Aditya Agrawal
aagraw14@ucsc.edu
Spring 2021

CSE13s
Assignment 4: The Circumnavigations of Denver Long
Design Document

This assignment which is often referred to as the travelling salesman problem will require the program to find the shortest possible path through all given locations that returns back to the origin, otherwise known as a Hamiltonian path. This will be accomplished using depth first search as well as the abstract data types stacks, graphs, and paths. All the necessary information to complete the problem like the cities will be read in from a file specified in the command line when executing the program.

graph.c:
typedef

```
struct Graph {
  uint32_t vertices;                    // Number of vertices.
  bool undirected;                      // Undirected graph?
  bool visited[VERTICES];               // Where have we gone?
  uint32_t matrix[VERTICES][VERTICES]; // Adjacency matrix.
};
```

```
Graph *graph_create(uint32_t vertices, bool undirected) {
        initialize and allocate space for graph using malloc
        set graph->vertices and undirected
        for z in visited:
                z =0
        for all elements in matrix
                matrix[][] = 0
}

void graph_delete(Graph **G) {

}

uint32_t graph_vertices(Graph *G) {
        return vertices
}

bool graph_add_edge(Graph *G, uint32_t i, uint32_t j, uint32_t k) {
        if (i or j are not in range)
                return false
        if( graph is undirected )
```

```
            set matrix[i][j] and matrix[j][i] to k
        else
            set matrix[i][j] to k
        return true
}

bool graph_has_edge(Graph *G, uint32_t i, uint32_t j) {
        if ( i or j are not in range )
            return false
        if( matrix[i][j] > 0 )
            return true
        else
            return false
}

uint32_t graph_edge_weight(Graph *G, uint32_t i, uint32_t j) {
        if( i or j are not in range or graph doesnt have edge )
            return 0
        return matrix[i][j]
}

bool graph_visited(Graph *G, uint32_t v) {
        return visited[v]
}

void graph_mark_visited(Graph *G, uint32_t v) {
        if (v is in range)
            visited[v] = true
}

void graph_mark_unvisited(Graph *G, uint32_t v) {
        if (v is in range)
            visited[v] = false
}

void graph_print(Graph *G);
```

path.c:

typedef

```
struct Path {
  Stack *vertices; // The vertices comprising the path.
  uint32_t length; // The total length of the path.
};
```

```
Path *path_create(void) {
        initialize path and allocate space for it
        initialize vertices by calling stack_create(VERTICES)
        initialize length to 0
}

void path_delete(Path **p) {

}

bool path_push_vertex(Path *p, uint32_t v, Graph *G) {
        int x
        if( stack_peek(vertices, &x)  && !stack_full)
                length += matrix[x][v]
        if( !stack_push(vertices, v) )
                return false

        return true
}

bool path_pop_vertex(Path *p, uint32_t *v, Graph *G) {
        if( !stack_pop(vertices, v) )
                return false
        int x
        if( stack_peek(vertices, &x) )
                length -= matrix[x][*v]
        return true
}

uint32_t path_vertices(Path *p) {
        return stack_size(vertices)
}

uint32_t path_length(Path *p) {
        return length
}

void path_copy(Path *dst, Path *src) {

}

void path_print(Path *p, FILE *outfile, char *cities[]) {

}
```

stack.c:
typedef

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

stack_create, stack_delete, stack_size, stack_empty, stack_full, stack_push, stack_pop have same implementation as in asgn3

bool stack_peek(Stack *s, uint32_t *x) {
    if(stack is empty)
        return false
    *x = items[top - 1]
    return true
}

void stack_copy(Stack *dst, Stack *src) {

}

code for stack_print provided by prof. Long in asgn4.pdf

```
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");
}
```

tsp.c

This file will contain the main() function which will parse and obtain the command line inputs with looped calls to getopt(). Some of the command line options the program will accept is -h for a help message, -v for verbose printing (prints all Hamiltonian paths as well as total number of recursive calls to dfs()), -u specifies that the graph will undirected, -i will specify the input file path that contains the cities and edges, -o will specify the output file path. This file will also contain the implementation for the depth first search function which will be used to obtain the Hamiltonian path.

```
1 void dfs(Graph *G, uint32_t v, Path *curr, Path *shortest
    , char *cities[], FILE *outfile);
```

```
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
```