Validating Distance Matrix of a Weighted Tree

Aufar Gilbran – 13513015
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13513015@std.stei.itb.ac.id

***Abstract*—Today's technology and bussiness could leave people 10 years in the past dumbfounded. The high advancement rate of these two fields are driven by new knowledges discovered by fields in computer science. These knowledges helps menial things that must be done by humans to be automated processes. Unfortunately, people cannot discovered any knowledges without any data. People can extract knowledges from some set of data (or dataset) that was obtained from real events. But these dataset might not come from our works. They could be from other experiments, government, or statistical surveys. In this case, the format of the dataset might not come with the desired format. In this paper, we're going to see how to determine if a matrix is a distance matrix of a weighted tree given a distance matrix. This is useful because matrices is very common to represent data and a weighted tree will come in handy for data classifiers in data mining.**

***Index Terms*—Data Manipulation, Weighted Tree, Distance Matrix, Depth First Search**

I. Introduction - Why is it important?

Today's technology is very advanced compared to 10 years ago. We have smartphones, internets, social medias, and other things that could leave people 10 years in the past dumbfounded. But the interesting part is not how we have advanced in 10 years, but how the “rate of advancement” has advanced in 10 years. The advancement of technology from year 2005-2015 is much bigger than 1995-2005. Sure, 1995-2005 have a big advancement too (database technology, object-oriented languages). But it's not as big as 2005-2015 (Google has automated image descriptor, it can identify pizzas and other things in pictures!). Even so, the advancement from year 1995-2005 is still much bigger than 1985-1995. This is called as accelerating change[1]. Accelerating change is based on observation that technological advancement is increase throughout the history. But why is this happening? There are many factors in it, but one of the interesting factor is the quality of data.

Data is a value or maybe a set of values that represent a quality or quantity that is restated into individual pieces of information. Data is commonly measured, collected, or analyzed in an observation. There are numerous methods of data collections and it determines feasibility and quality of the data. For example, you can measure a metal's temperature just by it's radiation of heat. It's easy and cheap to do that but the quality is very poor. As an alternative you can use a thermometer to measure the temperature. It wasn't until 1638 when Robert Fludd created the first thermometer, so early on we could say that using a thermometer isn't feasible solution at all. But given a thermometer, you can measure a temperature with a very good accuracy. With the advancement of technology, people can measure data in better accuracy. Accordingly, this leads to a better knowledge quality. Sadly, even the most accurate data won't be helpful if we cannot do easy operations in it.

Matrices is the most common way to represents data. It provides a very descriptive information in human-readable format. Unfortunately, operations in matrices are very costly. Another thing to consider is how we process the data afterward. One of the most common computation methods to classification and regression of data is Random Forests. This method utilize data in tree format and then do computation on the tree. We want to have each of their advantages that is, it's easier for humans to work on matrices and it's easier for computers to work on trees. We can easily convert matrices to trees if we know that the matrices come from a tree. But what if we don't know where the matrices come from? Here we're going to see some of the ways to classify if any given matrix is a distance matrix of a weighted tree.

II. Definitions, Assumptions and Algorithms

Throughout this paper, we will always assume that there's no data loss in matrices. That is for a matrix *M*, M_{ij} is always have a quantifiable value. Also, since we're validating matrices

Throughout this paper, we will always assume that there's no data loss in matrices. That is for a matrix *M*, M_{ij} is always have a quantifiable value. Also, since we're validating matrices as distance matrices of weighted trees, we know that *M must be* a *square matrix*. That is *M* will always have same number of rows and columns.

As for variables name conventions, it will defined before each use. If there's no variable name definitions, *N* will correspond to the number of rows for matrix *M* (which means it's also the number of columns). *V* will correspond to the number of vertices in a graph. The value of *V* will always equal to *N*, but it has different semantics. *E* will correspond to the number of edges in a graph.

