Penerapan BFS dalam Mengenerasi Perintah Kompilasi C++

Bintang Dwi Marthen - 13521144
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail) : marthen.bintangdwi@gmail.com

***Abstract*—Bahasa pemrograman C Plus Plus adalah bahasa pemrograman paling populer keempat di dunia. Dalam penggunaannya, pemrogram harus memasukkan perintah kompilasi ke *terminal*. Perintah kompilasi yang dibutuhkan akan menjadi semakin rumit dibuat seiring dengan meningkatnya kompleksitas program. Secara umum, perintah kompilasi C Plus Plus mengandung seluruh berkas C Plus Plus yang digunakan oleh program. Untuk menelusuri seluruh berkas yang digunakan, dapat digunakan penelusuran secara BFS untuk mendapatkan seluruh berkas. Algoritma yang digunakan adalah BFS bukan DFS untuk menghindari diperlukannya pengecekan *circular dependencies* pada program C Plus Plus yang dibuat. Penelusuran menggunakan berkas yang dinyatakan dalam perintah *include* pada awal berkas C Plus Plus.**

Kata Kunci*—*C Plus Plus; Perintah Kompilasi; BFS

I. Pendahuluan

Terdapat berbagai bahasa pemrograman yang dipelajari oleh mahasiswa Teknik Informatika selama masa perkuliahannya. Pada juruan Teknik Informatika di Institut Teknologi Bandung, bahasa pemrograman C++ digunakan dalam mata kuliah IF2211 Strategi Algoritma, IF2210 Pemrograman Berorientasi Objek, IF3210 Pengembangan Aplikasi pada Platform Khusus, dan lainnya. Selain itu, C++ juga merupakan bahasa terpopuler keempat di dunia saat ini berdasarkan data yang didapatkan dari jumlah *pull requests* dan *pushes* pada GitHub.

Selain aktif digunakan oleh mahasiswa Teknik Informatika, bahasa pemrograman C++ juga merupakan bahasa pemrograman paling populer keempat di dunia. Keberhasilan dan popularitas C++ dilatarbelakangi oleh keluwesan penggunaannya. C++ dapat digunakan dalam pengembangan game, pengembangan sistem operasi, grafika komputer, sistem tertanam, dan lainnya.

Dalam penggunaannya, pemrogram harus mengompilasi program setelah diprogram. Dalam kompilasinya, pemrogram harus menyatakan seluruh berkas kode yang akan dikompilasi menjadi satu kesatuan sehingga bila program yang diprogram terdiri atas berbagai berkas maka perintah kompilasi akan menjadi panjang dan sulit dibuat. Oleh karena itu, seringkali dibuat sebuah berkas Makefile untuk memudahkan pemrogram dalam mengompilasi program C++. Akan tetapi, dalam membuat berkas Makefile tersebut diperlukan perintah kompilasi juga sehingga pemrogram harus dapat mengenerasikan setidaknya perintah kompilasi yang tepat sekali.

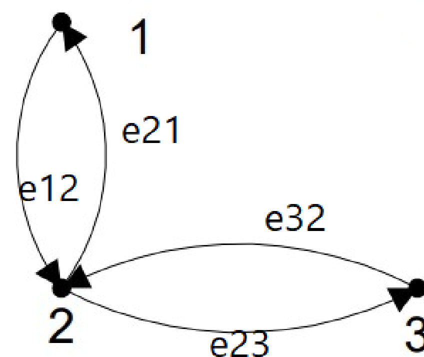
Dalam C++, *#include* dapat direpresentasikan dalam suatu graf. Oleh karena itu, algoritma penulusuran graf dapat digunakan untuk menelusuri seluruh *#include* pada suatu program C++ untuk menemukan seluruh berkas yang perlu dinyatakan dalam perintah kompilasi. Salah satu algoritma penulusuran graf yang dapat digunakan adalah BFS karena pendekatannya yang lebih terurut dalam menentukan *#include* dan penanganan *cyclic dependencies* yang lebih baik.

