Aplikasi *Topological Sort* dengan Pendekatan *Depth First Search* pada *Package Manager*

Faris Hasim Syauqi 13519050
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13519050@std.stei.itb.ac.id

***Abstract*—Package Manager adalah sebuah kakas yang banyak digunakan untuk melakukan manajemen terhadap paket-paket perangkat lunak terutama dalam pengembangan dan distribusi perangkat lunak. Sebelum sebuah paket di-instalasi oleh package manager, paket-paket lain yang dibutuhkan oleh paket tersebut juga akan di-instalasi terlebih dahulu, sehingga diperlukan sebuah cara untuk memastikan bahwa paket-paket yang dibutuhkan akan di-instalasi terlebih dahulu. Pada makalah ini akan dibahas mengenai algoritma topological sort dengan pendekatan DFS serta implementasinya pada package manager dengan studi kasus menggunakan Node Package Manager.**

Keywords—package, manajemen, topoplogical, DFS, instalasi

I. Pendahuluan

Di dalam dunia pemrograman, sebuah program yang dibangun biasanya bukanlah terdiri dari satu berkas file saja, melainkan terdiri dari kumpulan modul-modul yang bekerja bersama dalam satu kesatuan. Memecah sebuah program menjadi modul-modul memliki berbagai manfaat, diantaranya untuk meningkatkan modularitas dan *maintainability*. Hal ini dapat meningkatkan kualitas dari proses pengembangan perangkat lunak, yang mana pada akhirnya juga akan meningkatkan kualitas produk perangkat lunak yang akan dibangun.

Modul-modul aplikasi perangkat lunak dikumpulkan dan di kelompokkan menjadi sebuah paket perangkat lunak (*software package*). Suatu paket perangkat lunak terkadang tidak dapat berdidri sendiri, namun membutuhkan adanya paket-paket lain yang disebut sebagai *dependency*. Saat ini banyak paket-paket perangkat lunak yang sudah dipublikasi dan tersedia secara daring. Paket-paket tersebut dapat digunakan oleh siapapun, sehingga ketika ada seseorang yang ingin mengembangkan suatu perangkat lunak, alih-alih menciptakan semuanya dari awal, ia bisa memanfaatkan paket-paket yang sudah ada.

Seiring berkembangnya industri perangkat lunak, upaya untuk mengelola paket-paket perangkat lunak secara manual akan menjadi semakin sulit. Karena alasan inilah *package manager* diciptakan sebagai alat untuk mempermudah dalam melakukan manajemen paket perangkat lunak, seperti memasang, menghapus, memperbarui, atau melakukan konfigurasi paket perangkat lunak. Ada banyak contoh *package manager* yang saat ini banyak digunakan oleh para pengembang perangkat lunak, diantaranya npm, pip, rpm, dan sebagainya.

Hal yang menarik terjadi ketika suatu paket akan dipasang pada komputer. Seperti yang sudah disebutkan sebelumnya bahwa suatu paket dapat tergantung pada paket lain. Sehingga ketika suatu paket akan dipasang, harus bisa dipastikan bahwa semua *dependency*-nya terpenuhi. Diperlukan suatu cara untuk memperoleh urutan paket-paket yang harus di-instalasi yang memenuhi aturan bahwa untuk setiap paket yang dipasang, paket-paket lain yang dibutuhkan oleh paket tersebut harus dipasang terlebih dahulu. Salah satu cara yang dapat diterapkan untuk permasalahan ini adalah dengan menggunakan algortima *topological sort* dengan pendekatan *depth first search.*

II. Landasan Teori

A. Traversal Graf

Algoritma traversal graf adalah algoritam pencarian solusi pada graf dengan menggunakan suatu cara yang sistematik dalam mengunjungi simpul-simpul pada suatu graf[1]. Ada dua jenis algoritma traversal graf, yaitu

- Tanpa informasi (*uninformed/blind search*)

  Algoritma pencarian solusi yang tanpa adanya informasi tambahan pada persoalan. Contohnya yaitu Depth First Search, Breadth First Search, Depth Limited Search, Iterative Deepening Search, Uniform Cost Search.

- Dengan informasi (*informed search*)

  Algoritma pencarian solusi dengan adanya informasi tambahan pada persoalan, yang mana informasi tambahan tersebut dapat diproses secara heuristik. Contohnya yaitu Greedy Best First Search, A*.

Ada dua buah pendekatan yang dapat dilakukan proses pencarian solusi, yaitu

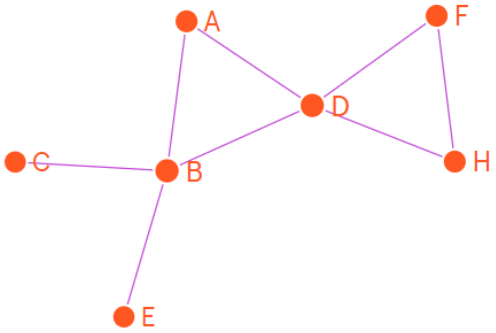
- Graf Statis, yaitu graf yang sudah terbentuk sebelum proses pencarian dilakukan. Graf direpresentasikan sebagai struktur data
- Graf Dinamis, yaitu graf yang terbentuk saat proses pencarian sedang dilakukan. Graf tidak tersedia sebelum dilakukannya pencarian dan dibangun selama proses pencarian.

