Application of Minimum Spanning Tree in Edge Detection

Aditya Putra Santosa - 13517013
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
adityaputrasantosa@itb.ac.id

***Abstract*—Computer vision is a terminology used when a computer extract meaningful information from an image, such as what object is in there or how many car in in the image. One of the important process of computer vision is edge detection, where the image is transformed into just the boundary of object that is in the image. In this paper, we will discuss the application of Minimum Spanning Tree in edge detection. We will use C++ and OpenCV to demonstrate the edge detection. OpenCV is a library that is typically used for image processing, but in here we will use it only as a way to convert color, access pixel data of an image and show it to the user.**

***Keywords*—computer vision, edge detection, graph, minimum spanning tree.**

I. Introduction

With the increasing development of technology that is in the area of computer graphic and also in artificial intelligence, computer vision is becoming a trending topic now. Many thing have used computer vision as it's core, such as in speed camera, instagram filter, self-driving car, face recognition, google image search, and many more. All of those things have one thing in common, they all used Object Detection.

It is easy for us human to detect and distinguish an object in an image, but for a computer, an image is just a bunch of number that represent a bunch of color. Computer may use algorithm such as machine learning to distinguish an object but there is also another way for computer to detect an object, that is by checking the object shape and match it with all the shape in the image. By using edge detection, the image will be transformed into a bunch of object shape that is visible in the image. For example, a ball will become a circle, or a half circle if the ball is only half visible in the image.

There are many ways to do edge detection, such as using Canny Edge Detector, Sobel Filter, Prewitt, or even by using Minimum Spanning Tree. The advantage of using Minimum Spanning Tree is it's easier to implement and easier to explain. We just need to find all the potential pixel that may be an edge and insert it into the graph,then we can use Kruskall Algorithm (O(E log V) in general) to find the path that connect all the potential pixel with minimum distance between each neighboring pixel.

II. Graph, Tree, Minimum Spanning Tree

A. Graph

Graph is a pair of set (V,E) where V stand for non-empty set of vertices and E stand for set of edge that connect pair of vertices [1]. Two vertices is said to be adjacent iff there is at least one edge connecting both of them.

Graph can be classified by it's properties. A graph that doesn't have any edge that connect vertex *V* to *V* (loop) or multiple edge that connect two vertices is called *simple graph*, otherwise it is called *unsimple-graph*. Graph whose edge doesn't have any direction is called *undirected graph*, otherwise it is called *directed graph*. In a *directed graph* the edges have start point and endpoint that show the source and destination of the edges. One cannot simply traverse from the destination to source in *directed graph* unless there is a path / edge from destination to source. Another properties of graph is weight. Graph that have weight on their edges is called *weighted graph*, otherwise it is called *unweighted graph*. The weight in edge is use to convey information about the two vertices that is connected by the edge. The graph that we will use later in this paper is *simple, undirected, and weighted* graph.

Graph, like other data structure has representation that will be used to describe the graph in program. Such representation is adjacency list, adjacency matrix, and list of edges and vertices.

Adjacency list is a way to describe a graph by storing information of every vertex *u* that is adjacent to vertex *v* in *v*'s adjacency list. A vertex *a* and *b* is said to be adjacent iff there is *b* in *a*'s adjacency list, if the graph is undirected then *a* also have to be in *b*'s adjacency list, otherwise it is possible for *a* not to be in *b*'s adjacency list, it implies that there is a edge from *a* to *b*, but no edge from *b* to *a*. In a weighted graph, the adjacency list will be filled with tuple(*u*, *w*) where *u* is a vertex and *w* is the weight of the edge.

Adjacency matrix is a two-dimensional array (matrix) that have same number of row and column that is the number of vertices in the graph. Vertex *v* and *u* is said to be adjacent iff $M_{i,j}$ is not 0, where M is the adjacency matrix and *i,j* is the id of vertex *v* and *u* respectively. In an *unweighted graph* we can simply put 1 in the cell if there are edge between two vertex and 0 otherwise. In a weighted graph, we have to assign the weight of the edge as the value of the cell and 0 if there are no edge connecting both vertex.

