CHAPTER 5
IMPLEMENTATION AND TESTING

5.1 Implementation

In this project, the data from the CSV file will be processed using **Networkx** library. Based from **Networkx** documentation, the data structure used is based on the adjacency list representation and implemented in Python by using the dictionary data structure.

```
1. def main():
2.     graph = nx.Graph
3.     edges = []
4.     nodes = []
5. with  open('Data_Sample2/49_USCT.csv',  newline='')  as  csvfile:
            spamreader = csv.reader(csvfile, delimiter=',')
6.          for row in spamreader:
7.              edges.append({
8.                  "node_satu": row[0],
9.                  "node_dua" : row[1],
10.                 "jarak"    : int(row[2])
11.               })
12.               if row[0] not in nodes:
13.                   nodes.append(row[0])
14.               if row[1] not in nodes:
15.                   nodes.append(row[1])
16.               graph.add_edge(row[0], row[1], len=int(row[2]))
```

From a piece of code above is used to perform the process to generate an initial graph that is still not processed by the algorithm, line 5 is used to open CSV file. Line 6-11 is used to list its edges first from “node_satu” and “node_dua” and also their length. While lines 12-15 are used to list nodes. Then line 15 is used to create the initial graph using **Networkx** library.

5.1.1 Reverse-Delete Algorithm

```
17.   import networkx as nx
18.   import time

19.   def main(unsorted_edges, rd_graph):
20.     start = time.time()
21.     edges = sorted(unsorted_edges, key = lambda i: i[‘jarak’],
  reverse = True)
22.     for edge in edges:
```

```
23.                          rd_graph.remove_edge(edge['node_satu'],
   edge['node_dua'])
24.          is_connected = nx.is_connected(rd_graph)
25.          if not is_connected:
26.           rd_graph.add_edge(edge['node_satu'], edge['node_dua',
   len=edge['jarak']
```

The process of the Reverse Delete algorithm starts from sorting the edges from the largest length to the smallest length, this can be seen in the code above on line 21. Then after that loop for each edge that has been sorted to try to delete the edge first, this can be seen from the code above on line 22-23. After trying to delete it, then first check whether the edge is removed makes the graph disconnected, if removing the edge cause will the graph disconnected, then add the edge that was deleted to the graph again, this can be seen from the code above on line 24-26.

```
27.    end = time.time()
28.    waktu_rd = end-start
29.    return [rd_graph, waktu_rd]
```

The program of Reverse Delete algorithm will finish working when all edges have been tried to delete.

5.1.2 Boruvka Algorithm

```
1. import networkx as nx
2. import time

3. def main(edges, distinct_nodes, graph):
4.   reserved = []
5.   return_graph = nx.Graph()
```

From a piece of code above, line 4 is used to create an empty variable which will used to save the list of edges that will be inserted into the graph. Meanwhile, line 5 is used to create a blank graph which will be drawn using the edges that are included in the list.

```
6.   start = time.time()
7.   for node in distinct_nodes:
8.     minlen = {
```

```
9.    “node_satu” = None,
10.   “node_dua”  = None,
11.   “jarak”    = 999999
12.   }
13.   thisNodeEdges = []
14.
```

From the code above, from line 7-12 are used to initiate “minlen” at each node, which later functions to compare length between edges. “jarak” 999999 is used as a comparative to find the smallest length for the first time, for example, is the value of 124 smaller than 999999, if so, set the value of 124 to be the new “minlen”.

```
15.   for edge in edges:
16.       if edge[“node_satu” == node or edge [“node_dua”] ==
  node:
17.         thisNodeEdges.append(edge)
```

From the code above, from line 14-16 it is used to retrieve the edges at a particular node. So for each edge, if the edge has the same node as the node in this loop, then add that edge to the “thisNodeEdges” variable that was created on line 13 to compare its distance to the other edges.

```
18.    for new in thisNodeEdges:
19.     if new[“jarak”] < [“minlen”]
20.       minlen = new
21.   reserved.append(minlen)
```

The code above functions to determine the smallest length of each edge on this node, so if the length is smaller than the minlen length that has been determined earlier, then set the edge to be the new minlen. After that insert the minlen on this node into a “reserved” variable.

```
22.  for res in reserved:
23.     return_graph.add_edge(res[“node_satu”],res[“node_dua”],le
  n = res[“jarak”])
```

The code above function to enter each edge that has been taken from line 20 into the blank graph that has been created.

```
24.  sorted_edges = sorted(edges, key = lambda i: i[“jarak”])
25.  while not nx.is_connected(return_graph):
26.     newSortedEdges = []
```

The code above functions to sort the edges from the smallest length to the largest length , this is done to find the edge with the smallest length to connect between graphs that are still not connected.

```
27.     for edge in sorted_edges:
28.          if  not  nx.has_path(return_graph,edge[“node_satu”],
   edge[“node_dua”]:
29.             newSortedEdges.append(edge)
```

The code above functions to retrieve edges with nodes that are still not connected, so for each edge that has been sorted on line 24, if the two nodes are not connected then insert this edge to the variable “newSortedEdges”. This will be used as a candidate for connecting graphs that are still not connected.

```
30.     for edge in newSortedEdges:
31.         if not nx.has_path(return_graph, edge[“node_satu”],
   edge[“node_dua]):
32.                  return_graph.add_edge(edge[“node_satu”],
   edge[“node_dua”], len = edge[“jarak”]
```

In the code above it functions for each edge that has been taken on line 29, to check one by one which edge that can connect the separate graph starting from edges with the smallest length first.

```
33. end = time.time()
34. waktu_boruvka = end-start
35. return [return_graph, waktu_boruvka]
```

Boruvka algorithm process will stop when the graph is no longer separate.

5.2 Testing

The testing carried out in this project uses three data that have been obtained. The first data contains a list of 12 cities in the UK as well as the 27 edges connecting these cities. Then in the second data there is a list of 22 city names in West Germany along with 45 edges that connect between these cities. In the third data, there are 49 list of cities in the United States along with 120 edges that connect these cities. After that the data will be processed using the Reverse Delete algorithm and Boruvka algorithm.

