Analisis Kompleksitas Algoritma Kruskal untuk Menentukan *Minimum Spanning Tree*

Fabian Radenta Bangun - 13522105[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13522105@std.stei.itb.ac.id

***Abstract*—Algoritma Kruskal adalah salah satu algoritma yang dapat memecahkan permasalahan pencarian pohon merentang minimum (*minimum spanning tree*) dari suatu graf berbobot. Pada makalah ini dikaji analisis kompleksitas algoritma Kruskal secara kompleksitas waktu. Hasil analisis menunjukkan Algoritma Kruskal memiliki kompleksitas *O*(*E log E*).**

***Keywords*—algoritma Kruskal, graf, pohon merentang minimum, kompleksitas**

I. PENDAHULUAN

Dalam keseharian kita, sebenarnya ada banyak hal yang dapat kita representasikan dalam bentuk graf. Misalnya adalah hubungan antarkota. Pada kasus ini kota dapat diibaratkan sebagai simpul (vertexs) dan jalan yang menghubungkan antarkota digambarkan sebagai busur (edges). Pohon (tree) merupakan graf yang tidak mengandung putaran (cycle). Pohon dapat dibuat dari suatu graf terhubung yang dihilangkan beberapa busurnya. Pohon yang dibentuk dari suatu graf dan mengandung semua simpul dari graf tersebut dinamakan pohon merentang (*spanning tree*). Dari satu graf terhubung yang berbobot dapat dibentuk banyak *spanning tree*. Pada permasalahan di kehidupan, terkadang kita perlu untuk mencari *spanning tree* dengan bobot minimum. Misal pada representasi hubungan antarkota tadi, kita bisa menentukan panjang jalur minimum yang dapat menghubungkan semua kota.

Ada beberapa algoritma yang dapat digunakan untuk menentukan pohon merentang minimal. Yang akan dibahas pada makalah ini adalah menentukan pohon merentang minimal dengan algoritma Kruskal. Algoritma ini dimulai dengan mengurutkan busur-busur pada graf mulai dari yang terkecil hingga terbesar. Kemudian dari itu, dipilihlah busur-busur yang akan membentuk pohon merentang minimum.

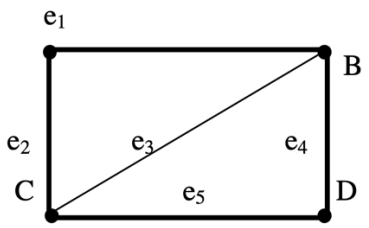
Dalam makalah ini, akan dikaji salah satu algoritma yang dapat digunakan untuk menentukan pohon merentang minimum, yaitu algoritma Kruskal dan kompleksitas algoritmanya.

II. DASAR TEORI

A. Graf

1) Definisi Graf

Graf adalah struktur data yang merepresentasikan kumpulan objek yang memiliki hubungan antar satu objek dengan objek yang lainnya dalam bentuk simpul dan busur. G membentuk suatu graf jika terdapat pasangan himpunan (V(G),E(G)), dimana V(G) (simpul pada graf G) tidak kosong dan E(G) (busur pada graf G).[2]



Gambar 1. Contoh graf [3]

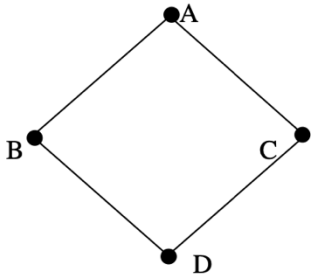
Dari gamar 1 kita bisa dapatkan himpunan V dan E yang menyatakan simpul dan busur sebagai berikut :

