Report on Dijkstra’s Shortest Path Algorithm

Yu Sheng

4/4/2018

In this lab, we are implementing the Dijkstra’s Shortest Path Algorithm in Java to find the shortest path from source vertex to destination vertex. Since vertexes are defined as integers, we only need edge classes and graph class in this project.

Edge class defines edges. Since this is a directed and weighted graph, an edge (from u to v with weight w) has three components, the initial(s) vertex, terminal(e) vertex and weight(w) of that edge. Initial and terminal vertexes in this lab are defined in integers and weights are some floats. Thus when constructing new edge, we need those three values s, e and w. And since in some case we would use one specific component of edge, I created three separate get value function for those three components which simply returns s, e and w separately. Besides, I set the value toString() function of edge to be “s->e w” so that we could print the edge the same as shown in the sample output.

As for graph class, I did not use adjacency matrix which will works easier when implementing the algorithm but wasting too much space (for those 0 entry) when dataset becomes large. My graph is an adjacency list where each line contains the vertex and all the directed edges start from this vertex. I used data structures as an array of Arraylist of edge to represent the graph. So there are two components in graph class, first is the size of the array which is an integer, second one is array adjlist. The ith element in adjlist is an Arraylist of edges which starts from vertex i, for example adjlist[0]={edge(0,4,0.38), edge(0,2,0.26)}in graph imported from tinyEWD.txt. I defined several also functions in graph class. First is init(), we need to initialize the graph first every time we create a new graph, as build a new Arraylist of edges for each vertex, because only after initialize then we can add edge to the Arraylist. Then I created a addedge(edge e) which add edge e to the edges Arraylist of initialize vertex of e(e.gete()). And function getedges(int i) just return all the edges starts from vertex i. Besides, getedge(int i, int j) will return the edge from i to j if there is one. And getsize() will tell the size of the graph. Last function in graph class is readGraph(Scanner f, int digits), where f is our data file and digits depends on the size of datafile. Like in tinyEWD.txt, every vertex is a one digit number than digits=1 here, but in mediumEWD.txt, vertexes are over hundred thus digits=3. This ‘digits’ is very useful when we create separate the information in each line. I observed that every line of scanner f contains three numbers separated by space which are s, e and w of an edge. However, since in mediumEWD.txt there will be more than one space before a vertex(when vertex<100), we cannot simply use parsing by space. Thus I found out the exact position of each number using digits, used substring() function to take them out and then used trim() to eliminate the space before the number and then assign to the correct component of an edge. Since s,w are integers and w is a float, I use Integer.parseInt and Float.parseFloat to convert substrings to the desired type. After doing that to the new line scanned in, we will have a new edge and we can add this edge to the graph using addedge(e). After scanning the whole file, our graph is now built.

Additionally, we need two structures to record shortest path. First is the *pred* array which tells you what is the exact path, thus the previous vertexes are stored in the *pred* array and pred will be an integer array. Second is the *shortest* array which records the overall weight of shortest path from source vertex to the vertex i. And since weights are in float, this *shortest* array will be a float array.

Once we have the graph, pred[], shortest, source vertex (we don’t need destination vertex here because Dijkstra’s Algorithm will produce shortest paths from source vertex to all other vertexes), we can now employ the Dijkstra’s Algorithm to find the shortest path. In order to avoid the use of global variables and also restricted by the fact that one function cannot return two arrays, I initialized the pred[] and shortest[] (shortest[v]=infinity) in main function and pass them as parameters to the DIJKSTRA(). In Dijkstra’s algorithm, we need a Q to store all the vertexes we have not visited, and we will need to choose the vertex in Q with lowest shortest value to do relaxation. Thus I used Arraylist to implement Q. And I wrote a min(Q,shortest[]) functions to extract the vertex(u) with lowest shortest value and remove u from Q. In book “Algorithm Unlocked”, this implementation of Q using Arraylist cause O(n^2) complexity for the DIJKSTRA() function. It works more efficiently than using priority queue when graph is dense.

In conclusion, all the data structures and abstractions that I used are listed follows:

graph G=new graph(n);

graph class:

**private** **int** V;

**private** Arraylist<edge> [] adjlist=(Arraylist<edge>[])**new** Arraylist[V];

**public** **void** init();

**public** **void** addEdge(edge e);

**public** ArrayList<edge> getedges(**int** s);

**public** edge getEdge(**int** u, **int** v);

**public** **int** getsize();

**public** graph readGraph(Scanner f,**int** digits);

edge class:

**private** **int** s;

**private** **int** e;

**private** **float** w;

**public** edge(**int** s,**int** e,**float** w) ;

**public** **int** gets();

**public** **int** gete();

**public** **float** getw();

float [] shortest=new float [n];

int [] pred=new int[n];

Arraylist <Integer> Q=new Arraylist <Integer>();

**Code**

//main program

public class task {

public static void main(String[] args) {

Scanner inS;

try {

URL webFile = new URL("https://cs.brynmawr.edu/Courses/cs330/spring2018/mediumEWD.txt");

inS = new Scanner(webFile.openStream());

int n = Integer.parseInt(inS.nextLine());

int m = Integer.parseInt(inS.nextLine());

graph G=new graph(n);

G.init();

int tiny=1;

int medium=3;

G.readGraph(inS,medium);

