Analisis Kompleksitas Algoritma Prim

Ariel Herfrison – 13522002[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13522002@std.stei.itb.ac.id

Abstrak*— Salah satu permasalahan yang ditemui dalam graf adalah penentuan pohon rentang minimum. Terdapat berbagai algoritma untuk menentukan pohon rentang minimum, masing-masing dengan kompleksitas algoritma yang berbeda-beda. Makalah ini mengkaji salah satu algoritma penentuan pohon rentang minimum, yaitu Algoritma Prim, dan kompleksitas algoritmanya. Kajian dilakukan dengan menganalisis tiga bentuk implementasi Algoritma Prim, yaitu implementasi secara Brute Force, implementasi dengan set, dan implementasi dengan Binary Heap. Hasil analisis menunjukkan bahwa Algoritma Prim dapat memiliki kompleksitas algoritma dari $O(n^4)$ sampai O(Elog(E)). Hasil tersebut kesangkilan Algoritma Prim dalam menentukan pohon rentang minimum, serta adanya kepentingan dalam pemilihan implementasi algoritma yang tepat.

***Kata Kunci*— algoritma, prim, graf, pohon, kompleksitas**

I. Pendahuluan

Graf adalah struktur yang merepresentasikan hubungan antara objek-objek diskrit. Salah satu kegunaan graf adalah menyatakan hubungan antar objek berdasarkan bobot, seperti dalam peta jaringan jalur kereta api atau peta jarak antar kota. Salah satu permasalahan yang sering ditemui adalah bagaimana cara mencari hubungan antara semua graf dengan total bobot terendah, atau yang disebut juga sebagai pohon rentang minimum.

Terdapat bermacam algoritma untuk mencari pohon rentang minimum, seperti Algoritma Prim dan Algoritma Kruskal. Dalam pemilihan algoritma untuk menyelesaikan masalah, diperlukan penilaian dan pertimbangan antara masing-masing algoritma. Salah satu objek penilaian yang dapat dilakukan adalah terhadap kompleksitas algoritma.

Dalam makalah ini, akan dikaji salah satu algoritma penentuan pohon rentang minimum, yaitu Algoritma Prim, dan kompleksitas algoritmanya.

II. Dasar Teori

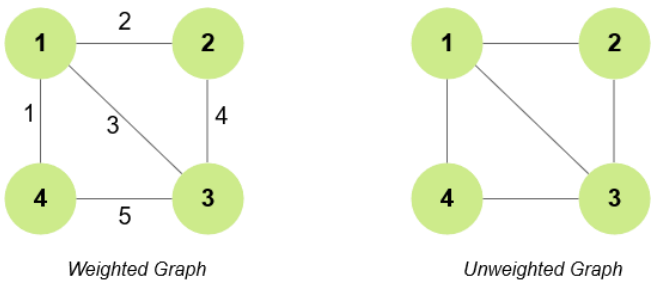
A. Graf

Graf adalah struktur yang merepresentasikan keterhubungan antara objek-objek diskrit. Beberapa contoh graf di dunia nyata adalah peta jaringan rel kereta api, sketsa rangkaian listrik, dan sketsa jaringan komputer. Graf terdiri atas simpul yang merepresentasikan objek-objek diskrit dan sisi yang menghubungkan simpul-simpul tersebut. Berdasarkan orientasi arah pada sisi, graf dapat berupa graf tak-berarah atau graf berarah.

Gambar 1. Contoh Graf
sumber: https://mathigon.org/course/graph-theory/applications

Beberapa terminologi penting di dalam graf adalah tetangga, derajat, lintasan, sirkuit, dan upagraf. Suatu simpul A bertetangga dengan simpul B apabila terdapat sisi yang menyambungkan simpul A tersebut dengan simpul B. Derajat suatu simpul adalah jumlah sisi yang bersisian dengan simpul tersebut. Lintasan adalah himpunan sisi-sisi yang menghubungkan simpul A dengan simpul B. Sirkuit adalah lintasan yang memiliki simpul awal dan akhir yang sama. Upagraf adalah graf yang simpul-simpul dan sisi-sisinya merupakah subset dari graf lain. Upagraf disebut upagraf merentang apabila upagraf mengandung semua simpul dari graf asalnya.