V = {A, B, C, D}
E = {(A,B),(A,C),(B,C),(B,D),(C,D)}

2) Jenis-Jenis Graf

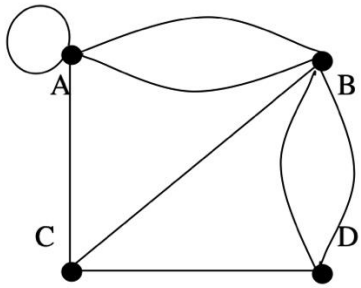
Berdasarkan ada atau tidaknya gelang atau sisi ganda, graf dapat dikategorikan menjadi dua jenis :

a. Graf sederhana
   Graf sederhana tidak mengandung gelang maupun sisi ganda



Gambar 2. Graf Sederhana[2]

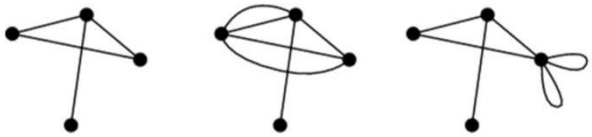
b. Graf tidak sederhana
   Graf tidak sederhana mengandung gelang atau sisi ganda.



Gambar 3. Graf tidak sederhana[2]

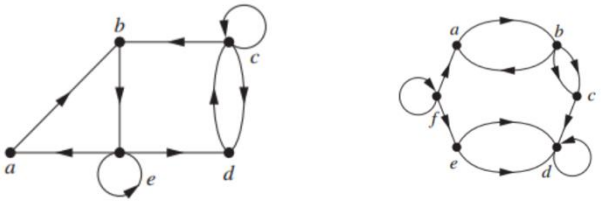
Berdasarkan orientasi arah pada busur, graf dapat dibedakan menjadi dua jenis, yaitu :

a. Graf tak-berarah
Graf yang sisinya tidak mempunyai orientasi arah.



Gambar 4. Graf tak-berarah[2]

b. Graf berarah
Graf yang sisinya mempunyai orientasi arah.



Gambar 5. Graf berarah[2]

3) Terminologi Graf

a. Ketetanggan
Dua simpul dikatakan bertetangga jika keduanya terhubung langsung oleh suatu busur.

b. Bersisian
Untuk sembarang busur $e = (v_j,v_k)$ dikatakan e bersisian dengan simpul v_j atau e bersisian dengan simpul v_k.

c. Simpul Terpencil
Sebuah simpul dikatakan simpul terpencil apabila tidak ada busur yang bersisian dengannya.

d. Graf Kosong
Graf kosong adalah graf yang himpunan busurnya merupakan himpunan kosong.

e. Derajat
Derajat dari suatu simpul merupakan hasil penjumlahan dari semua sisi yang bersisian dengan simpul tersebut.

f. Lintasan
Lintasan yang panjangnya n dari simpul awal v_0 ke simpul tujuan v_n di dalam graf G ialah barisan berselang-seling simpul-simpul dan sisi-sisi yang berbentuk $v_0,e_1,v_1,e_2,v_2,..,v_{n-1},e_n,v_n$ sedemikian sehingga $e_1 = (v_0,v_1)$, $e_2 = (v_1,v_2),...,e_n = c(v_{n-1},v_n)$ adalah sisi-sisi dari graf G.

g. Sirkuit
Sirkuit adalah lintasan yang simpul awal dan akhirnya sama.

h. Keterhubungan
Dua simpul, v1 dan v2, disebut terhubung jika ada jalur lintasan dari v1 ke v2. Sebuah graf G dikatakan terhubung (connected graph) jika untuk setiap pasangan simpul vi dan vj dalam himpunan V, terdapat jalur lintasan yang menghubungkan vi ke vj. Jika tidak ada lintasan yang menghubungkan setiap pasangan simpul dalam graf, maka G disebut sebagai graf tak-terhubung (disconnected graph).

i. Upagraf atau Komplemen Upagraf
Misalkan G = (V,E) adalah sebuah graf. $G_1 = (V_1,E_1)$ adalah upagraf dari G jika $V_1 \subseteq V$ dan $E_1 \subseteq E$.

j. Upagraf Merentang
Upagraf $G_1 = (V_1,E_1)$ dari G = (V,E) dikatakan upagraf rentang jika $V_1=V$ (yaitu G_1 mengandung semua simpul dari G).

