Penerapan Algoritma VF2 berbasis Backtracking untuk Deteksi Similaritas Kode Sumber Program Bahasa C

Makalah Strategi Algoritma

Muhamad Nazih Najmudin - 13523144
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: nazihnajmudin@gmail.com , 13523144@std.stei.itb.ac.id

***Abstrak*—Perkembangan teknologi di era globalisasi mempermudah akses informasi, namun menimbulkan tantangan etika, termasuk plagiarisme kode sumber dalam pemrograman. Makalah ini membahas penerapan algoritma VF2 berbasis backtracking untuk mendeteksi similaritas dua kode sumber berbahasa C guna mengatasi plagiarisme. Pendekatan dilakukan dengan mengekstrak kode menjadi graf berformat Program Dependence Graph (PDG) menggunakan alat Frama-C, memparsing graf ke dalam struktur data berbasis adjacency list, dan membandingkan dua graf menggunakan algoritma VF2. Algoritma VF2 memanfaatkan backtracking ganda untuk traversal graf, dengan struktur data Stack dan MappedNode untuk optimisasi pemetaan simpul. Hasil pengujian menunjukkan bahwa algoritma ini mampu mendeteksi similaritas tinggi pada kode dengan perubahan minor (seperti pergantian variabel) dan similaritas rendah pada kode dengan logika berbeda (misalnya, rekursif vs. iteratif). Meskipun implementasi berhasil, kompleksitas representasi struktur data pohon menyulitkan modifikasi program. Penelitian ini diharapkan dapat menekan plagiarisme kode, mendukung etika teknologi, dan meningkatkan keaslian karya dalam komunitas pemrograman.**

Keywords—VF2; backtracking; plagiarisme; simlaritas; program; C

I. PENDAHULUAN

Maraknya globalisasi dikuti denagn perkembangan teknologi membuat akses informasi menjadi semakin luas. Dengan teknologi, manusia dapat bertukar informasi dengan sangat mudah dan cepat. Akses informasi yang luas, memungkinkan pertukaran informasi tidak hanya terbatas pada lingkup dan golongan tertentu saja, melainkan seluruh penjuru dunia. Kemudahan dan keluasan akses informasi inilah yang mengubah pola pikir dan pola prilaku manusia. Meskipun memili banyak manfaat, keluasan akses informasi ini juga dapat menjadi tantangan baru, terutama bagi masyarakat yang belum siap mengadopsi teknologi dari sisi etika. Maraknya serangan siber, penggunaan media sosial yang tidak bijak, *cyber bullying*, dan tindakan kriminal lainnya merupakan beberapa contoh tantangan nyata dalam perkembangan teknologi yang pesat ini. Salah satu masalah penting mengenai etika berteknologi yang sering terlupakan adalah kecurangan yang dapat dilakukan dengan sangat mudah menggunakan teknologi yang mapan. Salah satu bentuk kecurangan yang cukup sederhana namun berdampak besar pada kemerosotan etika adalah plagiarisme.

Plagiarisme merupakan tindakan ilegal yang tidak bermoral dan tidak beretika. Plagiarisme adalah tindakan menjiplak, menyalin, atau menggunakan karya, ide, atau kata-kata orang lain tanpa memberikan penghargaan atau sumber yang sesuai, kemudian mengakuinya sebagai karya sendiri. Plagiarisme termasuk pelanggaran etika akademik dan hukum hak cipta, baik disengaja maupun tidak disengaja. Selain plagiarisme, ada pula tindakan kecurangan yang mulai banyak diperhatikan semenjak perkembangan AI yang semakin pesat, yaitu penggunaan jawaban yang dihasilkan AI untuk diklaim sebagai hasil jawaban sendiri. Meskipun tidak menjiplak karya orang lain secara langsung, hal ini tetap merupakan tindakan yang tidak dapat dibenarkan pada beberapa kasus, misalnya dalam ujian maupun tugas yang mengharuskan seseorang untuk mengumpulkan karyanya sendiri tanpa bantuan AI.

Dalam dunia profesi *programmer*, masalah plagiarisme dan penggunaan AI yang tidak bijak dapat berdampak buruk yang fatal bagi perusahaan maupun komunitas. Plagiarisme dan penggunaan AI untuk menjiplak kode sumber, lalu mengakuinya sebagai kode sendiri, pada bidang yang tidak memperbolehkan hal tersebut, merupakan pelanggaran yang dapat dikenai sanksi. Maka dari itu, diperlukan sebuah alat pengecekan plagiarisme atau similaritas kode program yang dapat menegcek keaslian suatu kode sumber.

Untuk menjawab tantangan tersebut, makalah ini dibuat untuk membahas mengenai penerapan algoritma VF2 berbasis *backtracking* untuk mendeteksi similaritas dua kode sumber. Pada bahasan ini, kode sumber yang akan dideteksi adalah program berbahasa C.

II. DASAR TEORI

A. *Graf*

Graf merupakan struktur matematis yang merepresentasikan objek beserta hubungannya dengan objek lain. Graf terdiri dari dua komponen utama, yaitu simpul (*vertex*) dan sisi (*edge*). Simpul merepresentasikan sebuah

objek, sedangkan sisi merepresentasikan hubungan antar objek[1]. Secara formal, graf ditulis sebagai

$$G = (V, E)$$

dengan V adalah himpunan simpul-simpul (*vertices*) yang ditulis sebagai

$$V = \{v_1, v_2, ...\}$$

dan E adalah himpunan sisi yang merupakan pasangan dua buah simpul, ditulis sebagai

$$E = \{(v_1, v_2), ...\}$$

Dalam kehidupan sehari-hari, graf dapat dijumpai pada berbagai hal yang berkaitan dengan lokasi dan jarak seperti peta. Selain itu, graf juga dapat dijumpai pada hal yang berkaitan dengan hubungan, seperti pemodelan hubungan pertemanan. Graf dapat pula dijumpai pada hal yang memiliki alur seperti karangka berpikir dan pemodelan perencanaan.

