Penerapan Algoritma *Topological Sorting* untuk Menyelesaikan *Dependency* Suatu Program

Yanuar Sano Nur Rasyid - 13521110[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1] *13521110@std.stei.itb.ac.id*

***Abstract*— Topological Sorting adalah algoritma untuk menyusun sebuah DAG sehingga tiap simpul berurutan sesuai dengan arah yang simpul itu terhadap simpul lain. DAG adalah directed acyclic graph yaitu graf yang berarah yang tidak memiliki loop. Algoritma ini dapat diterapkan untuk menyelesaikan dependency dari sebuah program yang akan di-install. Penerapan ini dilakukan dengan membuat program simulasi package manager yang dapat meng-install dan menghapus package yang ada di dalam sebuah database.**

***Keywords*— Dependency, Graf, Package Manager, Topological Sorting**

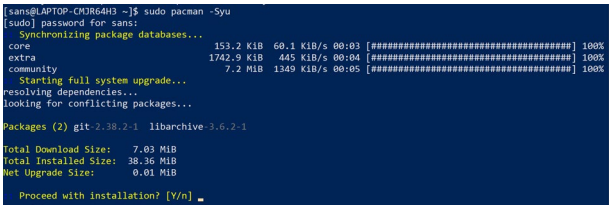
I. PENDAHULUAN

Dalam pembuatan sebuah program, seorang *programmer* dapat meng-*import library* yang tersedia di internet untuk dimasukkan ke dalam programnya. Akibat dari hal tersebut adalah fungsionalitas program yang dibuat akan bergantung kepada *library* yang sudah di-*import* tersebut. Jika *library* tersebut tidak bisa diakses oleh program maka program tidak akan berjalan dengan baik. Kebergantungan ini disebut dengan istilah *dependency*.

Dependency pada suatu program belum tentu merupakan sebuah hal yang buruk. Salah satu keuntungannya adalah dengan menggunakan *library,* seorang *programmer* tidak perlu untuk melakukan implementasi sendiri terhadap fitur yang ingin dibuat. *Programmer* cukup meng-*import* sebuah *library* yang memiliki fitur yang sesuai dengan keinginannya.

Akan tetapi, *dependency* juga dapat memberikan dampak buruk terhadap program. Selain dapat merusak fungsionalitas program jika *dependency* tidak sesuai, penggunaan *dependency* juga akan menyulitkan pengguna meng-*install* program terkait. Pengguna perlu melakukan instalasi terhadap *dependency* program. Jika program tersebut memiliki *dependency*-nya sendiri maka *user* perlu melakukan instalasi *dependency* kembali sampai ke program yang tidak memiliki *dependency.*

Untuk mengatasi permasalahan tersebut, pengguna dapat melakukan instalasi sebuah program dengan menggunakan *package manager. Package Manager* adalah sebuah program yang mengatur program yang ada di dalam sebuah komputer. *Package Manager* juga dapat mengotomatisasi proses instalasi sehingga pengguna tidak perlu melakukan instalasi *dependency* secara manual.



Gambar 1.1 *Package Manager* pacman
Sumber : Dokumentasi Pribadi

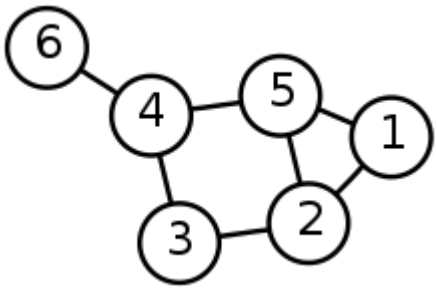
II. TEORI DASAR

A. Teori Graf

a. Definisi Graf Secara Umum

Sesuai yang disampaikan di [1] , graf merupakan struktur objek diskrit yang terdiri dari *vertices* atau simpul yang biasa digambarkan dengan sebuah titik dan *edge* atau sisi yang biasa digambarkan dengan sebuah garis. Secara formal G = (V, E) dengan

V = himpunan tidak-kosong dari simpul-simpul (vertices)
= { v1 , v2 , ... , vn }
E = himpunan sisi (edges) yang menghubungkan sepasang simpul
= {e1 , e2 , ... , en }



Gambar 2.1.1 Gambar Graf
Sumber :
https://upload.wikimedia.org/wikipedia/commons/thumb/5/5b/6n-graf.svg/440px-6n-graf.svg.png

b. Jenis-Jenis Graf

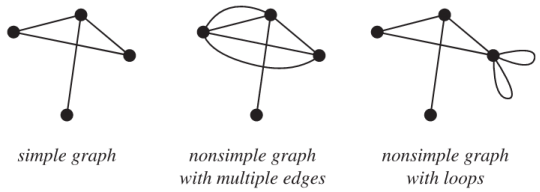
Graf memiliki beberapa jenis bergantung terhadap properti yang dimiliki oleh graf tersebut. Jenis-jenis graf antara lain

1. Graf sederhana (*simple graph)*

Merupakan graf yang tidak memiliki sisi ganda dan juga sisi gelang atau *loop*.