A. Definitions

This section will introduced the definitions that are need in order to comprehend this paper

A.1. Weighted Trees

A tree is a graph that contains no cycle in it. Formally, a tree is a graph $G = (V, E)$ such that there's no subset E' of E that could arrange a sequence of edges $e'_1, e'_2, \dots, e'_{k-1}, e'_k$ such that $e'_1=\{u, v\}, e'_2=\{v, w\}, \dots, e'_{k-1}=\{y, z\}, e'_k=\{z, u\}$. A weighted tree is a tree such that each edge in E is given a value.

A.2. Distance Matrices

A distance matrix is a matrix that each element of the matrix is measured by the distance of vertices in the corresponding row and column. Formally, a distance matrix is a matrix M such that $M_{ij}=dist(v_i, v_j)$ where v_i is the vertex identified with the number i. This means the following properties must hold for a matrix to come from a weighted tree:

1. $M_{ii}=0$
2. $M_{ij}=M_{ji}$
3. $M_{ik}+M_{kj}=M_{ij}$

These properties are *sufficient* and *necessary* conditions for a matrix to be a distance matrix. Thus we can based our proves from these properties (in fact we're going to abuse it).

B. Algorithms

This section will introduce the algorithms that are needed in order to comprehend this paper.

B.1. Greedy Algorithm[2]

Greedy Algorithm is an algorithm that always make locally optimal choice from a candidate set at each state to achieve the global optimum. The algorithm will never reconsiders the choices made so far and will continue until a complete solution is created. This algorithm is just a base introduction for the next algorithm, it will not be used in how to validate a distance matrix of a weighted tree.

In general, greedy algorithms have five components:

1. A candidate set, from which a choice is made to create the solution
2. A selection function, which chooses the best candidate to be added to the solution
3. A feasibility function, that is used to determine if a candidate can be used to create the solution
4. An objective function, which assigns a value to a solution, or a partial solution
5. A solution function, which will indicate when we have discovered a complete solution

Most of the time, Greedy Algorithm will produce good solutions for mathematical problems. Unfortunately, it's rarely produce good solutions for other type of problems.

For a Greedy Algorithm to succeed, the problem which the algorithm applied to need to satisfy the sufficient conditions. The most common conditions is:

1. Greedy Choice Property

The choice made by a greedy may not depend on future choices. It may depend on choices made so far.

2. Optimal Substructure

An optimal solution to the problem contains optimal solutions to the sub-problems.

These properties is a *sufficient* conditions but is not a *necessary* conditions. There are other conditions which can determine if the Greedy Algorithm will succeed or not. But none of them is proven to be *necessary* condition for the algorithm to get the desired result.

B.2. Kruskal's Algorithm[3]

Kruskal's Algorithm is used to construct a *minimum spanning tree* given a general connected graph. A *minimum spanning tree* is a tree with minimum sum of edges that consists of vertices from the original general graph. Formally, given connected graph $G = (V, E)$ we want to construct a graph $T = (V, E')$ where V is the set of vertices, E is the set of edges, E' is subset of E and total of the weights in E' is minimal.

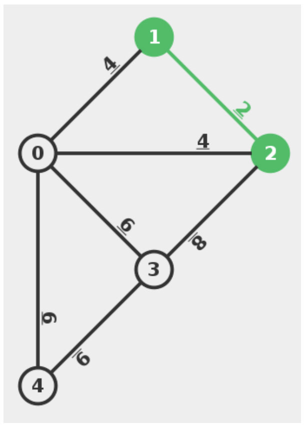
Kruskal's Algorithm is based on Greedy algorithm. The algorithm follows these steps:

1. Create a forest F such that each vertex is a tree and disjoint from other vertices. A forest is a set of trees.
2. Create a set S as candidate set.
3. Define a selection function which pick an edge with minimum weight in set S.
4. Define a feasible function that determine if an edge is appropriate to be added. An edge is appropriate if and only if it connects two different tree and combines it into a single tree.

While S is not empty and F is not yet spanning

5. Pick an edge with selection function
6. Determine if adding the edge is feasible to current solution using feasibility function.
7. If it is, add the edge to current solution and let objective function add value to new current solution.

