralph Miller

October 24th 2021



"One does not simply walk into Mordor."

Boromir

Contents

1 Objective

In this assignment I will be writing a program that produces a Hamiltonian path given an input graph of vertices and weights. A Hamiltonian path is defined as such, "In the mathematical field of graph theory, a Hamiltonian path is a path in an undirected or directed graph that visits each vertex exactly once."

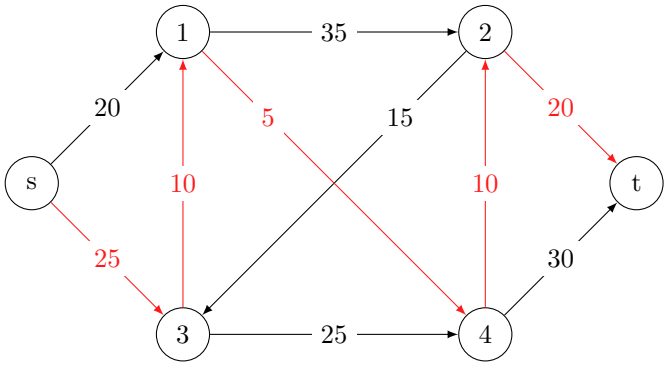
1.1 Deliverables

The program is comprised of five individual parts. Our Test Harness, contained in tsp.c, which will facilitate running the program. Three Abstract Data Types(ADT) and their corresponding functions, Graph ADT, Path ADT, and Stack ADT. Finally, Depth-first Search, a recursive algorithm used to find paths that pass through all vertices to solve for the shortest Hamiltonian Paths.

```
DELIVERABLES:
graph.h    -specifies the interface to the graph ADT
graph.c    -implements the graph ADT
path.h     -specifies the interface to the path ADT
path.c     -implements the path ADT
stack.h    -specifies the interface to the stack ADT
stack.c    -implements the stack ADT
tsp.c      -contains main and other functions to obtain functionality
vertices.h -defines macros regarding vertices
Makefile   -make, make clean, and make format function properly
README.md  -uses proper markdown syntax
```

1.2 Hamiltonian Path Diagram

In this illustration of our Hamiltonian Path, the s node is the starting point and the t node is the ending point. The arrows show the direction a path can take, if the arrow is one-directional the path can be made only from node x to the node y the arrow is pointing to. If the arrow is bi-directional a path can be made from either node. The Hamiltonian path is highlighted in red.



2 Graph ATD

Abstract Data Types are treated as opaque, ie we are not allowed to directly access the data structure. For implementing each ADT we will need five things; A struct definition, as well as; Constructor, Destructor, Accessor, and Manipulator functions.

```
//Graph struct included in Asgn doc
struct Graph {
    uint32_t vertices;
    bool undirected;
    bool visited[VERTICES];
    uint32_t matrix[VERTICES][VERTICES];
};
```

2.1 Constructor Function

```
//Graph Constructor included in Asgn doc
//uses the Graph ADT, takes in two variables, vertices and undirected
//outputs a pointer to the construced graph
Graph *graph_create(uint32_t vertices, bool undirected) {
    Graph *G = (Graph *)calloc(1, sizeof(Graph));
    G->vertices = vertices;
    G->undirected = undirected;
    return G;
}
```

2.2 Destructor Function

```
//Graph Destructor included in Asgn doc
void graph_delete(Graph **G) {
    free(*G);
    *G = NULL;
    return;
}
```

2.3 Accessor Functions

```
//supporting pseudocode included below
uint32_t graph_vertices(Graph *G);
bool graph_has_edge(Graph *G, uint32_t i, uint_32t j);
uint32_t graph_edge_weight(Graph *G, uint32_t i, uint_32t j);
bool graph_visited(Graph *G, uint32_t v);
void graph_print(Graph *G);
```

