Calculating Optimal Meeting Point using Modified Dijkstra's Algorithm

Felicia Sutandijo - 13520050[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13520050@std.stei.itb.ac.id

***Abstract*—In human society, a meeting between two or more people is a common occurrence. A modified Dijkstra's algorithm which results in data of all shortest paths from each vertex to every other vertex in a strongly connected, weighted, directed graph may be used to determine an optimal meeting point, where cost is minimized. Two types of cost minimizations that will be studied include total cost minimization and maximum cost minimization.**

***Keywords*—optimal meeting point, Dijkstra's algorithm, weighted graph, graph theory.**

I. Introduction

Human beings are social creatures. Therefore, a meeting between two or more people is a common occurrence in human society. By definition, a meeting is a gathering of two or more people that has been convened for the purpose of achieving a common goal through verbal interaction, such as sharing information or reaching an agreement [1]. Meetings happen for various reasons, such as to discuss work, find a suitable romantic partner, or simply to have fun together. However, there is always one thing in common between all meetings taking place in real life, which is a meeting point. In order to get together, people have to decide *where* to meet.

Deciding a suitable meeting point is usually done by a vague estimation of how far everyone is, without any mathematical basis. However, there are certain instances when efficiency or cost must be scrutinized. In these cases, there are two kinds of cost minimization. The first one is to pick a place where the total cost would be minimum, and the second is to pick a place where the maximum cost would be minimum [2]. The first may be used, for instance, when the total cost of travel is covered by a single person or organization. The second would be a better choice to ensure that no single person would travel too far.

This problem can be represented by a graph, where vertices represent places and edges represent routes. To illustrate the problem efficiently, the type of graph used is a strongly connected, weighted, directed graph. Thus, the implementation of Dijkstra's algorithm to determine the best routes everyone should travel is possible. Furthermore, to make things more efficient, there will be slight modifications to the classic Dijkstra's algorithm, where the resulting calculation would include all shortest paths from each vertex to every other vertex in the graph.

II. Graph Theory

A. Graph

Graphs are defined as mathematical structures which consist of a set of vertices (also known as nodes or points) connected by edges (also known as links or lines).

Formally [3], a graph can be written as $G = (V, E)$ where:

- V is a set of vertices (also called nodes or points), and
- E is a set of edges (also called links or lines), which are unordered pairs of vertices (that is, an edge is associated with two distinct vertices).

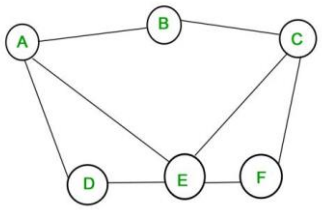

Fig. 1. Example of a simple graph. Source: https://www.geeksforgeeks.org/mathematics-graph-theory-basics/

In this paper, places which serve as possible meeting points are represented by vertices, and routes connecting these places are represented by weighted edges.

B. Directed Graph

Graphs can be further classified into directed or undirected graphs. A directed graph or a digraph is a graph in which edges have orientations.

An edge directed from vertex A to B has a *tail* and a *head*, where A is the tail of the edge and B the head of the edge. A and B are called the *endpoints* of the edge. In a directed graph, a *loop*, which is an edge directed from a vertex to itself, is not allowed [3].

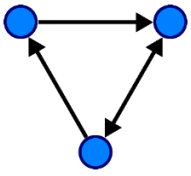

Fig. 2. Example of a directed graph. Source: https://en.wikipedia.org/wiki/Graph_theory#/media/File:Directed.svg

As opposed to undirected graphs, representing the problem discussed in this paper with directed graphs is more convenient and appropriate, as the route from place A to place B may differ from the route from place B to place A. With directed graphs, both routes may be represented with different edges with different orientations, so as not to confuse the two. This setting is not possible using an undirected graph.

C. Weighted Graph

Graphs can also be classified by their weight. A weighted graph is categorized by the existence of a numerical *weight* in each edge. These weights may represent various things such as the cost, length, or preference of each edge. A weighted graph is therefore a special type of labeled graph in which the labels are numbers, usually positive but may also be negative [4].