To better illustrate the process, take a look at the following table:

	Here we pick the minimum edges $e = \{1,2\}$ with weight = 2. Since vertices 1 and 2 are on different tree, we can connect and combines them

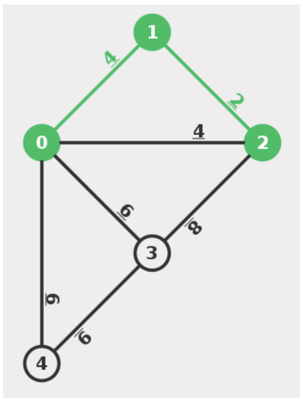
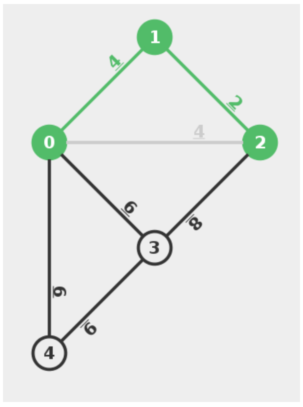
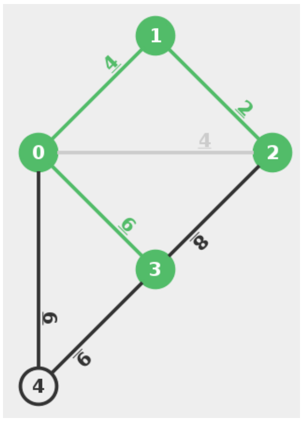
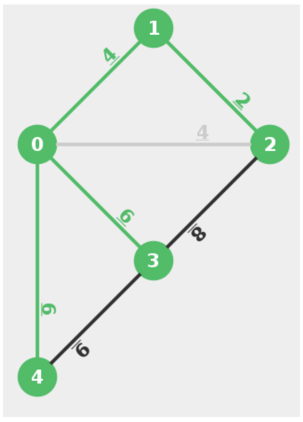
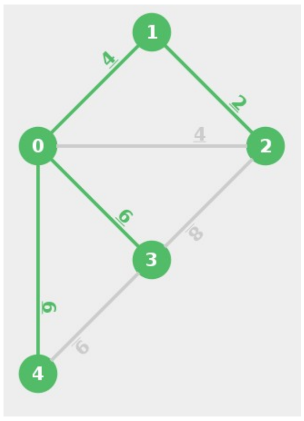
	This is the same with the last step. An alternative is we pick an edge $e = \{0, 2\}$. There can be many *minimum spanning trees* in a general graph. If we just want to find one, we can just pick whichever.
	In this step, since vertex 0 and 2 is already on the same tree, it won't be combined.
	Pick the minimum edge, that is $e = \{0, 3\}$. Since vertex 3 is not on the same as vertex 0, we can connect it and then combines it into a single tree.
	This is the same with the last step.
	The remaining edges won't be used since we already got our spanning tree.

Table 1. Kruskal's Algorithm Example

We can prove the correctness of this algorithm by proving the algorithm will produce spanning tree and by proving the algorithm will produce minimum total edges weight.

Let *T* is the result of this algorithm from connected graph *G*. *T* cannot have a cycle since feasible function in step 4 prevents it do so. *T* cannot be disconnected since he first encountered edge that joins two components of *T* would have been added by the algorithm. Thus *T* is spanning tree.

The pseudocode implementation of Kruskal's algorithm is as follows:

```
procedure KRUSKAL(G):
  T = ∅
  foreach v ∈ G.V
     MAKE-SET(v)
  foreach (u,v) order weight(u,v) increasing
    if FIND-SET(u) ≠ FIND-SET(v) then
      T = T ∪ {(u, v)}
      UNION(u, v)
  return T
```

If implemented correctly, Kruskal's Algorithm can run in $O(E \log V)$ time with simple data structures. In this paper, we're going to apply Kruskal's Algorithm for graphs with an edge for each pair of nodes. That is, $2|E| = |V| * (|V| - 1)$. Thus, we can say Kruskal's Algorithm run in $O(V^2 \log V)$.