2.4 Manipulator Functions

```
//supporting pseudocode included below
bool graph_add_edge(Graph *G, uint32_t i, uint_32t j, uint32_t k);
void graph_mark_visited(Graph *G, uint32_t v);
void graph_mark_unvisited(Graph *G, uint32_t v);
```

2.5 Pseudocode

```
//ACCESSOR FUNCTIONS//

//returns the number of vertices in the graph
graph_vertices {
   for all vertices in Graph {
       counter++
   }
   return counter
}

//returns true if vertices i and j are within bounds and there exists an edge from i to j
//an edge exists if it has a non zero, positive weight.
graph_has_edge {
    if vertices i,j are in bounds && an edge exists from i,j {
        return true
    }
    return false
}

//return the weight of the edge from vertex i to vertex j
//if either i or j aren't within bounds or if an edge doesn't exist, return 0
graph_edge_weight {
    if i or j is not within bounds || edge does not exist {
        return 0;
    }
    return edge_weight
}

//returns true if vertex v has been visited and false otherwise
graph_visited {
    if vertex v == visited {
        return true
    }
    return false
}

//debug function to print graph to check if graph ADT works correctly
graph_print {
    for each index in Graph {
        print Graph[index]
    }
}
```

```
//MANIPULATOR FUNCTIONS//

//adds an edge of weight k from vertex i to vertex j
//if graph is undirected add an edge, with weight k from j to i
//return true if both vertices are within bounds and the edges are sucessfully added
//return false otherwise
//an edge exists if it has a non zero, positive weight.
graph_add_edge {
    sucessful = false
    if j and i are in bounds {
        if k is non zero and pos {
            set matrix[j][i] to k
        }
        if g is undirected and k is non zero and pos {
            set matrix[i][j] to k
        }
        if sucessful {
            return true
        }
        return false
    }
}

//if vertex v is within bounds mark v as visited
graph_mark_visited {
    if vertex is visited {
        return true
    }
    return false
}

//if vertex v is within bounds mark v as unvisited
graph_mark_unvisited {
    if vertex is unvisited {
        return true
    }
    return false
}
```

3 Depth-first Search

DFS first marks the vertex v as having been visited, then it iterates through all of the edges $< v, w >$ recursively calling itself starting at w if w has not already been visited.

3.1 Pseudocode

```
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
}
```

4 Stack ADT

The Stack ADT shall implement an ADT that uses LIFO (last in, first out) policy.

```
//Stack Struct from Asgn Doc
struct Stack {
    uint32_t top;
    uint32_t capacity;
    uint32_t *items;
};
```

4.1 Constructor Function

```
//included in assignment doc
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 Function

```
//included in asgn doc
void stack_delete(Stack **s) {
    if (*s && (*s)->item) {
        free((*s)->items);
        free(*s);
        *s = NULL;
    }
    return;
}
```

4.3 Accessor Functions

```
//supporting pseudocode included below
bool stack_empty(Stack *s);
bool stack_full(Stack *s);
uint32_t stack_size(Stack *s);
bool stack_peek(Stack *s, uint32_t *x);
```

4.4 Manipulator Functions

```
//supporting pseudocode included below
bool stack_push(Stack *s, uint32_t x);
void stack_copy(Stack *dst, Stack *src);

//included in assignment doc
bool stack_pop(Stack *s, uint32_t *x) {
    *x = s->items[s->top];
}
//included in asgn doc
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, " -> ");
}
```

4.5 Pseudocode

```
//ACCESSOR FUNCTIONS//
stack_empty {
    if stack is empty
        return true
    otherwise
        return false
}
stack_full {}
    if stack is full
        return true
    otherwise
        return false
}
stack_size {
    return the number of items in the stack
}
stack_peek {
    return the value that top is pointing at
}

//MANIPULATOR FUNCTIONS//
stack_push {
    push an item onto the stack
}
stack_pop {
    pop an item off the stack
}
stack_copy {
    copy the stack from a source to destination
}
```

5 Path ADT