Sample 1

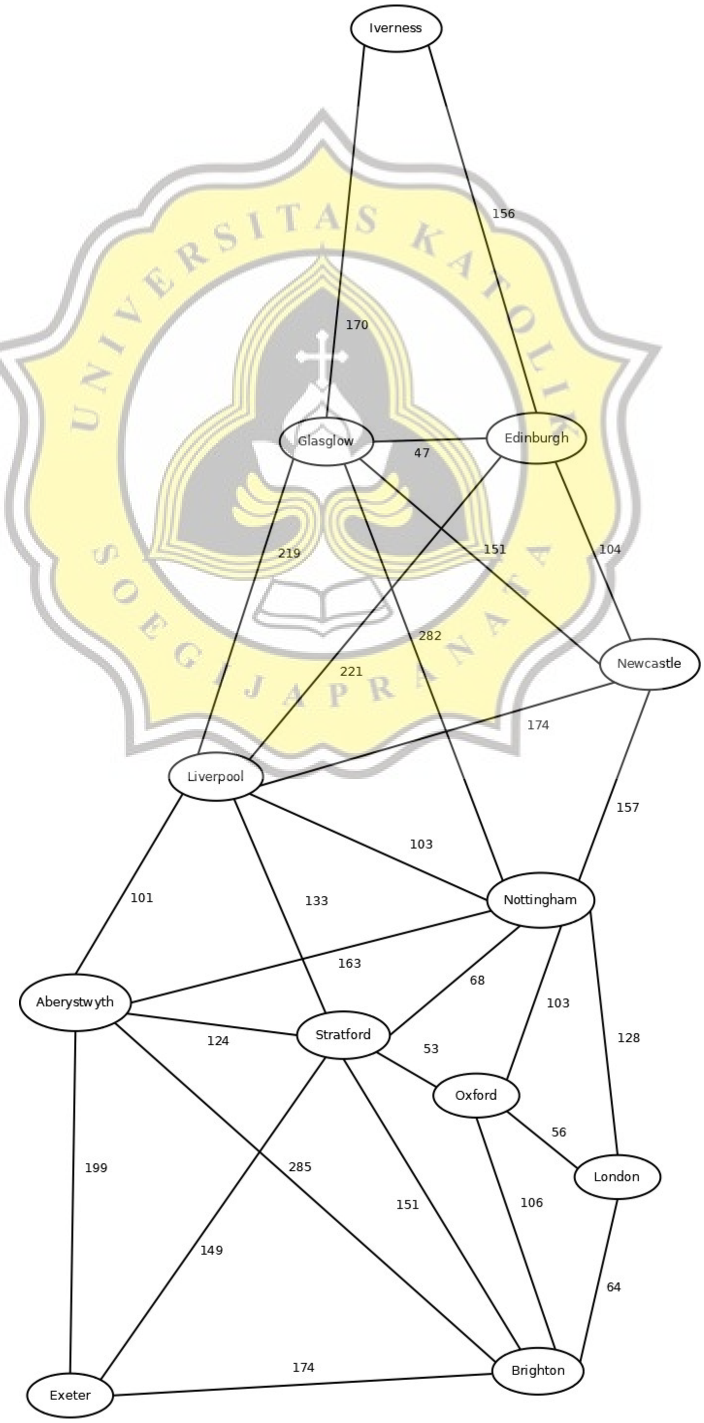

Illustration 5.1: Initial graph image of sample 1

From the picture above is the initial graph image of graph which represents 12 cities of UK along with the edges that connect the cities, which is still not processed by the algorithm, with an initial total length of 3841 miles. Below is the detailed table of the edges and their length.

Table 5.1: Edges and Weight Sample 1

Edge	**Length**	**Edge**	**Length**
Iverness - Edinburgh	156	Nottingham - Stratford	68
Iverness - Glasglow	170	Nottingham - Oxford	103
Edinburgh - Newcastle	104	Nottingham - London	128
Edinburgh - Glasglow	47	Aberystwyth - Stratford	124
Edinburgh - Liverpool	221	Aberystwyth - Brighton	285
Glasglow - Newcastle	151	Aberystwyth - Exeter	199
Glasglow - Nottingham	282	Stratford - Exeter	149
Glasglow - Liverpool	219	Stratford - Brighton	151
Newcastle - Nottingham	157	Stratford - Oxford	53
Newcastle - Liverpool	174	Oxford - Brighton	106
Liverpool - Nottingham	103	Oxford - London	56
Liverpool - Stratford	133	London - Brighton	64
Liverpool - Aberystwyth	101	Brighton - Exeter	174
Nottingham - Aberystwyth	163		

The data above will then be received from the CSV file and processed by the program of each algorithm, both Reverse Delete and Boruvka algorithm. After the data is processed by the program, the results obtained by the Reverse Delete algorithm length 1058 miles, the result are in accordance with the manual calculation of the Reverse Delete algorithm. The result from Reverse Delete program is shown on table bellow.

Table 5.2: Reverse Delete Program Results Table

Edge	Length
Iverness - Edinburgh	156
Edinburgh - Newcastle	104
Edinburgh - Glasglow	47
Newcastle - Nottingham	157
Liverpool - Nottingham	103
Liverpool - Aberystwyth	101
Nottingham - Stratford	68
Stratford - Exeter	149
Stratford - Oxford	53
Oxford - London	56
London - Brighton	64
Program Execution Time	0.0004737377166748

After the data is processed by the program, the results obtained by the Boruvka algorithm have a length of 1058 miles, the same as Reverse Delete algorithm. Manual calculation of the Boruvka algorithm also yields the same results. The results of Boruvka algorithm program is shown on the table bellow.

Table 5.3: Boruvka Program Results Table(time in seconds)

Edge	**Length**
Iverness - Edinburgh	156
Edinburgh – Glasglow	47
Edinburgh - Newcastle	104
Newcastle - Nottingham	157
Liverpool - Aberystwyth	101
Liverpool - Nottingham	103
Nottingham - Stratford	68
Stratford - Oxford	53
Stratford - Exeter	149
Oxford - London	56
London - Brighton	64
Program Execution Time	0.0003452301025390625

From the two table above, it can be seen that the Boruvka algorithm find MST first. After getting minimum spanning tree results from the program, visualization of the graph can be done, where visualization in this project is still done manually, not automatically directly through the program, both Reverse Delete and Boruvka produce the same graph. Bellow is the visualization image of Reverse Delete and Boruvka MST.

