In-Class End of Semester Learning

***(50 pts, Due Friday, May 20 )***

***BUT: with 3 classes and a lab left in the semester, you should be able to complete this during class time!!!***

***Difficulty:***

*(Note: the notes below correspond with week 13 videos on Graphing and Dijkstra’s algorithm)*

# Dijkstra: Shortest Path

Single-source shortest path:

■ *Finding the shortest path from a particular vertex to all other vertices on a graph*

* Think of taking a plane from Philadelphia International Airport to SEATAC? – what is the shortest path?
* Note that this can be easily modified to find the shortest path from a single source to any other single source

Other Shortest Path Applications:

|  |  |
| --- | --- |
| ■ | Maps(GPS): finding the shortest route, finding the fastest route   * *Vertices: intersections,* * *Edges: roads (cost: distance, speed, or something else)* |
| ■ | Networks: for routing packets (data) across a network or the internet   * *Vertices: routers* * *Edges: connections (cost: time for travel)* |
| ■ | Epidemiology: modeling the spread of infectious diseases |

* *Vertices: individuals with disease*
* *Edges: contacts (cost: maybe likelihood of infection?)* Assumptions:

|  |  |
| --- | --- |
| ■ | Costs of edges are positive numbers or zero |
| ■ | Not all vertices may be reachable from the single source |
| ■ | Weights are not necessarily distance – could be time, cost, likelihood, etc. |
| ■ | Shortest paths may not be unique |

Graphs can be **Directed** or **undirected**:

**Directed:** means the cost from vertex a to vertex b can be different than the cost from vertex b to vertex a (think of it this way: the cost from a flight from Philly to New Haven is different than the cost of a flight from New Haven to Philly

**Undirected:** means the cost from vertex a to vertex b and vice versa is the same. (Think of it this way: if the distance is the cost, then the distance from Philly to New Haven is the same either way)

*Dijkstra’s algorithm:*

Dijkstra:

* Edsger Dijkstra – Dutch computer scientist
* Responsible for the concept of structured programming, making dealing with complex data management possible.

Dijkstra’s shortest path algorithm:

* solves single-source shortest path problem
* Works on both directed and undirected graphs.
  + all edges must have nonnegative weights.
* Approach: **Greedy** (means make a locally optimal choice (i.e.,choose the next shortest path) in the hope that the ultimate result will be optimal)
* Input: Weighted graph o (weighted graph means edges have a weight assigned to them), E.g.,
  + - distance between 2 places
    - Cost of flight
    - Length of time taken to travel between the 2 edges ▪ Etc.
  + G={E,V}
    - Graph is a set of edges and vertices o source vertex *v*V,
    - this is the starting vertex in the graph
    - and the starting vertex has to be in the set of all the vertices o all edge weights are nonnegative
* Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex *v*V to all other vertices
* *Basic Idea:* 
  + We’ve got a graph with edges and distances between connected edges

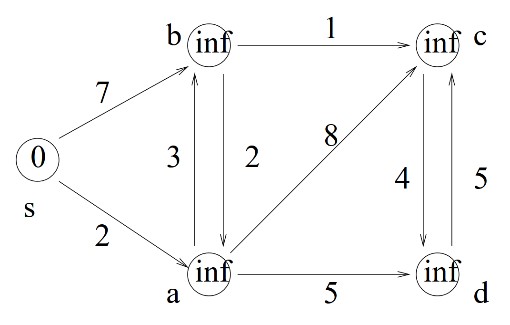
*(may be represented as a 2-dimensional array)* o Start by creating an array of distances, and initiating all shortest-path distances to infinity (or a really big number) except the distance from our starting vertex (0)

* + All vertexes go into a priority queue (bin heap perhaps) based on their distance from the initial vertex. (another array)
  + pick the unvisited vertex with the shortest distance, o Add it to the set of visited nodes.
  + Remove it from the queue.
  + Recalculate all distances from the original vertex to all other vertices not visited yet by taking the minimum of the distance from the original vertex to vx or the distance from ther original vertex to the current vertex + the distance from the current vertex to vx
  + Continue until all vertices have been visited (or no more vertices can be visited)
* We’re constructing the shortest path edge by edge, o Always adding the next vertex that is the shortest distance from the initial vertex
* Determining the shortest edge to add:
  + Assume we have an edge between vertices u and v, with a cost o We already know the cost to get from the original vertex to u o We also have a known cost to get from the original vertex to v o We update the costs so that if cost to u + cost from u to v is less than the cost to v, then we update the path so that it contains the new, shorter path to get to v