2. Graf tidak sederhana (un*simple graph)*

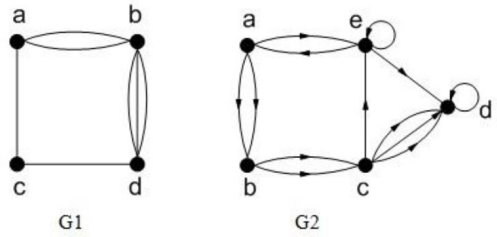
Merupakan kebalikan dari graf sederhana yaitu graf yang memiliki sisi ganda atau sisi *loop.*



Gambar 2.1.2 Graf Sederhana dan Graf tidak sederhana
Sumber : https://mathworld.wolfram.com/SimpleGraph.html

3. Graf Berarah (*directed graph)*

Sebuah graf yang sisinya memiliki suatu arah sehingga jika sebuah *vertex* terhubung bukan berarti *vertices* tersebut memiliki hubungan dua arah antara satu sama lain.



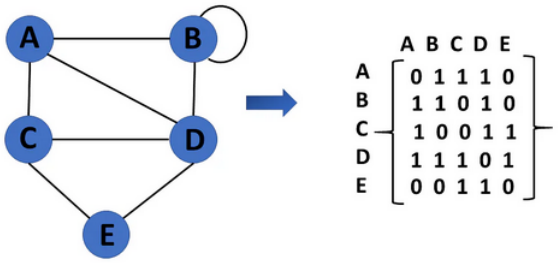
Gambar 2.1.3 Graf Tidak Berarah dan Graf Berarah
Sumber :
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

c. Representasi Graf

Menurut [2], terdapat beberapa cara untuk merepresentasikan suatu graf yaitu

1. Matriks Ketetanggaan (*adjacency matrix)*

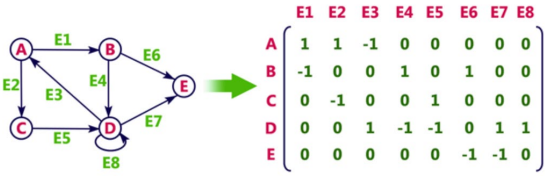
Graf direpresentasikan dengan matriks sebesar n x n dengan n adalah jumlah banyaknya simpul dan isi matriks tersebut diisi dengan status ketetanggaan antara simpul baris dan simpul kolom.



Gambar 2.1.4 Representasi Graf Matriks Ketetanggaan
Sumber : https://www.simplilearn.com/tutorials/data-structure-tutorial/graphs-in-data-structure

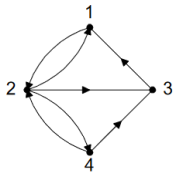
2. Matriks Bersisian (*incidency matrix)*

Seperti dengan matriks ketetanggaan, graf direpresentasikan dengan sebuah matriks , tetapi perbedaanya kolom matriks diganti dengan sisi graf sehingga dimensi matriks adalah n x m dengan n adalah jumlah simpul dan m adalah jumlah sisi.



Gambar 2.1.5 Representasi Graf Matriks Bersisian
Sumber : https://www.javatpoint.com/graph-theory-graph-representations

3. Senarai Ketetanggaan (*adjacency list)*

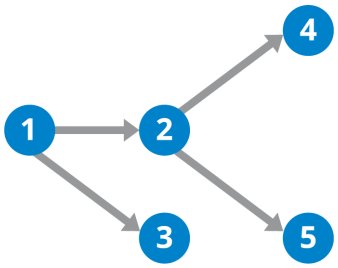


Simpul	Simpul Terminal
1	2
2	1, 3, 4
3	1
4	2, 3

Gambar 2.1.6 Representasi Graf *Adjacency List*
Sumber :
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian2.pdf

d. Definisi *Directed Acyclic Graph*

Menurut [3], sesuai dengan namanya, *directed* berarti berarah dan *acyclic* berarti tidak ada *cycle* atau *loop, Directed Acyclic Graph* atau (*DAC)* adalah graf berarah khusus yang tidak memiliki sisi berarah yang *loop* atau berputar.

