Obtaining the Most Effective Mass Public Transit Route through the Combination of Graph Theory and Dijkstra's Algorithm

Tirta Wening Rachman - 13512004[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]*tirtawr@std.itb.ac.id*

***Abstract*— As large cities like New York or Jakarta grow larger and busier, it becomes increasingly difficult to determine which route of public transit to take from point a to point B. A solution for this problem can be obtained through the combination of Graph Theory and Dijkstra's algorithm.**

***Index Terms*—Dijkstra's Algorithm, Computer Modeling, Graph Theory, Mass Public Transit**

I. INTRODUCTION

Aristotle concedes that, "human beings are social animals." Ever since the dawn of civilization, human beings have always felt the need to be involved in social activities. There was a time when for one to partake in a social activity, all that one requires is the will to go out of the house. However, when one resides in a big major city like New York City of Jakarta, it requires one to travel a considerable amount of distance to be involved in the social activity of your choosing.

While some people are able to get around trough the use of their personal vehicles, most of urban dwellers get around trough the use of public transit. Furthermore, as the city grows larger and busier, it becomes increasingly harder to determine which route of public transit to take. Picking a route at random not only will cause a person to lose a significant amount of time, but it may also lead a person to be lost.

With the coming of the Digital Revolution in the 1950s, more and more real world problems are solved through the use of modeling and computational algorithms. With this in mind, a combined use of modeling and algorithms should also be able to make it easier for a person to get around a major city using public transportation.

This paper shall discuss the possibility of combining the use of weighted directed graphs and Dijkstra's algorithm to find the most efficient way around a large urban city using public transit.

II. THEORETICAL APPROACH

A. *Graphs*

a. *Definition*

A graph is a mathematical model where objects known as vertices are connected with one another. The connections between two vertices are called edges.

b. *Classification of Graphs*

i. *Simple and Un-simple Graphs*

Simple graphs are graphs which don't contain loops (a vertex can't be connected to itself) and the number of edges connecting two vertices is limited to one.

Un-simple graphs are graphs which contain at least one loop or it has multiple edges connecting two distinct vertices.

ii. *Directed and Undirected Graphs*

Directed graphs are graphs in which an edge not only represents a connection between two vertices, but it also represent the direction of the connection. If vertex A is connected to vertex B, vertex B may not be necessarily connected to vertex A.

Edges in undirected graphs on the other hand, do not represent the direction of a connection. If two vertices are connected, they are connected both ways.

iii. *Weighted and Un-weighted Graphs*

Weighted graphs are graphs in which the edge also represents the "weight" of a connection between two vertices. The weight between two vertices can represent a number of things such as, but not limited to the difficulty of connecting two objects or the cost of traveling from point A to point B.

iv. *Planar and Non-planar Graphs*

Planar graphs are graphs which are when drawn on a Euclidian Plane, it does not contain any crossing between edges. If a graph contain at least one crossing between two edges it is considered a non-planar graph.

c. *Terms*

i. *Vertex*

A vertex is the individual object which can be connected to another vertex by an edge. In a drawing of a graph, it is represented as a node.

ii. *Edge*

An edge is the connection between two vertices. In a drawing of a graph, it is represented as a line between two nodes.

iii. *Isolated Vertex*

An isolated vertex is a vertex which is not connected to any other vertex.

iv. *Loop*

A loop is an edge that connects a vertex with itself.

v. *Adjacency*

Two vertices are considered adjacent when there is an edge connectiong them.

vi. *Incidence*

A vertex and an edge are considered incident when the edge connects the vertex with another vertex (or itself).

vii. *Degree*

The degree of a vertex is the number of edges incident to it. Loops are counted twice.

viii. *Order*

The order of a graph is how many vertices it has.

ix. *Walk*

A walk is a series of "steps" taken from one vertex to another. It is an alternating sequence between a vertices and edges, beginning and ending with a vertex. A *closed walk* or *cycle* is a walk where the start and the end of the walk is the same vertex. On the other hand, an *open walk* or *path* is a walk where the start and the end of the walk are different vertices.

x. *Sub-graph*

A sub-graph is a graph within another larger graph. Graph G1 is a sub-graph of graph G if the vertices of G1 are a subset of the vertices of G, and the edges of G1 are also a subset of the edges of G.

xi. *Spanning Subgraph*

A spanning sub-graph G1 is a sub-graph of G if and only if it contains all of the vertices of G.

d. *Special Types of Graphs*

i. *Null Graph*

A null graph is a graph which does not contain any edges.

ii. *Connected graph*

A connected graph is a graph that does not contain an isolated vertex.

iii. *Cycle graph*

A Cycle graph is a graph where all of the vertices are connected by a single cycle.

iv. *Wheel Graph*

A wheel graph is an order four or higher graph formed by connecting a single vertex to each vertex of a cycle graph.

v. *Tree*

A Tree is a connected graph which does not contain any cycle within it.

vi. *Complete Graph*

A complete graph is a graph is which all of the vertices are connected to all of the other vertices.

vii. *Regular Graph*

A regular graph is a graph where all of its vertices are connected to the same number of vertices. In other words, they all have the same degree.