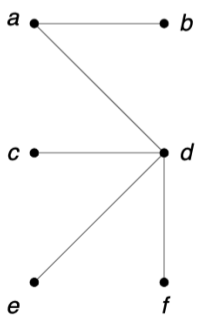
k. Cut-Set
Cut-Set dari graf terhubung G adalah himpunan sisi yang bila dibuang dari G menyebabkan G tidak terhubung. Jadi, *cut-set* selalu menghasilkan dua buah komponen.

l. Graf Berbobot
Graf berbobot adalah graf yang setiap sisinya diberi sebuah harga.

B. Pohon

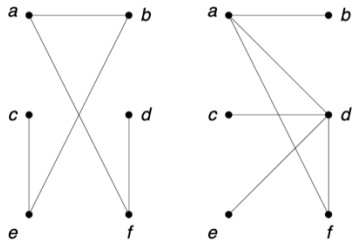
1) Definisi Pohon (tree)

Istilah Pohon (*tree)* pertama kali ditemukan pada tahun 1857 oleh seorang matematikawan asal Inggris, Arthur Cayley. Pohon adalah struktur data non-linier berbentuk hierarki yang terdiri dari simpul-simpul yang terhubung dan bentuknya seperti pohon. Pohon merupakan graf tak-berarah terhubung yang tidak mengandung sirkuit.[2]



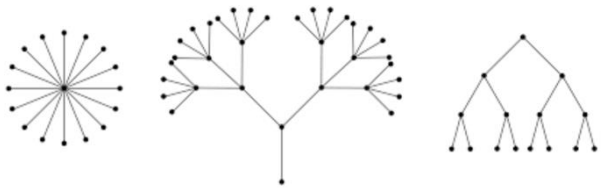
Gambar 6. Pohon[2]

Ada 3 syarat suatu graf dapat dikatakan pohon, yaitu tidak memiliki sirkuit, tidak berarah, dan harus terhubung. Sifat-sifat tersebut yang membedakan pohon dengan graf.



Gambar 7. Perbedaan graf dan pohon

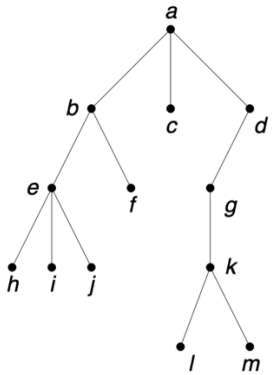
Pada gambar 7 dapat dilihat perbedaan antara graf dan pohon. Gambar yang sebelah kiri merupakan pohon dan gambar sebelah kanan merupakan graf. Yang menjadi penyebab gambar kanan tidak bisa dikatakan pohon adalah graf tersebut memiliki sirkuit. Pohon juga dapat digambarkan dalam berbagai bentuk seperti pada gambar berikut.



Gambar 8. Berbagai macam bentuk pohon

2) *Terminologi Pohon*

a. Lintasan (*path*)
Lintasan adalah jalur yang dilewati dari simpul awal hingga simpul akhir tujuan.



Gambar 9. Graf

Lintasan dari *a* ke *m* adalah *a,d,g,k,m*. Panjang dari lintasan tersebut adalah 4.

b. Derajat (*degree*)
Derajat sebuah simpul adalah jumlah anak pada simpul tersebut.

c. Anak (*child/children*)
Simpul anak adalah simpul turunan dari simpul diatasnya.