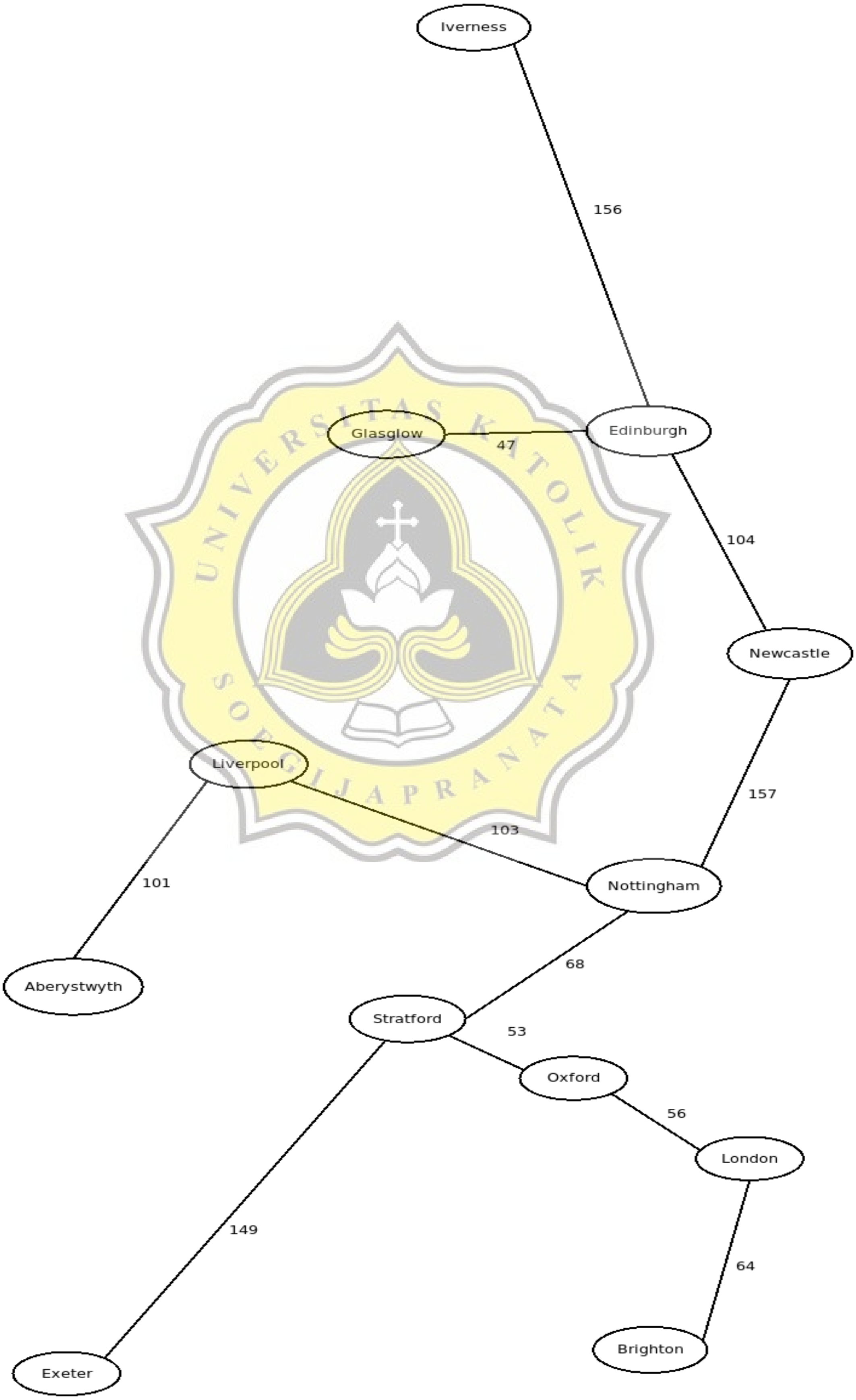

Illustration 5.2: Reverse Delete and Boruvka Sample 1 Visualization

Sample 2

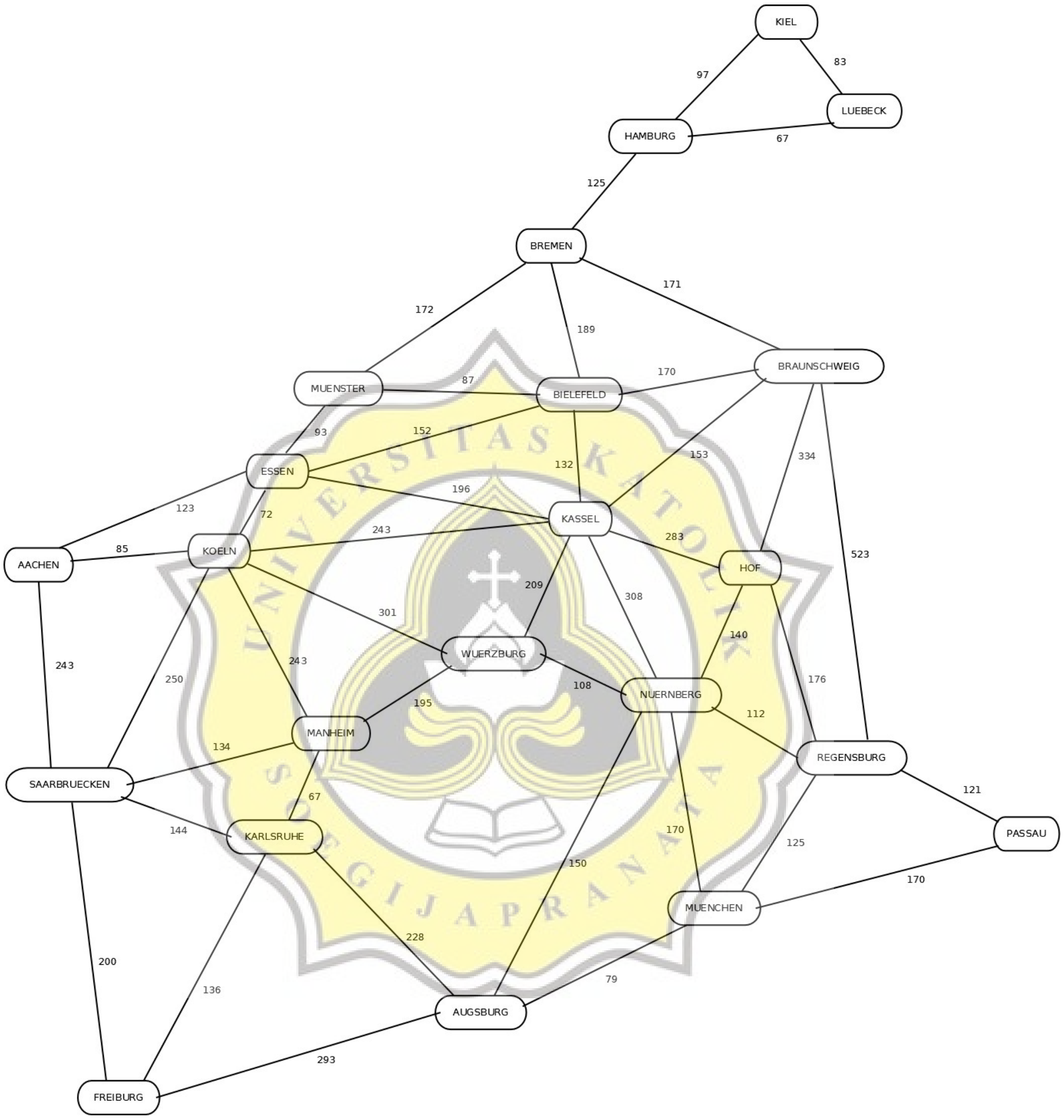

Illustration 5.3: Sample 2 Initial Graph

From the picture above is the initial graph which represents 22 cities in West Germany along with the edges that connect the cities, which is still not processed by the algorithm. With an initial total length of 7864 Km. Bellow is the detailed table of the edges and their length.