B. *Dijkstra's algorithm*

a. *Brief History*

Dijkstra's algorithm is an algorithm used to find a path within a weighted graph from one vertex to another with the least amount of weight. It was conceived by a Dutch computer scientist by the name of

Edsger Wybe Dijkstra in 1956.

Troughout his life, Dijkstra received numerous awards and honors for his work in computer science. Dijkstra was known for writing his manuscripts by hand, and then distributing a photocopy of it trough the computer science community. These manuscripts are known trough the community as EWDs (his initials). He was a brilliant scientist, but sadly he died in 2002.

b. *Algorithm*

Let the node at which we are starting be called the initial node. Let the distance of node Y be the distance from the initial node to node Y. Dijkstra's algorithm will then run as follows:

1. Assign a tentative distance value to every node. Set it zero for the initial node, and infinity for every other node.
2. Mark all nodes as unvisited. Set the initial node as current. Create a set of the unvisited nodes and call it the unvisited set.
3. For the current node, consider all of its unvisited neighbors and calculate their tentative distances. For example, if the current node X is marked with a distance of 4 (from the initial node), and the edge connecting it with neighbor Y has length 3, then the distance to B will be 4+3=7. If this distance is less than the previously recorded distance, then overwrite that distance. Even though a neighbor has been examined, it is not marked as visited at this time, and it remains the unvisited set.
4. Once every neighbor of the current node has been considered, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again; its distance recorded now is final and minimal.
5. If the destination node has been marked as visited or if the smallest tentative distance among the nodes in the unvisited set is infinity, then the algorithm has finished.
6. Set the unvisited node marked with the smallest tentative distance as the next current node and go back to step 3.

[2]

It is worth noting that Dijkstra's algorithm only work for weighted graphs in which the weight is a non-negative.

c. *Implementation in C*

In 2010, an Indian programmer by the name of Muffadal Makati implemented Dijkstra's algorithm in C programming language. To model the graph, he created a file format in which each line of code stands for a single vertex. The format for each line is as follows:

```
<node id>:<id of node it is
connected to>-<corresponding
cost>:…;
```

Figure 1.2.3 Line format of a vertex

Makati gave this file format a .ospf extension. A C implementation of Dijktra's algorithm and an example of a .ospf can be viewed at Appendix A and Fig. 3.2.1 resprectively.

C. *Public transit*

Public transit is a mode of transitation available to the masses. For the most part, public transit are provided by the government, but it is not rare for a private company to operate and/or own a fleet of public transit. Generally the customers pay a certain amount of money, and the provider transit the customer from point A to point B.

The vehicles used for public transitation varies greatly in different countries. In the city of Venice for example, the use of boats is extremely common, but in the city of Jakarta it would be almost impossible to spot a boat.

a. *Personal Public Transit*

Personal public transit is a type of public transit in which the passenger is transited to and from a place of their choosing. Also, they would not have to share their ride with other passenger.

Great example of a personal public transit would be a taxicab. People pay the driver a fare, and the driver takes the passenger wherever they want to go. In Southeast Asian countries such as Indonesia or the Philippines, motorcycle taxis (also known as *ojeg* in Indonesia) are extremely popular.

Generally, personal public transit is considerably more expensive than mass public transit because they can only serve one customer at a time.

b. *Mass Public Transit*

Unlike personal public transit, in mass public transit, passengers must follow a predetermined route and they would have to share the ride with other passengers.

A good example of mass public transit is the bus service. To get from point A to point B, passengers would typically have to take multiple routes. In the city of Bandung, Indonesia, the use of cars is favored instead of busses. These cars are called *angkot*, short for *angkutan kota* (city transit). Alternatively, beginning in January 25th 2004 in the city of Jakarta, Indonesia, the use of a bus rapid transit named TransJakarta is getting more and more popular.

Generally, mass public transit is considerably cheaper than personal public transit.

As the city grows more and more crowded, the need of finding the right combination of mass public transit routes is getting more and more crucial.