List of edges and vertices is a way to describe a graph by using two list E and V where E contain all the edge in the graph and V contain all the vertices in the graph. The edge in E is shown as a 3-tuple(u, v, w) where u and v show the vertex of the edge and w is the weight. If the graph is *unweighted graph* we can simply set all the weight of the edges to be the same. Two vertex u and v is said to be adjacent iff there is a tuple (u, v, w) in E. In this paper, we will use list of edges and vertices as a way to describe the graph in the program.

Graph $G' = (V', E')$ is said to be a sub-graph of $G = (V, E)$ iff V' is subset of V and E' is subset of E . Graph $G'' = (V'',E'')$ is said to be complement of G' iff $V'' = V - V'$ and $E'' = E - E'$ [1].

Subgraph $G' = (V', E')$ is said to be a spanning subgraph of $G = (V, E)$ iff $V' = V$ [1]. This property is important in this paper, as the graph that we will produces is a spanning subgraph.

Another terminology for graph is path. Path that has length n from v_0 to v_n is a sequence of vertex and edge $(v_0, e_1, v_1, e_2, \dots, v_{n-1}, e_n, v_n)$ such that $e_n = (v_{n-1}, v_n)$ is an edge in graph [1]. Two vertex is said to be connected if there is a path between them.

Cycle and circuit is a path that end in the same vertex as the start [1]. A graph that doesn't have any cycle / circuit is called a *Tree* which we will discuss after this.

B. Tree

Tree is simply a connected undirected graph that doesn't have any cycle / circuit [2]. By that definition we can see that every vertex had a path to another vertex except itself.

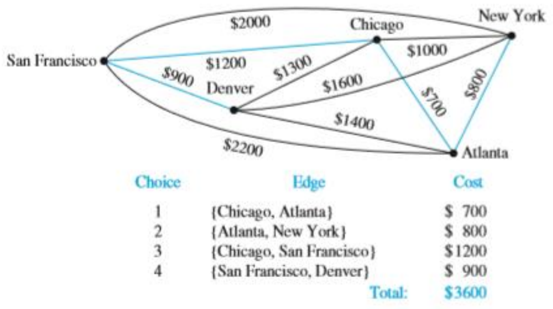
A graph that have multiple trees inside of them are called a forest. We can always make a forest from a tree just by removing one or more edges.

In the practice, we usually defined a vertex to be the root of the tree. By doing this, we have create a rooted tree, that is a type of tree that have a root and all other root as it's child / descendants. Child of u is defined to be all vertex v that have an edge from u to v. Descendant of u is all vertex v that has a path from u to v. Inversely, we can say that v is parent of u iff there is an edge from v to u and v is ancestor of u if there is a path from v to u. Vertex that doesn't have any child is called the leaf of the tree.

Tree is usually classified by number of child the vertex can have. By their number of child, a rooted tree is called an m-ary tree if every vertex have at most m child [2]. The special case of m-ary tree is when m=2, it is called a *binary tree*.

C. Minimum Spanning Tree

Minimum Spanning Tree is a special kind of tree that contain all vertex in the graph and have smallest possible sum of edge's weight [2]. Minimum Spanning Tree has a wide area of application such as clustering, network design, computer vision, road planning, and many more. We can use algorithm such as Kruskal's or Prim's.

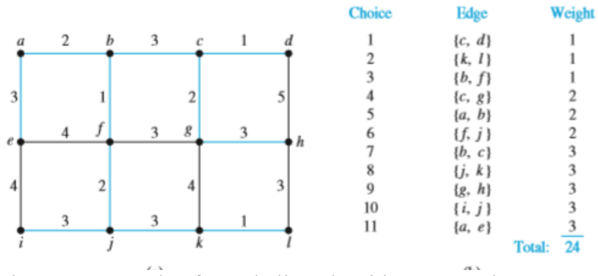


Choice	Edge	Cost
1	{Chicago, Atlanta}	$ 700
2	{Atlanta, New York}	$ 800
3	{Chicago, San Francisco}	$1200
4	{San Francisco, Denver}	$ 900
	Total:	$3600