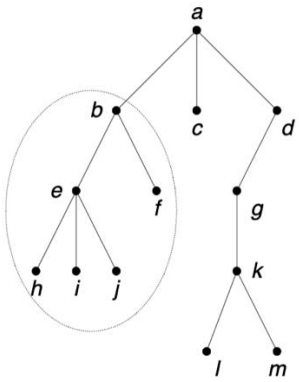
d. Orang tua (*parent*)
Simpul orang tua dari simpul a adalah simpul yang berada diatas simpul a. Simpul a berarti merupakan *child* dari simpul tersebut.

e. Saudara kandung (*sibling*)
Dua buah simpul atau lebih dapat dikatakan saudara kandung apabila simpul-simpul tersebut memiliki orang tua yang sama.

f. Daun (*leaf*)
Daun adalah simpul yang memiliki derajat nol (tidak memiliki anak).

g. Akar (*root*)
Root adalah simpul paling atas dari sebuah pohon.

h. Upapohon (*subtree*)
Upapohon adalah setiap simpul dari pohon beserta turunannya.



Gambar 10. Contoh upapohon

i. Simpul dalam (*internal nodes*)
Simpul dalam adalah semua simpul yang memiliki anak kecuali akar.

j. Aras (*level*)
Aras adalah tingkat dari graf. Aras dihitung 0 dari akar pohon dan terus bertambah sampai ke daun.

k. Tinggi (*height*)
Tinggi adalah aras maksimum dari suatu pohon (jumlah sisi dari *root* hingga ke *leaf* paling bawah).

C. Pohon Merentang Minimum (Minimum Spanning Tree)

Minimum Spanning Tree (MST) adalah upagraf dari sebuah graf berbobot terhubung yang mengandung semua simpul dari graf tersebut dan memiliki bobot paling kecil diantara alternatif upagraf lain. MST berbentuk pohon sehingga

upagraf yang dihasilkan tidak boleh memiliki sirkuit. Setiap graf pasti memiliki MST. Namun, tidak setiap graf hanya memiliki 1 solusi MST. Pada beberapa kasus, graf bisa jadi memiliki lebih dari 1 MST, akan tetapi jumlah bobot minimum yang dihasilkan pada MST tersebut tetaplah sama.

Terdapat 2 algoritma yang biasanya dipakai untuk menentukan MST dari sebuah graf, yaitu algoritma Prim dan algoritma Kruskal. Setiap algoritma tersebut memiliki lagkah pengerjaan dan kompleksitas yang berbeda.

D. Kompleksitas Algoritma

Kompleksitas algoritma menunjukkan seberapa efisien suatu algoritma berjalan. Kompleksitas algoritma dapat ditinjau dari waktu dan ruang. Kompleksitas waktu yang disimbolkan dengan *T(n)* mengukur tingkat efisiensi suatu algoritma dengan mengukur jumlah waktu yang dibutuhkan oleh algoritma berjalan. Namun, waktu yang dimaksud bukanlah waktu aktual algoritma tersebut berjalan, akan tetapi hanya sebuah fungsi dari panjang masukan. Sementara Kompleksitas ruang disimbolkan dengan *S(n)*. Kompleksitas ruang mengukur tingkat efisiensi suatu algoritma dari penggunaan memori dalam menyimpan data sementara selama algoritma tersebut berjalan.

Kompleksitas waktu dapat digolongkan menjadi 3, yaitu kompleksitas waktu untuk kasus terburuk ($T_{min}(n)$), kompleksitas waktu untuk kasus terbaik ($T_{max}(n)$), kompleksitas waktu rata-rata ($T_{avg}(n)$).

Notasi "*O*" disebut notasi "*O-Besar*" (*Big-O*) merupakan kompleksitas waktu asimptotik. "*T(n)=O(f(n))*" dibaca "*T(n)* adalah *O(f(n))*" , maksudnya adalah *T(n)* memiliki orde paling besar *f(n)* apabila terdapat konstanta *C* dan n_0 sedemikian sehingga $T(n) \leq C\ f(n)$ untuk $n \geq n_0$. Urutan spektrum kompleksitas waktu algoritma adalah :