III. ANALYSIS

A. *Modeling mass public transit routes through the use of weighted directed graphs*

In some cities, mass public transit only stop on certain locations, while in other cities people can get on or off the transit as they please. To simplify the matter, in this paper the subject matter will only be transits which stop at certain locations.

As apparent on the map in Appendix III, different locations are connected to one another by public transit routes. This phenomenon is similar to a graph. In a graph different vertices are connected to one another by edges. Therefore it would be not only possible, but also practical to utilize a graph to model the routes of mass public transport. Each vertex represents a location, and each edge represents a route of transit between them.

Another thing to consider is that the distance, time, and price needed to travel between two locations is not always the same. Therefore, in this model, the use of weighed graph is necessary.

The weight of the graph can represent either the distance, time, or price needed to travel between two locations. In determining which model to use, one would first need to have one's priority in order, whether it is cost, or time. If it is cost, then the weight of the graph represents the price to go from vertex A to vertex B, but if its time, then the weight of the graph can represents either the time needed to go from vertex A to vertex B or the distance between the two vertices.

If a person is able to get from point A to point B by riding on a certain route, it is not necessarily true that they can get back using the same route. Therefore, the use of directed graphs is also necessary.

As an example, we shall use the map below:

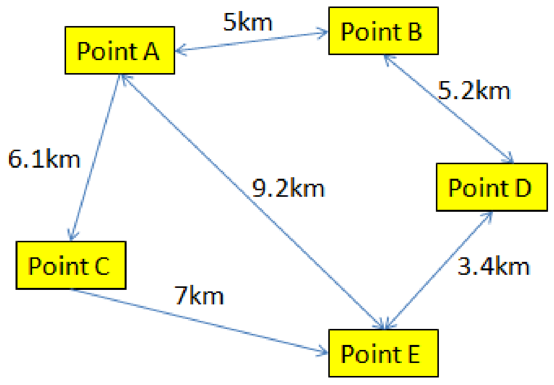

Figure 3.1.1 A map

After modeling it through the use of weighted directional graph, we are able to obtain the graph below (names of places and distances are converted into integer so that it may be easily computed):

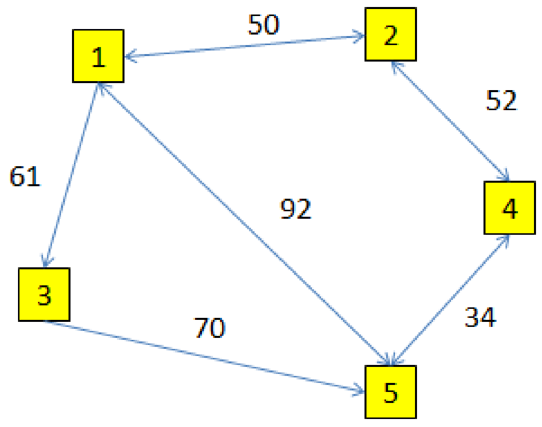

Figure 3.1.2 A weighted directional graph

B. *Obtaining the most effective combination through the use of Dijkstra's algorithm*

After a model is produced, all it takes is some simple computational calculations to find the most effective path.

By following the format of .ospf files, we are able to compute the most effective routes to take from point A to point B. A .ospf file representation of our map is shown below:

```
1:2-20:3-61:5-92;
2:1-50:4-52;
3:5-70;
4:2-52:3-34;
5:1-92:4-34;
```

Figure 3.2.1 .ospf representation of Fig. 3.1.2

As an example, we shall find the most effective route to take from Point A to Point D. In the model, it is represented by vertices 1 and 4 respectively. After running it trough Muffadal Makati's program, we are able to find out that the most effective route to take is first to go from Point A to Point D, then to go from Point D to Point E. The route is 10.4km far.

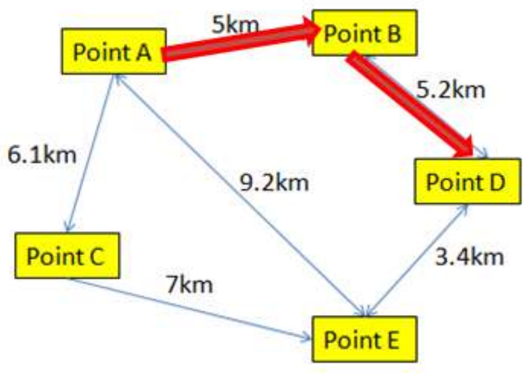

Figure 3.2.2 The most effective route from Point A to Point D

V. Conclusion

It is possible to obtain the most effective mass public transit route through the use of graph theory and Dijkstra's algorithm as shown in this paper. Nevertheless there are still numerous factors that need to be considered such as walking distance, weather, and comfort.