Figure 1. Example of minimum spanning tree, picture from Discrete Mathematics and Its Applications, 7th Ed. – Rosen

Kruskal's algorithm is a greedy algorithm, that is it always choose available edge with smallest weight and add it to the resulting tree. We can implement Kruskal's algorithm by using Disjoint Set Union(DSU). Kruskal's algorithm start by choosing an edge in graph with the minimum weight. Let u dan v be the vertex of the edge. If u and v is not connected in the tree and if the addition of the edge will not cause a circuit to appear, then add the edge to the tree, otherwise discard the edge. The pseudocode is as follow.

```
DSU d(numberOfVertices)
Graph result
sort all edges in graph by weight
for all edge E in graph:
     u = E.startVertex
     v = E.endVertex
     If(d.unite(u, v)) then:
            Add E to result
return result
```



Choice	Edge	Weight
1	{c, d}	1
2	{k, l}	1
3	{b, f}	1
4	{c, g}	2
5	{a, b}	2
6	{f, j}	2
7	{b, c}	3
8	{j, k}	3
9	{g, h}	3
10	{i, j}	3
11	{a, e}	3
	Total:	24

Figure 2. Example of Kruskall's Algorithm on graph. Image taken from Discrete Mathematics and Its Applications, 7th Ed. – Rosen

The complexity of the sorting is O(E log E) but it can be said that the sorting run in O(E log V) because in a tree E is at most V^2, so O(E log E) = O(E log V^2) = O(E. 2 log V) = O(E log V). The process unite / union inside the for loop is O(log V) at most because of the path compression algorithm. So the for loop run in O(E log V) time. In total the complexity of Kruskal's algorithm is O(2.E log V) which is O(E log V).

As for the Disjoint Set Union, we treat the set as a tree such that two subsets is said to be disjoint if their ancestor is different. The algorithm to find the ancestor is path compression algorithm that run in O(log V) time. Path compression algorithm work recursively, it exploits the property of tree such that if a is parent of b and b is parent of c then, a is ancestor of c. We just need to find the parent of the parent of the parent… until we reach a vertex that doesn't have any parent, that vertex is the ancestor of the vertex that we search. Below is the pseudo code for the DSU.

```
  DSU(int size):
       for i in [0..size]:
              parent[i] = i
              rank[i] = 0

  int getAncestor(int x):
       if(parent[x] <> x):
         parent[x] = getAncestor(parent[x])
       return parent[x]

  boolean unite(int x, int y):
       x = getAncestor(x)
       y = getAncestor(y)
       if(x <> y):
              //Check the rank, the smaller
rank will become the child
              if(rank[x] < rank[y]):
                     rank[y] += rank[x]
                     parent[x] = y
              else:
                     rank[x] += rank[y]
                     parent[y] = x
              return true
       else:
              //Failed, x and y is not disjoint
              return false
```

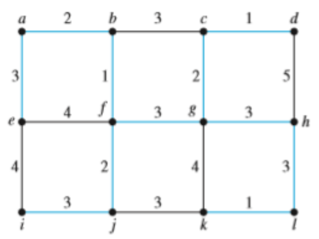
In the implementation of Kruskal's algorithm, it is easier to use the list of edges and vertices as the representation of the graph because the need to sort the edges.

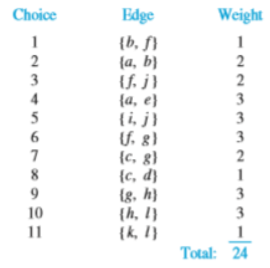
Prim's algorithm is also a greedy algorithm, it has similarity with Kruskal's, such as both choose the smallest weight. The only difference is, while Kruskal's choose the edge freely, Prim's algorithm only choose edge that is incident with the vertex in the minimum spanning tree. Prim's algorithm starts by choosing random starting vertex, flag is as not available, insert it into the tree and then insert every edges that is incident with the vertex into a priority queue. Then, while the priority queue is not empty, take the top edges and insert it into the tree and also flag the endpoint of the edges as not available. Repeat until all vertices is not available. Below is the pseudocode.

```
  priority_queue pq
  boolean taken[numberOfVertices] = true for
all vertices
  Graph result
  //We push tuple of int (weight, vertex)
  pq.push(0,0)
  taken [E[0].start] = false

  while pq is not empty:
       t = pq.top
       pq.pop()
       for every edge incident with t.vertex:
              if(taken[edge.end]):
                taken[edge.end] = false
                add edge to result
                pq.push(edge.weight, edge.end)

  return result
```