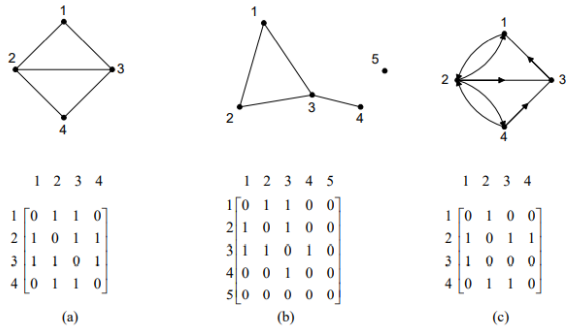
Graf juga dapat bersifat berbobot atau tak berbobot. Berbobot memiliki arti bahwa sisi dalam graf memiliki harga/nilai. Bobot graf memiliki banyak kegunaan, seperti untuk merepresentasikan jarak antara dua tempat berbeda. Salah kegunaannya adalah untuk mencari lintasan yang memiliki bobot terkecil. Bobot graf dapat bernilai positif maupun negatif, tetapi untuk keperluan peninjauan makalah ini, akan digunakan asumsi bahwa bobot graf tidak negatif.



Gambar 2. Graf Berbobot dan Tak-Berbobot
sumber: prepfortech.in/interview-topics/graphs/complete-guide-to-graphs/

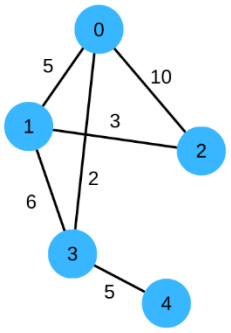
Graf dapat direpresentasikan dengan berbagai macam cara. Dua contohnya adalah dengan matriks ketetanggaan dan senarai ketetanggaan (*adjacency list*).

Matriks ketetanggaan adalah matriks NxN (N adalah jumlah simpul) dengan baris dan kolomnya merepresentasikan simpul dan elemen-elemennya merepresentasikan ketetanggaan antar simpul. Misal diberikan baris *i* dan kolom *j*, simpul *i* dan *j* bertetangga apabila elemen M_{ij} bernilai 1. Sebaliknya, simpul *i* dan *j* tidak bertetangga apabila elemen M_{ij} bernilai 0. Dalam kasus graf tak-berarah, matriks ketetanggaan bersifat simetrik.



Gambar 3. Matriks Ketetanggaan
sumber: www.belajarstatistik.com

Senarai ketetanggaan adalah set yang menyimpan tetangga dari masing-masing simpul. Dalam kasus graf berbobot, setiap elemen merepresentasikan pasangan simpul beserta bobot sisi yang menghubungkannya.



Adjacency List:

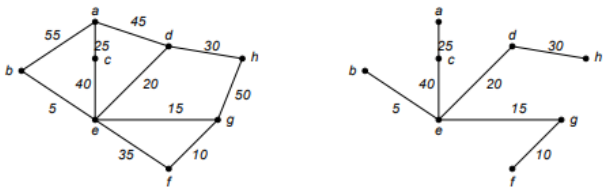
Simpul Awal	Simpul Akhir	Bobot
0	1	5
0	2	10
0	3	2
1	2	3
1	3	6
3	4	5

Gambar 4. Senarai Ketetanggaan

B. Pohon

Seperti graf, pohon adalah representasi keterhubungan antara objek-objek diskrit. Perbedaan pohon dari graf adalah semua simpul dalam pohon harus terhubung dengan lintasan tunggal, pohon tidak berarah, dan pohon tidak memiliki sirkuit.

Dari sebuah graf, dapat dibentuk pohon merentang yaitu upagraf merentang yang berupa pohon. Graf juga dapat membentuk hutan merentang, yaitu Kumpulan pohon merentang yang saling lepas. Salah satu manfaat pohon merentang adalah mencari pohon merentang minimum dari suatu graf berbobot, yaitu pohon merentang dengan total bobot terkecil. Salah satu metode penyelesaian masalah tersebut adalah dengan menggunakan Algoritma Prim.