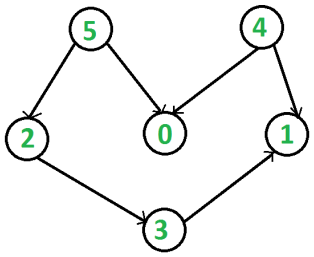


Gambar 2.1.7 *Directed Acyclic Graph*
Sumber : https://hazelcast.com/glossary/directed-acyclic-graph/

DAC biasa digunakan untuk merepresentasikan sebuah *workflow* yang memiliki suatu asal dan menuju tujuan dengan urutan langkah tertentu. Contohnya, representasi *dependency* suatu program.

e. Topological Sorting

Menurut [4], Sebuah algoritma *sorting* khusus untuk graf DAC yang akan menyusun semua simpul dari graf sesuai dengan urutan arah dari graf yang dimaksud.



Gambar 2.1.8 *Directed Acyclic Graph*
Sumber : https://www.geeksforgeeks.org/topological-sorting/

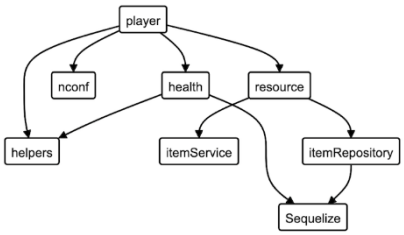
Sebagai contoh, jika gambar diatas dimasukan ke dalam algoritma *Topological Sort* susunan simpul hasil *sorting*-nya adalah 5 4 2 3 1 0 .

B. Dependency

Sesuai dengan [5], dependency adalah istilah yang menyatakan bahwa sebuah program bergantung terhadap program yang lain. Contohnya, jika program X bergantung terhadap program Y maka program A disebut *dependant* terhadap program B dan program B adalaah *dependency* dari program A.

Seperti yang dikatakan , *dependency* berdasarkan kontrol *programmer* terbagi menjadi dua yaitu

1. *Dependency* yang kita kontrol
   Merupakan *dependency* yang kita buat sendiri.
2. *Dependency* yang tidak kita kontrol
   Merupakan *dependency* yang berasal dari pihak ketiga.



Gambar 2.2.1 Contoh *dependency* graph
Sumber : https://understandlegacycode.com/blog/safely-restructure-codebase-with-dependency-graphs/

C. Package Manager

Sesuai dengan [6], *Package Manager* adalah sebuah program yang akan memudahkan *user* untuk meng-*install,* meng-*upgrade,* ataupun menghapus sebuah *software* yang ada di dalam komputer.

Pengertian *package* sendiri merupakan kumpulan *file* yang dapat berupa sebuah program atau juga sebuah kumpulan program.

Ada beberapa istilah yang mengatur hubungan suatu *package* dengan *package* yang lain yaitu

1. *Depend*
   Suatu *package* bergantung pada *packages* yang lain.
2. *Recommends*
   Packages yang direkomendasi akan menambahkan fitur yang relatif penting.
3. *Suggests*
   Packages ini akan menambahkan fitur yang mungkin digunakan oleh pengguna.
4. *Conflicts*
   Package yang dimaksud tidak dapat di-*install* dengan *package* tertentu karena merupakan variasi dari *package* yang sama.

III. IMPLEMENTASI

Penerapan *topology sort* dilakukan dengan mensimulasi sebuah Package Manager dengan bahasa pemrograman Python dengan struktur *source code* dalam bentuk *tree* seperti berikut.

```
└─── src
     ├─── graph.py
     ├─── main.py
     └─── manager.py
```

A. graph.py

Source code berisi sebuah *class* Graph yang memiliki beberapa *attribute* dan *method*.

```
from collections import defaultdict

class Graph:
    def __init__(self,vertices):
        self.graph = defaultdict(list)
        self.n_v = vertices
        self.list_v = list(range(vertices))

    def addEdge(self,u,v):···

    def topologicalSortUtil(self,v,visited,stack):···

    def topologicalSort(self):···
```

Gambar 3.1.1 *Atrribute* dan *Method* Graph
Sumber : Dokumentasi Pribadi

Attribute graph merupakan representasi graf menggunakan *adjacency list* yang diimplementasikan menggunakan struktur data *dictionary.* Kemudian, terdapat n_v yang berisi jumlah simpul dan list_v yang berisi nama dari setiap simpul.

