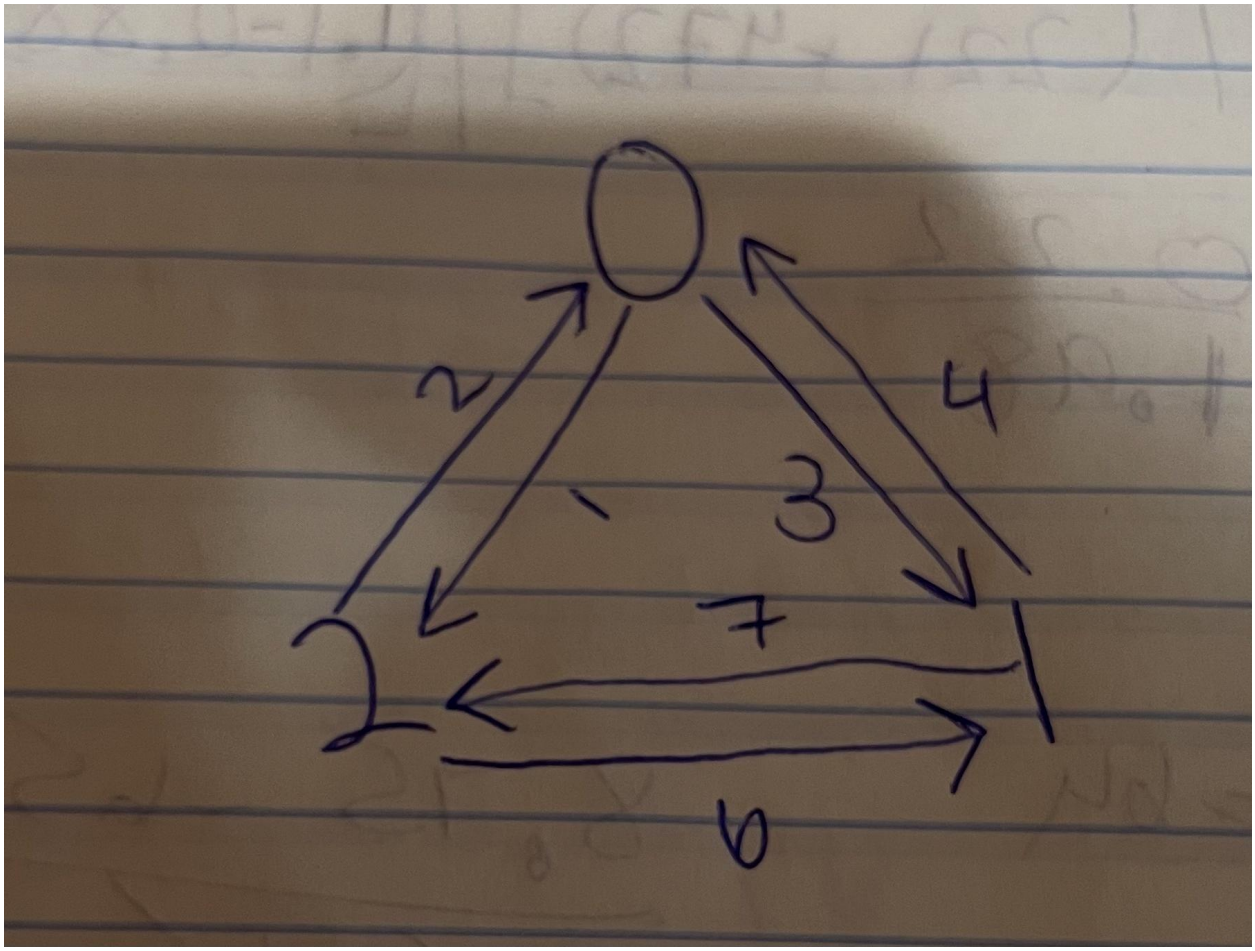
Rahul Vaidun
rvaidun@ucsc.edu
1 April 2021

CSE 13S Spring 2021
Assignment 4: The Circumnavigations of Denver Long
Design Document

I. Description

This program uses depth-first search to find the shortest hamiltonian path with a given graph.



The image above is a visual representation of an example graph. It shows that to go from 0 to 1 will have a weight of 3 but going from 1 to 0 will have a weight of 4 and so on. Given a starting point the program should be able to find the shortest path that goes through all the vertices and back to the starting point.

II. Pseudocode

The program will first parse the command line arguments through getopt. Then it starts reading from the file. The first line of the file is the number of vertices *n*. It will then read the next *n* lines and save those to an array of strings. Then it will read the rest of the file and add the appropriate edge weights. It will then call DFS.