Gambar 5. Pohon Merentang
sumber: https://www.belajarstatistik.com/blog/2021/10/15/pohon-merentang/

C. Algoritma Prim

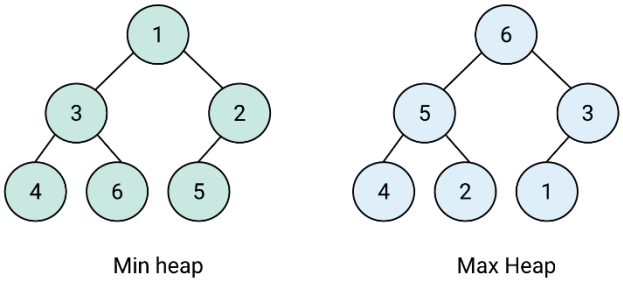
Algoritma Prim adalah algoritma untuk menentukan pohon merentang minimum dari suatu graf berbobot tak-berarah. Algoritma Prim dimulai dengan menentukan sisi, mungkin dari beberapa sisi, dengan bobot terkecil, lalu menggunakannya sebagai simpul awal dari pohon yang akan dihasilkan. Algoritma Prim lalu membangun pohon secara cabang per cabang, menambahkan sisi dengan bobot terkecil pada setiap tahap. Pada kasus khusus, yaitu apabila sisi dengan bobot terkecil sudah bersisian dengan simpul yang telah ditambahkan ke pohon hasil, maka sisi tersebut tidak ditambahkan ke dalam pohon, melainkan dicari sisi lain yang memiliki bobot terkecil berikutnya.

D. Kompleksitas Algoritma

Kompleksitas Algoritma adalah ukuran efisiensi/kesangkilan suatu algoritma. Kompleksitas algoritma dapat diukur dari waktu yang dibutuhkan bagi algoritma untuk menyelesaikan semua langkahnya atau ruang memori yang dibutuhkan algoritma. Kompleksitas algoritma cenderung diukur berdasarkan jumlah/ukuran data atau masukan (N). Kompleksitas algoritma krusial dalam penilaian dan pemilihan algoritma dalam penyelesaian suatu masalah.

E. Heap

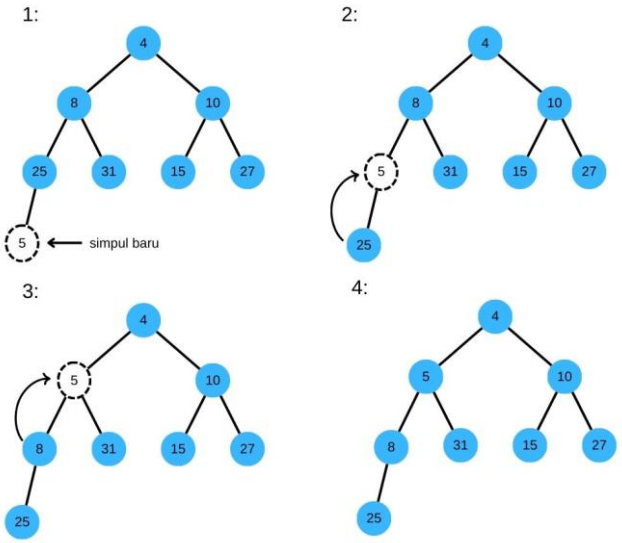
Heap adalah Pohon yang memiliki karakteristik khusus, yaitu setiap anak memiliki bobot yang lebih ekstrem (besar/kecil) atau sama dengan bobot ayahnya. Terdapat berbagai jenis Heap, seperti Binary Heap, K-ary Heap, Binomial Heap, dan Fibonacci Heap. Untuk keperluan peninjauan Algoritma Prim, hanya akan digunakan Binary Heap tipe min-heap, yaitu setiap simpul memiliki maksimal 2 anak dan setiap anak memiliki bobot yang lebih besar atau sama dengan bobot ayahnya.



Gambar 6. Heap
sumber: guides.codepath.com/compsci/Heaps

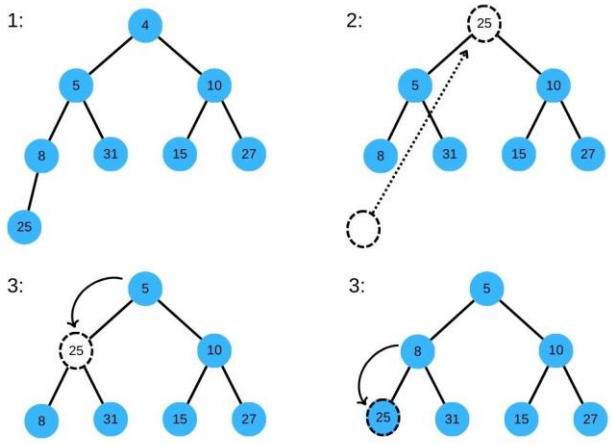
Terdapat dua operasi penting dalam heap, yaitu heap-push dan heap-pop.