B. Depth First Search

Algoritma *Depth First Search* adalah sebuah algoritma traversal graf yang mengunjungi simpul-simpul pada graf secara mendalam. Algoritma ini merupakan algoritma traversal graf dengan jenis *uninformed search*.

Berikut ini langkah-langkah algoritma *depth first search* secara umum:

1. Kunjungi simpul awal, misalkan v
2. Kunjungi simpul w yang bertetangga dengan simpul v.
3. Ulangi DFS mulai dari simpul w.
4. Ketika mencapai simpul u sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, pencarian dirunut-balik (backtrack) ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul w yang belum dikunjungi.
5. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.



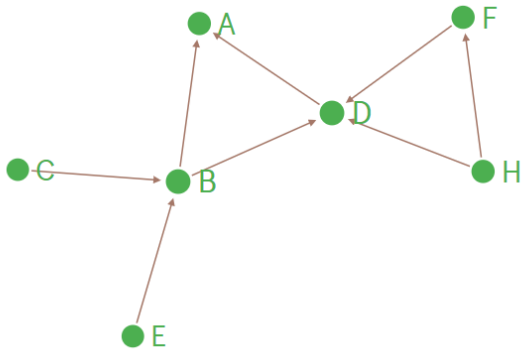
Gambar 1. Contoh penelusuran graf secara DFS (Sumber:Penulis)

Diberikan sebuah graf seperti pada gambar di atas. Misal simpul awal adalah simpul A, maka dengan penelusuran secara DFS, urutan simpul-simpul yang akan dikunjungi adalah A, B, C, D, F, H, E.

C. Topological Sort

Directed Acyclic Graph (DAG) adalah sebuah graf berarah yang tidak mengandung siklus di dalamnya[2]. Di dalam ilmu komputer, DAG banyak diterapkan terutama pada persoalan-persoalan yang berkaitan dengan penjadwalan dan konkurensi.

Topological sort dari suatu DAG adalah suatu larik yang setiap elemennya merupakan simpul pada DAG dan untuk setiap simpul v muncul pada larik sebelum setiap simpul lain yang dapat dicapai dari simpul v. Setiap DAG pasti memiliki *topological sort*[2]. Salah satu contoh DAG dan *topological sort* dapat dilihat pada gambar berikut.



Gambar 2. Contoh Directed Acyclic Graph (DAG) (Sumber:Penulis)

Jika diberikan sebuah DAG seperti gambar di atas, maka salah satu *topological sort* dari DAG tersebut adalah C, E, H, B, F, D, A. Disebut sebagai salah satu disini karena sebuah DAG dapat memiliki lebih dari satu *topological sort*.

D. Package Manager

Package atau yang selanjutnya akan disebut sebagai paket adalah kumpulan program atau modul yang bekerja bersama dalam satu kesatuan dengan tujuan tertentu. Sebuah perangkat lunak yang dibangun biasanya terdiri dari paket-paket perangkat lunak. Adanya paket pada pengembangan perangkat lunak memliki berbagai manfaat, diantaranya yaitu meningkatkan modularitas dan *maintainability*.

Selain *package*, terminologi lain yang penting untuk diketahui adalah *dependency*. *Dependency* adalah paket lain yang dibutuhkan oleh suatu paket supaya paket dapat bekerja. *Dependency* menunjukkan hubungan ketergantungan antar suatu paket dengan paket-paket lainnya.

Package manager adalah sebuah kakas perangkat lunak yang digunakan untuk manajemen paket perangkat lunak secara otomatis. Manajemen paket yang dimaksud terdiri dari pemasangan, pembaruan, konfigurasi, dan penghapusan paket perangkat lunak yang ada pada komputer[3].

Ada banyak *package manager* yang saat ini banyak digunakan dalam pengembangan perangkat lunak, beberapa contoh diantaranya:

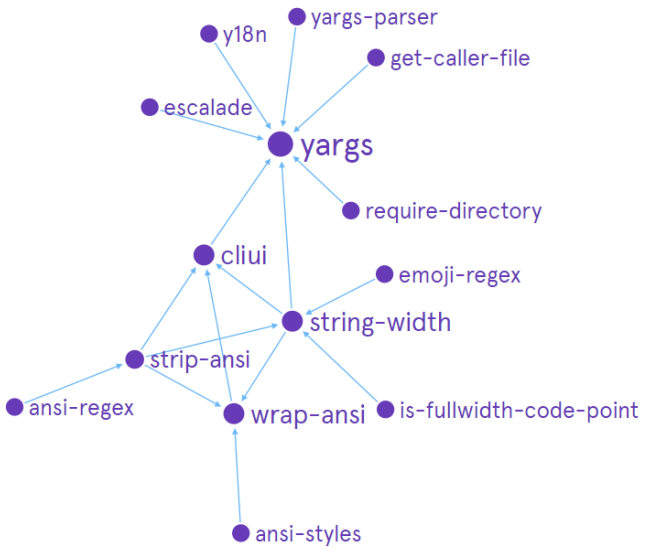
- NPM, adalah sebuah *package manager* untuk bahasa pemrograman JavaScript. npm adalah bagian dari Node.js, yang merupakan platform *runtime environtment* untuk pengembangan JavaScript. percobaan yang dilakukan oleh penulis dan dibahas pada makalah ini menggunakan npm sebagai *package manager*.
- APT, adalah sebuah *package manager* untuk sistem operasi linux berbasis debian, seperti Debian, Ubuntu dan yang lainnya.
- PIP, adalah sebuah *package manager* untuk *library* dan bahasa pemrograman Python.