Choice	Edge	Weight
1	$\{b, f\}$	1
2	$\{a, b\}$	2
3	$\{f, j\}$	2
4	$\{a, e\}$	3
5	$\{i, j\}$	3
6	$\{f, g\}$	3
7	$\{c, g\}$	2
8	$\{c, d\}$	1
9	$\{g, h\}$	3
10	$\{h, l\}$	3
11	$\{k, l\}$	1
	Total:	24

Figure 3. Example of Prim's Algorithm on graph. Image taken from Discrete Mathematics and Its Applications, 7th Ed. – Rosen

Same with the Kruskal's algorithm, Prim's algorithm run in O(E log V) time. Because the need of incident edge, Prim's algorithm usually use adjacency list as a way to represent the graph.

In this paper we will use Kruskal's algorithm to find the minimum spanning tree because it is easier to implement.

III. Edge Detection

A. Definition

Edge detection is a technique in image processing to find the boundary of an object in image by detecting significant change or discontinuity of brightness / color [3]. In the practice, people usually used kernel convolution to detect the change in brightness/color. Convolution in image processing is defined as a adding each pixel with it's neighbor with the weight defined by the kernel. Kernel is matrices of number that represent the weight of each pixel when we are doing convolution. In edge detection there are many kernel that we can choose, such as Sobel, Prewitt, Laplace, and many other. Other than single convolution there are also another method such as Canny Edge detector and Gabor Filter. But we will not discuss such method in this paper, instead we will use graph and minimum spanning tree to find the edge of an object.

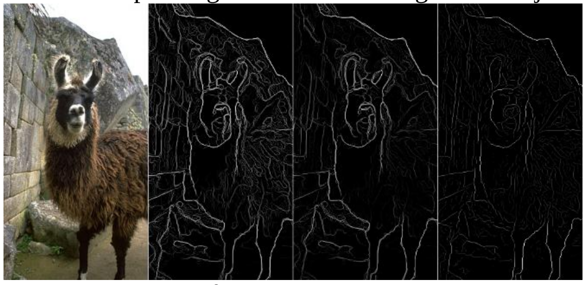


Figure 4. Example of edge detection using Canny Edge detector. Image taken from http://cs.brown.edu/courses/cs143/2011/proj2/ (access on 9/12/2018)

B. Minimum Spanning Tree Approach

By the definition of edge detection, an edge is collection of pixel that have significant change from their neighbor. We will first find all of such pixel in the image and then assign it as the vertex of the graph. After finding all potential pixel, we will need to assign edge between all of the vertex, with the manhattan distance between pixel as it's weight. After that, we just need to find the minimum spanning tree of the graph and that will become the edge of all object in the image. Because we assign edge between all vertices, the graph is a *complete graph*, a graph in which each pair of vertices is connected by an edge. However, it is easy to notice that by doing so there will be a line connecting two different object which we don't want. We can solve this problem by only assigning an edge to another pixel if it is less than certain amount of distance from the source, we will call this maximum distance as *distance threshold* or *distThres* for short. By doing so, not only we solve the problem of line connecting two different object, we also improve the running time of the program. This is because the complexity of the program is O(E log V) and by assigning a threshold to the edge, the number of edge is decreased from the initial, so E will be smaller and if E is smaller, the running time is faster than the initial.