System.***out***.println("Using file: tinyEWD.txt");

float [] shortest=new float [n];

int [] pred=new int[n];

Scanner cw = new Scanner(System.in);

System.out.print("Enter the source vertex: ");

int s=cw.nextInt();

DIJKSRRA(G,s,shortest,pred);

System.out.print("Enter the destination vertex: ");

int d=cw.nextInt();

if(shortest[d]!=Integer.MAX\_VALUE){

System.out.print("There is a path from "+s+" to "+d+".\n");

System.out.print("The shortest path has a cost "+shortest[d]+". Here it is:\n");

int x=d;

String path=G.getEdge(pred[x], x).toString();

while((x=pred[x])!=s){

path=G.getEdge(pred[x], x).toString()+"\n"+path;

}

System.out.println(path);

}else{

System.out.println("There is no path from "+s+" to "+d+".");

}

} catch (IOException e) {

e.printStackTrace();

System.exit(1);

} // catch

}

public static void DIJKSRRA(graph G,int s,float[]shortest,int[]pred){

int n=G.getsize();

//float [] shortest=new float [n];

//int [] pred=new int[n];

Arraylist <Integer> Q=new Arraylist <Integer>();

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

shortest[i]=Integer.MAX\_VALUE;

Q.add(i);

}

shortest[s]=0;

while(!Q.isEmpty()){

int u=min(Q,shortest);

for(int i=0;i<G.getedges(u).size();i++){

RELAX(u,G.getedges(u).get(i).gete(),shortest,pred,G);

}

}

}

public static void RELAX(int u,int v,float [] shortest,int [] pred,graph G){

if(shortest[u]+G.getEdge(u, v).getw()<shortest[v]){

shortest[v]=shortest[u]+G.getEdge(u,v).getw();

pred[v]=u;

}

}

public static int min(Arraylist <Integer> a,float[] shortest){

int min;

int minIndex=0;

for(int i=1;i<a.size();i++){

if (shortest[a.get(minIndex)]>shortest[a.get(i)])

minIndex=i;

}

min=a.get(minIndex);

a.remove(minIndex);

return min;

}

}

//graph class

**import** java.util.Arraylist;

**import** java.util.Collection;

**import** java.util.Scanner;

**public** **class** graph {

**private** **int** V;

**private** Arraylist<edge> [] adjlist=(Arraylist<edge>[])**new** Arraylist[V];

**public** graph(**int** n) {

**this**.V=n;

**this**.adjlist=(Arraylist<edge>[])**new** Arraylist[n];;

}//constructer

**public** **void** init(){

**for**(**int** i=0;i<**this**.V;i++){

**this**.adjlist[i]=**new** Arraylist<edge>();

}

}

**public** **void** addEdge(edge e) {

**this**.adjlist[e.gets()].add(e);

}

**public** Arraylist<edge> getedges(**int** s){

**return** **this**.adjlist[s];

}

**public** edge getEdge(**int** u, **int** v) {

**for**(**int** i=0;i<**this**.adjlist[u].size();i++){

**if**(**this**.adjlist[u].get(i).gete()==v)

**return** **this**.adjlist[u].get(i);

}

**return** **null**;

}

**public** **int** getsize(){

**return** **this**.V;

}

**public** graph readGraph(Scanner f,**int** digits){

**while** (f.hasNextLine()) {

String line = f.nextLine();

**int** s= Integer.*parseInt*(line.substring(0,digits+1).trim());

**int** e=Integer.*parseInt*(line.substring(digits+1,2\*digits+1).trim());

**float** w=Float.*parseFloat*(line.substring(2\*digits+1).trim());

edge Edge=**new** edge(s,e,w);

**this**.addEdge(Edge);

}

f.close();

**return** **this**;

}

}// place()

//edge class

**public** **class** edge {

**private** **int** s;

**private** **int** e;

**private** **float** w;

**public** edge(**int** s,**int** e,**float** w) {

**this**.s=s;

**this**.e=e;

**this**.w=w;

}//constructer

**public** **int** gets() {

**return** s;

}

**public** **int** gete() {

**return** e;

}

**public** **float** getw() {

**return** w;

}

**public** String toString() {

**return** s+"->"+e+" ("+w+")";

}

}// place()

**Output**

Using file: tinyEWD.txt

Enter the source vertex: 0

Enter the destination vertex: 6

There is a path from 0 to 6.

The shortest path has a cost 1.51. Here it is:

0->2 (0.26)

2->7 (0.34)

7->3 (0.39)

3->6 (0.52)

Using file: medium.EWD.txt

Enter the source vertex: 0

Enter the destination vertex: 1

There is a path from 0 to 1.

The shortest path has a cost 0.71178. Here it is:

0->44 (0.06471)

44->93 (0.06793)

93->187 (0.0764)

187->77 (0.10655)

77->78 (0.10966)

78->128 (0.03633)

128->69 (0.11896)

69->107 (0.0564)

107->1 (0.07484)

Using file: medium.EWD.txt

Enter the source vertex: 0

Enter the destination vertex: 4

There is a path from 0 to 4.

The shortest path has a cost 0.42292. Here it is:

0->44 (0.06471)

44->93 (0.06793)

93->187 (0.0764)

187->77 (0.10655)

77->4 (0.10733)