VI. Appendix

A. *Implementation of Dijkstra's algorithm in C*

```
/*
 * Author: Mufaddal Makati
   Official Post: http://www.rawbytes.com
    Copyright [2010] [Author: Mufaddal Makati]

   Licensed under the Apache License, Version 2.0 (the "License");
   you may not use this file except in compliance with the License.
   You may obtain a copy of the License at

       http://www.apache.org/licenses/LICENSE-2.0

   Unless required by applicable law or agreed to in writing, software
   distributed under the License is distributed on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   See the License for the specific language governing permissions and
   limitations under the License.
 *
 * Created on September, 2010.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_CONN 10
#define MAX_NODES 50
#define INFINITE 9999

void view(); /* just prints out the graph*/
void djkstra(); /*runs at the begining and when file is rescaned*/
void min_route();
void scanfile(); /*it does not check for wrong format of the file.it is up to the user to verify
it.*/
struct node /*data structure to store information of each node*/
{
    int id;
    struct node *l[MAX_CONN+1]; /*array that points to all other nodes connected with this
node*/
    int cost[MAX_CONN+1];  /*array that stores cost of all connections with this node*/
    struct node **next; /*address of the pointer of the next node to traverse, for the ith
```

```
destination node in the array of records datastructure*/
    int *mincost;/*minimum cost to traverse the ith destination node in the array of records
datastructure*/
};
typedef struct node nodes;
struct record /* data structure to map node id and corresponding address of nodes structure.*/
{
    int nid;
    struct node *add;
};
typedef struct record records;
records nlist[MAX_NODES+1]={0}; /*create array of records and mark the end of the array with
zero.*/
int count; /*stores number of nodes*/

int main()
{
    int c;
    nlist[MAX_NODES+1].nid=0;
    system("clear");

    scanfile();
    printf("\nSucessfully Scanned the graph.\n");
    djkstra();

    for(;;)
    {
        view();

        printf("\n1.Rescan file.");
        printf("\n2.View minimum route between nodes.");
        printf("\n3.Exit.");
        printf("\nEnter:");
        scanf("%d",&c);
        switch(c)
        {
            case 1:
            {
                scanfile();
                djkstra();
                system("clear");
                break;
            }
            case 2:
            {
                min_route();
                system("clear");
                break;
            }
            case 3:
            {
                exit(0);
            }
            default:
            {
                printf("\nEnter proper choice.\n");
                break;
            }
        }
    }
    return (EXIT_SUCCESS);
}

void scanfile()
{
    FILE *f;
    int d;
    int i=0,j=0,n_id,n_cost;
    nodes *temp=0,*temp1=0;

    if((f=fopen("graph.ospf","r"))== NULL)
    {
        printf("Error opening file.\n");
        exit(1);
    }
    memset(nlist, 0, sizeof(struct record) * MAX_NODES);
    count=0;
    do /*first get the id and address of all nodes*/
    {
        fscanf(f,"%d",&n_id);
        for(i=0;nlist[i].nid!=0;i++)
        {
            if(n_id==nlist[i].nid)
            {
                printf("Id already exists.");
                return;
            }
        }
        temp=(nodes *)malloc(sizeof(nodes));
        if (temp == 0)
        {
           printf("ERROR: Out of memory\n");
           return;
        }
        memset(temp, 0, sizeof(struct node));
        temp->id=n_id;
        temp->l[MAX_CONN+1]=0;
        temp->cost[MAX_CONN+1]=0;
        for(i=0;nlist[i].nid!=0;i++)
        {}
        nlist[i].nid=n_id;
        nlist[i].add=temp;
        count++;
        while((d=fgetc(f)!=';'))
        {}
    }while((d=fgetc(f))!=EOF);

    rewind(f);

    for(i=0;i<count;i++) /*now get the information of all nodes connections.*/
    {
        fscanf(f,"%*d");
        temp=nlist[i].add;
        while((d=fgetc(f)!=';'))
        {
            fscanf(f,"%d-%d",&n_id,&n_cost);
            for(j=0;nlist[j].nid!=0;j++)
            {
                if(nlist[j].nid==n_id)
                {
                    temp1=nlist[j].add;
                    break;
                }
            }
            for(j=0;temp->cost[j]!=0;j++)
            {}
            temp->cost[j]=n_cost;
            temp->l[j]=temp1;