<orig,….u,v> pred[v] = u

Example:

**Step 1.** Initialize:



The above matrix would be represented as the matrix, below (I chose to use -1 to represent infinity, meaning there’s no direct edge between two vertices. You could easily choose a number larger than all the paths combined, or just a really really large number):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | S | A | B | C | D |
| S | 0 | 2 | 7 | -1 | -1 |
| A | -1 | 0 | 3 | 8 | 5 |
| B | -1 | 2 | 0 | 1 | -1 |
| C | -1 | -1 | -1 | 0 | 4 |
| D | -1 | -1 | -1 | 5 | 0 |

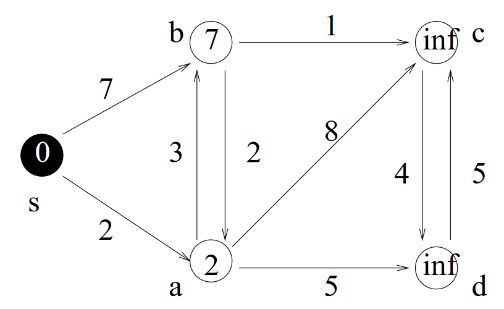
Assume all edges are infinitely away from the original vertex s (our starting vertex) Set visited to False for all vertices

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| V | S | A | B | C | D |
| D[v] | 0 | Inf | Inf | Inf | Inf |
| Prev[v] | null | null | null | null | null |
| Visited | False | False | False | False | False |

Priority Queue:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| V | S | A | B | C | D |
| D[v] | 0 | Inf | Inf | Inf | inf |

**Step 2:** update adjacent edges to the initial vertex:



B and a are adjacent to a vertex in the minimum spanning tree so far (in this case, s), so we’ll update information for a and b

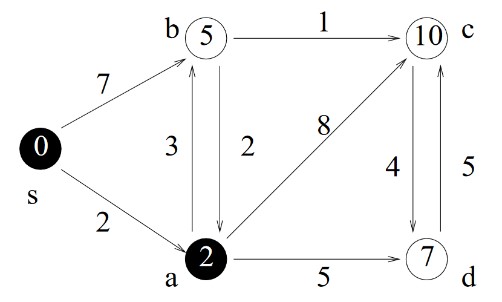
Update s as being visited

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| V | S | A | B | C | D |
| D[v] | 0 | 2 | 7 | Inf | Inf |
| Prev[v] | null | s | s | null | null |
| Visited | True | False | False | False | False |

Priority Queue (note that s has been removed from the priority queue because it has been visited:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| V | A | B | C | D |
| D[v] | 2 | 7 | Inf | inf |

**Step 3:** A has the minimum edge in the priority queue, so we’ll add A to the minimum spanning tree so far. B,c,d are adjacent to a vertex in the minimum spanning tree so far, so we’ll update the information for b,c,d :



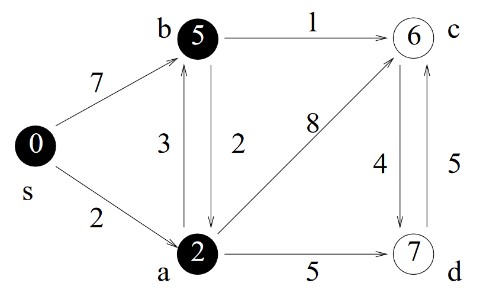
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| V | S | A | B | C | D |
| D[v] | 0 | 2 | 5 | 10 | 7 |
| Prev[v] | null | s | a | a | a |
| Visited | True | True | False | False | False |