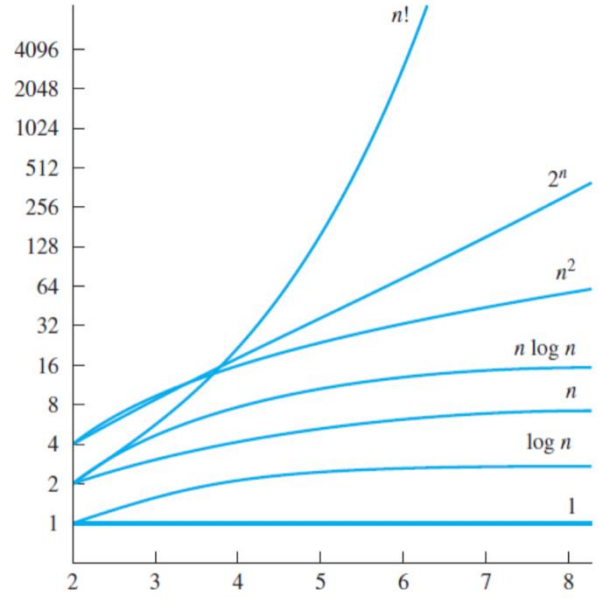
Kelompok Algoritma	Nama
$O(1)$	konstan
$O(\log n)$	logaritmik
$O(n)$	linier
$O(n \log n)$	linier logaritmik
$O(n^2)$	kuadratik
$O(n^3)$	kubik
$O(2^n)$	eksponensial
$O(n!)$	faktorial

Tabel 1. Pengelompokan algoritma berdasarkan Notasi *O*-Besar

Algoritma dengan kompleksitas $O(1)$ hingga $O(n^3)$ (urutasn sesuai pada tabel 1) dapat dikelompokkan menjadi algoritma polinomial sesuai dengan sifat pertumbuhan nilai hasil fungsinya. Sementara algoritma dengan kompleksitas $O(2^n)$ dan $O(n!)$ disebut algoritma eksponensial sebab pertumbuhan nilai hasil fungsinya eksponen. Algoritma polinomial dinilai lebih baik karena pertumbuhan nilai hasil fungsinya tidak sepesat algoritma eksponensial.

$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
0	1	0	1	1	2	1
1	2	2	4	8	4	2
2	4	8	16	64	16	24
3	8	24	64	512	256	362880
4	16	64	256	4096	65536	20922789888000
5	32	160	1024	32768	4294967296	(terlalu besar untuk ditulis)

Tabel 2. Perbandingan pertumbuhan kompleksitas algoritma



Gambar 11. Pertumbuhan kompleksitas algoritma

III. ALGORITMA KRUSKAL

Dalam teori graf, algoritma kruskal adalah algoritma yang digunakan untuk mencari pohon merentang minimum atau yang nama lainnya adalah *minimum spanning tree* (MST). Algoritma kruskal menggunakan pendekatan *greedy*, artinya adalah algoritma ini memilih busur dengan bobot terkecil kemudian menambahkan busur hingga semua simpul terhubung. Namun, algoritma ini harus memastikan bahwa busur yang tersambung tidak membentuk sirkuit.

Langkah-langkah membuat MST dengan algoritma Kruskal adalah sebagai berikut :

1) Urutkan busur-busur dari graf menaik berdasarkan nilai bobot busur.
2) Inisiasi *tree T* kosong.
3) Pilih busur (u,v) yang bobotnya paling kecil diantara busur yang lain kemudian tambahkan (u,v) ke dalam *T*.
4) Jika busur (u,v) membentuk sirkuit maka jangan masukkan busur (u,v) kedalam *T* dan lewati busur tersebut walaupun nilai bobotnya minimum.
5) Ulangi langkah sebanyak $n - 1$ kali.

