Floyd-Warshall Algorithm Application on Optimizing Bandung City Travelling Path Based on Traffic Density

Muhammad Harits Shalahuddin Adil Haqqi Elfahmi/13511046
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13511046@std.stei.itb.ac.id

***Abstract*—Floyd-Warshall is a Dynamic Programming algorithm to find All Pair Shortest Path. This paper focuses in trying to convert Bandung map from Google Maps into undirected graph, with live traffic detection turned on, and approximate the weight based on the traffic density. After that it tries to find all possible path with shortest route, thus helping tourist/locals finding shortest paths between main intersections in the city.**

***Index Terms*—floyd-warshall, bandung, shortest path, traffic, density.**

1. INTRODUCTION

1.1 PATH FINDING ALGORITHM

Path finding is one of the most interesting problems in programming, as such, there have been numerous solution proposed to solve many kinds of its variants. One of its variants, shortest path, is the one discussed in this paper.

One of the shortest path algorithm with Dynamic Programming (DP) approach is the Floyd-Warshall algorithm, which calculates the shortest distance from every node to every other node in a graph (All-Pair Shortest Path).

1.3 GOOGLE MAPS

Google Maps is one of Google products that offers location finder tool, street maps and a route planner for traveling by foot, car, bike (beta), or with public transportation. It also includes a locator for urban businesses in numerous countries around the world. One of the interesting feature that Google Maps offer is the almost live traffic density detection feature, which Google Maps detects based on smartphone user location.[1]

1.2 BANDUNG CITY AS A TRAVEL DESTINATION

Bandung is the capital city of West Java, one of the famous city for travelling tourist because of its fresh air, unique landmarks, and kind locals.

The Dutch colonials first established tea plantations around the mountains in the eighteenth century, and a road was constructed to connect the plantation area to the capital (180 kilometres (112 miles) to the northwest). The Dutch inhabitants of the city demanded establishment of a municipality (gemeente), which was granted in 1906, and Bandung gradually developed itself into a resort city for plantation owners. Luxurious hotels, restaurants, cafes and European boutiques were opened, hence the city was nicknamed Parijs van Java (Dutch: "The Paris of Java").

Since Indonesia achieved independence in 1945, the city has experienced rapid development and urbanisation, transforming Bandung from idyllic town into a dense 16,500 people/km2 metropolitan area, a living space for over 2 million people. Natural resources have been exploited excessively, particularly by conversion of protected upland area into highland villas and real estate. Although the city has encountered many problems (ranging from waste disposal, floods to chaotic traffic system, etc.), Bandung still attracts immigrants and weekend travelers.

But just like other capital cities, Bandung has a common problem: traffic jam.

Because of the reason stated above, this paper will focus on optimizing travelling path between main street intersection in Bandung based on Google Maps live traffic detection feature.

2. THEORIES

2.1 FLOYD-WARSHALL ALGORITHM

In computer science, the Floyd–Warshall algorithm (also known as Floyd's algorithm, Roy–Warshall algorithm, Roy–Floyd algorithm, or the WFI algorithm) is a graph analysis algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles).

The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with $\Theta(|V|^3)$ comparisons in a graph. This is remarkable considering that there may be up to

$\Omega(|V|^2)$ edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal.

The basic idea behind Floyd Warshall's is to gradually allow the usage of intermediate vertices to form the shortest paths. Let the vertices be labeled from 0 to 'V - 1'. We start with direct edges only, i.e. shortest path of vertex i to vertex j, denoted as sp(i,j) = weight of edge (i,j). We then find shortest paths between any two vertices with help of restricted intermediate vertices from vertex [0 ... k]. First, we only allow k = 0, then k = 1, ..., up to k = V-1.