Priority Queue (note that a has been removed from the priority queue because it has been visited:

|  |  |  |  |
| --- | --- | --- | --- |
| V | B | C | D |
| D[v] | 5 | 10 | 7 |

**Step 4:**

Now b has the minimum edge in the priority queue, so we’ll add B to our minimum spanning tree so far. A and c are adjacent to at least one vertex in our minimum spanning tree, so update information on A and c



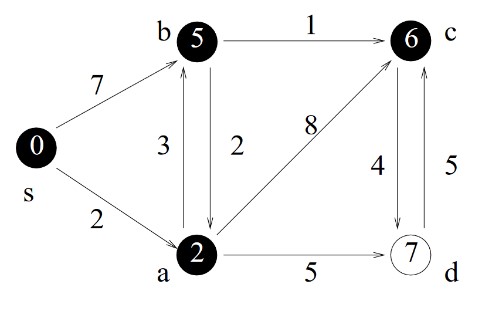
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| V | S | A | B | C | D |
| D[v] | 0 | 2 | 5 | 6 | 7 |
| Prev[v] | null | s | a | b | a |
| Visited | True | True | True | False | False |

Priority Queue (note that b has been removed from the priority queue because it has been visited:

|  |  |  |
| --- | --- | --- |
| V | C | D |
| D[v] | 6 | 7 |

**Step 5:**

Now c has the minimum edge in the priority queue, so we’ll add c to our minimum spanning tree so far. D is adjacent to at least one vertex in our minimum spanning tree, so update information on d



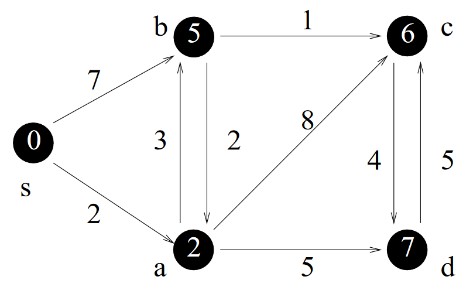
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| V | S | A | B | C | D |
| D[v] | 0 | 2 | 5 | 6 | 7 |
| Prev[v] | null | s | a | b | a |
| Visited | True | True | True | True | False |

Priority Queue (note that c has been removed from the priority queue because it has been visited:

|  |  |
| --- | --- |
| V | D |
| D[v] | 7 |

**Step 6:**

Now d has the minimum edge in the priority queue, so we’ll add d to our minimum spanning tree so far. Note that c is adjacent to d, so we need to update information on c



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| V | S | A | B | C | D |
| D[v] | 0 | 2 | 5 | 6 | 7 |
| Prev[v] | null | s | a | b | a |
| Visited | True | True | True | True | True |

Priority Queue is now empty so we’re done

**To find the minimum path** from any vertex to the starting vertex, just take the last vertex and follow the prev nodes back to s

So the minimum path from s to c:

C’s prev is b. B’s prev is A. A’s prev is S. We have our path!

**PROBLEMS:**

1. **(5) Given the following path, find the shortest path from vertex e to vertex h in the graph using Dijkstra’s algorithm. Show the steps:**

**O**

**R**

**H**

**5**

**E**

**S**

**3**

**A**

**T**

**L**

2

6

6

7

1

2

9

5

4

3

5

2

2

1. **(5) Given the following matrix, draw the directed graph:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **R** | **I** | **L** | **A** | **P** | **T** |
| **R** | **0** | **-1** | **-1** | **-1** | **-1** | **-1** |
| **I** | **5** | **0** | **3** | **-1** | **-1** | **-1** |
| **L** | **14** | **-1** | **0** | **-1** | **-1** | **-1** |
| **A** | **-1** | **13** | **-1** | **0** | **2** | **-1** |
| **P** | **-1** | **5** | **12** | **-1** | **0** | **-1** |
| **T** | **-1** | **-1** | **-1** | **4** | **9** | **0** |

1. **(4) In this graph, there are more than one minimum path between T and R. Show the path that would be found using Dijkstra’s algorithm, including the in-between steps:**

1. **(3) The above graph is directed, meaning the path from a to b can have a different cost than the path from b to a. What would the matrix look like if the graph was undirected?**

1. **(4) What would be the problem with modifying the above algorithm so that it found the largest-cost path instead of the smallest cost path?**