II. Teori Dasar

A. Graf

Graf dapat didefinisikan sebagai sebuah pasangan himpunan simpul-simpul (himpunan tidak-kosong) dan sisi yang menghubungkan simpul. Simpul pada graf dapat memiliki arah dan bobot, maupun tidak. Karena pendefinisiannya sebagai himpunan simpul dan sisi, graf dapat direpresentasikan dalam sebuah matriks ketetanggaan, matriks bersisian, dan senarai ketetanggaan.

Berikut merupakan contoh representasi graf dalam matriks ketetangaan, matriks bersisian, dan senarai ketetanggaan berdasarkan graf di bawah.



Gambar 1. Contoh Graf Berarah Tak Berbobot

Sumber: Diktat Kuliah *Graf (Bag 1)*

	1	2	3
1	0	1	0
2	1	0	1
3	0	1	0

Matriks Ketetanggaan

	e12	e21	e23	e32
1	1	0	0	0
2	0	1	1	0
3	0	0	0	1

Matriks Bersisian

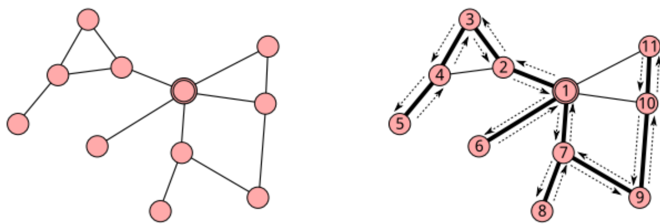
Simpul	Simpul Tetangga
1	2
2	1, 3
3	2

Senarai Ketetanggaan

B. *Penjelajahan Graf*

Penjelajahan graf merupakan penelusuran simpul-simpul pada suatu graf menurut suatu aturan tertentu. Terdapat dua metode umum dalam penjelahan graf: *breadth-first search* (BFS) dan *depth-first search* (DFS). Kedua jenis penjelajahan tersebut merupakan penjelajahan tanpa informasi atau disebut juga dengan *blind search* atau *uninformed search*. Pada DFS, terdapat pula modifikasi penelusuran kedalaman pencarian yang dikenal dengan algoritma *Iterative Deepening Search* yang batas kedalaman pencarian DFS akan semakin mendalam dengan meningkatnya iterasi. Terdapat pula penjelajahan yang lebih teredukasi seperti A*, *Uniform Cost Search* (UCS), dan *Greedy Best First Search*.

C. *Depth-First Search*

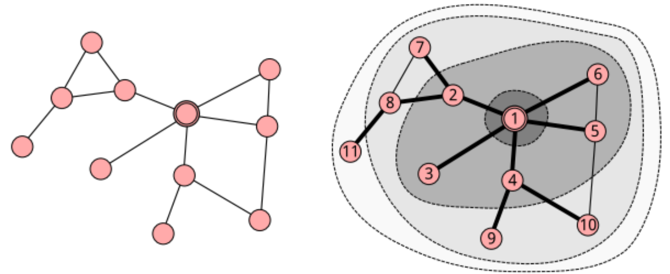


Gambar 2. Ilustrasi Penjelajahan DFS

Sumber: Buku Pemrograman Kompetitif Dasar

Depth-first search, sesuai namanya merupakan penelusuran simpul-simpul pada graf secara mendalam terlebih dahulu. Yang dimaksud dengan mendalam terlebih dahulu adalah pengunjungan akan menuju ke *child node* dari suatu simpul terlebih dahulu sebelum ke *sibling node*-nya. Selama masih terdapat simpul tetangga yang belum dikunjungi, tetangga tersebut akan dikunjungi terlebih dahulu. Ketika sudah mencapai suatu simpul yang seluruh tetangganya telah dikunjungi maka kembali ke simpul sebelumnya hingga kembali ke simpul yang masih memiliki tetangga yang belum dikunjungi. Dalam implementasinya, DFS seringkali menggunakan struktur data *stack* atau juga secara rekursif karena pemanggilan fungsi bersifat seperti *stack* juga. Beberapa permasalahan umum yang diselesaikan dengan DFS antara lain: mendeteksi siklus pada graf, mencari apakah ada rute antara dua simpul, pengecekan bipartit, *topological sort*, dan *web crawlers*. Dari persoalan-persoalan yang diselesaikan oleh DFS, DFS biasanya digunakan untuk menyelesaikan persoalan apakah ada sesuatu ataupun urutan tertentu.