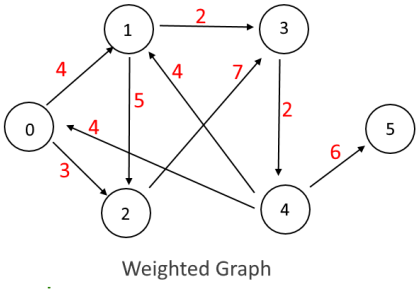

Fig. 3. Example of a weighted graph. Source: https://algorithms.tutorialhorizon.com/weighted-graph-implementation-java/

In this paper, the weights assigned to each edge portray the distance or the cost which is needed to travel the road between two places. This weight is crucial since the goal of the algorithm is to find a place where the cost of travel for all parties is minimized.

C. Graph Terminology

There are a few graph terminologies used in this paper to describe graphs, which are listed below [5].

1. Adjacency
   Two vertices are said to be *adjacent* when they are directly connected.
2. Incidence
   For any edge $e = (v_1, v_2)$, e is *incident* with vertex v_1, or e is *incident* with vertex v_2.
3. Isolated Vertex
   An *isolated vertex* is any vertex which has no incident edge.

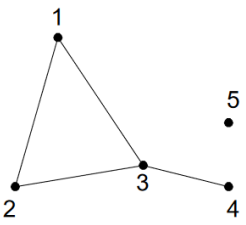

Fig. 4. Vertex 5 is an isolated vertex. Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

4. Null Graph (Empty Graph)
   A graph is said to be a *null graph* when it is comprised of an empty set.

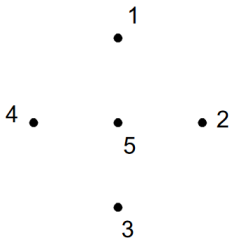

Fig. 5. Example of a null graph. Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

5. Degree
   The *degree* of a certain vertex is the count of incident edges of the said vertex.
6. Path
   A *path* from starting vertex v_0 to target vertex v_n is a sequence of alternating vertices and edges such that each successive vertex is connected by the edge. The length of a path is the number of edges in the path.
7. Cycle (Circuit)
   A *cycle* or a *circuit* is a path that starts and ends with the same vertex.
8. Connectedness
   Two vertices v_0 and v_n is said to be *connected* if there is a path from v_0 to v_n. A graph is connected when for every pair of vertices v_i and v_j, there exists a path from v_i to v_j. If not, the graph is called a disconnected graph. In a directed graph, there are two kinds of connectedness: weakly connected and strongly connected. Two vertices u and v are called strongly connected if there exists a path from u to v and also a path from v to u. A connected directed graph is called strongly connected when all pairs of vertices are strongly connected. Otherwise, the graph is weakly connected.

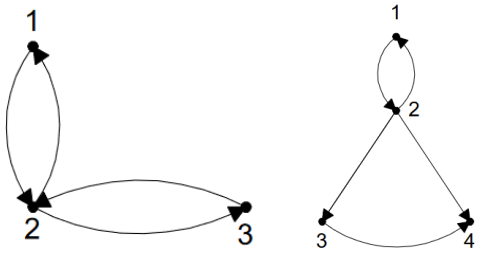

Fig. 6. A strongly connected graph (left) and a weakly connected graph (right). Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

9. Subgraph
   A *Subgraph* is a subset of vertices and edges in a graph.
10. Spanning Subgraph
    A *Spanning subgraph* is a special type of subgraph which consists of all the vertices.

D. Graph Representation

A graph may be represented in various ways, three of which are listed below [6].

1. Adjacency Matrix

   $A = [a_{ij}]$,

   $$aij = \begin{cases} 1, & \text{if vertex i and j are adjacent} \\ 0, & \text{if vertex i and j are not adjacent} \end{cases}$$