B.3. Depth-First Search[4]

Depth-First Search is an algorithm for traversing or searching a particular vertex in graphs. The algorithm work as follow:

1. Start at any vertices. We call the start vertices as *root*.
2. From current vertex *v*, find all vertices adjacent vertices *S*.

While not all vertices in *S* is processed

3. From *S* pick any vertex *u* that is not labeled as discovered yet.
4. Recursively do this algorithm from step 2 with current node set as *u*.

To illustrate the process, take a look at the following table:

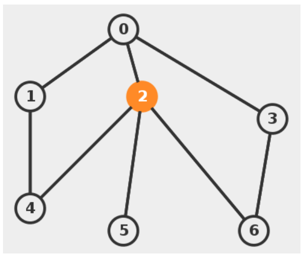
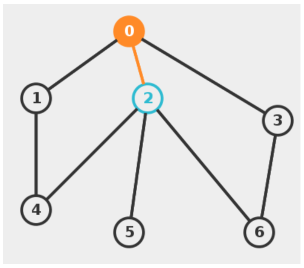
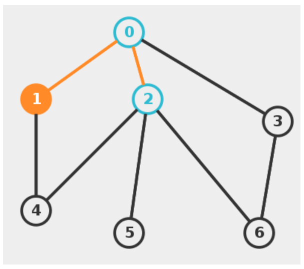
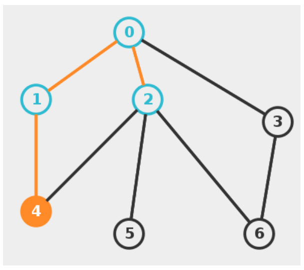
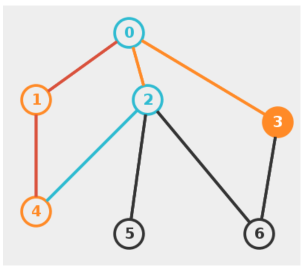
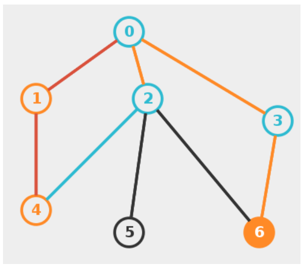
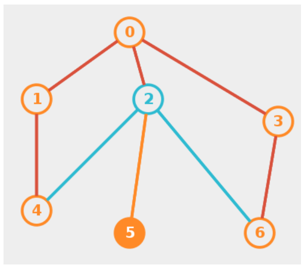
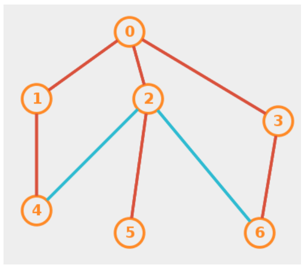
	First we start at root node, that is in this case node 2. We call recursive call to each of it's children. The next call is node 0.
	Since node 0 is not yet flagged, we go to it then do the same thing.
	Do the same thing as before, but this time from node 1.
	Now, when node 4 wants to call node 2, it is already flagged so it won't go over there. Since there is no more children of node 4, it will just backtrack to 1, and 1 will backtrack to 0.
	Now zero call Depth-First Search on it's remaining childs. That is, node 3.
	Node 6 has the same case as node 4. It wants to call Depth-First Search on Node 2 but it is already flagged. Since there's no more children, it will backtrack to node 3, and then to node 0, and then to node 2.
	We're currently on the node 5 after the call for node 2's last child. Since there's no more expansion from node 5, we're backtracking
	We're back at the root node and there's no more child to expand. We've completed the Depth-First Search.

Table 2. Depth-First Search Example

The pseudocode implementation of Depth-First Search algorithm is as follows:

```
procedure DFS(v):
  label v as discovered
    for all edges from v to w in v.neighbors()
      if w is not labeled as discovered then
        recursive call DFS(w)
```

The complexity of Depth-First Search is determined by the purpose of the algorithm. In this paper, we'll use it as a traversing algorithm not as searching algorithm. Thus, the complexity of Depth-First Search is $O(V)$.