D. *Breadth-First Search*



Gambar 3. Ilustrasi Penjelajahan BFS

Sumber: Buku Pemrograman Kompetitif Dasar

Breadth-first search, sesuai dengan namanya merupakan penelusuran simpul-simpul pada graf yang dilakukan secara melebar atau lapis demi lapis. Dalam implementasinya, BFS biasanya menggunakan struktur data *queue*. *Queue* yang digunakan menyimpan daftar simpul yang akan dikunjungi selanjutnya oleh algoritma BFS. Karena sifatnya yang menelusuri lapis demi lapis, BFS dapat digunakan untuk menentukan jarak dari satu simpul ke semua simpul lainnya. Akan tetapi, BFS lebih sulit diimplementasikan dibandingkan dengan DFS. Beberapa permasalahan yang umumnya menggunakan BFS antara lain: mengecek apakah semua simpul dapat dikunjungi pada graf, waktu minimum untuk mengunjungi semua simpul pada graf setidaknya sekali, dan waktu minimum untuk menginfeksi seluruh simpul pada sebuah *binary tree*. Melalui persoalan-persoalan yang umumnya diselesaikan oleh BFS, BFS biasanya digunakan sebagai algoritma untuk mencari solusi optimal.

E. *Kompilasi C++*

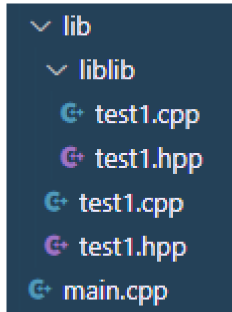
Dalam kompilasi program C++ menjadi suatu berkas yang dapat dieksekusi dapat dipecah menjadi dua tahap: pengompilasian berkas .cpp menjadi berkas objek (.o atau .obj) dan *linking* berkas objek menjadi suatu berkas *executable* (.exe). Pada tahap pengompilasian, *compiler* melakukan analisis lexical, *parsing*, optimisasi, dan mengenerasikan kode yang cocok dengan *hardware* yang cocok (sebagai contoh, kode untuk perangkat dengan processor AMD dan Intel akan berbeda kodenya dalam menangani *float numbers*). Setelah berkas obyek dikompilasi dan digenerasi, *linker* akan menghubungkan seluruh berkas objek tersebut dengan *library-library* yang diperlukan. Sebagai contoh, bila dalam program C++ terdapat "*#include <vector>*" maka *linker* akan menghubungkan seluruh berkas objek tersebut dengan *library* atau modul *vector* milik C++. Umumnya, perintah kompilasi C++ adalah " g++ -o main (seluruh berkas kode cpp) ".

III. Implementasi

A. *Pengolahan Include pada Berkas CPP*

Untuk mendapatkan seluruh *include* dalam suatu berkas CPP, diperlukan suatu proses *parsing*. Salah satu metode *parsing* yang dapat dilakukan adalah menggunakan regex. Regex yang digunakan adalah #include\\s*\"(.*)\". Regex tersebut akan menerima seluruh baris yang diawali dengan "#include", diikuti oleh spasi maupun tidak, dan terdapat kata yang diapit oleh dua petik dua ("). Salah satu contoh yang diterima oleh regex tersebut adalah #include "module1.hpp".

Proses *parsing* ini akan seolah-olah mengenerasikan graf *include* pada program C++. Sebagai contoh, untuk berkas C++ dengan struktur folder sebagai berikut.