Table 5.4: Edges and Weight Sample 2

Edge	Length	Edge	Length
Kiel - Luebeck	83	Hof - Regensburg	176
Kiel - Hamburg	97	Koeln - Aachen	85
Luebeck - Hamburg	67	Koeln - Saarbuecken	250
Hamburg - Bremen	125	Koeln – Manheim	243
Bremen - Braunschweig	171	Koeln - Wuerzburg	301
Bremen - Bielefeld	189	Aachen - Saarbuecken	243
Bremen - Muenster	172	Saarbuecken - Manheim	134
Bielefeld - Braunschweig	170	Saarbuecken - Karlsruhe	144
Bielefeld - Muenster	87	Saarbuecken - Freiburg	200
Bielefeld - Essen	152	Manheim - Karlsruhe	67
Bielefeld - Kassel	132	Manheim - Wuerzburg	195
Braunschweig - Kassel	153	Wuerzburg - Nuernberg	108
Braunschweig - Hof	334	Nuernberg - Augsburg	150
Braunschweig - Regensburg	523	Nuernberg - Muenchen	170
Muenster - Essen	93	Nuenrberg - Regensburg	122
Essen - Aachen	123	Karlsruhe - Freiburg	136
Essen - Koeln	72	Karlsruhe - Augsburg	228
Essen - Kassel	196	Regensburg -Muenchen	125
Kassel - Koeln	243	Regensburg - Passau	121
Kassel - Hof	285	Freiburg - Augsburg	293
Kassel - Wuerzburg	209	Augsburg - Muenchen	79
Kassel - Nuernberg	308	Muenchen - Passau	170
Hof - Nuernberg	140		

After that the data will be processed by the program to form its minimum spanning tree. The result obtained by the Reverse Delete algorithm have a length of 2504 Km. These results are in accordance with the manual calculation of the Reverse Delete algorithm. The results from Reverse Delete program is shown on table bellow.

Table 5.5: Reverse Delete Program Results Table(time in seconds)

Edge	Length
Kiel - Luebeck	83
Luebeck - Hamburg	67
Hamburg - Bremen	125
Bremen - Braunscweig	171
Braunscweig - Kassel	153
Bielefeld - Kasel	132
Bielefeld - Muenster	87
Muenster - Essen	93
Essen - Koeln	72
Kassel - Wuerzburg	209
Hof - Nuernberg	140
Regensburg - Muenchen	125
Regensburg - Nuernberg	122
Regensburg - Passau	121
Aachen - Koeln	85
Wuerzburg - Manheim	195
Wuerzburg - Nuernberg	108
Saarbucken - Manheim	134
Manheim - Karlsruhe	67
Karlsruhe - Freiburg	136
Augsburg - Muenchen	79
Program Execution Time	0.0011403560638427734

After the data is processed by the program, the results obtained by the Boruvka algorithm have a length of 2504 Km which is the same as the results obtained by the Reverse Delete algorithm. The result from the manual calculation of the Boruvka algorithm is also same as that produced by the program. The result from Boruvka program is shown on table bellow.

Table 5.6: Boruvka Program Results Table(time in seconds)

Edge	Length
Kiel - Luebeck	83
Luebeck - Hamburg	67
Hamburg - Bremen	125
Bremen - Braunscweig	171
Braunscweig - Kassel	153
Kassel - Bielefeld	132
Kassel - Wuerzburg	209
Bielefeld - Muenster	87
Muenster - Essen	93
Essen - Koeln	72
Koeln - Aachen	85
Hof - Nuernberg	140
Nuernberg - Wuerzburg	108
Nuernberg - Regensburg	122
Regensburg - Passau	121
Regensburg - Muenchen	125
Wuerzburg - Manheim	195
Saarbucken - Manheim	134
Manheim - Karlsruhe	67
Karlsruhe - Freiburg	136
Augsburg - Muenchen	79
Program Execution Time	0.0006718635559082031

From the two table above, it can be seen that the Boruvka algorithm find MST first. After the minimum spanning tree has been formed from the program, visualization of the path that has been selected by the Boruvka algorithm can be done, where visualization stage in this project is still done manually, not automatically directly from the program, both Reverse Delete and Boruvka produce the same graph. Bellow is visualization image from Reverse Delete and Boruvka MST.