Berikut adalah implementasi algoritma Kruskal dalam bentuk class dalam bahasa python. Program ini ditulis oleh Neelam Yadav dan dikembangkan kembali oleh James Graça-Jones .

```python
# Python program for Kruskal's algorithm to find
# Minimum Spanning Tree of a given connected,
# undirected and weighted graph


# Class to represent a graph
class Graph:

    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    # Function to add an edge to graph
    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    # A utility function to find set of an element i
    # (truly uses path compression technique)
    def find(self, parent, i):
        if parent[i] != i:

            # Reassignment of node's parent
            # to root node as
            # path compression requires
            parent[i] = self.find(parent, parent[i])
        return parent[i]

    # A function that does union of two sets of x and y
    # (uses union by rank)
    def union(self, parent, rank, x, y):

        # Attach smaller rank tree under root of
        # high rank tree (Union by Rank)
        if rank[x] < rank[y]:
            parent[x] = y
        elif rank[x] > rank[y]:
            parent[y] = x

        # If ranks are same, then make one as root
        # and increment its rank by one
        else:
            parent[y] = x
            rank[x] += 1

    # The main function to construct MST
    # using Kruskal's algorithm
    def KruskalMST(self):

        # This will store the resultant MST
        result = []

        # An index variable, used for sorted edges
        i = 0

        # An index variable, used for result[]
        e = 0

        # Sort all the edges in
        # non-decreasing order of their
        # weight
        self.graph = sorted(self.graph,
                            key=lambda item: item[2])

        parent = []
        rank = []

        # Create V subsets with single elements
        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        # Number of edges to be taken is less than to V-1
        while e < self.V - 1:

            # Pick the smallest edge and increment
            # the index for next iteration
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            # If including this edge doesn't
            # cause cycle, then include it in result
            # and increment the index of result
            # for next edge
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)
            # Else discard the edge

        minimumCost = 0
        print("Edges in the constructed MST")
        for u, v, weight in result:
            minimumCost += weight
            print("%d -- %d == %d" % (u, v, weight))
        print("Minimum Spanning Tree", minimumCost)
```

Gambar 12. Implementasi algoritma Kruskal dalam python

Graf pada implementasi algoritma Kruskal di gambar 13 berbentuk *class. Class* tersebut memiliki 2 properti, yaitu :

1) V atau Vertices
   Property ini digunakan untuk menyimpan nilai jumlah simpul pada graf.
2) graph
   graph bertipe data list dan digunakan untuk menyimpan busur pada graf beserta .

Class Graph pada gambar juga memiliki 4 *methods*, yaitu :

1) addEdge(self, u, v, w)
   Method ini berfungsi untuk menambahkan busur tepi dengan simpul u, simpul v, dan bobot w. Kompleksitas dari *method* ini adalah $O(1)$.
2) find(self, parent, i)
   Method ini berfungsi untuk menemukan himpunan dari elemen i dalam struktur *union-find*. Kompleksitas dari *method* ini adalah $O(\log V)$ dengan V adalah jumlah simpul.
3) union(self, parent, rank, x, y)
   Method ini berfungsi untuk melakukan penggabungan 2 himpunan dengan mempertimbangkan *rank* untuk mengoptimalkan struktur pohon. Kompleksitas dari *method* ini adalah $O(1)$.
4) KruskalMST(self)
   Method ini adalah *method* utama yang akan digunakan untuk membangun MST menggunakan algoritma Kruskal. Kompleksitas dari *method* ini adalah $O(E \log E)$ dengan E adalah jumlah busur.