```
//Path struct included in Asgn doc
struct Path {
    Stack *vertices;
    uint32_t length;
};
```

5.1 Constructor Function

```
//supporting pseudocode included below
Path *path_create(void);
```

5.2 Destructor Function

```
//supporting pseudocode included below
void path_delete(Path **p);
```

5.3 Accessor Functions

```
//supporting pseudocode included below
uint32_t path_vertices(Path *p);
uint32_t path_length(Path *p);
void path_print(Path *p, FILE *outfile, char *cities[]);
```

5.4 Manipulator Functions

```
//supporting pseudocode included below
bool path_push_vertex(Path *p, uint32_t v, Graph *G);
bool path_pop_vertex(Path *p, uint32_t *v, Graph *G);
void path_copy(Path *dst, Path *src);
```

5.5 Pseudocode

```
//CONSTRUCTOR FUNCTION//
//initializes length to 0
//set vertices as a stack with capacity vertices
path_create {
    set length 0
    vertices = stack_create(vertices)
}
//DESTRUCTOR FUNCTION//
path_delete {
    delete path
    free memory
//ACCESSOR FUNCTIONS//
}
path_vertices {
    return number of vertices in stack
}
path_length {
    return length of path
}
path_print {
    print length
    call stack_print
}

//MANIPULATOR FUNCTIONS//
path_push_vertex{
    push a vertex onto the stack
}
path_pop_vertex{
    pop a vertex off the stack
}
path_copy {
    copy the path from a source to destination
}
```

6 Test Harness

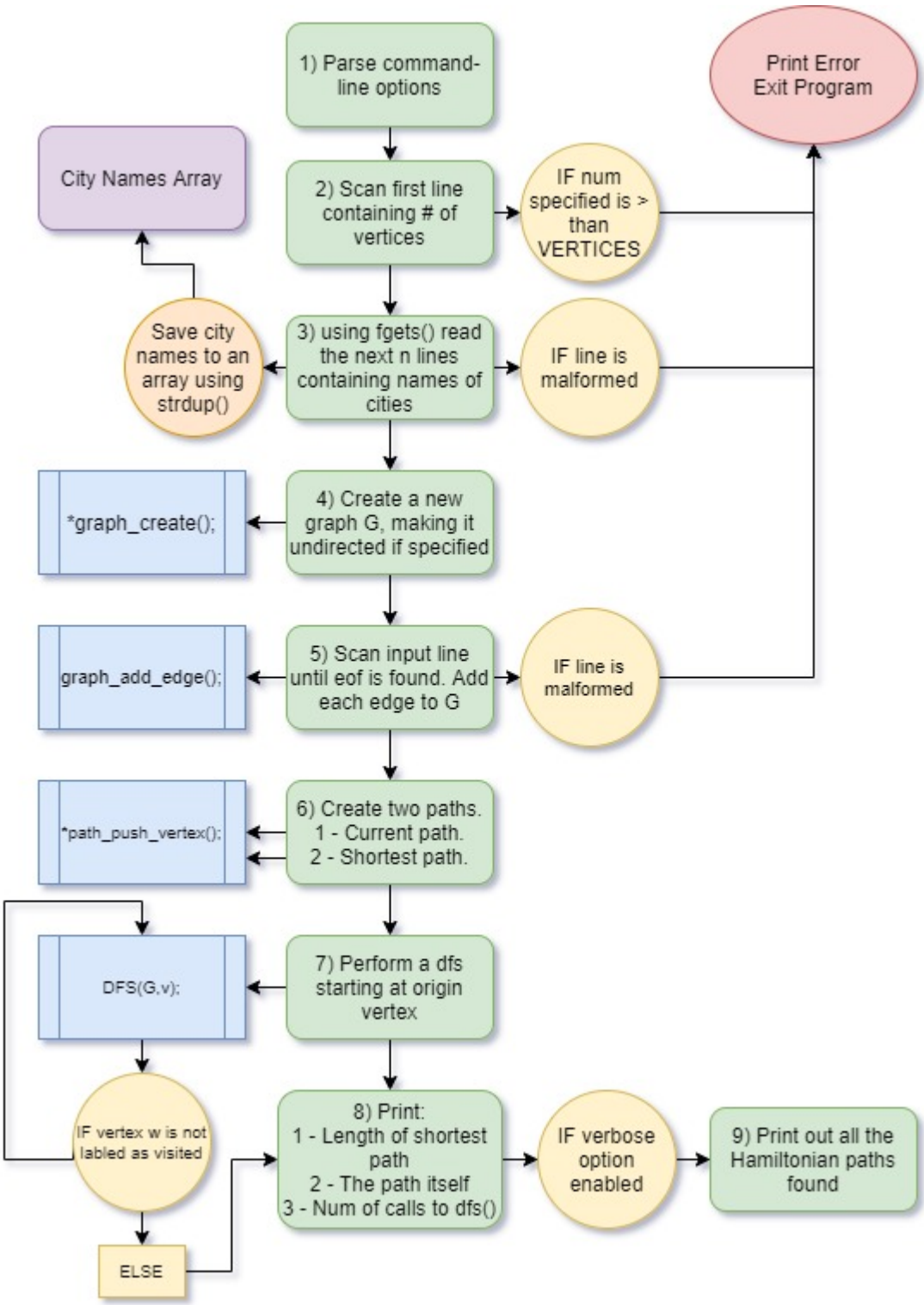
Our test harness will facilitate finding the Hamiltonian path. This will contain our main() function and will use getopt() command-line options to interact and setup the program.

6.1 Command-line Options