Gambar 4. Contoh Struktur Folder Berkas C++

Sumber: Dokumentasi Penulis

Dalam projek tersebut, "main.cpp" melakukan *include* terhadap berkas "test1.hpp" pada *folder* "lib/liblib" dan berkas "test1.hpp" tersebut melakukan *include* terhadap berkas "test1.hpp" pada *folder* "lib". Dengan informasi tersebut, maka graf *include* yang dihasilkan adalah sebagai berikut (dalam senarai ketetanggaan).

Simpul	Simpul Tetangga
main.cpp	lib/liblib/test1.hpp
lib/liblib/test1.hpp	lib/test1.hpp
lib/test1.hpp	-

Dalam *parsing folder* projek C++, terdapat beberapa asumsi yang dibuat:

1. Struktur projek dalam *best-practice*, berkas .cpp dan berkas .hpp diletakkan dalam satu *folder* yang sama dengan nama yang sama (misal ada berkas test1.hpp maka akan terdapat pula berkas test1.cpp pada *folder* yang sama)
2. Seluruh *include* selain *standard library* akan terdapat dalam berkas .hpp
3. Tidak terdapat *circular dependencies* (akan muncul *error message* dari IDE bila terdapat *circuclar dependencies*)

Melalui asumsi itu, maka dapat dibuat sebuah kode untuk melakukan *parsing* untuk mendapatkan seluruh simpul tetangga dari suatu berkas. Berikut merupakan *pseudocode* untuk *parsing* tersebut.

```
function getAllIncludes (string filePath)
    includes ← []
    for (setiap baris dalam berkas)
        if ( baris match ""#include\\s*\"(.*)\"")
            includes ← includes + baris
    ➔ includes
```

Berikut merupakan kodenya dalam CPP.

```
vector<string> getAllIncludes(const string& filePath){
    vector<string> includes;
    regex includeRegex("#include\\s*\"(.*)\"");
    ifstream file(filePath);
    if (file.is_open()){
        string line;
        while (getline(file, line)){
            smatch match;
            if (regex_search(line, match, includeRegex)){
                includes.push_back(match[1]);
            }
        }
        file.close();
    } else {
        cout << "Unable to open file: " << filePath << endl;
    }
    return includes;
}
```

B. *Penelusuran Berkas Include*

Setelah seluruh simpul tetangga atau berkas *include* berhasil didapatkan, maka akan dilakukan penelusuran BFS pada simpul – simpul tetangga tersebut. Karena dalam satu projek atau program C++, satu berkas .hpp dapat di-*include* oleh banyak berkas lainnya maka akan dicatat apakah berkas tersebut telah masuk ke dalam himpunan berkas yang di-*include* maka penelusuran pada simpul itu akan dibunuh. Berikut merupakan *pseudoecode* penelusuran BFSnya.

```
procedure bfsIncludes (string berkas)
    includeQueue.push({berkas, parentPath(berkas)})
    while ( not includeQueue.empty() )
```

```
        {x, y} ← includeQueue.pop()
        if x dalam visited:
                continue
        visited.insert(x)
        includes ← getAllIncludes(x)
        for setiap include dalam includes
                if not exist include
                        continue
                includeQueue.push( {include, parentPath(include)} )
    printPerintahKompilasi(visited)
```

Berikut merupakan penelusuran BFS dalam C++.

```
namespace fs = std::filesystem;
string getAbsoluteFilePath(const string& relativePath, const string& currentDir) {
        fs::path absolutePath = fs::path(currentDir) / relativePath;
        return absolutePath.string();
    }

    void bfsOnIncludes(const string& filename) {
        queue<pair<string, string>> includeQueue;  // (file path, current dir)
        set<string> visitedIncludes;

        fs::path filePath = fs::path(filename).lexically_normal();
        includeQueue.push({ filePath.string(), filePath.parent_path().string() });

        while (!includeQueue.empty()) {
            string currentFile = includeQueue.front().first;
            string currentDir = includeQueue.front().second;
            includeQueue.pop();

            if (visitedIncludes.count(currentFile) > 0)
                continue;

            visitedIncludes.insert(currentFile);
            cout << "Processing " << currentFile << endl;

            vector<string> includes = getAllIncludes(currentFile);
            for (const string& include : includes) {
              cout << "Include: " << include << endl;

              string absolutePath = getAbsoluteFilePath(include, currentDir);
              if (!fs::exists(absolutePath)) {
                cout << "Include file not found: " << absolutePath << endl;
                continue;
              }

              includeQueue.push({ absolutePath, fs::path(absolutePath).parent_path().string() });
            }
        }

        cout << "g++ -o main ";

        for (const string& include : visitedIncludes) {
            // if include ends with .hpp change to .cpp
            string includeCpp = include;
            if (includeCpp.size() > 4 && includeCpp.substr(includeCpp.size() - 4) == ".hpp") {
              includeCpp = includeCpp.substr(0, includeCpp.size() - 4) + ".cpp";
            }
            cout << "\"" << includeCpp << "\" ";
        }   cout << endl;
}
```