Penambahan sisi pada graf dilakukan dengan memanggil *method* addEdge yang akan memasukkan simpul u dan v dengan u sebagai *key* dan v sebagai *value* dari *dictionary.*

Implementasi *topological sort* dilakukan dengan menggunakan fungsi rekursif topologicalSortUtil.

```
def topologicalSortUtil(self,v,visited,stack):

    visited[v] = True
    name = self.list_v[v]

    for n in self.graph[name]:
        idx = self.list_v.index(n)
        if visited[idx] == False:
            self.topologicalSortUtil(idx,visited,stack)

    stack.insert(0,name)
```

Gambar 3.1.2 *Method* topologicalSortUtil
Sumber : Dokumentasi Pribadi

Pertama *method* akan menandai bahwa simpul sudah dikunjungi kemudian fungsi akan memanggil dirinya sendiri dengan argument *value* dari simpul awal yang dipanggil. Jika

sudah menempuh semua *edge* dari graf maka nama dari simpul akan dimasukkan ke dalam *stack*.

```
def topologicalSort(self):
    visited = [False]*self.n_v
    stack = []

    for i in range(self.n_v):
        if visited[i] == False:
            self.topologicalSortUtil(i,visited,stack)

    stack.reverse()
    return stack
```

Gambar 3.1.3 *Method* topologicalSort
Sumber : Dokumentasi Pribadi

Method rekursif itu dipanggil saat menjalankan *method* topologicalSort untuk setiap *node* dari *Attribute* list_v. Hasil di *stack* butuh dibalik karena graf *dependency* menunjuk dari *dependant* ke *dependency* sementara *dependency* harus di-*install* terlebih dahulu sehingga hasil *stack* akhir perlu dibalik untuk menghasilkan urutan yang tepat.

B. manager.py

```
class PackageManager():

    global package_list
    global dependency_list
    global installed_list

    def __init__ (self, db_folder, usr_folder):
        self.db_folder = db_folder
        self.usr_folder = usr_folder
        self.g = Graph(0)

    def update_db(self): ···

    def update_installed(self): ···

    def get_package(self, package_name): ···

    def get_dependencies(self, package_name): ···

    def install_package(self, package_name): ···

    def remove_package(self, package_name): ···

    def list_packages(self): ···

    def list_installed(self): ···

    def delete_all(self): ···
```

Gambar 3.2.1 *Atrribute* dan *Method* Package Manager
Sumber : Dokumentasi Pribadi

Class Package Manager mengatur fungsi yang ada pada sebuah *Package Manager* mengenai proses *install* dan *remove packages.* Data-data mengenai *packages* disimpan di *Attribe* Package Manager, seperti daftar *packages* di *database* di dalam package_list dan folder database di dalam db_folder.

Dalam peng-*install*-an suatu program, tidak perlu meng-*install* sebuah *packages* yang tidak dibutuhkan sehingga saat melakukan *topological sort* cukup melakukan kepada upagraf yang sesuai dengan *dependency package* permintaan *user.* Hal tersebut sudah dimplementasikan dalam *method* get_dependencies.

```
def get_dependencies(self, package_name):
    d_list = []
    with open(self.db_folder + "\\" + package_name + ".txt", "r") as f:
        dependency = ''
        for line in f:
            line = line.strip()
            if line.strip(mark) == "dependency":
                continue
            else:
                dependency += line
        if dependency != '':
            for dep in dependency.strip(mark).split(mark):
                d_list.append(dep)
                d_list += self.get_dependencies(dep)

    d_list = list(dict.fromkeys(d_list))

    for dep in d_list:
        self.g.addEdge(package_name, dep)

    return d_list
```

Gambar 3.2.2 *Method* get_dependecies
Sumber : Dokumentasi Pribadi

Method ini akan membaca *dependency* dari nama *package* yang dimasukkan ke dalam fungsi. Setelah mendapatkan *list* dari semua *dependency*, tiap elemen dari *list* tersebut akan dimasukkan ke dalam *method* get_dependecies lagi sehingga terjadi rekursif. *Method* ini juga akan memasukkan sisi dari package_name dan juga d_list ke dalam Graph yang nantinya akan digunakan sebagai data yang akan diurutkan.