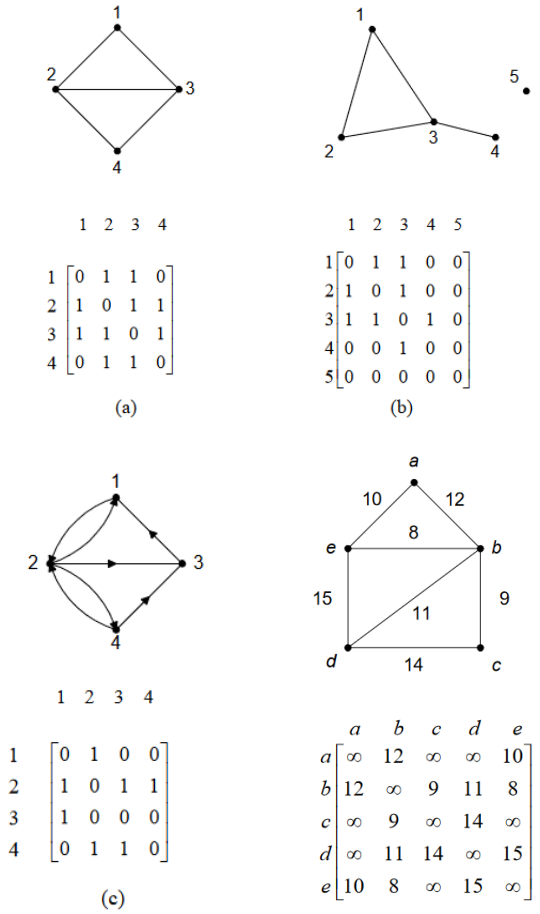

Fig. 7. Examples of adjacency matrices. Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian2.pdf

2. Incidency Matrix

   $A = [a_{ij}]$,

   $$aij = \begin{cases} 1, & \text{if vertex i and edge j are incident} \\ 0, & \text{if vertex i and edge j are not incident} \end{cases}$$

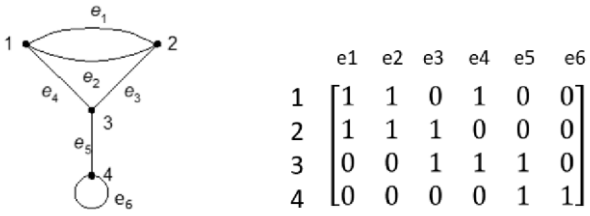

Fig. 8. Examples of incidency matrix. Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian2.pdf

3. Adjacency List

   For each vertex i, all vertices which are adjacent to i are listed.

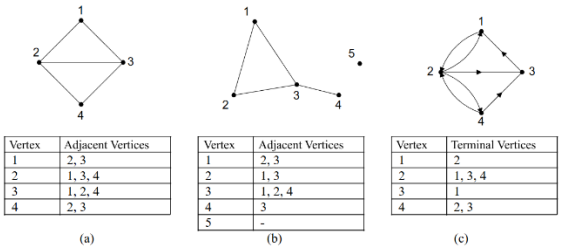

Fig. 9. Examples of adjacency list. Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian2.pdf

III. DIJKSTRA'S ALGORITHM

Created by computer scientist Edsger W. Dijkstra in 1956, Dijkstra's algorithm is an algorithm used to calculate the shortest or lowest-cost path between two points in a weighted graph. This algorithm is designed for weighted graphs which edges have non-negative weights. Dijkstra's algorithm is widely used in network routing protocols and geographical maps [7][8][9].

Even though the original purpose of the algorithm is to determine the shortest path from one vertex to another, Dijkstra's algorithm, as a whole, automatically calculates the cheapest paths from one vertex to *every other vertex* in the graph [10].

Dijkstra's algorithm works by first determining the initial or starting vertex and setting it as an anchor. The algorithm works in these steps.

1. All vertices are marked unvisited and stored in a set called the *unvisited set.*
2. Every vertex is assigned a *tentative distance* value. The distance from the starting vertex to itself is set to zero, whereas to every other vertex is set to infinity. The tentative distance of a vertex *v* is the length of the shortest path between the starting vertex and the vertex *v* calculated so far. Initially, there are no known paths from the starting vertex to any other vertex. Therefore, the tentative distances of all vertices (except for the starting vertex itself) are set to infinity.
3. The algorithm begins by visiting the starting vertex and marking it as the *current vertex.*
4. For the current vertex, all its unvisited adjacent vertices' tentative distances are calculated through the current node. The newly calculated tentative distance is then compared to the current assigned tentative distance. Then, the algorithm keeps the smaller of the two.
5. After all unvisited adjacent vertices are evaluated, the current node is marked as visited and removed from the unvisited set. A visited vertex will not be checked again.
6. The vertex which is marked with the smallest tentative distance is selected from the unvisited set and steps 4-6 are repeated until a stop condition is met.
7. There are two conditions when the algorithm stops depending on the goal:

1. When calculating the shortest path between one vertex to another, the algorithm stops when the destination vertex has been reached.
2. When calculating the shortest path between one vertex and all other vertices, the algorithm stops when the smallest tentative distance among the vertices in the unvisited set is infinity. This means that there is no connection between the initial vertex and the remaining unvisited vertices.