IV. Analisis

A. *Ketepatan Perintah Kompilasi*

Setelah program berhasil mendapatkan perintah kompilasi yang diingkan, maka kompilasi dapat dilakukan. Berikut merupakan beberapa tangkapan layar uji coba pengujian

program.

```
PS D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Generator> ./generator
Enter the path of the main file (absolute path): D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Genera
tor\test\main.cpp
Processing D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Generator\test\main.cpp
Include: lib/liblib/test1.hpp
Processing D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Generator\test\lib/liblib/test1.hpp
Include: ../test1.hpp
Processing D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Generator\test\lib/liblib\../test1.hpp
g++ -o main "D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Generator\test\lib/liblib/test1.cpp" "D:\K
uliah\Semester 4\Stima\Tugas\C++ Compile Generator\test\lib/liblib\../test1.cpp" "D:\Kuliah\Semester
 4\Stima\Tugas\C++ Compile Generator\test\main.cpp"
PS D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Generator> g++ -o main "D:\Kuliah\Semester 4\Stima\T
ugas\C++ Compile Generator\test\lib/liblib/test1.cpp" "D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile
Generator\test\lib/liblib\../test1.cpp" "D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Generator\test
\main.cpp"
PS D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Generator> ./main
Success
```

Gambar 5. Uji Coba dengan Contoh Kode pada *Repository*

Sumber: Dokumentasi Penulis

```
PS D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Generator> ./generator
Enter the path of the main file (absolute path): D:\Kuliah\Semester 4\Stima\Tugas\tucil-1\Tucil1_135
21144\src\main.cpp
Processing D:\Kuliah\Semester 4\Stima\Tugas\tucil-1\Tucil1_13521144\src\main.cpp
Include: ioHandler.hpp
Include: solver.hpp
Processing D:\Kuliah\Semester 4\Stima\Tugas\tucil-1\Tucil1_13521144\src\ioHandler.hpp
Processing D:\Kuliah\Semester 4\Stima\Tugas\tucil-1\Tucil1_13521144\src\solver.hpp
g++ -o main "D:\Kuliah\Semester 4\Stima\Tugas\tucil-1\Tucil1_13521144\src\ioHandler.cpp" "D:\Kuliah\
Semester 4\Stima\Tugas\tucil-1\Tucil1_13521144\src\main.cpp" "D:\Kuliah\Semester 4\Stima\Tugas\tucil
-1\Tucil1_13521144\src\solver.cpp"
PS D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Generator> g++ -o main "D:\Kuliah\Semester 4\Stima\T
ugas\tucil-1\Tucil1_13521144\src\ioHandler.cpp" "D:\Kuliah\Semester 4\Stima\Tugas\tucil-1\Tucil1_135
21144\src\main.cpp" "D:\Kuliah\Semester 4\Stima\Tugas\tucil-1\Tucil1_13521144\src\solver.cpp"
PS D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Generator> ./main
Input dari user atau random? (user/random): I
```

Gambar 6. Uji Coba dengan Kode Tucil1 Penulis (https://github.com/Marthenn/Tucil1_13521144)