DFS will first mark a vertex as visited and push to the path. Then it will loop through all the vertices and stop when it is finished or the current path is longer than the current shortest path. If the graph is not visited it will recursively call DFS on itself. If the graph is visited but the vertex we are trying to go to is the origin and we have already gone through every vertex, we find a hamiltonian path and push it to the path. Now we check if the current path is shorter than the shortest path currently found. If it is we set the shortest path to the current path and if verbose is enabled we print the path

The following pseudocode is in Python. It shows the implementation of a stack, graph, path and the main drivers.

A. Stack

```
class Stack:
    def __init__(self, capacity):
        self.capacity = capacity
        self.top = 0
        self.items = []

        def stack_empty(self):
            return self.top == 0

        def stack_print(self):
            print(self.items)

        def stack_full(self):
            return self.top == self.capacity

        def stack_size(self):
            return self.top

        def stack_push(self, x):
            if (self.stack_full()):
                return False
            self.items.append(x)
            self.top += 1
```

```
            return True

        def stack_pop(self, x):
            if (self.stack_empty()):
                return False
            self.top -= 1
            x = self.items.pop()
            return True

        def stack_peek(self, x):
            return self.items[len(self.items)]

        def stack_copy(self, x, y):
            y = copy.deepcopy(x)
```

B. Graph

```
class Graph:
    def __init__(self, vertices, undirected):
        self.vertices = vertices
        self.undirected = undirected
        self.visited = []
        self.matrix = [[0 for x in range(vertices)] for
y in range(vertices)]

    def graph_vertices(self):
        return self.vertices

    def graph_add_edge(self, i, j, k):
        self.matrix[i][j] = k
        if self.undirected:
            self.matrix[j][i] = k
```

```
    def graph_has_edge(self, i, j):
        return self.matrix[i][j] > 0

    def graph_edge_weight(self, i, j):
        return self.matrix[i][j]

    def graph_visited(self, v):
        return self.visited[v] == True

    def graph_mark_visited(self, v):
        self.visited[v] = True
        return True

    def graph_mark_unvisited(self, v):
        self.visited[v] = False
        return True

    def graph_print(self, v):
        print(self.matrix)
```

C. Path

```
class Path:
    def __init__(self):
        vertices = 26
        self.vertices = []
        self.length = 0

    def path_push_vertex(self, v, G):
        t = self.vertices[-1]
        self.vertices.append(t)
        if (t != v):
            self.length += G.graph_edge_weight(t, v)
```

```
            return True

    def path_pop_vertex(self, v, G):
        v = self.vertices.pop()
        t = self.vertices[-1]
        if (t != v):
            self.length += G.graph_edge_weight(t, v)
            return True

    def path_vertices(self):
        return len(self.vertices)

    def path_length(self):
        return self.length

    def path_copy(self, dst):
        return copy.deepcopy(self)

    def path_print(self):
        print(self.vertices)
```

D. Main

```
import getopt
import sys
from pseudocode import Graph, Path
verbose = False
undirected = False
recursive_calls = 0
in_fp = sys.stdin
out_fp = sys.stdout
cities = []
curr = None
```

```
shortest = None
graph = None


def dfs(v):
    recursive_calls += 1
    graph.mark_visited(v)
    curr.path_push_vertex(v, graph)
    nottoolong = (shortest.path_length() == 0 or (
        curr.path_length() < shortest.path_length()))

    if nottoolong == False:
        return
    for w in range(graph.graph_vertices):
        if graph.graph_has_edge(v, w):
            if (graph.visited == False):
                dfs(w)
            elif(w == 0):
                if (curr.path_vertices ==
graph.graph_vertices):
                    if (shortest.path_length() == 0 or
(
                            curr.path_length() <
shortest.path_length())):
                        curr.path_push_vertex(v, graph)
                        curr.path_copy(shortest)
                        if verbose:
                            curr.path_print()
                        curr.path_pop_vertex(v)
    curr.path_pop_vertex(v,graph)
    graph.graph_mark_unvisited(v)
    return
```

```
def main():
   global in_fp, verbose, undirected, recursive_calls,
out_fp, cities, curr, shortest, graph
   s = set()
   try:
       opts, args = getopt.getopt(sys.argv[1:],
"vui:o:", ["help"])
   except getopt.GetoptError:
       sys.exit(2)
   for o, a in opts:
       if o == "-h":
           # print_help()
           sys.exit(2)
       elif o == '-v':
           verbose = True
       elif o == '-u':
           undirected = True
       elif o == '-i':
           in_fp = open(a, 'r')
       elif o == '-o':
           out_fp = open(a, 'w')
       else:
           # print_help()
           sys.exit(2)
   first = int(in_fp.readline())
   print(first)
   graph = Graph(first, undirected)
   cities = [in_fp.readline().strip() for i in
range(first)]
   for line in in_fp:
```

```
        line = line.strip().split()
        line = [int(x) for x in line]

        graph.graph_add_edge(line[0], line[1], line[2])
    shortest = Path()
    curr = Path()
    dfs(0)


if __name__ == '__main__':
    main()
```