**Part B (The coding part): (26 pts – 5 pts for getting everything working together)**

**I’m giving you the outline and basic steps of Dijkstra’s algorithm. Your job is to fill in the 3 methods in the Graph class to make Dijkstra’s algorithm work: Graph.hpp:**

#ifndef GRAPH\_HPP\_

#define GRAPH\_HPP\_

#include <iostream> #include <stdlib.h> using namespace std;

class Graph {

int \*\*adjMatrix; int numOfVerts; int start; string \*dataArr; //BinHeap priorityQueue(); bool \*visited; int \*distances;

int \*prev; public:

Graph(int n, int first, string vertexnames[]);

void dijkstra(); void setDistances(int lv); int minDistance(); void printAdjMat(); void printInfoSoFar(); void printPath(int end);

void genGraph();

};

#endif /\* GRAPH\_HPP\_ \*/

**Graph.cpp:**

#include "Graph.hpp"

#include <iostream>

#include <stdlib.h> #include <time.h> using namespace std;

Graph::Graph(int n, int first, string vertexnames[]) { numOfVerts = n;

start = first; dataArr = new string[n]; for (int i = 0; i < n; i++) {

dataArr[i] = vertexnames[i];

}

distances = new int[n]; adjMatrix = new int\*[n]; visited = new bool[n]; prev = new int[n];

for (int i = 0; i < n; i++) {

adjMatrix[i] = new int[n];

}

genGraph();

for (int i = 0; i < n; i++) {

distances[i] = 1000000; visited[i] = false;

prev[i] = -1;

}

printAdjMat();

dijkstra();

printPath(1);

}

void Graph:: dijkstra(){ // 5 pts

//Step 1:

//set the distance to the starting vertex to 0 and set

//the visited array to true for the start index;

//Step 2:

// Initialize the distances to the cost of going to each node from the

//start index (this is done using the adjacency matrix)

//Step 3:

//loop until every vertex has been visited, calling the methods

//minDistance to find the next unvisited vertex with the minimum

//distance, and then calling setDistances method for every vertex

//to update distances for the unvisited vertices. (I called printInfoSoFar()

//in this loop to see the progress of the algorithm)

}

void Graph::setDistances(int latestVert) { //8 pts

// This method updates the distances array with the costs being

//updated to either their cost so far, or the cost of

//traveling through the recently visited vertex + the cost of

//traveling from that vertex to the new vertex (whichever is the

//minimum). If the minimum is through the recently visited vertex,

//then update the previous array so that it holds the latest visited

//vertex's index number

}

int Graph::minDistance( ){ //8 pts

//This method finds the next unvisited vertex with the minimum

//distance.

//Once the minimum is found (along with its index in the distance //array), the visited array at that index is set to True and that index is //returned from this method.

}

//This method prints out the final path from the starting vertex to the end vertex,

//which is the index passed into this method void Graph::printPath(int end){

int \*temppath = new int[numOfVerts];

int ct = 0; temppath[ct] = end; int dist = distances[end];

int prevnode = prev[end];

ct++;

while (prevnode != start) {

temppath[ct] = prevnode;

prevnode = prev[prevnode];

ct++;

}

temppath[ct] = start; cout << "Shortest Path: " << dist<<endl; for (int i = ct; i >= 0; i--) {

cout << dataArr[temppath[i]] << "("<< temppath[i]<<")->";

}

cout << endl;

}

//This method prints out the adjacency matrix with the distances void Graph::printAdjMat() {

cout <<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl << "Adjacency Matrix (Graph):"<<endl; for (int i = 0; i< numOfVerts;i++) {

for (int j = 0; j < numOfVerts; j++) {

cout << adjMatrix[i][j] << "\t";

}

cout << endl;

}

cout <<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

}

//This method prints out the information in the distance array, the previous array, and the visited array

//It is called so you can watch the progress of the construction of the shortest path void Graph::printInfoSoFar() {

cout <<"\t";

for (int i = 0; i < numOfVerts; i++) {

cout << "\t"<<i;