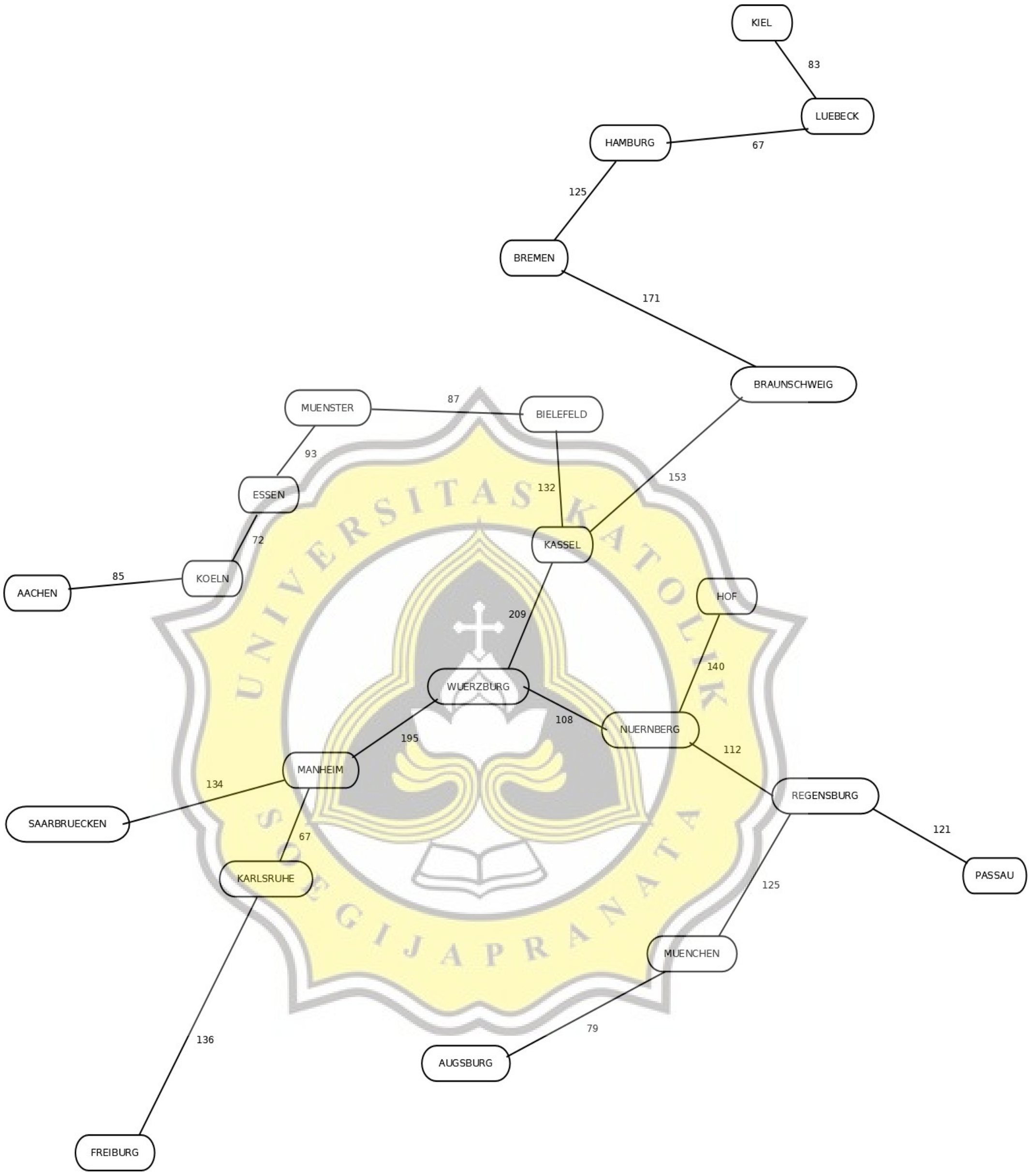

Illustration 5.4: Visualization Of Reverse Delete and Boruvka MST sample 2

Sample 3

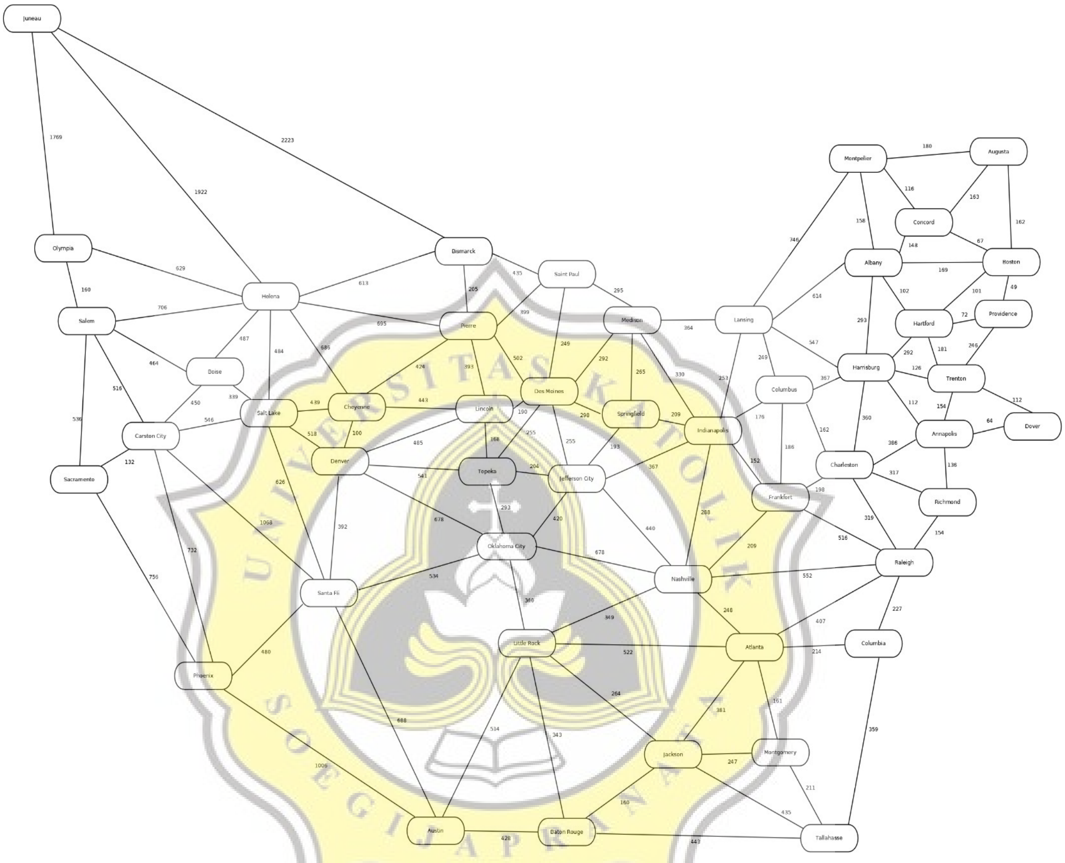

Illustration 5.5: Sample 3 Initial Graph

From the picture above is the initial graph that represents 49 cities in America along with edges that connect these cities, which are still not processed by the algorithm. With a total initial length of 47173 miles. The table bellow details the edges and length.