III. Analisis Permasalahan

Ketika *package manager* melakukan instalasi suatu paket perangkat lunak, selain melakukan instalasi paket tersebut, *package manager* juga akan melakukan instalasi paket-paket lain yang dibutuhkan oleh paket tersebut, atau dalam kata lain paket-paket lain yang menjadi *dependency* akan juga dipasang oleh *package manager*. Ketika suatu paket akan dipasang, maka paket-paket lain yang menjadi *dependency* juga harus terpasang sebelumnya, sehingga untuk mencapai hal ini, diperlukan sebuah cara untuk menghasilkan urutan paket-paket yang harus dipasang oleh *package manager*. Persoalan ini dapat diselesaikan dengan menerapkan algoritma *topological sort* dengan pendekatan DFS.

Sebelum menerapkan *topological sort*, persoalan tersebut akan dimodelkan terlebih dahulu menjadi sebuah DAG. Setiap simpul pada graf akan merepresentasikan sebuah paket. Kemudian setiap sisi berarah pada graf akan merepresentasikan hubungan ketergantungan antar paket. Jika pada graf terdapat suatu sisi dari simpul u ke simpul v, maka dapat diartikan paket v bergantung pada paket u, atau dalam kata lain paket u adalah *dependency* dari paket v.

Berikut ini adalah contoh instans dari persoalan ini, dengan menggunakan npm sebagai *package manager* dan paket yargs sebagai paket yang akan di-instalasi.



Gambar 3. Graf ketergantungan paket yargs
(Sumber:Penulis)

Pada gambar di atas dapat terlihat bahwa paket yargs yang akan dipasang, memliki ketergantungan pada beberapa paket lainnya, diantaranya paket *y18n*, *cliui*, *string-width*, *yargs-parser*, dan selanjutnya. Hal ini ditandai dengan adanya sisi berarah yang keluar dari simpul-simpul tersebut masuk ke simpul yargs. Kemudian dapat terlihat juga bahwa paket cliui, yang merupakan *dependency* bagi paket yargs, juga memilki ketergantungan pada paket-paket lainnya.

Algoritma *topological sort* dapat diterapkan pada persoalan ini untuk menghasilkan larik yang merepresentasikan urutan paket-paket yang harus dipasang oleh *package manager* sehingga dapat dipastikan *package manager* akan melakukan instalasi paket-paket yang menjadi *dependency* terlebih dahulu sebelum paket-paket yang lain.

Langkah-langkah algoritma topological sort yang dapat diterapkan dengan pendekatan DFS adalah sebagai berikut.

1. Kunjungi simpul awal, misalnya v
2. Tandai v sudah dikunjungi.
3. Inisiasikan sebuah larik kosong
4. Untuk setiap simpul u yang menjadi *dependency* bagi simpul v, dan u belum dikunjungi, maka lakukan topological sort pada u, dan masukkan hasilnya pada larik.
5. Jika tidak ada lagi simpul *dependency* yang harus dikunjungi, maka masukkan simpul v sebagai elemen terakhir larik.
6. Kembalikan larik tersebut yang merupakan *topological sort* dari persoalan.

IV. Implementasi & Percobaan

A. Hasil Implementasi

Berikut ini adalah hasil implementasi dari algoritma *topological sort* yang sudah dijelaskan sebelumnya

```
def topological_sort(graph: nx.DiGraph, pkg_name: str, visited: set) -> list:

    # tandai simpul sudah dikunjungi
    visited.add(pkg_name)

    # inisiasi larik kosong
    result = list()

    for edge in graph.edges:
        if edge[1] == pkg_name:
            if edge[0] not in visited:
                # untuk setiap sisi berarah yang keluar simpul u
                # dan masuk simpul ini, dan u belum dikunjungi
                # lakukan topological sort untuk simpul u
                result = result + topological_sort(graph, edge[0], visited)

    result.append(pkg_name)
    return result
```

Gambar 4. Hasil implementasi topological sort
(Sumber:Penulis)

Dengan menggunakan instan persoalan yang sama seperti yang sudah dijelaskan pada bab sebelumnya, program hasil implementasi tersebut akan diuji untuk mendapatkan hasil dari algoritma *topological sort* untuk persoalan tersebut. Hasil dari percobaan tersebut dapat dilihat pada gambar di bawah ini.