Pada operasi push, program akan pertama-tama menambahkan elemen yang dipush sebagai simpul terakhir dari pohon. Lalu, program akan membandingkan bobot simpul baru tersebut dengan bobot simpul ayahnya. Jika bobot simpul tersebut lebih kecil, simpul tersebut akan ditukar dengan simpul ayahnya. Tahap tersebut kemudian diulang sampai ditemukan simpul ayah yang memiliki bobot lebih kecil. Apabila pohon tersebut biner, pohon akan tersusun dalam log(N) tingkat. Karena operasi push mengiterasi sepanjang tingkat pohon, kompleksitas operasi push adalah O(log(N)).



Gambar 7. Operasi Push Heap

Pada operasi pop, program pertama-tama akan menghapus simpul akar dengan menukarkannya dengan simpul akhir dalam pohon, lalu menghapus simpul akhir yang telah ditukar tersebut. Kemudian, program akan membandingkan simpul akar yang baru dengan kedua anaknya. Apabila salah satu atau dua simpul anaknya memiliki bobot lebih kecil, simpul tersebut ditukarkan dengan simpul anak yang memiliki bobot terkecil. Simpul tersebut kemudian dikaji ulang sampai kedua anaknya tidak memiliki bobot yang lebih kecil. Seperti operasi push, karena operasi pop mengiterasi sepanjang tingkat pohon, kompleksit operasi pop adalah O(log(N)).



Gambar 8. Operasi Pop Heap

III. ANALISIS

Terdapat berbagai cara implementasi Algoritma Prim. Pada makalah ini, akan dikaji kompleksitas algoritma dari tiga bentuk implementasi Algoritma Prim, yaitu secara *Brute Force*, dengan bantuan Set, dan dengan bantuan Binary Heap.

A. Analisis Kompleksitas Algoritma Prim dengan Implementasi Brute Force

Apabila graf direpresentasikan dengan matriks ketetanggaan, berikut adalah tahap prosedur Algoritma Prim dengan implementasi *Brute Force*:

1. Menentukan suatu simpul dengan bobot terkecil sebagai simpul awal dari pohon
2. Meninjau setiap sisi dari setiap simpul yang sudah berada di dalam pohon.
3. Menentukan satu sisi dengan bobot minimum yang tidak membentuk sirkuit apabila ditambahkan ke dalam pohon.
4. Menambahkan sisi tersebut beserta simpulnya ke dalam pohon.
5. Mengulangi 3angkah 2-5 sampai semua simpul telah berada di dalam pohon.

Berikut adalah perumusan tahap-tahap tersebut di dalam bahasa Python:

```
def prim(graph):
# ALGORITMA
    N = len(graph)
    simpulPohon = [0 for I in range(N)]
    sisiPohon = [(0,0) for I in range(N-1)]
    min = 0
    for I in range(N):
        for j in range(N):
            if (graph[i][j] != 0 and (min == 0 or graph[i][j] < min)):
                min = graph[i][j]
                simpulPohon[0] = i
                simpulPohon[1] = j
                sisiPohon[0] = (I,j)

    for k in range(2,N):
        min = 0
        for u in simpulPohon:
            for j in range(N):
                if (graph[u][j] != 0 and not (j in simpulPohon) and (min == 0 or graph[u][j]<min)):
                    min = graph[u][j]
                    simpulPohon[k] = j
                    sisiPohon[k-1] = (u,j)

    return sisiPohon
```

Kompleksitas untuk

```
    N = len(graph)                          O(1)
    simpulPohon = [0 for I in range(N)]     O(n)
    sisiPohon = [(0,0) for I in range(N-1)] n-1 = O(n)
    min = 0                                 O(1)
    for I in range(N):                      O(n)
        for j in range(N):                      O(n)
            if (graph[i][j] != 0 and (min == 0 or graph[i][j] < min)):  O(1)
```

```
            min = graph[i][j]          O(1)
            simpulPohon[0] = i         O(1)
            simpulPohon[1] = j         O(1)
            sisiPohon[0] = (i,j)       O(1)
```