}

cout << endl;

cout << "Dist:\t";

for (int i = 0; i < numOfVerts; i++) {

cout << "\t"<<distances[i];

}

cout << endl;

cout << "Prev:\t";

for (int i = 0; i < numOfVerts; i++) {

cout << "\t"<<prev[i];

}

cout << endl; cout << "Visited:";

for (int i = 0; i < numOfVerts; i++) {

cout << "\t"<<visited[i];

}

cout << endl; cout << endl;

}

//This method generates the distances for the graph. My way of representing vertices that are

//not connected was to set those distances to 100 (it could just as easily been 1000000, but

//that didn’t look as nice when I printed out the adjacency matrix) void Graph::genGraph() {

srand(time(NULL));

for (int i = 0; i < numOfVerts; i++) { for (int j = 0; j < numOfVerts; j++) {

if (i == j) { adjMatrix[i][j] = 0;

}

else {

adjMatrix[i][j] = rand() % 9 + 1; if (adjMatrix[i][j] == 9) {

adjMatrix[i][j] = 100;

}

}

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ graphMain.cpp

#include "Graph.hpp"

#include <iostream>

#include <stdlib.h> #include <time.h>

using namespace std;

int main() {

srand (time(NULL)); string names[12] =

{"Albany","Buffalo","Chicago","Detroit","Encino","Fargo","Gotham","Houston","Indianapolis","Jackson","Kenosha","Lewiston"}; int size =4;

Graph g1(size, 3, names);

cout<<endl<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl; size = 7;

Graph g2(size,6,names);

cout<<endl<<"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<<endl; size = 12;

Graph g3(size,11,names);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**Potential Output: (it’s random, so it will be different for each run)**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Adjacency Matrix (Graph):

0 100 8 1

5 0 3 2

1. 2 0 1
2. 3 3 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

0 1 2 3

Dist: 4 3 3 0

Prev: 3 3 3 -1

Visited: 0 0 0 1

0 1 2 3

Dist: 4 3 3 0

Prev: 3 3 3 -1

Visited: 0 0 1 1

0 1 2 3

Dist: 4 3 3 0

Prev: 3 3 3 -1

Visited: 0 1 1 1

0 1 2 3

Dist: 4 3 3 0

Prev: 3 3 3 -1

Visited: 1 1 1 1

Shortest Path: 3

Detroit(3)->Buffalo(1)->

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Adjacency Matrix (Graph):

0 100 8 1 5 3 2

3 0 2 1 4 3 3

5 3 0 7 2 100 100

4 4 6 0 2 2 8

4 7 1 2 0 100 6

1. 6 7 4 4 0 4
2. 8 8 1 5 7 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

0 1 2 3 4 5 6

Dist: 2 8 8 1 5 7 0

Prev: 6 6 6 6 6 6 -1

Visited: 0 0 0 0 0 0 1

0 1 2 3 4 5 6

Dist: 2 5 7 1 3 3 0

Prev: 6 3 3 6 3 3 -1

Visited: 0 0 0 1 0 0 1

0 1 2 3 4 5 6

Dist: 2 5 7 1 3 3 0

Prev: 6 3 3 6 3 3 -1

Visited: 1 0 0 1 0 0 1

0 1 2 3 4 5 6

Dist: 2 5 7 1 3 3 0

Prev: 6 3 3 6 3 3 -1

Visited: 1 0 0 1 0 1 1

0 1 2 3 4 5 6

Dist: 2 5 4 1 3 3 0

Prev: 6 3 4 6 3 3 -1

Visited: 1 0 0 1 1 1 1

0 1 2 3 4 5 6

Dist: 2 5 4 1 3 3 0

Prev: 6 3 4 6 3 3 -1

Visited: 1 0 1 1 1 1 1

0 1 2 3 4 5 6

Dist: 2 5 4 1 3 3 0

Prev: 6 3 4 6 3 3 -1

Visited: 1 1 1 1 1 1 1

Shortest Path: 5

Gotham(6)->Detroit(3)->Buffalo(1)->

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Adjacency Matrix (Graph):