8. When the stop condition is met, the algorithm would have returned the shortest paths between the starting point and every other vertex in the graph (unless the goal is to find the shortest path between one vertex with another, which in this case the algorithm is stopped early).

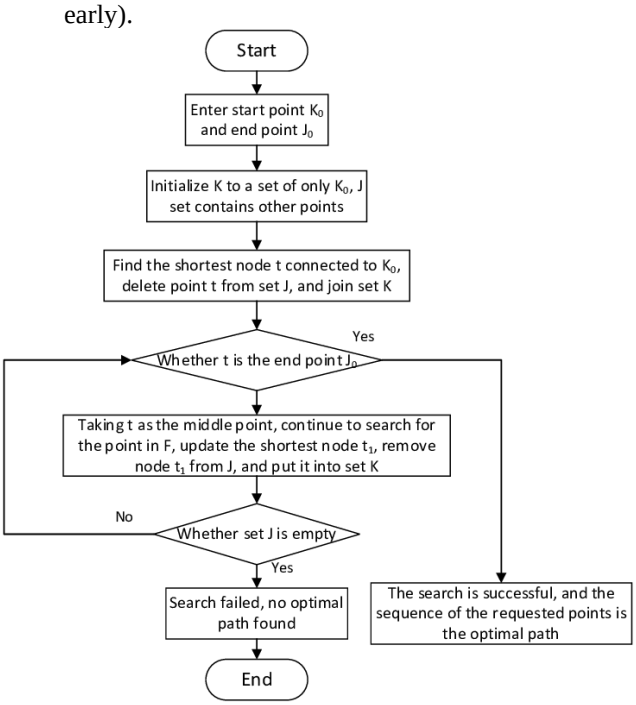

Fig. 10. A flowchart of Dijkstra's algorithm. Source: https://www.researchgate.net/figure/Improved-Dijkstra-algorithm-flowchart_fig3_348091761

IV. Representation of the Optimal Meeting Point Problem using Weighted Directed Graph

Suppose there are five people living in five different cities (Atlanta, Boston, Chicago, Denver, and El Paso) who want to decide on a city to meet with the minimal plane ticket cost. The ticket costs are listed below.

Table 1. Plane ticket prices for this scenario

From	To	Price
Atlanta	Boston	$190
Atlanta	Chicago	$250
Boston	Atlanta	$200
Boston	Chicago	$80
Boston	Denver	$150
Chicago	Boston	$90
Chicago	Denver	$80
Denver	El Paso	$120
El Paso	Chicago	$120
El Paso	Denver	$130

Each city will be represented by a vertex, while the costs of plane tickets will be represented by the weight of directed edges between the vertices.

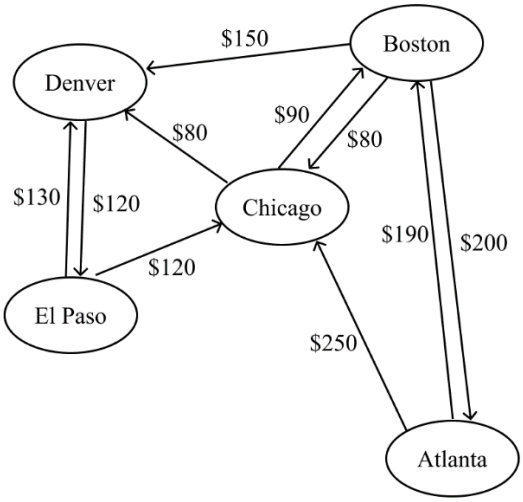

Fig. 11. A graph representation of plane ticket costs. Source: Author

This graph will then be represented in a format that can be read by the program. Because the point of interest is the inbound edges, the representation should revolve around the inbound edges rather than the outbound edges like the classic Dijkstra does. The representation below is written in the form of an adjacency list.