Kompleksitas =

$$O(1) + O(n) + O(n) + O(1) + O(n)O(n)\{O(1) + O(1) + O(1) + O(1) + O(1)\}$$
$$= O(n) + O(n^2)O(1)$$
$$= O(n) + O(n^2)$$
$$= O(n^2)$$

Kompleksitas untuk

```
for k in range(2,N):
    min = 0                                  O(1)
    for u in simpulPohon:
        for j in range(N):
            if (graph[u][j] != 0 and not (j in simpulPohon) and
(min == 0 or graph[u][j]<min)):              O(1)
                min = graph[u][j]            O(1)
                simpulPohon[k] = j           O(1)
                sisiPohon[k-1] = (u,j)       O(1)
```

- for k in range(2,N): … for u in simpulPohon → membentuk deret aritmatika dengan jumlah iterasi = 2+3+…+N-1 = n(n+3)/2
- for k in range(2,N): … for j in range(N): … not (j in simpulPohon) → operasi searching dengan kesangkilan O(n)

Kompleksitas =

$$\left(O(n)O(1) + \frac{n(n+3)}{2}\right)O(n)(O(n) + O(1))$$
$$= \left(O(n) + O(n^2)\right) * O(n)O(n)$$
$$= O(n^2) * O(n^2)$$
$$= O(n^4)$$

Kompleksitas keseluruhan:

```
…
for i in range(N):          O(n^2)
    …

for k in range(2,N):        O(n^4)
    …
```

Kompleksitas akhir =

$$O(n^2) + O(n^4)$$
$$= O(n^4)$$

Berdasarkan hasil perhitungan, kompleksitas Algoritma Prim dengan implementasi *Brute Force* adalah $O(n^4)$.

B. Analisis Kompleksitas Algoritma Prim dengan Set

Salah satu penyebab ketidaksangkilan implementasi Algoritma Prim secara Brute Force adalah operasi *searching* untuk memastikan simpul belum berada di dalam pohon. Untuk mengatasi ini, dapat dimanfaatkan set perantara sehingga operasi *searching* dapat digantikan dengan operasi *reading* yang hanya membutuhkan waktu O(1). Apabila graf direpresentasikan dengan matriks ketetanggaan, berikut adalah tahap prosedur Algoritma Prim dengan set:

1. Inisialisasi set untuk mengingat keberadaan simpul dalam pohon (isInPohon), set untuk menyimpan ayah dari simpul yang telah ditambahkan ke dalam pohon (ayah), dan set untuk menyimpan bobot dari setiap simpul yang telah ditambahkan ke dalam pohon (bobot) dengan nilai awalnya adalah tak hingga/*infinite*.
2. Tentukan suatu simpul dengan bobot terkecil sebagai simpul awal dari pohon
3. Tentukan suatu simpul yang belum berada di dalam pohon yang memiliki bobot minimum
4. Masukkan simpul tersebut ke dalam pohon
5. Tinjau setiap tetangga dari simpul tersebut. Apabila bobotnya lebih kecil dari bobot yang tercatat dalam set 'bobot', perbarui bobot dan ayahnya dalam set tersebut
6. Ulangi tahap 2-5 sebanyak N-1 kali

Berikut adalah perumusan tahap-tahap tersebut di dalam bahasa Python:

```
def prim2(graph):
# KAMUS
import math
inf = math.inf
# ALGORITMA
    N = len(graph)
    sisiPohon = []
    isInPohon = [False for i in range(N)]
    ayah = [-1 for i in range(N)]
    bobot = [inf for i in range(N)]

    min = -1
    for i in range(N):
        for j in range(N):
            if (graph[i][j] != 0 and (min == -1 or graph[i][j] <
min)):
                min = graph[i][j]
                min_idx = i
    bobot[min_idx] = 0

    for i in range(N-1):
        min = -1
        for j in range(N):
            if (isInPohon[j] == False and (min == -1 or bobot[j] <
min)):
                min = bobot[j]
                min_idx = j
        isInPohon[min_idx] = True

        for j in range(N):
            if (graph[min_idx][j] != 0 and isInPohon[j] == False
and graph[min_idx][j]<bobot[j]):
                ayah[j] = min_idx
                bobot[j] = graph[min_idx][j]

    for i in range(N):
        if (ayah[i] != -1) :
            sisiPohon.append((ayah[i],i))

    return sisiPohon
```