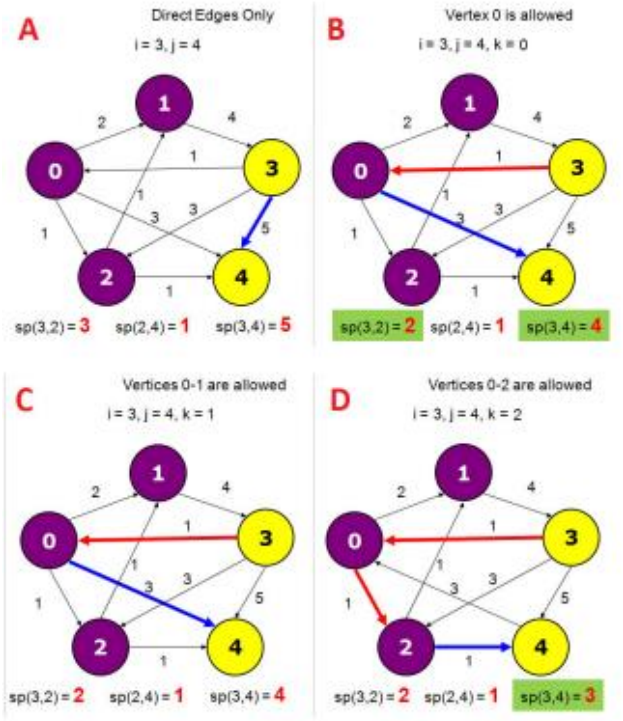


Figure 2.1.1: Floyd Warshall's Explanation

In Figure 2.1.1, we want to find sp(3,4). The shortest possible path is 3-0-2-4 with cost 3. But how to reach this solution? We know that with direct edges only, sp(3,4) = 5, as in Figure 4.18.A. The solution for sp(3,4) will eventually be reached from sp(3,2)+sp(2,4). But with only direct edges, sp(3,2)+sp(2,4) = 3+1 is still > 3.

When we allow k = 0, i.e. vertex 0 can now be used as an intermediate vertex, then sp(3,4) is reduced as sp(3,4) = sp(3,0)+sp(0,4) = 1+3 = 4, as in Figure 4.18.B. Note that with k = 0, sp(3,2) – which we will need later – also drop from 3 to sp(3,0)+sp(0,2) = 1+1 = 2. Floyd Warshall's will process sp(i,j) for all pairs considering only vertex 0 as the intermediate vertex.

When we allow k = 1, i.e. vertex 0 and 1 can now be used as the intermediate vertices, then it happens that there is no change to sp(3,4), sp(3,2), nor to sp(2,4).

When we allow k = 2, i.e. vertices 0, 1, and 2 now can be used as the intermediate vertices, then sp(3,4) is reduced again as sp(3,4) = sp(3,2)+sp(2,4) = 2+1 = 3. Floyd Warshall's repeats this process for k = 3 and finally k = 4 but sp(3,4) remains at 3.

We define $Dk_{i,j}$ to be the shortest distance from i to j with only [0..k] as intermediate vertices.

Then, Floyd Warshall's recurrence is as follows:

$D^{-1}{}_{i,j}$ = weight(i, j). This is the base case when we do not use any intermediate vertices.

$D^{k}{}_{i,j}$ = min($D^{k-1}{}_{i,j}$, $D^{k-1}{}_{i,k}$ + $D^{k-1}{}_{k,j}$) = min(not using vertex k, using k), for $k \geq 0$, see Figure 2.1.2.

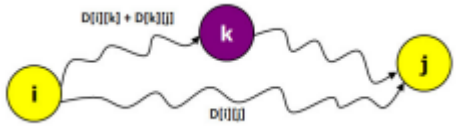


Figure 2.1.2: Using Intermediate Vertex to (Possibly) Shorten Path

This DP formulation requires us to fill the entries layer by layer. To fill out an entry in the table k, we make use of entries in the table k-1. For example, to calculate $D^{2}{}_{3,4}$, (row 3, column 4, in table k = 2, index start from 0), we look at the minimum of $D^{1}{}_{3,4}$ or the sum of $D^{1}{}_{3,2}$ + $D^{1}{}_{2,4}$. See Figure 2.1.3 for illustration.

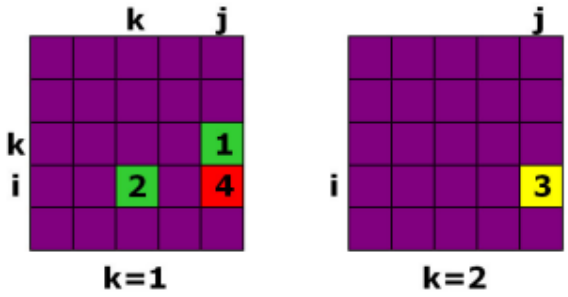


Figure 2.1.3: Floyd Warshall's DP Table

The naive implementation is to use 3-dimensional matrix D[k][i][j] of size $O(V^3)$. However, we can utilize a space-saving trick by dropping dimension k and computing D[i][j] 'on-the-fly'.Thus, the Floyd Warshall's algorithm just need $O(V^2)$ space although it still runs in $O(V^3)$.[2]

The pseudecode implementation of the Floyd-Warshall algorithm is as shown below.

```
let dist be a |V| × |V| array of
minimum distances initialized to ∞
(infinity)
 for each vertex v
    dist[v][v] ← 0
 for each edge (u,v)
    dist[u][v] ← w(u,v)  // the weight
of the edge (u,v)
 for k from 1 to |V|
    for i from 1 to |V|
       for j from 1 to |V|
          if dist[i][j] > dist[i][k] +
dist[k][j]
             dist[i][j] ← dist[i][k] +
dist[k][j]
         end if
```

After executing the code above, we can get the shortest distance from node A to node B in the variable `dist[A][B]`.