Selain itu, jika sebuah *package* yang sudah di-*install* merupakan *dependency* dari *package* yang lain maka *package* yang sudah di-*install* tidak perlu di-*install* kembali oleh *package manager.* Hal itu sudah diimplementasikan pada *method* install_package dengan mengecek ke dalam *file* installed yang ada di dalam *source code.*

C. main.py

```
import manager

cwd = manager.os.getcwd()
db_path = "\\db"
db = cwd + db_path

def menu(): ···

if __name__ == "__main__":
    pm = manager.PackageManager(db, cwd)
    pm.update_db()

    print("Welcome to package manager!")

    while(1):

        try:
            choice = int(menu())
        except ValueError: ···

        if choice == 1: ···
        elif choice == 2: ···
        elif choice == 3: ···
        elif choice == 4: ···
        elif choice == 5: ···
        elif choice == 6: ···
        else: ···

        print()
        print("Press enter to continue...")
        input()
```

Gambar 3.3.1 *Source Code* main.py
Sumber : Dokumentasi Pribadi

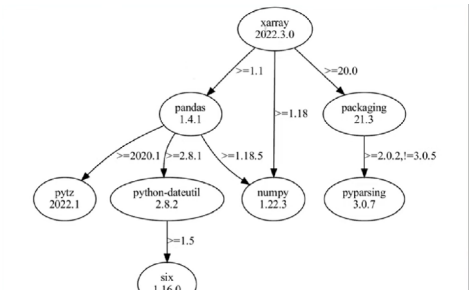
File ini merupakan *interface* yang menghubungkan pengguna dengan program. Di sini pengguna memberikan perintah dasar untuk *package manager* seperti *install package, remove package,* dan melihat daftar *packages* yang tersedia.

IV. PERCOBAAN

Percobaan dilakukan dengan memasukkan data graf ke dalam *database* program dan mencoba meng-*install packages* dari data graf yang sudah dimasukkan.

A. Percobaan 1

Test Case 1 merupakan *dependency* graf dari beberapa *library* di Python.



Gambar 4.1.1 *Test Case* 1
Sumber :
https://www.youtube.com/watch?v=DMS6IIMloZ8

a. *Install* "pyparsing"

```
Input the name of the package: pyparsing
Solving dependencies:
Need to install:
-
Proceed with installation? [Y/n] y
Installing pyparsing

Installation complete
```

Gambar 4.1.2 *Test* a
Sumber : Dokumentasi Pribadi

Sesuai dengan graf *Test Case* 1, pyparsing tidak memiliki *dependency* sehingga tidak ada *package* lain yang perlu di-*install.*

b. *Install* "pandas"

```
Input the name of the package: pandas
Solving dependencies:
Need to install:
pytz
six
python-dateutil
numpy
Proceed with installation? [Y/n] y
Installing pytz
Installing six
Installing python-dateutil
Installing numpy
Installing pandas

Installation complete
```

Gambar 4.1.3 *Test* b
Sumber : Dokumentasi Pribadi

Sesuai dengan graf *Test Case* 1, pandas memiliki *dependency* pytz, python-dateutil, dan numpy. Kemudian, python-dateutil memiliki *dependency* six sehingga diperlukan instalasi *six* terlebih dahulu sebelum python-dateutil yang sudah sesuai dengan output program.

B. Percobaan 2

Test Case 2 merupakan *dependency tree* dari bash yang didapatkan dari *command* pactree.

```
[sans@LAPTOP-CMJR64H3 dependency]$ pactree bash -d 2 -c
bash
├─readline
│ ├─glibc
│ ├─ncurses
│ └─ncurses provides libncursesw.so=6-64
├─readline provides libreadline.so=8-64
├─glibc
│ ├─linux-api-headers>=4.10
│ ├─tzdata
│ └─filesystem
└─ncurses
  ├─glibc
  └─gcc-libs
```

Gambar 4.2.1 *Test Case* 2
Sumber : Dokumentasi Pribadi

a. *Install* "bash"

```
Input the name of the package: bash
Solving dependencies:
Need to install:
linux-api-headers
tzdata
filesystem
glibc
gcc-libs
ncurses
readline
Proceed with installation? [Y/n] y
Installing linux-api-headers
Installing tzdata
Installing filesystem
Installing glibc
Installing gcc-libs
Installing ncurses
Installing readline
Installing bash

Installation complete
```

Gambar 4.2.2 *Test* a
Sumber : Dokumentasi Pribadi

Untuk meng-*install* bash, Instalasi pertama adalah *dependency* dari glibc dan glibc itu sendiri karena glibc merupakan *dependency* dari *packages* yang lain. Kemudian, instalasi *dependency* dari ncurses dan ncurses itu sendiri karena merupakan *dependency* dari dua *packages* lain. Terkahir, instalasi readline karena hanya menjadi *dependency* dari satu package. Hasil *output* program sudah sesuai dengan hasil analisis *tree.*

