#### Jonathan Liao

#### CS 305 HW 6 Report

**1. Questions (include these in your write-up):**

1a. Meets all specifications.

1b.

|  |  |  |  |
| --- | --- | --- | --- |
| **Source** | **Destination** | **Path** | **Miles** |
| PDX | MCO | PDX -> ATL -> MCO | 2572 |
| LAX | PVD | LAX -> SAN -> JFK -> PVD | 3426 |
| ATL | JFK | ATL -> LAX -> SAN -> JFK | 4491 |
| SEA | LAX | SEA -> PDX -> LAX | 965 |
| PHX | DEN | PHX -> SEA -> PDX -> DEN | 5586 |
| PVD | JFK | PVD -> JFK | 114 |
| DFW | ATL | DFW -> SAN -> LAX -> ATL | 3219 |

2. (1 pt) Use **Prim**’s algorithm to find the minimum spanning tree for the following subset of the airport graph. Start the algorithm using vertex LAX.

DFW

1658

SEA

1168

SAN

1051

130

109

2440

LAX

PDX

835

1942

2447

2168

JFK

ATL

In your answer, show the final minimum spanning tree edges and all the vertices.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | DFW | SEA | SAN | PDX | LAX | JFK | ATL |
| Color | white | white | white | white | white | white | white |
| dValue | inf | inf | inf | inf | 0 | inf | inf |
| Pred | NULL | NULL | NULL | NULL | NULL | NULL | NULL |

Process destinations of LAX

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | DFW | SEA | SAN | PDX | LAX | JFK | ATL |
| Color | white | white | white | white | black | white | white |
| dValue | inf | inf | 109 | 835 | 0 | inf | 1942 |
| Pred | NULL | NULL | LAX | LAX | NULL | NULL | LAX |

Process destinations of SAN

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | DFW | SEA | SAN | PDX | LAX | JFK | ATL |
| Color | white | white | black | white | black | White | white |
| dValue | 1277 | 1160 | 109 | 835 | 0 | 2549 | 1942 |
| Pred | SAN | SAN | LAX | LAX | NULL | SAN | LAX |

Process destinations of PDX

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | DFW | SEA | SAN | PDX | LAX | JFK | ATL |
| Color | white | white | black | black | black | white | white |
| dValue | 1277 | 965 | 109 | 835 | 0 | 2549 | 1942 |
| Pred | SAN | PDX | LAX | LAX | NULL | SAN | LAX |

Process destinations of ATL

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | DFW | SEA | SAN | PDX | LAX | JFK | ATL |
| Color | white | white | black | black | black | white | black |
| dValue | 1277 | 965 | 109 | 835 | 0 | 2549 | 1942 |
| Pred | SAN | PDX | LAX | LAX | NULL | SAN | LAX |

Process destinations of DFW

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | DFW | SEA | SAN | PDX | LAX | JFK | ATL |
| Color | Black | white | black | black | black | white | black |
| dValue | 1277 | 965 | 109 | 835 | 0 | 2549 | 1942 |
| Pred | SAN | PDX | LAX | LAX | NULL | SAN | LAX |

Process destinations of DFW

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | DFW | SEA | SAN | PDX | LAX | JFK | ATL |
| Color | Black | white | black | black | black | white | black |
| dValue | 1277 | 965 | 109 | 835 | 0 | 2549 | 1942 |
| Pred | SAN | PDX | LAX | LAX | NULL | SAN | LAX |

Process destinations of JFK

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | DFW | SEA | SAN | PDX | LAX | JFK | ATL |
| Color | Black | white | black | black | black | black | black |
| dValue | 1277 | 965 | 109 | 835 | 0 | 2549 | 1942 |
| Pred | SAN | PDX | LAX | LAX | NULL | SAN | LAX |

Process destinations of SEA

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | DFW | SEA | SAN | PDX | LAX | JFK | ATL |
| Color | Black | black | black | black | black | black | black |
| dValue | 1277 | 965 | 109 | 835 | 0 | 2549 | 1942 |
| Pred | SAN | PDX | LAX | LAX | NULL | SAN | LAX |

DFW

SEA

1168

SAN

2440

130

109

LAX

PDX

835

1942

JFK

ATL

3. This algorithm lists the edges from least cost to most cost in a queue. It then creates edges starting from the top of the queue unless the edge creates a loop.

SAN-LAX: 109

SEA-PDX: 130

PDX-LAX: 835

SEA-SAN: 1051 //edge not created since it would create a loop

DFW-SAN: 1168

SEA-DFW: 1658 //edge not created since it would create a loop

LAX-ATL: 1942

ATL-PDX: 2168 //edge not created since it would create a loop

SAN-JFK: 2440

PDX-JFK: 2447 //edge not created since it would create a loop

DFW

SEA

1168

SAN

130

109

LAX

PDX

835

2440

1942

JFK

ATL

4 a. This was supposed to be an easy homework assignment, but it was confusing at first. In total about 5 hours.

b. Understanding the pre-existing code. All of the different structs were challenging to understand at first.

**Appendix A:** I verify that the code and this write-up were authored by me. I have documented the help I have received in comments in the code files. I have not distributed my code to anyone else except via this homework submission.

**Appendix B**:

/\*

\* CS 305 HW 6

\* Author: Jonathan Liao, based on prior CS 303 HW by Tanya Crenshaw

\* dijkstra.c

\* Graph function implementations, based on adjacency list

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "graph.h"

#include "main.h"

#include <math.h>

void dijkstra(Graph \* g, char \* source) {

//find current node index. We are given it's string name, but need to find the index number in Graph g

int i;

for(i=0; i<g->V; i++) {

if(strcmp(g->array[i].label, source) == 0) { //found the index place

g->array[i].dValue = 0; //found the source. Set its distance to 0 and leave loop

break;

}

}

while(isEmpty(g) == 0) {//while all the nodes are still white

int U = getMin(g);

while(g->array[U].head != NULL) {//while the list of U's neighbors hasn't been exhausted

int neighbor = g->array[U].head->dest; //store the neighbor's node value here

if(g->array[neighbor].dValue > (g->array[U].dValue + g->array[U].head->cost) ) {

g->array[neighbor].dValue = g->array[U].dValue + g->array[U].head->cost;

g->array[neighbor].pred = U;

}

g->array[U].head = g->array[U].head->next; //go on to next neighbor

}

g->array[U].color = BLACK;

}

}

int isEmpty(Graph \* g) {

if(g == NULL) {

return 1; //this graph is empty

}

int i;

for(i=0; i<g->V; i++) {

if(g->array[i].color == WHITE) { //found at least one white node

return 0;

}

}

return 1; //loop unsucesfully found a white node if it made it here

}

int getMin(Graph \* g) {

if(g == NULL) {

return -1; //this graph is empty

}

if(isEmpty(g) == 1) {

return -1; //there are no white nodes

}

int i;

int min = INF; //initialize as first dValue

for(i=0; i<g->V; i++) {

if(g->array[i].color == WHITE && g->array[i].dValue < min ) { //found white node and with node smaller than min

min = i; //index of this node is saved

}

}

return min;

}