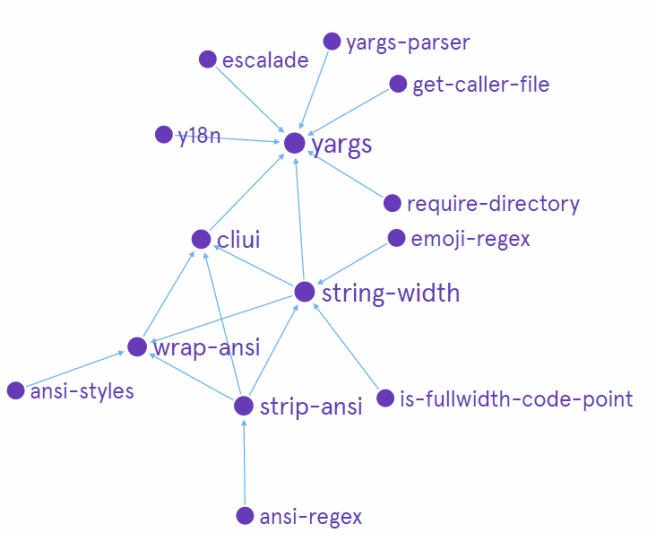
```
PS C:\Users\HASIM\Kuliah\04-Stima\Makalah\npm-topological-sort> python npm.py

Masukkan nama package yang akan di-instal: yargs

Urutan instalasi:

emoji-regex
is-fullwidth-code-point
ansi-regex
strip-ansi
string-width
ansi-styles
wrap-ansi
cliui
escalade
get-caller-file
require-directory
y18n
yargs-parser
yargs
```

Gambar 5. Hasil Percobaan Program (Sumber:Penulis)

Pada gambar di atas dapat dilihat bahwa ketika paket yargs akan di-instalasi, paket-paket lain yang menjadi *dependency* bagi yargs akan juga di-instalasi dengan urutan yaitu *emoji-regex, is-fullwidth-code-point, ansi-regex, strip-ansi, string-width, ansi-styles, wrap-ansi, cliui, escalade, get-caller-file, require-directory, y18n, yargs-parser, yargs*.

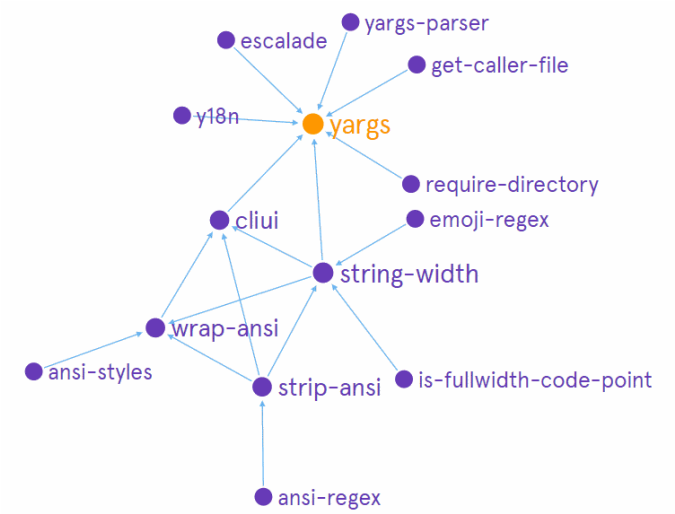
B. Langkah-langkah Penelusuran

Algoritma yang diterapkan merupakan algoritma traversal graf dengan pendekatan graf statis, sehingga pada keadaan awal, graf sudah dibentuk dan belum ada satu pun simpul yang dikunjungi. Keadaan awal graf dapat dilihat pada gambar di bawah ini.



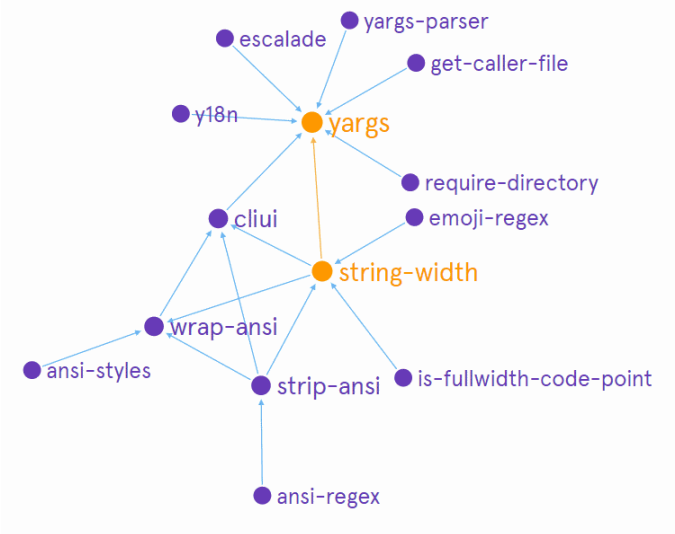
Gambar 6.. Kondisi awal graf (Sumber:Penulis)

Langkah pertama yang akan dilakukan adalah mengunjungi simpul awal. Karena pada awalnya paket yang akan di-instalasi adalah paket yargs, maka penelusuran dilakukan dengan simpul yargs sebagai simpul pertama yang dikunjungi. Status graf saat ini dapat dilihat pada gambar berikut.