Sumber: Dokumentasi Penulis

```
PS D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Generator> ./generator
Enter the path of the main file (absolute path): D:\Kuliah\Semester 4\OOP\Praktikum\praktikum-2\tool
\main.cpp
Processing D:\Kuliah\Semester 4\OOP\Praktikum\praktikum-2\tool\main.cpp
Include: Pickaxe.hpp
Include: ReinforcedAxe.hpp
Processing D:\Kuliah\Semester 4\OOP\Praktikum\praktikum-2\tool\Pickaxe.hpp
Include: Tool.hpp
Processing D:\Kuliah\Semester 4\OOP\Praktikum\praktikum-2\tool\ReinforcedAxe.hpp
Include: Tool.hpp
Processing D:\Kuliah\Semester 4\OOP\Praktikum\praktikum-2\tool\Tool.hpp
g++ -o main "D:\Kuliah\Semester 4\OOP\Praktikum\praktikum-2\tool\Pickaxe.cpp" "D:\Kuliah\Semester 4\
OOP\Praktikum\praktikum-2\tool\ReinforcedAxe.cpp" "D:\Kuliah\Semester 4\OOP\Praktikum\praktikum-2\to
ol\Tool.cpp" "D:\Kuliah\Semester 4\OOP\Praktikum\praktikum-2\tool\main.cpp"
PS D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Generator> g++ -o main "D:\Kuliah\Semester 4\OOP\Pra
ktikum\praktikum-2\tool\Pickaxe.cpp" "D:\Kuliah\Semester 4\OOP\Praktikum\praktikum-2\tool\Reinforced
Axe.cpp" "D:\Kuliah\Semester 4\OOP\Praktikum\praktikum-2\tool\Tool.cpp" "D:\Kuliah\Semester 4\OOP\Pr
aktikum\praktikum-2\tool\main.cpp"
PS D:\Kuliah\Semester 4\Stima\Tugas\C++ Compile Generator> ./main
tool crafted! 3 2
pickaxe crafted! 3 2
swing! swing! diamonds found!
pickaxe enchanted!
```

Gambar 7. Uji Coba dengan Kode Praktikum Penulis

Sumber: Dokumentasi Penulis

Berdasarkan pengujian tersebut, selama seluruh asumsi pada bagian “III.A.Pengolahan Include pada berkas CPP” dipenuhi maka perintah kompilasi akan berhasil.

B. *BFS dibandingkan DFS*

Algoritma BFS akan menangani seluruh *dependencies* atau *include* dari suatu berkas secara langsung dibandingkan DFS. Pendekatan ini akan menyebabkan perintah kompilasi menjadi lebih terstruktur secara hierarki dan sistemasi. Hal ini menyebabkan program dapat mendeteksi berkas *include* yang tidak ada lebih awal dibandingkan dengan DFS.

Dalam program C++, terkadang terdapat *circular dependencies*. *Circular dependencies* terjadi ketika terdapat dua atau lebih berkas yang melakukan *include* dalam suatu siklus. Bila tidak terdapat asumsi tidak akan terjadi *circular dependencies*, BFS akan menghindari *circular dependencies* tersebut dengan lebih natural dibandingkan dengan DFS. BFS dapat menghindari dengan lebih natural pendekatannya yang tahap per tahap.

Selain itu, dengan menggunakan BFS program menghindari rekursi yang terlalu dalam. Untuk program C++ yang cukup besar, DFS akan menelusuri hingga kedalaman yang terlalu dalam sehingga tingkat rekursinya akan dalam. Akan tetapi, pendekatan BFS yang tingkat per tingkat akan menghindari rekursi yang terlalu dalam. Rekursi yang terlalu dalam dapat menyebabkan *stack overflow* pada program sehingga menghindarinya menjadi hal yang ideal.

C. *Kompleksitas Waktu dan Ruang*

Secara umum, program dapat dibagi menjadi tiga tahapan: mendapatkan seluruh simpul tetangga, melakukan BFS, dan mencetak perintah kompilasi. Dalam menghitung kompleksitas waktu dan ruang, tahap pencetakan perintah kompilasi ke layar atau *terminal* tidak akan dihitung. Dengan asumsi terdapat N buah berkas yang di-*include* dan terdapat rata-rata M baris pada setiap berkas maka kompleksitas waktu dari program akan menjadi O(NM). Kompleksitas waktu tersebut didapatkan dengan perhitungan bahwa tidak akan dilakukan *parsing* pada berkas yang sama lebih dari sekali dan akan dilakukan pengecekan pada setiap baris dari berkas.