0 100 8 1 5 3 2 3 2 1 4 3

3 0 5 3 7 2 100 100 4 4 6 2

2 8 0 4 7 1 2 100 6 1 6 7

4 4 4 0 2 8 8 1 5 7 6 6

1 3 5 5 0 1 100 4 8 2 1 4

1 2 8 4 4 0 5 1 1 1 100 100

6 8 4 8 7 8 0 3 5 100 6 3

8 8 100 100 100 1 8 0 1 4 1 2

8 8 8 2 6 4 6 8 0 6 1 1

8 1 1 100 8 6 7 6 2 0 6 6

7 1 100 100 8 2 8 8 1 4 0 4

3 100 6 3 1 3 5 5 5 4 7 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

0 1 2 3 4 5 6 7 8 9 10

11

Dist: 3 100 6 3 1 3 5 5 5 4 7

0

Prev: 11 11 11 11 11 11 11 11 11 11 11

-1

Visited: 0 0 0 0 0 0 0 0 0 0 0 1

0 1 2 3 4 5 6 7 8 9 10

11

Dist: 2 4 6 3 1 2 5 5 5 3 2

0

Prev: 4 4 11 11 11 4 11 11 11 4 4

-1

Visited: 0 0 0 0 1 0 0 0 0 0 0 1

0 1 2 3 4 5 6 7 8 9 10

11

Dist: 2 3 6 3 1 2 5 5 3 3 2

0

Prev: 4 10 11 11 11 4 11 11 10 4 4

-1

Visited: 0 0 0 0 1 0 0 0 0 0 1 1

0 1 2 3 4 5 6 7 8 9 10

11

Dist: 2 3 6 3 1 2 5 3 3 3 2

0

Prev: 4 10 11 11 11 4 11 5 10 4 4

-1

Visited: 0 0 0 0 1 1 0 0 0 0 1 1

0 1 2 3 4 5 6 7 8 9 10

11

Dist: 2 3 6 3 1 2 4 3 3 3 2

0

Prev: 4 10 11 11 11 4 0 5 10 4 4

-1

Visited: 1 0 0 0 1 1 0 0 0 0 1 1

0 1 2 3 4 5 6 7 8 9 10

11

Dist: 2 3 4 3 1 2 4 3 3 3 2

0

Prev: 4 10 9 11 11 4 0 5 10 4 4

-1

Visited: 1 0 0 0 1 1 0 0 0 1 1 1

0 1 2 3 4 5 6 7 8 9 10

11

Dist: 2 3 4 3 1 2 4 3 3 3 2

0

Prev: 4 10 9 11 11 4 0 5 10 4 4

-1

Visited: 1 0 0 0 1 1 0 0 1 1 1 1

0 1 2 3 4 5 6 7 8 9 10

11

Dist: 2 3 4 3 1 2 4 3 3 3 2

0

Prev: 4 10 9 11 11 4 0 5 10 4 4

-1

Visited: 1 0 0 0 1 1 0 1 1 1 1 1

0 1 2 3 4 5 6 7 8 9 10

11

Dist: 2 3 4 3 1 2 4 3 3 3 2

0

Prev: 4 10 9 11 11 4 0 5 10 4 4

-1

Visited: 1 0 0 1 1 1 0 1 1 1 1 1

0 1 2 3 4 5 6 7 8 9 10

11

Dist: 2 3 4 3 1 2 4 3 3 3 2

0

Prev: 4 10 9 11 11 4 0 5 10 4 4

-1

Visited: 1 1 0 1 1 1 0 1 1 1 1 1

0 1 2 3 4 5 6 7 8 9 10

11

Dist: 2 3 4 3 1 2 4 3 3 3 2

0

Prev: 4 10 9 11 11 4 0 5 10 4 4

-1

Visited: 1 1 0 1 1 1 1 1 1 1 1 1

0 1 2 3 4 5 6 7 8 9 10

11

Dist: 2 3 4 3 1 2 4 3 3 3 2

0

Prev: 4 10 9 11 11 4 0 5 10 4 4

-1

Visited: 1 1 1 1 1 1 1 1 1 1 1 1

Shortest Path: 3

Lewiston(11)->Encino(4)->Kenosha(10)->Buffalo(1)->