Program dapat dijalankan misalnya dengan *driver* sebagai berikut.

```
100  # Driver code
101  if __name__ == '__main__':
102      g = Graph(4)
103      g.addEdge(0, 1, 10)
104      g.addEdge(0, 2, 6)
105      g.addEdge(0, 3, 5)
106      g.addEdge(1, 3, 15)
107      g.addEdge(2, 3, 4)
108
109      # Function call
110      g.KruskalMST()
```

Gambar 13. *Driver code*

Kode *driver* tersebut menunjukkan terdapat sebuah graf dengan simpul, busur, dan bobot sebagai berikut :

V = {0, 1, 2, 3}
E = {(0,1,10),(0,2,6),(0,3,5),(1,3,15),(2,3,4)}

Kemudian dari graf tersebut dicari MST-nya dengan *method* KruskalMST(). Hasil dari program tersebut adalah sebagai berikut

```
Edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Spanning Tree 19
```

Gambar 14. Hasil *running program*

IV. KESIMPULAN

Dari hasil analisis yang dilakukan pada implementasi algoritma Kruskal pada gambar 12, didapat bahwa kompleksitas waktu algoritma Kruskal adalah $O(E \log E)$. Saat menjalankan program tersebut, yang pertama dilakukan adalah mengurutkan busur-busur pada graf berdasarkan bobotnya mulai dari yang terkecil hingga terbesar. Proses pengurutan ini memiliki kompleksitas $O(E \log E)$. Kemudian diterapkan algoritma *find-union*, proses ini menggunakan waktu paling banyak sebesar $O(\log V)$. Sehingga kompleksitas waktu keseluruhan dari algoritma ini adalah $O(E \log E + E \log V)$. Nilai E maksimal yang bisa didapat adalah sebesar $O(V^2)$ sehingga $O(\log V)$ dan $O(\log E)$ adalah sama. Oleh karena itu, kompleksitas waktu keseluruhan adalah $O(E \log E)$ atau $O(E \log V)$.

V. UCAPAN TERIMA KASIH

Puji dan syukur yang sebesar-besarnya penulis panjatkan kepada Allah SWT karena dengan rahmat dan karunia-Nya penulis dapat menyelesaikan makalah yang berjudul ‘Analisis Kompleksitas Algoritma Kruskal untuk Menentukan *Minimum Spanning Tree*‘ ini dengan baik. Penulis juga menyampaikan terima kasih yang sangat besar kepada tim dosen pengajar mata kuliah Matematika Diskrit, terutama kepada Ibu Dr. Fariska Zakhralativa Ruskanda, S.T., M.T. yang mengajar di kelas K02 karena dengan ilmu yang bapak dan ibu sekalian ajarkan penulis dapat menyelesaikan makalah ini. Penulis juga ingin berterima kasih kepada teman-teman penulis yang sudah membantu penulis mulai dari memilih topik dan memberi dukungan dalam proses penyelesaian makalah ini.

REFERENSI

[1] “Kruskal’s Minimum Spanning Tree (MST) Algorithm” GeeksforGeeks, October 5, 2023. https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-*greedy*-algo-2/. (diakses pada 11 December 2023).
[2] Munir, Rinaldi. “Homepage Rinaldi Munir”. https://informatika.stei.itb.ac.id/~rinaldi.munir/. (diakses pada 9 Desember 2023).
[3] II-1 BAB II LANDASAN TEORI 2.1 Teori Graf. (n.d.). Available at: http://repository.uin-suska.ac.id/3815/3/Bab%20II.pdf (diakses pada 13 Dec. 2023).
[4] Munir, Rinaldi. “Homepage Rinaldi Munir”. https://informatika.stei.itb.ac.id/~rinaldi.munir/. (diakses pada 12 Desember 2023).
[5] Trivusi (2022). Struktur Data Tree: Pengertian, Jenis, dan Kegunaannya. [online] Trivusi. Available at: https://www.trivusi.web.id/2022/07/struktur-data-tree.html?m=1.
[6] arvindpdmn, dineshpathak (2017). Algorithmic Complexity. [online] Devopedia. Available at: https://devopedia.org/algorithmic-complexity.
[7] “Time Complexity and Space Complexity” GeeksforGeeks, August 9, 2023. https://www.geeksforgeeks.org/time-complexity-and-space-complexity/. (diakses pada 12 December 2023).

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, 10 Desember 2023

Fabian Radenta Bangun
13522105