Anastasia Shpurik

Algorithms CSCI406

Project 3 Deliverable

**I. Problem Modeling**

a) Explain how you modeled the problem as a graph

We were given a set of edges from the map, and I converted each edge into a vertex essentially. I also stored the opposite direction of each edge. When I was finding adjacent paths, I was finding the adjacent paths based on the second location in each edge. Then, with a set of vertices (which are really edges), and adjacency lists for each vertex, I was able to do a depth-first-search to find the correct path. Each vertex had a parent vertex and a state of whether or not it was visited yet. When the search was complete, I back-traced through all the vertices, starting from the ending point, to get the path (based on the parent vertices). This path consists of actual individual points (villages), printed in the correct order, which thus showed the edges to take to connect vertices.

b) Draw enough of the resulting graph to convince us that you have modeled the graph correctly

See attached drawing.

c) Identify the graph algorithm needed to solve the problem

I could have used depth-first-search or breadth-first-search to solve this problem. I used depth-first-search, and this algorithm executed in about 0.003 seconds.

d) Argue that this algorithm will actually solve the problem

This algorithm will solve the problem because DFS finds the shortest path by exhaustively searching all possibilities. I create adjacency lists for each vertex based on the constraints we are given, where we can only travel with free transfers. This is done by checking either if the color is the same or the type of transit is the same for adjacent paths.

**II. Code**

**graph.rb**

#!/usr/bin/env ruby

# Author: Anastasia Shpurik

# Professor: Dinesh Mehta

# Course: CSCI406

# Project3: Graph Modeling

require\_relative 'edge'

#open file with coordinates

file = File.open("input.txt", 'r')

lines = File.read(file).lines

input = []

#place each line of file into array

lines.each do |line|

input.push(line) if (line.length > 0)

end

#store first line of file as number of villages

num\_villages = input[0].split(' ')[0]

#store second line of file as number of routes

num\_lines = input[0].split(' ')[1]

layout = []

i=1

j=0

#store each edge into an array of edges (both directions)

while i<input.length

village1 = input[i].split(' ')[0]

village2 = input[i].split(' ')[1]

color = input[i].split(' ')[2]

type = input[i].split(' ')[3]

layout[j] = Edge.new(village1, village2, color, type, 0)

j+=1

layout[j] = Edge.new(village2, village1, color, type, 0)

j+=1

i+=1

end

#create adjacency lists for each edge

def make\_adjlists(layout)

layout.each do |edge|

start = edge.village1

ending = edge.village2

color = edge.color

type = edge.type\_transit

layout.each do |surround|

if surround.village1 == ending && surround.village2 != start

if surround.color == color || surround.type\_transit == type

edge.add\_adjlist(surround)

end

end

end

end

end

#dfs algorithm

def dfs(layout)

make\_adjlists(layout)

#for each vertex, if it hasn't been visited, visit it

layout.each do |edge|

if edge.visited == 0

dfs\_visit(edge)

end

end

end

def dfs\_visit(edge)

#mark the vertex you are currently on

edge.set\_visited(1)

#for each adjacent vertex, if it hasn't been visited, visit it

edge.adj\_list.each do |adj|

if adj.visited == 0

#assign the parent vertex of each vertex

adj.set\_parent(edge)

dfs\_visit(adj)

end

end

#mark the edge as visited

edge.set\_visited(2)

end

#get all of the parent vertices and construct a path

def storepath(layout, path)

n=0

#the path starts at the end; 'j'

p = layout[138].parent

while p != nil

str = "layout[138]"+".parent"\*n

p = eval(str)

if p != nil

str += ".village2"

path.push(eval(str))

else

break

end

n+=1

end

path.push('A')

end

#print out the path, from beginning to end

def printpath(path)

puts "Path: From Startsburg (A) to Endenville (j)"

n = path.size

while n >= 0

print("#{path[n]} ")

n-=1

end

puts "\n"

puts "Path length: #{path.size-2}"

end

#initialize empty path, do dfs on it

path = []

time = Time.now

dfs(layout)

tot = Time.now - time

puts tot

storepath(layout, path)

printpath(path)

**edge.rb**

class Edge

attr\_accessor :village1

attr\_accessor :village2

attr\_accessor :color

attr\_accessor :type\_transit

attr\_accessor :visited

attr\_accessor :parent

attr\_accessor :adj\_list

def initialize(village1, village2, color, type\_transit, visited)

@village1 = village1

@village2 = village2

@color = color

@type\_transit = type\_transit

#0 means not visited

#1 means currently visiting

#2 means visited

@visited = visited

@parent = nil

@adj\_list = []

end

def add\_adjlist(vertex)

@adj\_list.push(vertex)

end

def print\_edge

puts "Village1: #{@village1}, Village2: #{@village2}, Color: #{@color}, Type: #{@type\_transit}, Visited: #{@visited}, Parent: #{@parent} "

end

def print\_adjlist

@adj\_list.each do |edge|

edge.print\_edge

end

end

def set\_visited(state)

@visited = state

end

def set\_parent(edge)

@parent = edge

end

end

**III. Results**

Path: From Startsburg (A) to Endenville (j)

A B E O U V a f Z U O E B C D J N T S R Q P K F G

L Q W b a Z V W X S Y e d S M I D G K O V b h i j

Path length: 48