C. Implementation

To implement the program, we will first need to detect all the potential pixel, that is all pixel that have significant change in brightness/color from their neighbor. We will use the CIELAB color space, defined by three component in the pixel

$$0 \le L \le 100, -127 \le a \le 127, -127 \le b \le 127.$$

We use CIELAB because it is uniform with our eyes color vision, that is two color that's different in CIELAB color space will be different to our eyes. Two color that is the same in CIELAB color space will be the same too in our eyes. Because we use OpenCV in the implementation of the program, and OpenCV change the pixel intensity range from normal CIELAB pixel range, we need to convert OpenCV pixel intensity to normal CIELAB pixel range. According to OpenCV documentation, for 8-bit image

$$L \leftarrow L * \frac{255}{100}, a \leftarrow a + 127, b \leftarrow b + 127 \text{ [4]}$$

We can get the normal CIELAB pixel range from OpenCV pixel range by using this formula.

$$L \leftarrow L * \frac{100}{255}, a \leftarrow a - 127, b \leftarrow b - 127$$

The formula for distance/similarity between two color in CIELAB color space that we use is the Delta E 76 version that is defined as follow.

$$distance = \sqrt{(L_1 - L_2)^2 + (a_1 - a_2)^2 + (b_1 - b_2)^2}$$

A distance of 0 means that those two color are the same. Now we just need to find all pixel that is a certain amount of distance from it's neighbor. We will call the minimum color distance by *epsilon*. The pseudocode is as follow.

```
double distance(Vertex u, Vertex v):
        double L1 = u.L * 100/255
        double a1 = u.a – 128
        double b1 = u.b – 128
        double L2 = v.L * 100/255
        double a2 = v.a – 128
        double b2 = v.b – 128
        return sqrt(sqr(L1-L2)+sqr(a1-
a2)+sqr(b1-b2))

void findPotentialPixel():
        convert image to CIELAB
        for i in [0..image.rows]:
            for j in [0..image.cols]:
               if(distance(pixel(i, j),neighbor) >
                 epsilon:
                  add (i,j) to graph
```

after finding all the potential pixel, we will need to assign edge between each of them. By the idea before, we will assign an edge iff the manhattan distance between a pair of pixel is less than *thresDist*. The pseudocode is as follow.

```
double mDist(Vertex u, Vertex v):
        return abs(u.x - v.x) + abs(u.y - v.y)

void assignEdges():
for i in [0..numberOfVertices]:
        for j in [i+1..numberOfVertices]:
                Vertex V1 = V[i]
                Vertex V2 = V[j]
                if(mDist(V1,V2) < thresDist):
                   add edge(V1,V2, mDist (V1,V2))
```

the complexity of assigning edges is O(V^2) .After that, we just need to find the minimum spanning tree of the graph using Kruskal's algorithm and then create an image showing the graph. The resulting image will show the edges of object in the source image. Below is the example of source image and the resulting image after doing the algorithm.

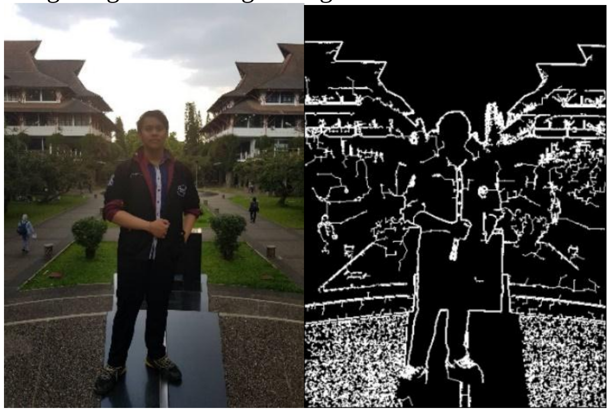


Figure 5. Example of before (left) and after (right) edge detection with *epsilon 20* and *thresDest* 30. Photo taken by author in Plaza Widya Nusantara ITB.