        }
    }
    fclose(f);
}

void view()
{
    int i,j;
    nodes *temp=0,*temp1=0;

    printf("\nID\tConnceted to- ID:cost");
    for(i=0;nlist[i].nid!=0;i++)
    {
        printf("\n%d",nlist[i].nid);
        temp=nlist[i].add;
        for(j=0;temp->l[j]!=0;j++)
        {
            temp1=temp->l[j];
            printf("\t%d:%d",temp1->id,temp->cost[j]);
        }
    }
    printf("\n \n \n");
}

void djkstra()
{
    int i,j,k,num,num1=0,min=INFINITE;
    int *tcost=0,*done=0;
    nodes *temp=0,*temp1=0,**tent=0;

    tcost=(int*)calloc(count, sizeof(int));
    if (tcost == 0)
    {
       printf("ERROR: Out of memory\n");
       return;
    }
    done=(int*)calloc(count, sizeof(int));
    if (done == 0)
    {
       printf("ERROR: Out of memory\n");
       return;
    }
    tent=(nodes**)calloc(count, sizeof(nodes));
    if (tent == 0)
    {
       printf("ERROR: Out of memory\n");
       return;
    }

    for(i=0;nlist[i].nid!=0;i++)
    {
         for(j=0;j<count;j++)
         {
            tcost[j]=INFINITE;
            done[j]=0;
         }
        temp=nlist[i].add;
        temp->next=(nodes**)calloc(count, sizeof(nodes));
        temp->mincost=(int*)calloc(count, sizeof(int));
        tcost[i]=0;
        done[i]=1;
        temp->mincost[i]=0;
        temp1=temp;
        for(;;)
        {
            for(num1=0;nlist[num1].nid!=0;num1++)
            {
                if(nlist[num1].add==temp1)
                    break;
            }
            for(k=0;temp1->l[k]!=0;k++)
            {
                for(num=0;nlist[num].nid!=0;num++)
                {
                    if(nlist[num].add==temp1->l[k])
                        break;
                }

                if(tcost[num] > (tcost[num1]+temp1->cost[k]))
                {
                    tcost[num]= tcost[num1] + temp1->cost[k];
                    if(temp1==temp)
                        tent[num]=temp1->l[k];
                    else
                        tent[num]=tent[num1];
                }
            }
            min=INFINITE;num1=0;
            for(j=0;j<count;j++)
            {
                if(tcost[j]<min && done[j]!=1 && tcost[j]!=0)
                {
                    min=tcost[j];
                    num1=j;
                }
            }
            if(min==INFINITE)
                break;

            temp1=nlist[num1].add;
            temp->mincost[num1]=tcost[num1];
            temp->next[num1]=tent[num1];
            done[num1]=1;
        }
    }
}
```

```
void min_route()
{
    int i,sid,did,num,chk=0;
    nodes *temp=0,*temp1=0;

        printf("\nEnter source node ID:");
        scanf("%d",&sid);
        printf("\nEnter destination node ID:");
        scanf("%d",&did);

        for(i=0;nlist[i].nid!=0;i++)
        {
            temp=nlist[i].add;
            if(temp->id==sid)
            {
                chk=1;
                break;
            }
        }
        if(chk==0)
        {
            printf("\nSource Id not found.\n");
            return;
        }
        chk=0;
        for(num=0;nlist[num].nid!=0;num++)
        {
            temp1=nlist[num].add;
            if(temp1->id==did)
            {
                chk=1;
                break;
            }
        }
        if(chk==0)
        {
            printf("\nDestination Id not found.\n");
            return;
        }
        printf("%d-",temp->id);
        temp1=temp;
        for(;;)
        {
            if(temp1->id==did)
                break;
            if(temp1->next[num]!=0)
            {
                temp1=temp1->next[num];
                printf("-%d-",temp1->id);
            }
            else
            {
                printf("No Route");
                break;
            }
        }
        printf("\nTotal cost:%d",temp->mincost[num]);
}
```

[4]

REFERENCES

[1] M. Rinaldi, “Diktat Kuliah IF 2091 Struktur Diskrit”, Program Studi Teknik Informatika, 2008, Bandung, Indonesia.

[2] Nugraha, Tubagus A. "Pathfinding through Urban Traffic Using Dijkstra’s Algorithm." 2011. TS. Institut Teknologi Bandung, Bandung, Indonesia.

[3] Marrana, Joao. "Transport and Urban Life: Developments and Trends." International Association of Public Transport. UITP, n.d. Web. 14 Dec. 2013.

[4] Makati, Mufaddal. "Dijkstra’s Algorithm in C." Web log post. Rawbytes. N.p., 20 Dec. 2012. Web. 15 Dec. 2013.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 15 November 2013

[illegible]

Tirta Wening Rachman - 13512004