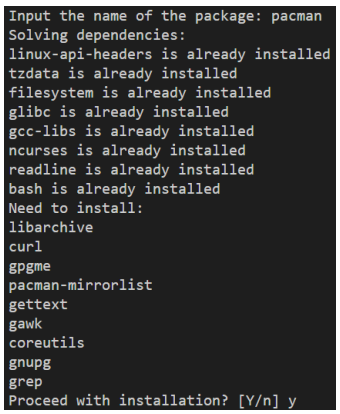
C. Percobaan 3

Test Case 3 merupakan *dependency tree* dari pacman yang didapatkan dari *command* pactree.

```
[sans@LAPTOP-CMJR64H3 dependency]$ pactree pacman -d 1 -c
pacman
├─bash
├─glibc
├─libarchive
├─curl
├─gpgme
├─pacman-mirrorlist
├─gettext
├─gawk
├─coreutils
├─gnupg
└─grep
```

Gambar 4.3.1 *Test Case* 3
Sumber : Dokumentasi Pribadi

a. *Install* "pacman"



Gambar 4.3.2 *Test* a
Sumber : Dokumentasi Pribadi

Untuk meng-*install* pacman, Instalasi pertama adalah bash dan *dependency*-nya, tetapi karena sudah di-*install* dari percobaan sebelumnya sehingga tidak perlu di-*install* kembali. Sementara itu, *packages* yang belum terinstalasi semuanya tidak memiliki *dependency* sehingga tidak perlu dilakukan *topological sorting* cukup di-*install* satu per satu tanpa memperhatikan urutan. Analisis tersebut sudah sesuai dengan *output* program.

V. KESIMPULAN

Topological sorting adalah sebuah algoritma yang mengurutkan *directed acyclic graph* atau DAC sehingga setiap simpul pada graf tersebut tersususun sesuai dengan arah dari tiap simpulnya. Salah satu aplikasi dari *topological sorting* adalah untuk mengatasi *dependency* pada sebuah program.

Untuk mengimplementasinya, program yang mensimulasikan *package manager* dibuat dengan menggunakan bahasa pemrograman Python. Dari hasil analisis, program yang dibuat sudah dapat menentukan urutan instalasi *package* yang seusai yaitu menginstalasi *package* yang berada di ”ujung” graf atau yang tidak memiliki *dependency* kemudian diikuti dengan tetangga dari *package* tersebut hingga sampai ke *package* awal. Selain itu, implementasi hanya menginstalasi *package* yang dibutuhkan yaitu hanya menginstalasi *package* yang dibutuhkan yaitu *package* yang belum di-*install* dan *dependency package*-nya saja.

VI. UCAPAN TERIMA KASIH

Penulis mengucapkan puji syukur kepada Tuhan Yang Maha Esa karena berkat izin-Nya pengerjaan makalah Matematika Diskrit dengan judul “Penerapan Algoritma *Topological Sorting* untuk Menyelesaikan *Dependency* Suatu Program” telah berhasil diselesaikan. Penulis juga mengucapkan terima kasih dosen pengampu mata kuliah IF 2120 Matematika Diskrit beserta asisten yang turut membantu pelaksanaan mata kuliah.

REFERENSI

[1] R. Munir, "Graf (Bag.1)," informatika.stei.itb.ac.id. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf (diakses pada 12 Desember 2022).

[2] R. Munir, "Graf (Bag.2)," informatika.stei.itb.ac.id. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian2.pdf (diakses pada 12 Desember 2022).

[3] Hazelcast, "Directed Acyclic Graph (DAG)", hazelcast.com. https://hazelcast.com/glossary/directed-acyclic-graph/ (diakses pada 12 Desember 2022).

[4] Geeks for Geeks, "Topological Sorting," geeksforgeeks.org. https://www.geeksforgeeks.org/topological-sorting/ (diakses pada 11 Desember 2022).

[5] Siddiqi, "What are Dependencies in Programming," coderslegacy.com. https://coderslegacy.com/what-are-dependencies-in-programming/ (diakses pada 12 Desember 2022)

[6] Debian, "What is a package manager?" debian.org. https://www.debian.org/doc/manuals/aptitude/pr01s02.en.html (diakses pada 12 Desember 2022)

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, 12 Desember 2022

Yanuar Sano Nur Rasyid/135321110