Atlanta : Boston, 200
Boston : Atlanta, 190; Chicago, 90
Chicago : Atlanta, 250; Boston, 80; El Paso, 120
Denver : Boston, 150; Chicago, 40; El Paso, 130
El Paso : Denver, 120

This list means there is one inbound edge going to Atlanta from Boston that costs $200, two inbound edges to Boston from Atlanta and Chicago, and so on.

V. The Use of Modified Dijkstra's Algorithm in Calculating an Optimal Meeting Point

The first step in determining the optimal meeting point of a graph is to ensure the connectivity of the graph. A suitable meeting point can only be obtained if the resulting graph representation is strongly connected, which means people can travel from one city to another. If the resulting graph is not connected or weakly connected, no suitable meeting point can be derived, and the program terminates. The above example is a strongly connected graph.

Every vertex is then assigned a tentative price value, which is listed in the table below. The first row represents the initial city, and the first column represents the destination city. The tentative price value assigned to any city from itself is zero, while the tentative price value to any city from all other cities are infinity.

Table 2. Initial tentative distance value

	Atlanta	Boston	Chicago	Denver	El Paso
Atlanta	0	∞	∞	∞	∞
Boston	∞	0	∞	∞	∞
Chicago	∞	∞	0	∞	∞
Denver	∞	∞	∞	0	∞
El Paso	∞	∞	∞	∞	0

Next, an arbitrary starting point is chosen. In this example, Atlanta shall be picked as the starting city. Like in classic Dijkstra's algorithm, the starting city is first set to be the current city and marked visited.

Unvisited cities need to be kept track. For efficiency, a priority queue may be used in code since the next visited city is the city cheapest to the visited cities.

For each time a city is visited, all inbound edges are inspected and compared to the list of cheapest prices which has already been discovered.

In the first step, the cheapest price to Atlanta from Boston is discovered, which is $200. This price is added to the tentative price table and Boston is added to the unvisited cities priority queue.

Fig. 12. Step 1: Atlanta. Source: Author

Table 3. Step 1: Atlanta

	Atlanta	Boston	Chicago	Denver	El Paso
Atlanta	0	*200*	∞	∞	∞
Boston	∞	0	∞	∞	∞
Chicago	∞	∞	0	∞	∞
Denver	∞	∞	∞	0	∞
El Paso	∞	∞	∞	∞	0

After all adjacent cities have been checked, in this case only Boston, the algorithm moves on to the next unvisited city. Checking the unvisited cities queue, the next destination would be Boston.

After marking Boston as visited, the inbound edges are checked. Boston has two inbound edges, which are from Atlanta ($190) and from Chicago ($90). These prices are still the cheapest to Boston, so they are added to the cheapest prices table, replacing the infinity marks.

In this modified algorithm, Atlanta should also be taken into account. Chicago can get to Atlanta through Boston. Therefore, Chicago's price is calculated. The price to Atlanta from Chicago through Boston is $90 + $190, which is $290. Comparing this value to the value in the table (infinity), this value is cheaper. Therefore, $290 is also updated as the price to Atlanta from Chicago. This marks the end of step 2.

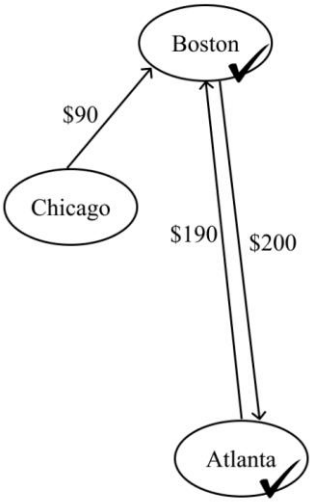

Fig. 13. Step 2: Boston. Source: Author

Table 4. Step 2: Boston

	Atlanta	Boston	Chicago	Denver	El Paso
Atlanta	0	200	*290*	∞	∞
Boston	*190*	0	*90*	∞	∞
Chicago	∞	∞	0	∞	∞
Denver	∞	∞	∞	0	∞
El Paso	∞	∞	∞	∞	0

Again, the unvisited cities queue is checked. There is one discovered but unvisited city, which is Chicago. The algorithm then moves on to Chicago.