III. Validation - The Naive Method

According to chapter II, we must satisfy the three properties to validate a matrix. The first two properties are trivial to be checked. We can just iterate all the elements in $O(N^2)$. Note that we cannot have a better time complexity than $O(N^2)$ for validation because at the very least we need to read the matrix, which has N^2 elements.

These steps are used to check if the 1st property satisfied:

1. Iterate from 1 to N.

For each iteration on a variable i

2. Check if $M_{ii}=0$. If it's true, go to the next iteration or else we know that the matrix is not a distance matrix.

This algorithm runs in O(N) time. Here's a pseudocode implementation for this algorithm:

```
function CheckSelf(M):
  for all number from 1 to N
    if $M_{ii} \neq 0$ then
      return false
    return true
```

The next step is to check if the 2nd property is satisfied. We check for each ordered pair of vertices, and then decide if the corresponding element is equal to the corresponding element if the order of the pair is changed. If it is not equal, then it's not a distance matrix, since the 2nd property of a distance matrix is violated. The algorithm runs as follows:

1. Iterate for each pair of *(i, j)* where $1 \le i, j \le N$

For each iteration on pair (i, j)

2. Check if $M_{ij}=M_{ji}$. If it's true, go to the next iteration or else we know that the matrix is not a distance matrix

This algorithm runs in $O(N^2)$ time. Here's the pseudocode implementation of the algorithm.

```
function CheckTranspose(M):
  for all pair (i, j) where 1 ≤ i,j ≤ N
    if $M_{ij} \neq M_{ji}$ then
      return false
  return true
```

There is some problem with this solution. That is in most computer system, this algorithm has high overhead for having a poor cache-hit rate. Unfortunately, the only way to have better cache-hit rate is to aggregate the transposed matrix so M_{ij} is right before M_{ij} or vice versa. That is, the lower triangle of matrix is transposed and each element on the same row column of upper triangle and transposed lower triangle is put adjacent. This method is much more harder to implement with low speed gain. Consider this approach if we need to validate frequently. Checking the 2nd property of distance matrix can also be done while reading although it doesn't reduce anything but length of code.

Our last job is to determine the 3rd property is satisfied. The naive idea to do this is to check the transitivity of the matrix value. In latter chapter however, we will see how to check if the 3rd property is satisfied without checking all the transitivity pair. As for now, we should work on how to check the transitivity in the matrix.

From the 3rd property definition, we can spontantenously come up with the naive idea as follows:

1. Iterate for each pair of *(i, j, k)* where $1 \le i, j, k \le N$

For each iteration on pair (i, j)

2. Check if $M_{ik}+M_{kj}=M_{ij}$. If it's true, go to the next iteration or else we know that the matrix is not a distance matrix

The algorithm above runs in $O(N^3)$ time. Here's a pseudocode implementation for this algorithm:

```
function CheckTransitive(M):
  for all pair (i, j, k) where 1 ≤ i, j, k ≤ N
    if $M_{ik}+M_{kj}=M_{ij}$ then
      return false
  return true
```

This algorithm suffer the same poor cache hit-rate as with the algorithm to check if 2nd property is satisfied. To make it worse, this algorithm has much lower cache hit-rate because there's three tuple in it, each may be far away from the others.

As we can see, all the algorithms above is easy to implement. In fact, they are very similar to each other. One thing to note is that this naive algorithm has bad complexity and poor cache hit-rate. So in a big dataset, this approach shouldn't even be considered. Next, we will see how to improve the validation algorithm with some observations and Greedy Algorithm, or to be specific, the Kruskal's Algorithm.

IV. Validation - The Improved Method

We have seen how to validate if a matrix is a distance matrix of a weighted tree with little to no effort. The naive method has already given us a correct solution for this problem. But the problem is the naive method is very slow, and we want computations to be fast.

From the complexities of each algorithm to check the corresponding property, it's obvious that the 3rd property is the slowest to be checked. The 1st and 2nd property cannot be improved further (or maybe we can, but any improvement on these algorithms will not give us much). This is because the 1st and 2nd property is checked by algorithm with complexity lower or equal to $O(N^2)$ *without calculating the overhead.* Since the overhead is constant, we will not discuss it in this paper. Thus, we only need to be concerned on how to improve on algorithm for checking if the 3rd property is satisfied.