Komplekitas keseluruhan:

```
def prim2(graph):
```

```
# KAMUS
import math
inf = math.inf
# ALGORITMA
    N = len(graph)
    sisiPohon = []          O(1)
    isInPohon = [False for i in range(N)]    O(n)
    ayah = [-1 for i in range(N)]            O(n)
    bobot = [inf for i in range(N)]          O(n)

    min = -1                O(1)
    for i in range(N):          O(n)
        for j in range(N):          O(n)
            if (graph[i][j] != 0 and (min == -1 or graph[i][j] < min)):                 O(1)
                min = graph[i][j]       O(1)
                min_idx = I             O(1)
    bobot[min_idx] = 0              O(1)

    for i in range(N-1):        O(n)
        min = -1                O(1)
        for j in range(N):          O(n)
            if (isInPohon[j] == False and (min == -1 or bobot[j] < min)):               O(1)
                min = bobot[j]      O(1)
                min_idx = j         O(1)
        isInPohon[min_idx] = True       O(1)

        for j in range(N):          O(n)
            if (graph[min_idx][j] != 0 and isInPohon[j] == False and graph[min_idx][j]<bobot[j]):    O(1)
                ayah[j] = min_idx           O(1)
                bobot[j] = graph[min_idx][j]        O(1)

    for i in range(N):          O(n)
        if (ayah[i] != -1) :        O(1)
            sisiPohon.append((ayah[i],i))       O(1)

    return sisiPohon
```

Dengan menghiraukan operasi *O(1)* yang tidak dominan, kompleksitas akhir =

$$O(n) + O(n) + O(n) + O(n)O(n) + O(n)\big(O(n) + O(n)\big) + O(n)$$
$$= O(n^2) + O(n^2) + O(n)$$
$$= O(n^2)$$

Berdasarkan hasil perhitungan, kompleksitas Algoritma Prim dengan Set adalah $O(n^2)$.

C. Analisis Kompleksitas Algoritma Prim dengan Binary Heap

Apabila graf direpresentasikan dengan List Ketetanggaan, dapat disusun algoritma yang lebih efisien dengan memanfaatkan Binary Min-Heap. Berikut adalah tahap prosedur algoritma tersebut:

1. Ubah adjacency list menjadi set (convList) yang menyimpan *pair* berupa tetangga dan bobot sisi menuju tetangga tersebut dari setiap simpul
2. Inisialisasi heap, lalu pilih suatu simpul asal sebagai simpul awal dan push simpul tersebut ke dalam heap
3. Pop simpul dari heap. Karena menggunakan min-heap, simpul memiliki bobot minimum.
4. Kunjungi simpul tersebut. Apabila belum dikunjungi sebelumnya, push semua simpul tetangga yang belum dikunjungi ke dalam heap.
5. Ulangi tahap 3-5 sampai heap kosong.

Berikut adalah perumusan tahap-tahap tersebut di dalam bahasa Python:

```
def primHeap(N,adjList):
# KAMUS
import heapq
# ALGORITMA
    convList = [[] for _ in range(N)]

    for i in range(len(adjList)):
        a, b, bobot = adjList[i]
        convList[a].append((b, bobot))
        convList[b].append((a, bobot))

    heap = []
    isInPohon = [False for _ in range(N)]
    sisiPohon = []

    heapq.heappush(heap, (0, -1, 0))

    while (heap):
        bobot,ayah,a = heapq.heappop(heap)
        if (not isInPohon[a]):
            if (ayah != -1):
                sisiPohon.append((ayah,a))
            isInPohon[a] = True

            for (b,bobot) in convList[a]:
                if (not isInPohon[b]):
                    heapq.heappush(heap,(bobot,a,b))

    return sisiPohon
```