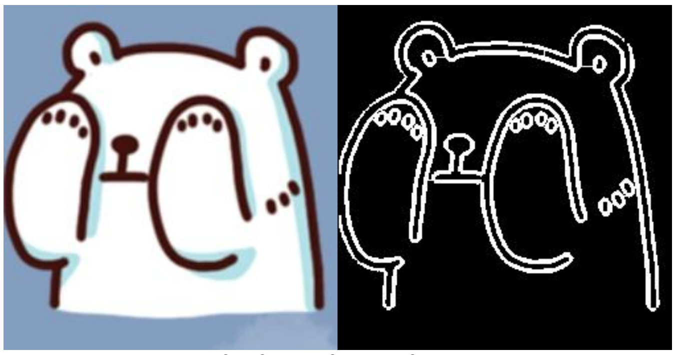


Figure 6. Example of before (left) and after (right) edge detection with *epsilon* 20 and *thresDest* 20. Screenshot taken from author's chat on line. Bac-Bac Sticker is courtesy of Daryl Cheung

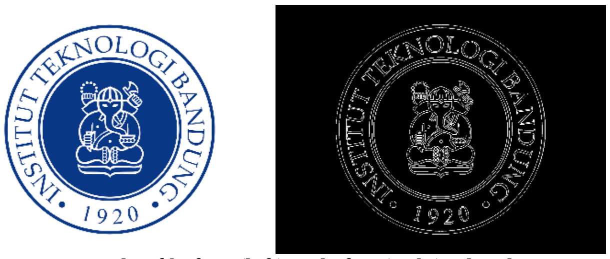


Figure 7. Example of before (left) and after (right) edge detection with *epsilon 20* and *thresDest* 20. Bandung Institute of Technology logo taken from https://gudrilogo.blogspot.com/2017/11/logo-institut-teknologi-bandung-itb.html (access on 8/12/2018)

It can be seen from the example that the algorithm work quite well with simple image (image that contain a few object and little to no noise). With image that have many kind of object and noise the result will be like figure 1. The total running time of the program is quite varied. With the first image in figure 1, the program run about 1.5 minute, that is because of the image resolution (1108 x 1477) so the program have to process about 1.600.000 pixel and about 80.000.000 edges. With the second image, the running time is about 20 seconds, that is because of the resolution (253 x 261) so the program will process less pixel than the first one, about 66.000 pixel and 5.000.000 edges. The third image run about 30 second, faster than the first image even though it has bigger resolution (1600x1200), that is because most of the pixel is white and monotone so the program can skip those pixel, the edges for the third image is about 20.000.000 edges. By experimenting with various number of *epsilon* and *thresDest*, we have found 20 for *epsilon* and *30* for *thresDest* is enough for most image.

The program running time can be further improved by utilizing Graphical Processing Unit (GPU) such as Nvidia Graphic Card by using OpenCV CUDA module. The author choose not to do this because of the sheer complexity of the code and the limited time available to do the research and actual coding of the program. The running time of the algorithm also can be improved by neglecting the direct neighbor of the pixel that we are currently checking. We can implement this by doing a modification on the findPotentialPixel function, on the loop we increment *i* and *j* by 2 or another number bigger than 1. By doing this, we decrease the number of the vertices, thus decrease the number of the edges, which in turn decrease the complexity of the program. The downside of this method is the accuracy of the program decreased, some edges may not be detected in comparison when we increment *i* and *j* by 1.

IV. Application Of Edge Detection

Edge detection has many application in image processing and computer vision, that is because it is easier to compare object shape (it's edges) rather that comparing all of the pixel in the image. Edge detection also minimized the information shown in an image, from complete information (shape, color, texure, and many more) into just shape / edges of the object. One such example where edge detection is important is in humanoid soccer robot. The robot need to differentiate between field line, soccer ball, goal post, and other object. To detect the goal post, the robot can just do edge detection on it's vision and find all vertical line in it's sight.