Gambar 7.. Kondisi graf saat penelusuran (Sumber:Penulis)

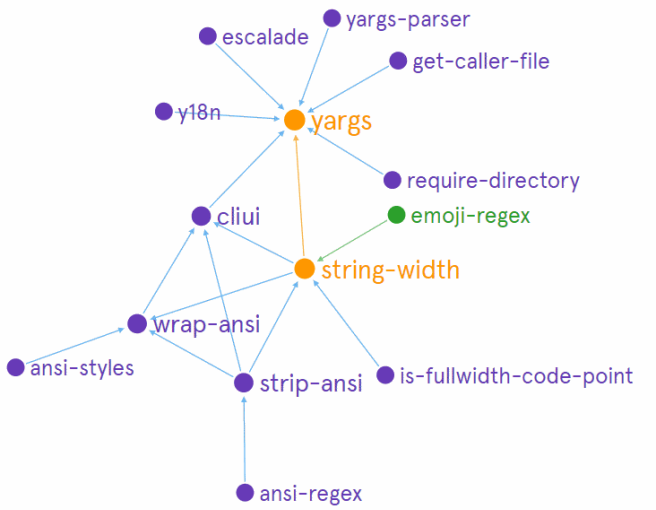
Setelah simpul yargs dikunjungi, sebelum memasukkan paket yargs ke larik, terlebih dahulu akan dilakukan proses *topological sort* untuk setiap paket yang menjadi *dependency* bagi paket yargs. Simpul selanjutnya yang akan dikunjungi adalah simpul *string-width*. Keadaan graf selanjutnya dapat dilihat pada gambar di bawah ini.



Gambar 8.. Kondisi graf saat penelusuran (Sumber:Penulis)

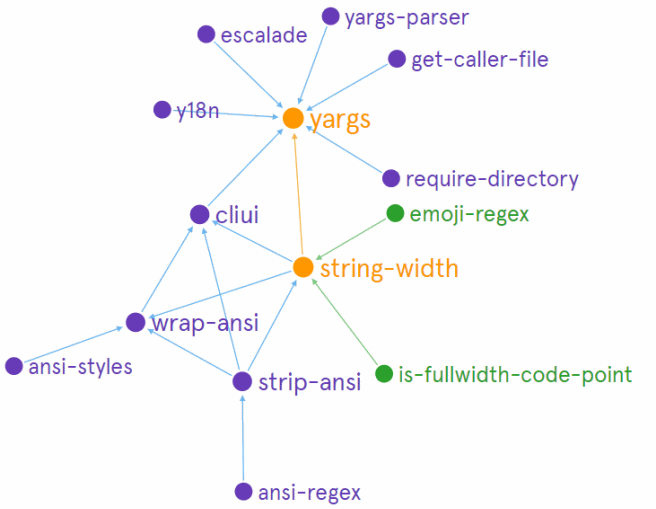
Setelah simpul *string-width* dikunjungi, simpul-simpul lain yang menjadi *dependency* bagi *string-witdh* dan belum dikunjungi akan dikunjungi pada langkah berikutnya. Simpul yang akan dikunjungi selanjutnya adalah simpul emoji-regex. Ketika mengunjungi simpul emoji-regex, dapat dilihat bahwa simpul tersebut tidak memiliki satupun *dependency*, sehingga simpul emoji-regex dapat dimasukkan ke larik, kemudian akan dilakukan *backtrack* ke simpul string-width, sehingga

menghasilkan kondisi graf yang dapat dilihat pada gambar di bawah ini.



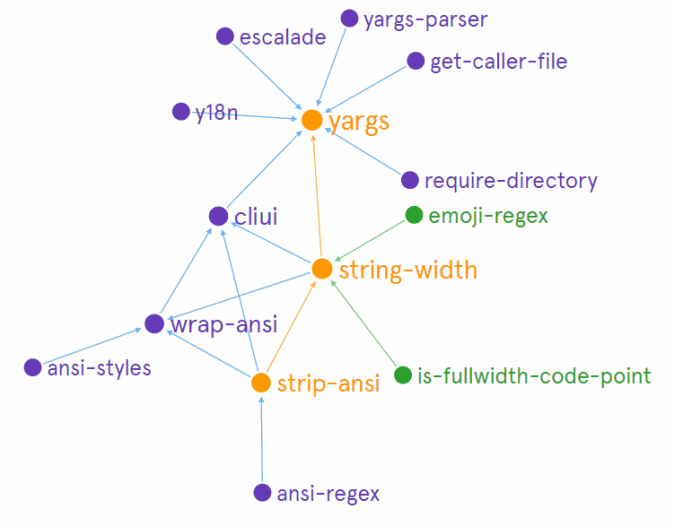
Gambar 9.. Kondisi graf saat penelusuran
(Sumber:Penulis)

Setelah *backtrack* ke simpul *string-width*, selanjutnya akan dilakukan penelusuran ke simpul *is-fullwidth-code-point* yang merupakan *dependency* bagi simpul *string-width*. Sama seperti simpul *emoji-regex* sebelumnya, simpul ini juga tidak memiliki *dependency*, sehingga langkah selanjutnya adalah masukkan simpul ini ke dalam larik, kemudian *backtrack* ke simpul *string-width*. Status penelusuran dapat dilihat pada gambar berikut.