Untuk kompleksitas ruang dari program dapat dihitung dengan menkonsiderasi dua struktur data yang digunakan: himpunan *include* yang telah dicatat dan *queue* untuk berkas yang akan dicek. Pada himpunan *include* yang telah dicatat, isinya akan sebanyak seluruh berkas yang di-*include* sehingga menjadi O(N). Pada *queue,* pada kasus terburuk akan mencatat seluruh berkas yang di-*include* sekaligus kecuali berkas “main.cpp” (pada kasus ini “main.cpp” meng-*include* seluruh berkas lainnya) sehingga kompleksitasnya menjadi O(N). Oleh karena itu, kompleksitas ruang dari program ini adalah O(N).

V. Penutup

A. *Kesimpulan*

Algoritma BFS dapat digunakan untuk mengenerasi perintah kompilasi program C++. Dengan program yang dihasilkan dengan program C++, pemrogram tidak perlu lagi pusing untuk mengenerasi perintah kompilasi sendiri. Algoritma BFS untuk mengenerasi perintah kompilasi program C++ memiliki kompleksitas waktu O(NM) dan kompleksitas ruang O(N). Akan tetapi, program yang dibuat pada makalah ini memiliki keterbatasan pada asumsi yang terdapat pada bagian “III.A.Pengolahan Include pada berkas CPP”.

B. *Saran*

Pada realitanya, *best-practice* dari struktur program C++ memiliki banyak versi dan sering dilanggar. Alangkah lebih baik bila program dapat mencari berkas .cpp yang sesuai dengan berkas .hpp yang di-*include*. Selain itu, pada realita tidak jarang berkas .cpp juga melakukan *include* ke berkas .cpp lainnya sehingga perlu dilakukan *parsing* pada berkas .cpp pula. Selain itu, tidak seluruh pemrogram menggunakan IDE yang dapat mendeteksi *circular dependencies* sehingga bila program dapat mendeteksinya akan meningkatkan kemudahan penggunaan bagi pemrogram yang menggunakannya.

Link Video Youtube

https://youtu.be/C6T46dwevLs

Link Repository Github

https://github.com/Marthenn/Simple-CPP-Compile-Generator

Ucapan Terima Kasih

Puji syukur penulis panjatkan kepada Tuhan Yang Maha Esa karena dengan rahmatnya penulis dapat menyelesaikan makalah "Penerapan BFS dalam Mengenerasi Perintah Kompilasi C++". Penulis juga mengucapkan terima kasih kepada orang tua, sahabat, dan teman penulis yang senantiasa membantu penulis dengan memberikan dukungan emosional sehingga makalah ini dapat cepat selesai. Penulis juga ingin mengucapkan terima kasih kepada Dr. Nur Ulfa Maulidevi, S.T, M.Sc. selaku dosen IF2211 Strategi Algoritma kelas K02 yang sudah mengajari penulis mengenai algoritma BFS, DFS, dan regex. Penulis juga ingin mengucapkan terima kasih kepada semua pihak yang telah membantu penulis yang tidak dapat penulis sebutkan satu per satu.

VI. Referensi

Bryant, R. E., & O'Hallaron, D. R. (2011). *Computer Systems A Programmer's Perspective.* Boston: Pearson Education.

Github Language Stats. (2023, Mei 20). Retrieved from https://madnight.github.io/githut/#/pull_requests/2023/1

Munir, R. (2023, Mei 22). *Graf (Bagian 1)*. Retrieved from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

Munir, R. (2023, Mei 22). *Graf (Bagian 2)*. Retrieved from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian2.pdf

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, 22 Mei 2023

Bintang Dwi Marthen - 13521144