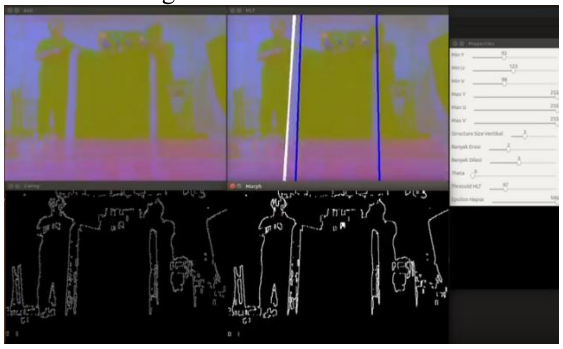


Figure 8. Goal Post detection using edge detection. The white and blue line show the position of goal post. Screenshot taken from author's program in Dago Hoogeschool (ITB Humanoid Soccer Robotic Team for KRSBI-Humanoid)

In figure 4 above, the robot use edge detection, but not using minimum spanning tree approach, the robot use canny edge detector approach. Thit is because of the long running time when we are using minimum spanning tree approach. The minimum spanning tree approach is terrible when speed is more important than quality, so it cannot be used to do edge detection in video, but it can be used against still image.

Another application for edge detection other than object detection is counting the number of object in an image. We can use edge detection to find the boundary of the shape of all object and then calculate how much connected line in there.

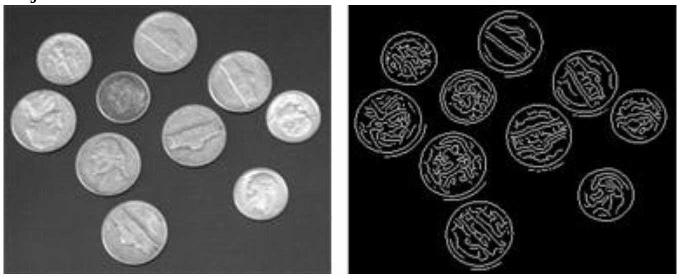


Figure 9. Bunch of coin and it's edge after being processed by edge detection algorithm. Image taken from https://www.mathworks.com/content/mathworks/www/en/discovery/edge-detection/jcr:content/mainParsys/image_1.adapt.full.high.jpg/1541581516580.jpg (accessed 9/12/2018)

From figure 9 we can count how many coins in there simply by counting the number of circle found in the processed image.

V. Conclusion

Edge detection is an important process in image processing and computer vision. It has many application such as object detection, preprocessing for advanced computer vision, image recognition and many more. In this paper, we have use minimum spanning tree approach to implement edge detection on still image. The total time complexity of the program is O(V^2 + E log V) which is quite slow to used against video, so another approach like Canny / Gabor is recommended to use in video. But the running time is quite good on still image, considering it's result quality which find most edges whether the image is noised / not.

VI. Appendix

Author’s implementation of the algorithm that is discussed in this paper can be found on author’s github : https://github.com/AdityaPutraS/Deteksi-Tepi-MST

The program require C++ and OpenCV to compile. The link to install and use OpenCV can be accessed on https://opencv.org/ . Follow the readme.md inside author’s github repository to compile the program and run it. Samples and samples result are already available in the repository.

VII. Acknowledgement

The author would like to express it’s thanks to Dr. Ir. Rinaldi Munir, MT. as lecturer for Discrete Mathematic IF1210 for author’s class. The author would also thanks the contributors to OpenCV library which helped the author in implementing the algorithm. The author would also like to express it’s thanks to Dago Hoogeschool (ITB Humanoid Soccer Robotic Team for KRSBI-H) for the inspiration and motivation for this paper.

References

[1] Rinaldi Munir, *Matematika Diskrit”*, 6th ed. Bandung: Informatika Bandung, 2016,ch. 8.

[2] K.H. Rosen, *Discrete Mathematics and Its Application*, 7th ed. New York: McGraw-Hill, 2012, ch.10.

[3] Priyanto Hidayatullah, *Pengolahana Citra Digital Teori dan Aplikasi Nyata*, 1st ed. Bandung: Informatika Bandung, 2017, ch 10.

[4] OpenCV Documentations. (2015). Retrieved from https://docs.opencv.org/3.1.0/de/d25/imgproc_color_conversions.html

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, 9 Desember 2018

[illegible]

Aditya Putra Santosa - 13517013