Dengan E = jumlah sisi, kompleksitas untuk

```
    convList = [[] for _ in range(N)]           O(N)

    for i in range(len(adjList)):               O(E)
        a, b, bobot = adjList[i]                O(1)
        convList[a].append((b, bobot))          O(1)
        convList[b].append((a, bobot))          O(1)

    heap = []                                   O(1)
    isInPohon = [False for _ in range(N)]       O(n)
    sisiPohon = []                              O(1)

    heapq.heappush(heap, (0, -1, 0))            O(log(E))
```

Kompleksitas =

$$O(N) + O(E)\big(O(1) + O(1) + O(1)\big) + O(1) + O(n) + O(1) + O(\log(E)))$$
$$= O(E)O(1)$$
$$= O(E)$$

Kompleksitas untuk

```
    while (heap):
        bobot,ayah,a = heapq.heappop(heap)
```

```
…
    for (b,bobot) in convList[a]:
        if (not isInPohon[b]):
            heapq.heappush(heap,(bobot,a,b))
```

- for (b,bobot) in convList[a] → push sebanyak elemen dalam convList
- Total elemen dalam convList = jumlah derajat semua simpul = 2*E
- Jumlah iterasi push = 2*E
- Jumlah iterasi pop = jumlah iterasi push = 2*E
- Maka, jumlah operasi di dalam while loop = 2*E+2*E = 4*E

Kompleksitas = jumlah operasi * kompleksitas operasi heappush/heappop = 4E* *O(log(E)) = O(Elog(E))*

Kompleksitas keseluruhan

```
convList = [[] for _ in range(N)]      O(E)
…
while (heap):                          O(Elog(E))
    …
```

Kompleksitas akhir =

$$(E) + O(\text{Elog}(E))) = O(Elog(E))$$

Berdasarkan hasil perhitungan, kompleksitas Algoritma Prim dengan Set adalah $O(Elog(E))$.

IV. KESIMPULAN

Berdasarkan hasil analisis, ditemukan bahwa Algoritma Prim dapat memiliki kompleksitas waktu dari $O(n^4)$ sampai $O(Elog(E))$. Algoritma Prim dengan implementasi *Brute Force* memiliki kompleksitas waktu $O(n^4)$, tetapi dapat dipercepat dengan penerapan set menjadi $O(n^2)$. Algoritma dapat dipercepat lagi dengan implementasi Binary Heap sehingga memiliki kompleksitas waktu $O(Elog(E))$. Hal ini menunjukkan bahwa Algoritma Prim cukup sangkil dalam menentukan pohon rentang minimum dari suatu graf. Perbedaan kompleksitas waktu antara metode implementasi juga menunjukkan terdapatnya kepentingan dalam memilih implementasi algoritma yang tepat.

V. UCAPAN TERIMA KASIH

Segala puji dan syukur penulis panjatkan kepada Tuhan Yang Maha Esa oleh karena hikmat dan anugerahnya penulis dapat menyelesaikan makalah ini. Penulis menyampaikan terima kasih kepada Dr. Nur Ulfa Maulidevi selaku dosen pengampu IF2120 Matematika Diskrit kelas K01 atas bimbingan dan pemberian ilmu yang menjadi landasan penulisan makalah ini. Penulis juga berterimakasih kepada teman-teman penulis serta pihak-pihak lain yang telah mendorong dan menolong penulis dalam penyelesaian makalah ini.

REFERENSI

[1] "Prim's Algorithm for Minimum Spanning Tree (MST)." GeeksforGeeks, December 4, 2023. https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/. (diakses pada 9 December 2023).

[2] Chris L. Kuszmaul, "heap", in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed. 6 April 2023. Available from: https://www.nist.gov/dads/HTML/heap.html. (diakses pada 9 Desember 2023).

[3] Munir, Rinaldi. "Homepage Rinaldi Munir". https://informatika.stei.itb.ac.id/~rinaldi.munir/. (diakses pada 9 Desember 2023).

[4] Rosen, Kenneth H. Rosen. *Discrete Mathematics and Application to Computer Science*. 8th Edition. Mc Graw-Hill, Inc. 2018.

[5] Wengrow, Jay. *A Common-Sense Guide to Data Structures and Algorithms*. 2nd Edition. Pragmatic Bookshelf. 2020.

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, 11 Desember 2023

Ariel Herfrison 13522002