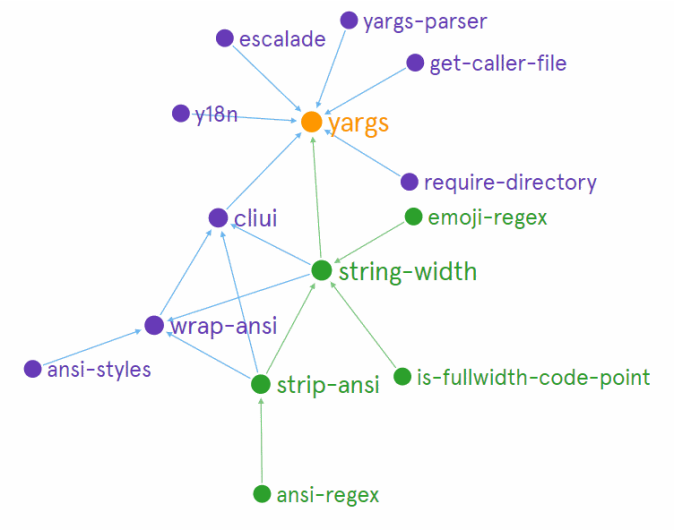
Gambar 10.. Kondisi graf saat penelusuran
(Sumber:Penulis)

Setelah kembali lagi ke simpul *string-width*, langkah penelusuran selanjutnya adalah mengunjungi simpul *strip-ansi* yang merupakan *dependency* bagi simpul *string-width*. Kondisi penelusuran saat ini dapat dilihat pada gambar dibawah ini.



Gambar 11.. Kondisi graf saat penelusuran
(Sumber:Penulis)

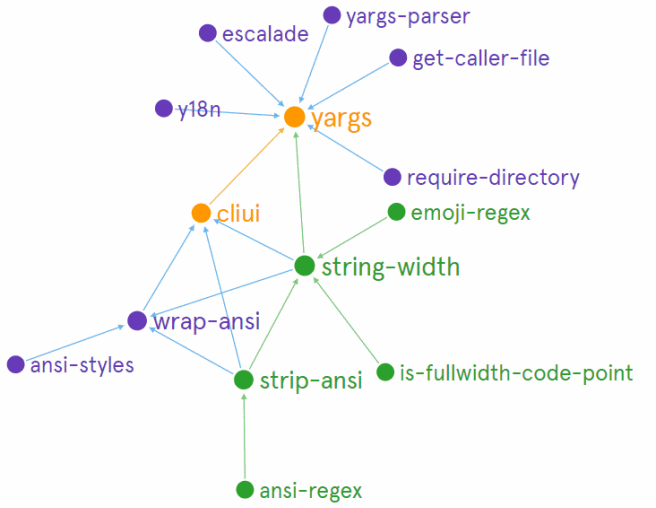
Ketika simpul *strip-ansi* sudah dikunjungi, maka langkah selanjutnya adalah mengunjungi simpul lain yang menjadi *dependency* bagi simpul *strip-ansi* yaitu simpul *ansi-regex*. Kemudian dapat dilihat pada graf bahwa simpul *ansi-regex* tidak memiliki satupun *dependency*, sehingga simpul *ansi-regex* dapat dimasukkan ke dalam larik, kemudian *backtrack* ke simpul *strip-ansi*. Kemudian saat kembali ke simpul *strip-ansi* juga tidak ditemukan adanya simpul *dependency* bagi simpul *strip-ansi* yang perlu dikunjungi, sehingga simpul tersebut akan dimasukkan ke dalam larik dan akan dilakukan *backtrack* ke simpul *string-width*. Karena pada simpul *string-width* juga tidak ditemukan simpul *dependency* yang belum dikunjungi maka selanjutnya akan dilakukan *backtrack* ke simpul yargs, yang mana merupakan simpul pertama yang dikunjung saat proses penulusuran dimulai. Kondisi graf saat ini dapat dilihat pada gambar berikut.



Gambar 12.. Kondisi graf saat penelusuran
(Sumber:Penulis)

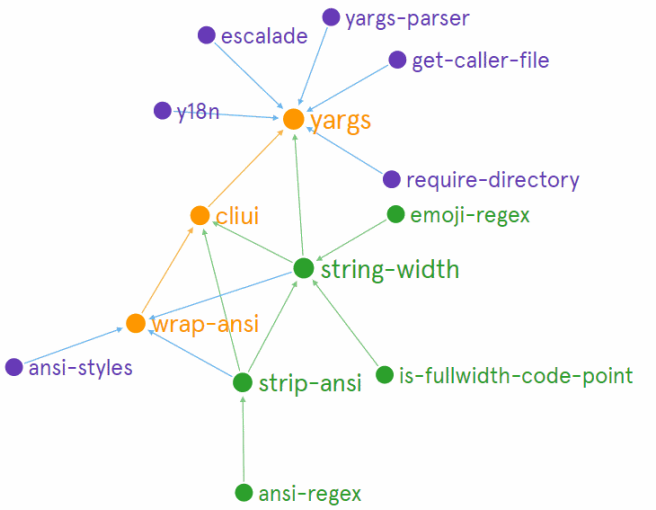
Pada saat ini, perlu dicatat bahwa elemen-elemen pada larik secara berurutan dari elemen pertama terdiri dari *emoji-regex, is-fullwidth-code-point, ansi-regex, strip-ansi dan string-width.*

Proses penelusuran selanjutnya yang akan dilakukan adalah mengunjungi simpul lain yang belum dikunjungi dan merupakan *dependency* dari simpul *yargs*. Simpul selanjutnya yang akan dikunjungi adalah simpul *cliui*. Status graf saat ini dapat dilihat pada gambar berikut.