Similar to the first and second steps, all inbound edges to Chicago are examined. In this case, there are three inbound edges, which are from Atlanta ($250), Boston ($80), and El Paso ($120). All prices through Chicago are calculated and the cheapest prices table is updated.

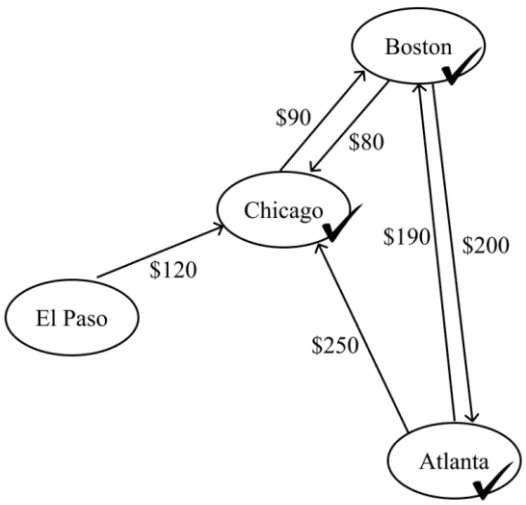

Fig. 14. Step 3: Chicago. Source: Author

Table 5. Step 3: Chicago

	Atlanta	Boston	Chicago	Denver	El Paso
Atlanta	0	200	290	∞	*410*
Boston	190	0	90	∞	*210*
Chicago	*250*	*80*	0	∞	*120*
Denver	∞	∞	∞	0	∞
El Paso	∞	∞	∞	∞	0

The next city to visit is El Paso. There is only one inbound edge, which is from Denver and costs $210. All prices through El Paso are calculated and compared with the values in the cheapest prices table. If any is found to be cheaper, the table is updated.

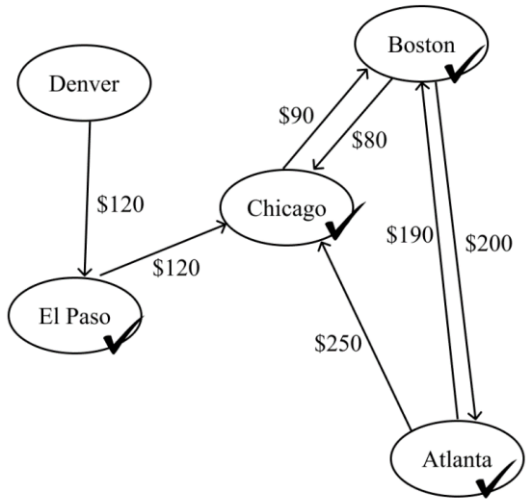

Fig. 15. Step 4: El Paso. Source: Author

Table 6. Step 4: El Paso

	Atlanta	Boston	Chicago	Denver	El Paso
Atlanta	0	200	290	*530*	410
Boston	190	0	90	*330*	210
Chicago	250	80	0	*240*	120
Denver	∞	∞	∞	0	∞
El Paso	∞	∞	∞	*120*	0

The last unvisited city in this case is Denver. Checking all three inbound edges, the cheapest prices table is updated.

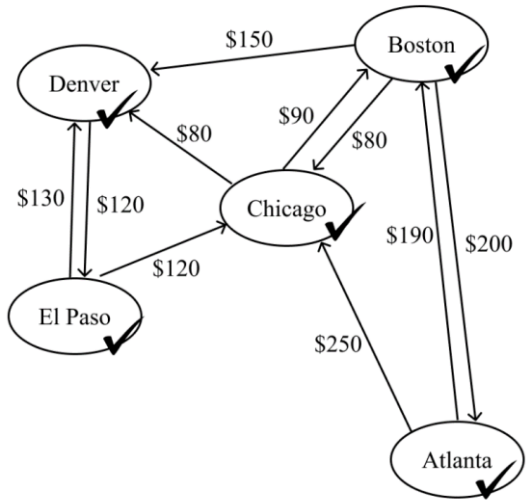

Fig. 16. Step 5: Denver. Source: Author

Table 7. Step 5: Denver

	Atlanta	Boston	Chicago	Denver	El Paso
Atlanta	0	200	290	530	410
Boston	190	0	90	330	210
Chicago	250	80	0	240	120
Denver	∞	*150*	*40*	0	*130*
El Paso	∞	∞	∞	120	0