Table 5.7: Table Detail Sample 3

Edge	Length	Edge	Length
Juneau - Olympia	1769	Des Moines - Topeka	255
Juneau - Helena	1922	Des Moines - Lincoln	190
Juneau - Bismarck	2223	Lincoln - Topeka	168
Olympia - Salem	160	Lincoln - Denver	485
Olympia - Helena	629	Lincoln - Cheyenne	443
Helena - Salem	706	Cheyenne - Denver	100
Helena - Boise	487	Cheyenne - Salt Lake	439
Helena - Salt Lake	484	Salt Lake - Denver	518
Helena - Cheyenne	686	Salt Lake - Santa Fe	626
Helena - Piere	695	Salt Lake - Boise	339
Helena - Bismarck	613	Salt Lake - Carston City	546
Bismarck - Pierre	205	Boise - Carston City	450
Bismarck - Saint Paul	435	Boise - Salem	464
Saint Paul - Pierre	399	Salem - Carston City	516
Saint Paul - Des Moines	249	Salem - Sacramento	536
Saint Paul - Medison	295	Sacramento - Carston City	132
Pierre - Cheyenne	424	Sacramento - Phoenix	756
Pierre - Lincoln	393	Carston City - Phoenix	732
Piere - Des Moines	502	Carston City - Santa Fe	1068
Medison - Des Moines	292	Denver – Santa Fe	392
Medison - Springfield	265	Denver - Topeka	541
Medison - Indianapolis	330	Denver - Oklahoma City	678
Medison - Lansing	364	Topeka - Oklahoma City	293
Lansing - Indianapolis	253	Topeka - Jefferson City	204
Lansing - Columbus	249	Jefferson City - Oklahoma City	420
Lansing - Harrisburg	347	Jefferson City - Nashville	440
Lansing - Albany	614	Dover - Annapolis	64
Lansing - Montpelier	746	Annapolis - Richmond	136
Montpelier - Albany	158	Annapolis - Charleston	386
Montpelier - Concord	116	Charleston - Richmond	317
Montpelier - Augusta	180	Charleston - Raleigh	319
Augusta - Concord	163	Charleston Frankfort	198
Augusta - Boston	162	Frankfort - Raleigh	516
Concord - Albany	148	Frankfort - Nashville	209
Concord – Boston	67	Richmond - Raleigh	154
Albany - Boston	169	Raleigh - Columbia	227
Albany - Harrisburg	293	Raleigh - Atlanta	407
Albany - Hartford	102	Raleigh - Nashville	552
Boston - Hartford	101	Nashville - Atlanta	248
Boston - Providence	49	Nashville - Little Rock	349
Providence - Hartford	72	Nashville - Oklahoma City	678

Providence - Trenton	181	Oklahoma City - Little Rock	340
Hartford - Harrisburg	126	Oklahoma City - Santa Fe	534
Hartford - Trenton	181	Santa Fe - Austin	688
Trenton - Harrisburg	126	Santa Fe - Phoenix	480
Trenton - Annapolis	154	Phoenix - Austin	1006
Trenton - Dover	112	Little Rock - Austin	514
Harrisburg - Annapolis	112	Little Rock – Batton Rouge	343
Harrisburg – Charleston	360	Little Rock - Atlanta	522
Harrisburg - Columbus	367	Little Rock - Jackson	264
Columbus - Charleston	162	Atlanta - Jackson	381
Columbus - Frankfort	186	Atlanta - Montgomery	161
Columbus - Indianapolis	176	Atlanta - Columbia	214
Indianapolis - Frankfort	152	Columbia – Tallahasse	359
Indianapolis - Nashvile	288	Montgomery - Tallahase	211
Indianapolis - Jefferson City	367	Montgomery - Jackson	247
Indianapolis - Springfield	209	Jackson - Tallahasse	435
Springfield - Jefferson City	193	Jackson – Batton Rouge	160
Springfield - Des Moines	298	Austin - Batton Rouge	428
Des Moines - Jefferson City	255	Batton Rouge - Tallahasse	443

After that the data will be processed by the program to form its minimum spanning tree. The result obtained by the Reverse Delete algorithm have a length of 12437 Miles. These results are in accordance with the manual calculation of the Reverse Delete algorithm. The results from Reverse Delete program is shown on table bellow.

Table 5.8: Table Reverse Delete Saample 3 Results(time in seconds)

Edge	Length
Juneau – Olympia	1769
Olympia - Salem	160
Helena – Salt Lake	484
Bismarck - Pierre	205
Salem - Boise	464
Boise – Carston City	450
Boise – Salt Lake	339
Salt Lake - Cheyenne	439
Cheyenne - Pierre	424
Cheynne - Denver	100
Pierre - Lincoln	393
Saint Paul – Des Moines	249
Des Moines - Lincoln	190
Medison - Springfield	265
Lincoln - Topeka	168
Springfield - Indianapolis	209
Springfield – Jefferson City	193
Indianapolis – Columbus	176
Indianapolis - Frankfort	152
Lansing - Columbus	249
Columbus - Charleston	162
Harrisburg - Annapolis	112
Albany - Hartford	102
Montpelier - Concord	116
Concord - Boston	67
Augusta - Boston	162
Boston - Providence	49
Hartford - Trenton	181
Hartford - Providence	72
Trenton - Dover	112
Annapolis - Richmond	136
Annapolis - Dover	64
Frankfort - Nashville	209
Nashville - Atlanta	248
Jefferson City - Topeka	204
Topeka – Oklahoma City	293
Denver – Santa Fe	392
Santa Fe - Phoenix	480
Carston City - Sacramento	132
Richmond - Raleigh	154
Raleigh - Columbia	227
Columbia - Atlanta	214
Atlanta - Montgomery	161

Little Rock - Jackson	264
Austin – Batton Rouge	428
Baton Rouge - Jackson	160
Jackson - Montgomery	247
Montgomery - Tallahasse	211
Program Execution Time	0.006117105484008789

After the data is processed by the program, the results obtained by the Boruvka algorithm have a length of 12437 Miles, which is the same as the results obtained by the Reverse Delete algorithm. The result from the manual calculation of the Boruvka algorithm is also same as that produced by the program. The result from Boruvka program is shown on table bellow.

Table 5.9: Table Boruvka Sample 3 Results(time in seconds)

Edge	Length
Juneau - Olympia	1769
Olympia - Salem	160
Salem - Boise	464
Helena – Salt Lake	484
Salt Lake - Boise	339
Salt Lake - Cheyenne	439
Bismarck - Pierre	205
Pierre - Lincoln	393
Pierre - Cheyenne	424
Boise – Carston City	450
Cheyenne - Denver	100
Denver – Santa Fe	392
Saint Paul – Des Moines	249
Des Moines - Lincoln	190
Lincoln - Topeka	168
Medison - Springfield	265
Springfield – Jefferson City	193
Springfield - Indianapolis	209
Topeka – Oklahoma City	293
Topeka – Jefferson City	204
Indianapolis - Frankfort	152
Indianapolis - Columbus	176
Frankfort - Nashville	209
Lansing - Columbus	249
Columbus - Charleston	162
Harrisburg - Annapolis	112
Annapolis - Dover	64
Annapolis - Richmond	136
Albany - Hartford	102
Hartford - Providence	72
Montpelier - Concord	116
Concord - Boston	67

Boston - Augusta	162
Boston - Providence	49
Trenton - Dover	112
Nashville - Atlanta	248
Santa Fe - Phoenix	480
Sacramento - Carston City	132
Richmond - Raleigh	154
Raleigh - Columbia	227
Atlanta - Columbia	214
Atlanta - Montgomery	161
Montgomery - Tallahasse	211
Montgomery - Jackson	247
Little Rock - Jackson	264
Jackson – Baton Rouge	160
Austin – Baton Rouge	428
Program Execution Time	0.0021924972534179688

From the two table above, it can be seen that the Boruvka algorithm find MST first. After the minimum spanning tree has been formed from the program, visualization of the path that has been selected by the Boruvka algorithm can be done, where visualization stage in this project is still done manually, not automatically directly from the program, both Reverse Delete and Boruvka produce the same graph. Bellow is visualization image from Reverse Delete and Boruvka MST.