```
-h : Help message. Prints the usage of the program
-v : Verbose Printing. Prints each new optimal path as it is found
-u : Uses an undirected graph
-i infile : specifies the input file
-o outfile : specifies the output file
```

6.2 Test Harness Diagram

1. First, parse command-line options with looped calls to getopt(). (See Command-line Options)
2. Scan in the first line from input which contains the number of vertices in the graph. It will print an error if the number specified is greater than macro VERTICES, defined in vertices.h
3. Assuming the number of specified verticies n, read the next n lines from the input using fgets(). Each of these lines will be the name of a city. Then save the name of each city to an array using strdup(). If the line is malformed, print an error and exit.
4. Create a new graph G, making it undirected if specified. (See Graph ADT Design)
5. Scan the input line using fscanf() until eof is found. Add each edge to G. (See Graph ADT Design, Manipulator) If the line is malformed, print and error and exit.
6. Create two paths, One for tracking the current traveled path and the other for tracking the shortest path found. (See Path ADT Design)
7. Starting at the origin vertex, defined by the macro START_VERTEX in vertices.h. Perform a depth-first search on G to find the shortest Hamiltonian path. (See DFS ADT Design)
8. After the DFS, print out the length of the shortest path found, the path itself, and the number of calls to dfs(). (See DFS ADT Accessor)
9. Lastly, If the verbose command-line option was enabled, print out all the Hamiltonian paths that were found as well.


1) Parse command-line options
Print Error Exit Program
City Names Array
2) Scan first line containing # of vertices
IF num specified is > than VERTICES
Save city names to an array using strdup()
3) using fgets() read the next n lines containing names of cities
IF line is malformed
*graph_create();
4) Create a new graph G, making it undirected if specified
graph_add_edge();
5) Scan input line until eof is found. Add each edge to G
IF line is malformed
*path_push_vertex();
6) Create two paths.
1 - Current path.
2 - Shortest path.
DFS(G,v);
7) Perform a dfs starting at origin vertex
IF vertex w is not labled as visited
8) Print:
1 - Length of shortest path
2 - The path itself
3 - Num of calls to dfs()
IF verbose option enabled
9) Print out all the Hamiltonian paths found
ELSE