But because we would like to know the shortest path between intersections (node) we need to store the shortest past from A to B in the DP matrix, because of that we will use such node struct (in C++ code):

```
struct typedef{
      vector<int> path;
      int dist;
} Node;
```

With such data structure, we can make the Floyd-Warshall algorithm that saves the intersection visited in the shortest path between all intersection.

The algorithm is as follows:

```
let dist be a |V| × |V| array of minimum
distances initialized to ∞ (infinity)
 for each vertex v
    dist[v][v] ← 0
    path[v][v] ← {}
 for each edge (u,v)
    dist[u][v] ← w(u,v)  // the weight
of the edge (u,v)
 for k from 1 to |V|
    for i from 1 to |V|
       for j from 1 to |V|
          if dist[i][j] > dist[i][k] +
dist[k][j]
             dist[i][j] ← dist[i][k] +
dist[k][j]
             path[i][j] ←
append(path[i][k], path[k][j])
         end if
```

And so with the code above we can get the path of the shortest distance from node A to node B in the variable *path[A][B]*.

2.2 GOOGLE MAPS

Figure 2.2.1: Google Map's Live Traffic Option

In the Google Maps interface, after enabling the traffic option on the top left corner of the interface, on the bottom left corner user can see the live traffic guide, where the road will be marked as the corresponding color: the slowest is in red-striped black pattern, followed by red, yellow, and the fastest lane is marked green.

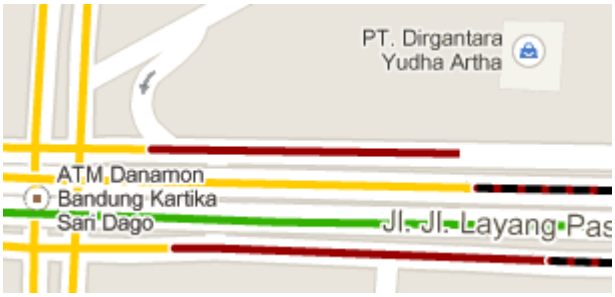
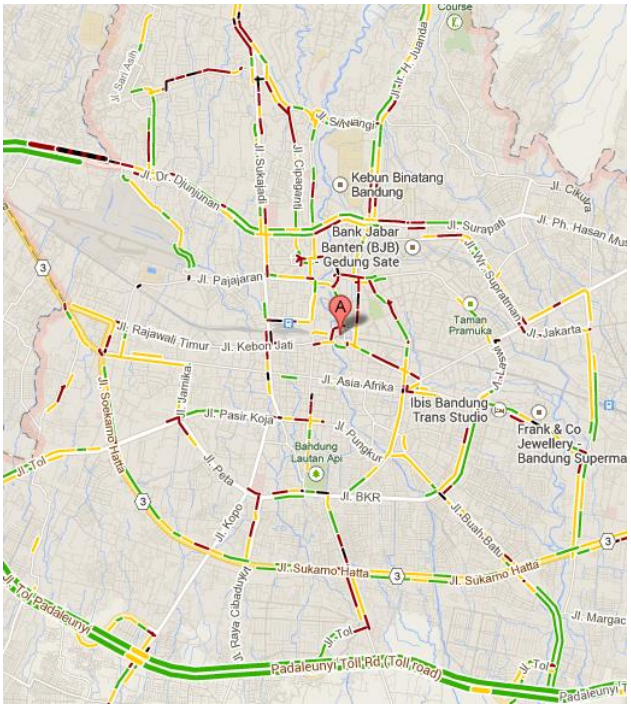


Figure 2.2.2: Google Map's Live Traffic Detection Example

The following is Bandung city map preview with traffic detection enabled on Google Map:



3. IMPLEMENTATION

3.1 BANDUNG CITY GRAPH SIMPLIFICATION

In this paper we will use the traffic map from Monday, 12.00pm (Data is estimated based on past conditions)[3]. The actual map is shown below.