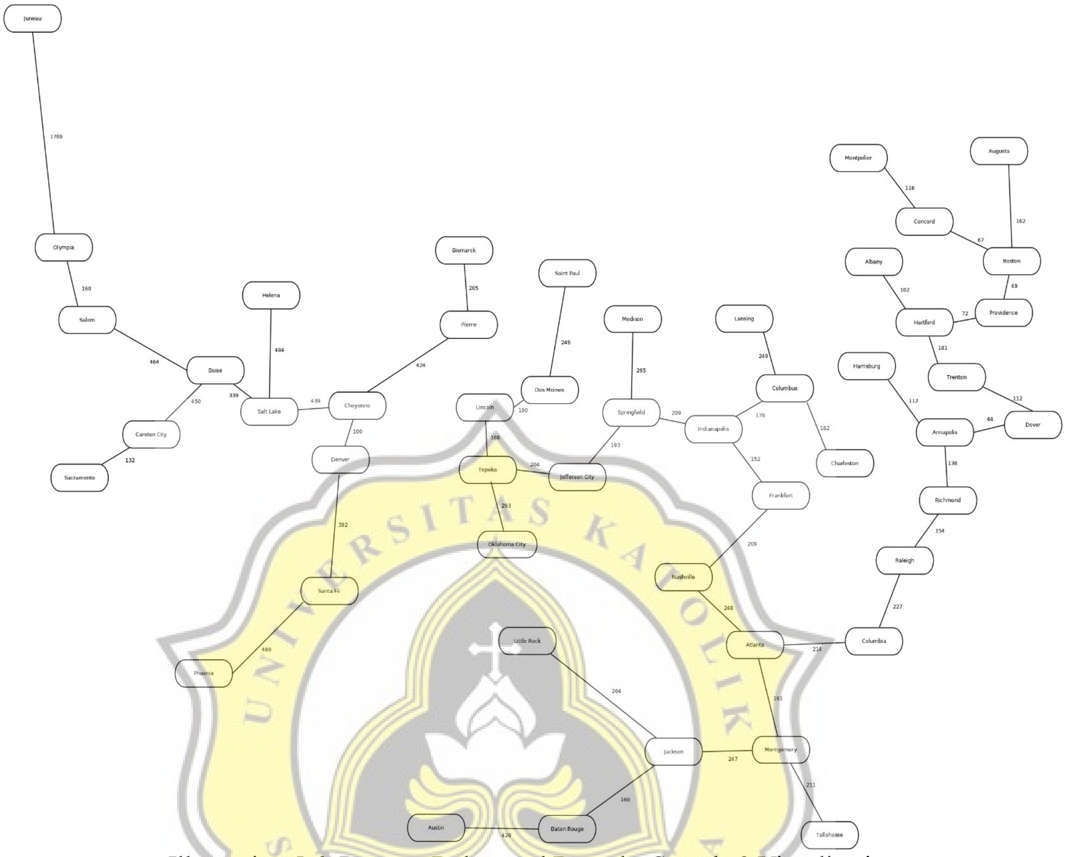

Illustration 5.6: Reverse Delete and Boruvka Sample 3 Visualization

Sample 4

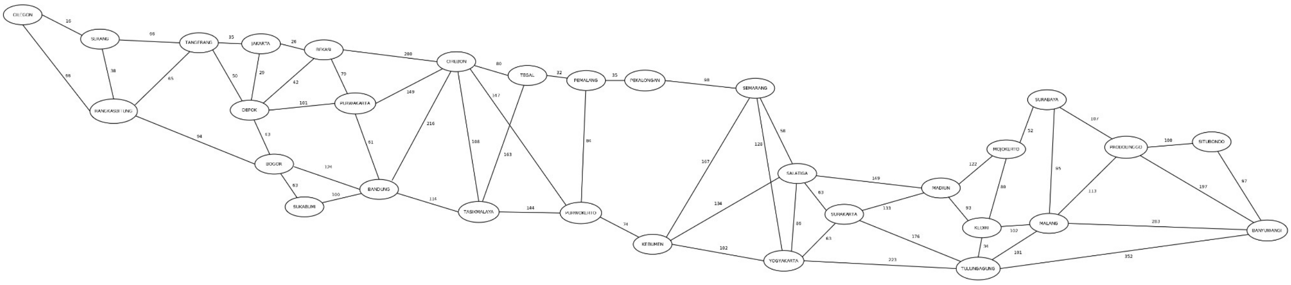

Illustration 5.7: Sample 4 Initial Graph

The function of sample 4 is to find out how much accuracy the algorithm results are compared to the original toll road on the island of Java. The image above represents 31 cities on the island of Java, Indonesia with 60 edges that connect these cities, which have not been processed by the algorithm with a total distance of 6178 Km. Bellow is the detailed table of the edge and their length.

Table 5.10: Table Detail Sample 4

Edge	Length	Edge	Length
Cilegon - Serang	16	Purwokerto - Kebumen	74
Cilegon - Rangkasbitung	66	Pemalang - Pekalongan	35
Serang - Tangerang	66	Pekalongan - Semarang	98
Serang - Rangkasbitung	38	Kebumen - Yogyakarta	102
Rangkasbitung - Bogor	94	Kebumen - Semarang	167
Rangkasbitung - Tangerang	65	Kebumen - Salatiga	134
Tangerang - Jakarta	35	Semarang - Salatiga	58
Tangerang - Depok	50	Semarang - Yogyakarta	128
Jakarta - Bekasi	42	Salatiga -Yogyakarta	86
Jakarta - Depok	29	Salatiga - Surakarta	63
Depok - Bogor	43	Salatiga - Madiun	149
Depok - Bekasi	42	Yogyakarta - Tulungagung	223
Depok - Purwakarta	101	Yogyakarta - Surakarta	63
Bekasi - Purwakarta	79	Surakarta - Madiun	113
Bekasi - Cirebon	200	Surakarta - Tulungagung	176
Bogor - Bandung	124	Madiun - Kediri	93
Bogor - Sukabumi	63	Madiun - Mojokerto	122
Purwakarta - Cirebon	149	Mojokerto - Surabaya	52
Purwakarta - Bandung	61	Kediri - Tulungagung	34
Sukabumi - Bandung	100	Kediri - Malang	102
Bandung - Cirebon	216	Kediri - Mojokerto	80
Bandung - Tasikmalaya	114	Tulungagung - Malang	101