Gambar 13.. Kondisi graf saat penelusuran
(Sumber:Penulis)

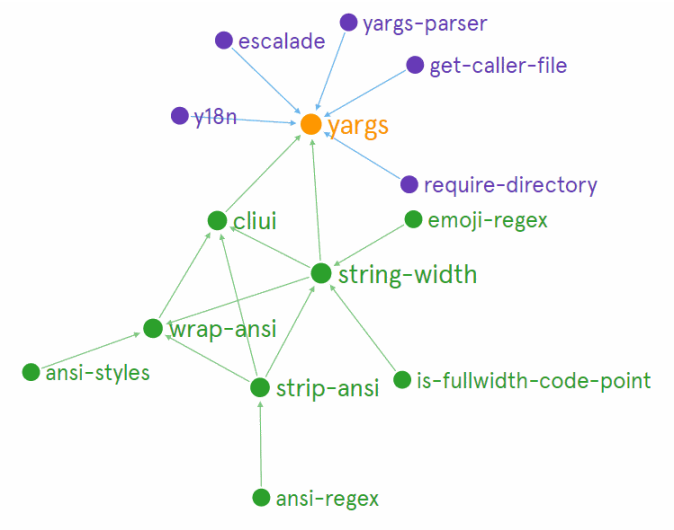
Dari gambar tersebut, dapat terlihat bahwa simpul *cliui* memiliki beberapa simpul *dependency*. Diantara yang menjadi simpul dependency bagi *cliui* adalah simpul *string-width, strip-ansi* dan *wrap-ansi.* Namun karena simpul *string-width* dan *strip-ansi* sudah dikunjungi, maka simpul selanjutnya yang akan dikunjungi adalah simpul *wrap-ansi.* Kondisi graf selanjutnya dapat dilihat pada gambar di bawah ini.



Gambar 14.. Kondisi graf saat penelusuran
(Sumber:Penulis)

Simpul yang saat ini sedang dikunjungi adalah simpul *wrap-ansi.* Simpul tersebut memiliki beberapa simpul *dependency*, namun simpul dependency yang belum dikunjungi hanyalah simpul *ansi-styles* sehingga simpul selanjutnya yang akan dikunjungi adalah simpul *ansi-styles.*

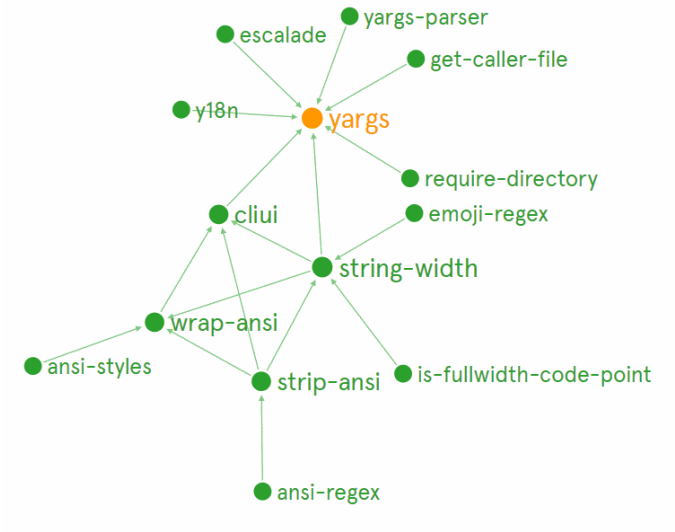
Simpul *ansi-styles* tidak memiliki satu pun *dependency* sehingga simpul tersebut dapat dimasukkan ke dalam larik dan dilakukan *backtrack* ke simpul wrap-ansi. Semua simpul dependency untuk wrap-ansi sudah dikunjungi, sehingga *wrap-ansi* akan dimasukkan ke dalam larik dan akan dilakukan *backtrack.* Begitu juga dengan simpul *cliui*, sehingga status penelusuran saat ini kembali lagi ke simpul *yargs.* Kondisi penelusuran saat ini secara visual dapat dilihat pada gambar berikut.