Since all cities have been visited, the first loop has been completed. In the classic Dijkstra's algorithm, the program would have stopped here. However, as apparent from the table of cheapest prices that has been accumulated, some of the cells have not been filled yet (the value is still infinity) even though it is known that each vertex has at least a single path to any other vertex since the graph has been established as a strongly connected graph.

To fill in the missing cells as well as update the current cheapest prices table with any path that will be discovered to be cheaper, the program loops one more time with the same order as the first loop (Atlanta, Boston, Chicago, El Paso, Denver). The comparison algorithm is also the same. The resulting table is shown below.

Table 8. Final result

	Atlanta	Boston	Chicago	Denver	El Paso
Atlanta	0	200	290	530	410
Boston	190	0	90	330	210
Chicago	250	80	0	240	120
Denver	*290*	*120*	40	0	130
El Paso	*410*	*240*	*160*	120	0

After all cheapest paths between every vertex are known, the calculation to determine the optimal meeting point can begin.

There are two types of minimizing cost that is being studied in this paper. The first is to determine the optimal meeting point based on the city that has the minimum total cost from all other cities, and the second one is to decide based on the minimum maximum-cost of each city, so that no one person should travel too expensively.

In the first method, the costs from each city are summed and compared. The calculated total costs to meet in each city are listed below.

Atlanta : $1,430
Boston : $820
Chicago : $690
Denver : $580
El Paso : $930

Therefore, using the first method, the most suitable meeting point would be *Denver*, with a total cost of $580.

The second method proves to be simpler. The only thing needed to be done is to compare the maximum costs of each city, which are listed below.

Atlanta : $530
Boston : $330
Chicago : $250
Denver : $290
El Paso : $410

The resulting optimal meeting point calculated using the

second method differs from the first one, as the cost from Atlanta to Denver is too expensive. Therefore, *Chicago* is the best choice if no one person should travel too expensively. The maximum cost needed to travel to Chicago is $250, which is the cost from Atlanta to Chicago.

VI. Conclusion

A modification to the classic Dijkstra's algorithm could be used to determine an optimal meeting point in a weighted directed graph. However, to use this modification, the graph must be a strongly connected graph, a precondition which does not exist in the classic Dijkstra's algorithm.

The modification involves checking the shortest path from *all vertices* instead of just one vertex for every pass. In code, a nested loop may be utilized to achieve this. In addition to this, a second loop similar to the first must be completed in order to complete the check after all vertices are visited.

The result of this modification is a set of data of all known shortest paths from each vertex to every other vertex, which can be used in a variety of schemes, not only to determine an optimal meeting point.