Tasikmalaya - Purwokerto	144	Tulungagung - Banyuwangi	352
Tasikmalaya - Cirebon	108	Malang - Surabaya	95
Tasikmalaya - Tegal	163	Malang - Probolinggo	113
Cirebon - Tegal	80	Malang - Banyuwangi	283
Cirebon - Purwokerto	147	Surabaya - Probolinggo	107
Tegal - Purwokerto	98	Probolinggo - Situbondo	100
Tegal - Pemalang	32	Probolinggo - Banyuwangi	197
Purwokerto - Pemalang	84	Situbondo - Banyuwangi	97

After that the data will be processed by the program to form its minimum spanning tree. The result obtained by the Reverse Delete algorithm have a length of 2035 Km. These results are in accordance with the manual calculation of the Reverse Delete algorithm. The results from Reverse Delete program is shown on table bellow.

Table 5.11: Table Reverse Delete Sample 4 Results(time in seconds)

Edges	**Length**
Cilegon - Serang	16
Serang - Rangkasbitung	38
Rangkasbitung - Tangerang	65
Tangerang - Jakarta	35
Bogor - Sukabumi	63
Bogor - Depok	43
Jakarta - Depok	29
Jakarta - Bekasi	26
Bekasi - Purwakarta	79
Purwakarta - Bandung	61
Cirebon - Tasikmalaya	108
Cirebon - Tegal	80
Bandung - Tasikmalaya	114
Purwokerto - Pemalang	84
Purwokerto - Kebumen	74
Tegal - Pemalang	32
Pemalang - Pekalongan	35
Pekalongan - Semarang	98
Semarang - Salatiga	58
Yogyakarta - Surakarta	63
Salatiga - Surakarta	63
Surakarta - Madiun	113
Madiun - Kediri	93
Tulungagung - Kediri	34

Kediri - Mojokerto	80
Mojokerto - Surabaya	52
Surabaya - Probolinggo	107
Surabaya - Malang	95
Banyuwangi - Situbondo	97
Probolinggo - Situbondo	100
Program Execution Time	0.0019724369049072266

After the data is processed by the program, the results obtained by the Boruvka algorithm have a length of 2035 Km, which is the same as the results obtained by the Reverse Delete algorithm. The result from the manual calculation of the Boruvka algorithm is also same as that produced by the program. The result from Boruvka program is shown on table bellow.

Table 5.12: Boruvka Program Results(time in seconds)

Edge	Length
Cilegon - Serang	16
Serang - Rangkasbitung	38
Rangkasbitung - Tangerang	65
Tangerang - Jakarta	35
Jakarta - Bekasi	26
Jakarta - Depok	29
Depok - Bogor	43
Bogor - Sukabumi	63
Bekasi - Purwakarta	79
Purwakarta - Bandung	61
Bandung - Tasikmalaya	114
Cirebon - Tegal	80
Cirebon - Tasikmalaya	108
Tegal - Pemalang	32
Purwokerto - Kebumen	74
Purwokerto - Pemalang	84
Pemalang - Pekalongan	35
Pekalongan - Semarang	98
Semarang - Salatiga	58
Salatiga - Surakarta	63
Yogyakarta - Surakarta	63
Surakarta - Madiun	113
Madiun - Kediri	93
Kediri - Tulungagung	34
Kediri - Mojokerto	80
Mojokerto - Surabaya	52
Surabaya - Malang	95
Surabaya - Probolinggo	107
Situbondo - Banyuwangi	97
Situbondo - Probolinggo	100
Program Execution Time	0.0009052753448486328

From the two table above, it can be seen that the Boruvka algorithm find MST first. After the minimum spanning tree has been formed from the program, visualization of the path that has been selected by the Boruvka algorithm can be done, where visualization stage in this project is still done manually, not automatically directly from the program, both Reverse Delete and Boruvka produce the same graph. Bellow is visualization image from Reverse Delete and Boruvka MST.

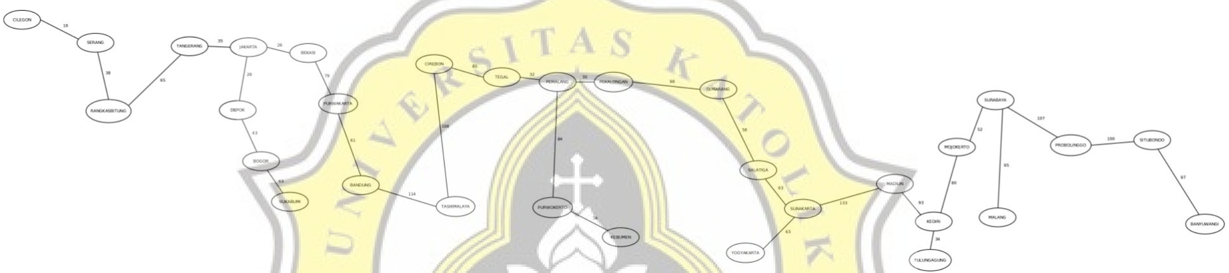

Illustration 5.8: Reverse Delete and Boruvka Visualization Sample 4

From the table and visualization above, compared to the original toll road on the island of Java, of the total 30 lanes formed, 19 of them have the same lanes as the original toll roads. Bellow is a table of lanes formed which is the same as the original toll road.

Table 5.13: Table From The Same Line Of The Original Toll

Edge	**Length**
Cilegon - Serang	16
Tangerang - Jakarta	35
Jakarta - Bekasi	26
Jakarta - Depok	29
Depok - Bogor	43
Bekasi - Purwakarta	79
Purwakarta - Bandung	61
Cirebon - Tegal	80
Tegal - Pemalang	32
Pemalang - Pekalongan	35
Pekalongan - Semarang	58
Semarang - Salatiga	58
Salatiga - Surakarta	63
Surakarta - Madiun	113
Madiun - Kediri	93
Kediri - Mojokerto	80
Mojokerto - Surabaya	52
Surabaya - Malang	95
Surabaya - Probolinggo	107
Accuracy	63%

From the table above we can see that the accuracy rate of both the Reverse Delete and Boruvka algorithm is 63%. This result is obtained from 19 divided by 30 then multiplied by 100.