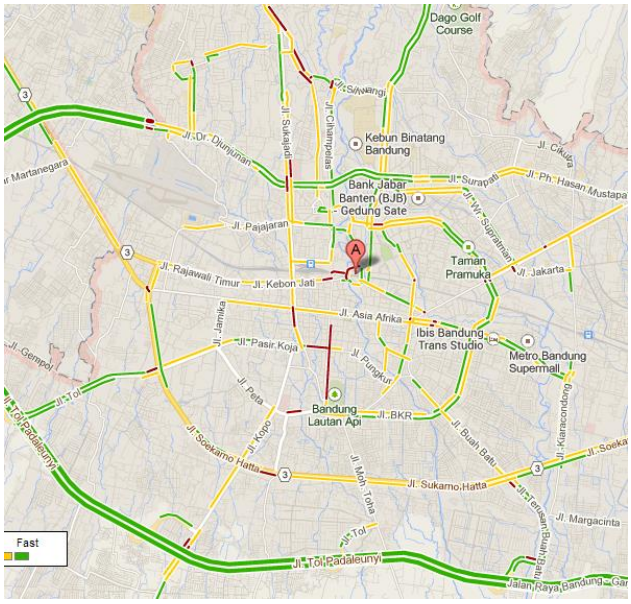


Figure 3.1.1: Bandung City Traffic on Monday 12:00pm

In the map we can see that some of the road is colorless, that is because Google Maps doesn't have enough data from smartphones that currently resides in that area, either there isn't any or they just don't give the permission for Google to locate their place. And so, for easier implementation purpose, we will assume that the colorless road is a green road.

With that assumption, the new map looks like the following:

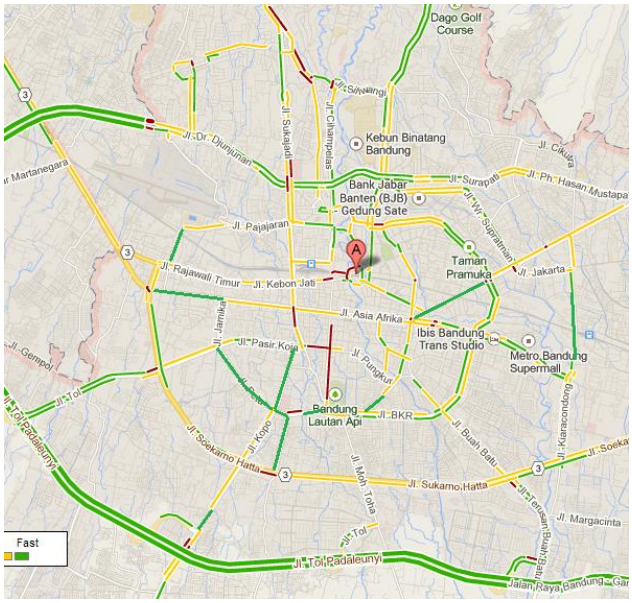


Figure 3.1.2: Bandung City Traffic Simplified (1)

For the next step we will (1) divide the road by a small, uniform segment which we will base the approximation of the edge weight on, and (2) pick several main intersection as the node for the graph and assign number to it.

With those two modifications applied, the new map looks like following:

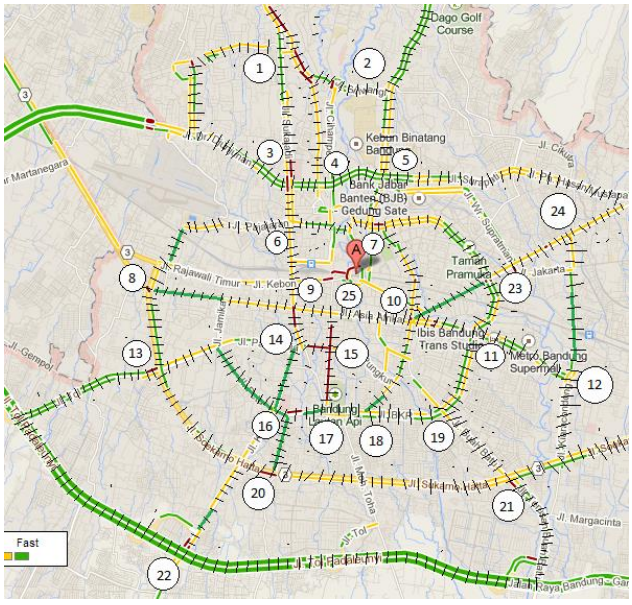


Figure 3.1.3: Bandung City Traffic Simplified (2)

The segmentation on the graph will have values as follows:

No	Color	Values
1	Red-Black	4
2	Red	3
3	Yellow	2
4	Green	1

Table 3.1.1: Segmentation Weighting

By the information provided by the table above and applying it on the Figure 3.1.3, the redrawn graph looks like this:

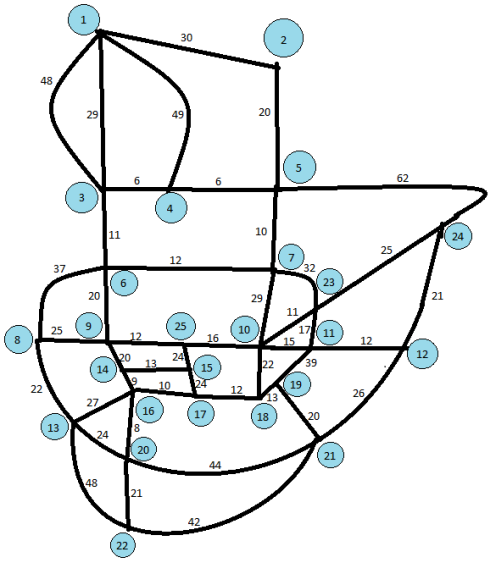


Figure 3.1.4: Bandung City Traffic Simplified (3)

3.2 Applicating Floyd-Warshall On The Simplified Graph

The core C++ implementation from the Floyd-Warshall with path recognition algorithm is as follows:

```
for(int k=0;k<maxNode;k++){
        for(int i=0;i<maxNode;i++){
            for(int
j=0;j<maxNode;j++){
                if(adjMat[i][j].dist
> adjMat[i][k].dist +
adjMat[k][j].dist){

adjMat[i][j].dist =
adjMat[i][k].dist +
adjMat[k][j].dist;
                    vector<int>
dummy(adjMat[i][k].path.begin(),
adjMat[i][k].path.end()-1);

adjMat[i][j].path = append(dummy,
adjMat[k][j].path);
                }}}}
```

From the figure 3.1.4, we can extract the edges for the Floyd-Warshall input below:

```
41 //number of edges
1 2 30 //from node 1 to 2 weighted 30
1 3 29
1 4 49
3 4 6
4 5 6
2 5 20
5 24 62
3 6 11
6 7 12
5 7 10
7 23 32
23 24 25
6 8 37
8 9 25
9 6 20
9 25 12
25 10 16
10 7 29
10 23 11
10 11 15
11 23 17
11 12 12
24 12 21
8 13 22
13 16 27
14 16 9
14 9 20
14 15 13
16 17 10
15 17 24
25 12 24
17 18 12
18 19 13
19 11 39
21 12 26
13 20 24
20 16 8
20 21 44
13 22 48
22 20 21
22 21 42
```

After executing the Floyd-Warshall algorithm, the program will asks for start and finish node, which we will use for the next section of this paper.

3.3 Results

Some testcase output result example from the program execution is as follows:

Input:
1 24
Output:
Time required for Floyd-Warshall algorithm execution:
0.002s
adjMat[1][24]: 103
Intersection(s) traveled: 1-3-4-5-24.

Input:
1 15
Output:
Time required for Floyd-Warshall algorithm execution:
0.002s
adjMat[1][15]: 93
Intersection(s) traveled: 1-3-6-9-14-15.

Input:
6 12
Output:
Time required for Floyd-Warshall algorithm execution:
0.002s
adjMat[6][12]: 56
Intersection(s) traveled: 6-9-25-12.

4. ALGORITHM ANALYSIS

Floyd-Warshall is an algorithm with $O(V^3)$ complexity. Although it is very simple to implement, Floyd-Warshall is an ineffective algorithm for problems with $V > 100$.

After the simplification of the Bandung City map, we acquired a graph with 41 edges and 25 vertices. Because of its low vertices count, this problem is suitable to be solved with Floyd-Warshall algorithm.

By the result of the execution of the program, it is indeed true that this problem was solved very fast (0.002s).

5. CONCLUSION

Bandung city map can be converted into a weighted graph with road as the edge and the intersection as the vertice. From that we can calculate the shortest path between two intersection.

For that purpose Floyd-Warshall is a suitable algorithm for finding the fastest travelling path between two intersections in Bandung city based on traffic density. It is very fast based on execution time.

REFERENCES

[1] http://www.theconnectivist.com/2013/07/how-google-tracks-traffic/, 19 Desember 2013, 12:43

[2] Steven Halim, Felix Halim. Competitive Programming 3: The New Lower Bound of Programming Contests. 2010.

[3] http://maps.google.com, 20 Desember 2013, 13:59

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, 20 Desember 2013

Muhammad Harits Shalahuddin Adil Haqqi Elfahmi
13511046