VII. Appendix

The following is a python code of the modified Dijkstra's algorithm used in this paper. The full code may be accessed from https://github.com/FelineJTD/Optimal-Meeting-Point-using-Dijkstras-Algorithm.

```
def modifiedDijkstra(vertices, start_vertex):
  # a function that requires a vertices (dictionary) and a start_vertex (string)
  # and returns cheapest_costs (dictionary) and previous_stopover_vertex (dictionary)
  # through modified Dijkstra's algorithm which calculates all cheapest routes
  # to every city from every city (or vice versa, depending on the vertices data)

  # variables to be returned
  cheapest_costs = {}
  previous_stopover_vertex = {}
  # variables to hold temporary states while Dijkstra runs
  unvisited_vertices = PriorityQueue() # a priority queue, to be used in initial loop
  revisit_vertices = [] # a queue, containing the same elements with the same order, to be used in secondary loop
  visited_vertices = {}

  # enqueue
  unvisited_vertices.insert(start_vertex)

  # initial loop
  while (not unvisited_vertices.isEmpty()):
    # initiation
    # dequeue
    curr_vertex = unvisited_vertices.delete()
    revisit_vertices.append(curr_vertex)

    cheapest_costs[curr_vertex] = {}
    previous_stopover_vertex[curr_vertex] = {}
    cheapest_costs[curr_vertex][curr_vertex] = 0
    visited_vertices[curr_vertex] = True

    for adj_vertex, price in vertices[curr_vertex].items(): # iterate each adj vertices
      # add vertex to queues if not yet visited
      if (adj_vertex not in visited_vertices):
        unvisited_vertices.insert(adj_vertex)

      for vv in visited_vertices.keys():
        try:
          # if there is a cheapest cost through curr_vertex
          price_through_current_vertex = cheapest_costs[vv][curr_vertex] + price
          try:
            # compare the prices if there is an existing one
            if(cheapest_costs[vv][adj_vertex] > price_through_current_vertex):
              cheapest_costs[vv][adj_vertex] = price_through_current_vertex
              previous_stopover_vertex[vv][adj_vertex] = curr_vertex
          except:
            # price not yet initialized
            cheapest_costs[vv][adj_vertex] = price_through_current_vertex
            previous_stopover_vertex[vv][adj_vertex] = curr_vertex
        except:
          pass

  # revisiting vertices
  # this secondary loop is important so that all vertices 'visits' all vertices, currently only the start_vertex has done so
```

```
  for p in range(len(revisit_vertices)):
    curr_vertex = revisit_vertices[p]
    for q in range(p+1,len(revisit_vertices)):
      vv = revisit_vertices[q]
      for adj_vertex, price in
vertices[revisit_vertices[p]].items(): #
iterate each adj vertices
        try:
          price_through_current_vertex =
cheapest_costs[vv][curr_vertex] + price
          try:
            if(cheapest_costs[vv][adj_vertex]
> price_through_current_vertex):
              cheapest_costs[vv][adj_vertex] =
price_through_current_vertex
              previous_stopover_vertex[vv][adj
_vertex] = curr_vertex
          except:
            cheapest_costs[vv][adj_vertex] =
price_through_current_vertex
            previous_stopover_vertex[vv][adj_v
ertex] = curr_vertex
        except:
          pass

  return(cheapest_costs,
previous_stopover_vertex)
```

VIII. ACKNOWLEDGMENT

The author is very grateful to Mr. Rinaldi Munir for his guidance and lessons throughout the semester, which insights have been invaluable to the completion of this project. Also, the author would like to thank a friend who has tested and reviewed the code for this paper.

REFERENCES

[1] Meeting and Convention Planners. (2009, December 17). U.S. Bureau of Labor Statistics. Retrieved April 21, 2010.

[2] narek Bojikian (https://cs.stackexchange.com/users/83933/narek-bojikian), Optimal meeting point, URL (version: 2018-07-08): https://cs.stackexchange.com/q/94024

[3] *Bender, Edward A.; Williamson, S. Gill (2010). Lists, Decisions and Graphs. With an Introduction to Probability*

[4] Weisstein, Eric W. "Weighted Graph." From *MathWorld*--A Wolfram Web Resource. https://mathworld.wolfram.com/WeightedGraph.html

[5] Munir, Rinaldi (2020). Graf (Bagian 1). https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

[6] Munir, Rinaldi (2020). Graf (Bagian 2). https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian2.pdf

[7] Richards, Hamilton. *"Edsger Wybe Dijkstra"*. A.M. Turing Award. Association for Computing Machinery. *Retrieved 16 October 2017*. At the Mathematical Centre a major project was building the ARMAC computer. For its official inauguration in 1956, Dijkstra devised a program to solve a problem interesting to a nontechnical audience: Given a network of roads connecting cities, what is the shortest route between two designated cities?"

[8] Frana, Phil (August 2010). *"An Interview with Edsger W. Dijkstra"*. Communications of the ACM. **53** (8): 41–47. *doi:10.1145/1787234.1787249*

[9] Dijkstra, E. W. *(1959).* "A note on two problems in connexion with graphs" (PDF). *Numerische Mathematik. **1**: 269–271.* doi:10.1007/BF01386390. S2CID 123284777

[10] Wengrow, Jay (2020). *A Common-Sense Guide to Data Structures and Algorithms, Second Edition: Level Up Your Core Programming Skills.*

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, 14 Desember 2020

Felicia Sutandijo
13520050