To improve the algorithm, we need to observe some facts about distance matrix. These are the following facts needed for improving the algorithm:

1. The minimum edge must be part of the weighted tree.
2. The k-th minimum edge must be included unless there's already a path from each end points of the edge.

These facts are pretty much describing a rule for Kruskal's Algorithm. Thus the solution to the 3rd property checking is build a *Minimum Spanning Tree* from the matrix. But the fact is given, we need to verify it by proving it's corectness.

Suppose that, in a specific stage of Kruskal's Algorithm, there's a forest *F.* Kruskal's Algorithm would pick an edge e_i which has the minimum edge on the current set of edges *E'*. Let w_i is the weight of edge e_i. Suppose that, instead of picking e_i, we pick e_j such that $w_i < w_j$ and discard w_i in the process. That means the vertices on the endpoint of edge e_i will have a distance $d \ge w_j$ on the tree we're creating and holds $w_i < w_j \le d$. This is violating the 3rd rule because the d should be equal to w_i given by $M_{ii}+M_{ij}=M_{ij}=w_i=d$. Thus, by contradiction, it is proved that the original tree must be one of the *Minimum Spanning Trees* of the matrix.

But we have only proved the correctness of a *necessary* conditions, it can still be a wrong tree afterall since the *Minimum Spanning Tree* is only proven correct for adjacent vertices. Fortunately, after Kruskal's Algorithm, we already have a tree. Thus, to calculate the distance d between vertex v_i and other vertices, we can do a Depth-First Search starting at vertex v_i as the root node. We need an additional parameter to keep track of the distance from v_i to the current vertex. We can do this because the structure of a tree make sure that there's only one path for a pair of vertice, unlike the structure of general graph which may contains many path. If the path contains no contradiction after we do Depth-First Search on every vertex, then the matrices satisfy the 3rd propery of a distance matrix.

The time needed to do computation is $O(N)+O(N^2)+O(N \log N)+O(N)$ since $V=N$. Thus, we can determine if a matrix is a distance matrix of a weighted tree with $O(N \log N)$ time.

There are other methods to check the 3rd property with less overheads. The advantage of this algorithm over others is we can get the tree constructed without further algorithm. Also note that we can do this with Prim's Algorithm instead of Kruskal's Algorithm, because what we need is to construct the *minimum spanning tree* and both them will do just fine.

V. Conclusions

Data manipulation is crucial for serving data in the desired format. Since data comes frequently and abundant, we need to process data faster. To do so, we need to make a method to compute matrix faster. We can actually validate wether or not a matrix is a distance matrix or not. Once we know that it is a distance matrix, we can construct the tree and use calculating method based on trees such as Random Classifiers. One such way to validate and construct is to use Kruskal's/Prim's Algorithm

Apparently, there's many other alternatives such as *Sparse Table for Lesser Common Ancestor, Floyd-Warshall Algorithm,* and many others. Some advantages of using Kruskal's/Prim's Algorithm is that we get the constructed tree with fast enough computation time with a single run and it's very easy to implement the algorithm.

VII. Acknowledgment

I thank Allah S.W.T. for health and strength so I can finish this paper, Mr. Rinaldi Munir and Mrs. Nur Ulfa Maulidevi for their teaching in IF2211 course during this semester, and also my friends and fellow students in Informatics Institut Teknologi Bandung for their supports and assistance.

References

[1] Kurzweil. Ray, “The Singularity Is Near: When Humans Transcend Biology”, London: Penguin Books, 2006.
[2] Cormen et al, “Introduction to Algorithms”, 3rd ed. Cambridge, MA: MIT, 2009, ch.16.
[3] Cormen et al, “Introduction to Algorithms”, 3rd ed. Cambridge, MA: MIT, 2009, ch.23.
[4] Cormen et al, “Introduction to Algorithms”, 3rd ed. Cambridge, MA: MIT, 2009, pg. 603-612.

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, 5 Mei 2015

[illegible]

Aufar Gilbran - 13513015