Gambar 15.. Kondisi graf saat penelusuran
(Sumber:Penulis)

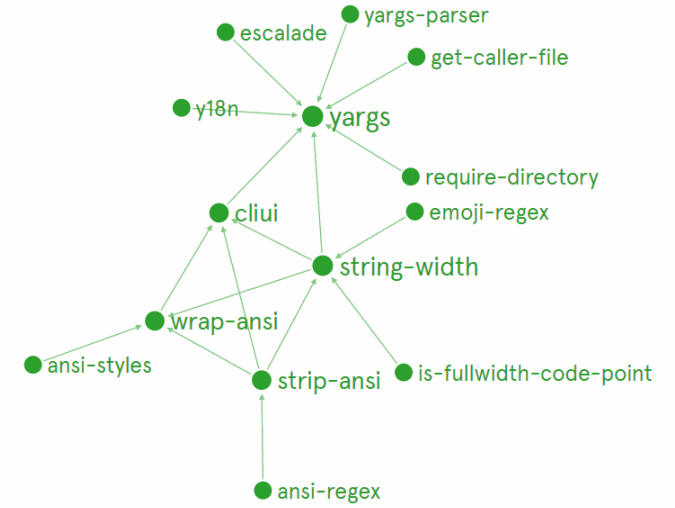
Setelah dilakukan proses DFS pada simpul *cliui* dan *backtrack* ke simpul *yargs*, elemen-elemen larik saat ini secara berturut-turut dari elemen pertama terdiri dari *emoji-regex, is-fullwidth-code-point, ansi-regex, strip-ansi, string-width, ansi-styles, wrap-ansi, dan cliui.*

Pada saat ini, status penelusuran kembali ke simpul yargs dan simpul-simpul yang tersisa untuk dikunjungi diantaranya yaitu simpul *escalade, get-caller-file, require-directory, y18n, dan yargs-parser*. Untuk setiap simpul tersebut akan dikunjungi satu per satu dengan proses penelusuran yang sama seperti sebelum-sebelumnya. Simpul-simpul tersebut tidak memiliki *dependency*, sehingga untuk setiap simpul dari simpul-simpul tersebut, simpul akan dikunjungi, kemudian simpul akan dimasukkan ke dalam larik, kemudian *backtrack* ke simpul yargs, untuk memproses simpul *dependency* lainnya.



Gambar 16.. Kondisi graf saat penelusuran (Sumber:Penulis)

Karena setiap simpul *dependency* bagi simpul yargs sudah dikunjungi dan dimasukkan ke dalam larik terurut sesuai dengan aturan *dependency*, maka selanjutnya simpul yargs dapat dimasukkan ke dalam larik. Sehingga kondisi akhir dari penulusuran ini dapat dilihat pada gambar berikut.



Gambar 17.. Kondisi graf saat penelusuran (Sumber:Penulis)

Setelah proses penulusuran berakhir, maka algoritma topological sort ini akan menghasilkan sebuah larik yang terdiri dari simpul-simpul, dimana untuk semua simpul v muncul pada larik terlebih dahulu sebelum simpul-simpul lain yang dapat dicapai dari v. Pada kasus ini, elemen-elemen larik yang dihasilkan terdiri dari *emoji-regex, is-fullwidth-code-point, ansi-regex, strip-ansi, string-width, ansi-styles, wrap-ansi, cliui, escalade, get-caller-file, require-directory, y18n, yargs-parser, dan yargs*.

V. KESIMPULAN

Algoritma *topological sort* dan *depth first search* memiliki banyak penerapan di berbagai bidang, baik pada bidang ilmu komputer, pada persoalan-persoalan matematis, maupun pada problematika kehidupan sehari-hari. Salah satu contoh penerapan dari algoritma tersebut ialah pada cara kerja sebuah *package manager*. Ketika *package manager* akan melakukan instalasi suatu paket perangkat lunak, *package manager* tidak hanya melakukan instalasi paket tersebut, tetapi juga maleakukan instalasi paket-paket lain yang menjadi *dependency* sebelum paket tersebut. Algoritma *topological sort* dengan pendekatan DFS digunakan untuk menghasilkan urutan paket-paket yang harus di-instalasi oleh *package manager*.

VIDEO LINK AT YOUTUBE

https://youtu.be/ZmTR8nMJvQo .

UCAPAN TERIMA KASIH

Puji syukur kepada Allah atas rahmat dan karunia-Nya, penulis dapat menyelesaikan makalah dengan judul "Aplikasi *Topological Sort* dengan Pendekatan *Depth First Search* pada *Package Manager*". Penulis juga menghaturkan ucapan terima kasih kepada berbagai pihak yang sudah membantu dalam penulisan makalah ini, kepada Dr. Eng. Rila Mandala, selaku dosen pengampu mata kuliah IF2211 Strategi Algoritma kelas K1 semester 2 tahun ajaran 2020/2021, kepada Dr. Ir. Rinaldi Munir, MT. yang telah menyediakan situs yang bermanfaat dan membantu dalam penyusunan makalah ini, juga kepada pihak-pihak lainnya yang tidak bisa disebutkan satu-persatu.

DAFTAR PUSTAKA

[1] R. Munir and N. U. Maulidevi, "Breadth / Depth First Search," no. Bagian 1, pp. 1–29, 2021.

[2] MIT, "Directed Acyclic Graphs (DAGs) & Scheduling: Chapter 9.5," pp. 335–344, 2015.

[3] A. Varun, R. M. Patil, P. Kantanavar, and G. Shobha, "A Comparative Study of various Linux Package-Management Systems," vol. 12, no. 1, pp. 37–44, 2019.

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 Mei 2021

Faris Hasim Syauqi
13519050