B. *Subgraf dan Isomorfisme Graf*

Selayaknya himpunan yang memiliki himpunan bagian, dalam teori graf, terdapat istilah yang disebut sebagai subgraf atau upagraf. Subgraf merupakan graf yang simpul-simpul dan sisi-sisinya dibentuk dari sebagian graf lain. Secara formal, graf dengan notasi

$$H = (V', E')$$

dikatakan sebagai subgraf dari G = (V, E) apabila V' dan E' memenuhi

$$V' \subseteq V$$

$$E' \subseteq E$$

Subgraf dapat dibagi menjadai beberapa jenis, diantaranya adalah subgraf biasa (*spanning subgraf*), subgraf pohon (*spanning tree*), serta subgraf terinduksi (*induced subgraph*)[2].

Konsep lain graf yang tidak jauh dari subgraf adalah konsep isomorfisme. Isomorfisme graf merupakan kesamaan antara dua buah graf yang menunjukkan bahwa kedua graf tersebut memiliki korespondensi objek-objek yang sama bseserta hubungan-hubungannya. Secara formal, dua buah graf

$$G = (V, E) \text{ dan } H = (V', E')$$

dikatakan isomorfik apabila keduanya memenuhi korespondensi satu-satu (bijeksi)

$$f : V \rightarrow V'$$

yang mempertahankan struktur graf, yaitu

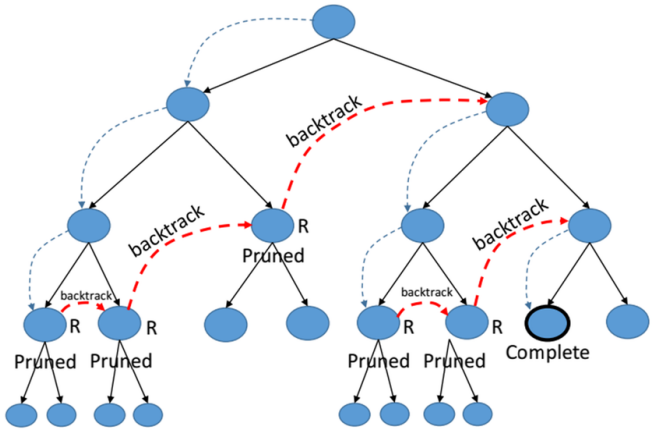
$$(u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$$

Dengan kata lain, dua buah graf dikatakan isomorfik apabila kedua graf memiliki struktur yang sama, baik simpul maupun sisi, beserta arahnya jika graf berarah, meskipun memiliki label atau merepresentasikan objek dan hubungan yang berbeda. Hal ini berarti bahwa isomorfisme graf harus memenuhi jumlah simpul dan sisi yang sama, derajat dan urutan derajat (*degree sequence*) yang sama, konektivitas yang sama, jenis yang sama seperti siklik, asiklik, atau bipartit, serta arah-arah sisi yang sama untuk graf berarah[2].

C. *Algoritma Backtracking*

Algoritma *backtracking* merupakan metode pencarian sistematis yang digunakan untuk menyelesaikan masalah kombinatorik dengan menelusuri seluruh kemungkinan solusi secara bertahap, dan kembali ke tahap sebelumnya (*backtrack*) apabila tidak menemukan solusi dari tahap tersebut. Algoritma *backtracking* bekerja dengan membangun solusi secara bertahap yang inkremental dengan memastikan apakah solusi parsial memenuhi syarat untuk mencapai solusi total. Apabila solusi parsial tidak mungkin dapat membangun solusi total yang valid, proses dihentikan dan kembali ke tahap atau solusi parsial sebelumnya[3].

Algoritma *backtracking* memiliki prinsip yang mirip dengan algoritma DFS (*Depth First Search*) yang berbasis *brute force*. Hanya saja, algortima *backtracking* mengeliminasi kemungkinan solusi parsial yang tidak mungkin mendatangkan solusi total yang valid, dan tidak memproses tahap selanjutnya dari solusi parsial tersebut ketika sudah dianggap tidak valid[3].



Gambar 1. Algoritma Backtracking

Sumber: https://www.researchgate.net/figure/Backtracking-algorithm-taken-from-another-study47_fig4_332041630

Secara formal, algoritma *backtracking* digambarkan sebagai proses traversal pohon keputusan dengan simpul merupakan solusi parsial, sisi merupakan keputusan pengambilan solusi parsial, dan daun merupakan solusi lengkap yang dapat berupa solusi yang valid atau tidak. Terdapat beberapa istilah penting mengenai algoritma *backtracking*, diantaranya adalah *constraint*, *pruning,* dan *backtrack*. Mudahnya, traversal pohon mengecek solusi parsial (simpul) berdasarkan *constraint*. Apabila solusi parsial memenuhi kendala (*constraint*), traversal dilanjut untuk cabang-cabangnya. Apabila solusi parsial tidak memenuhi *constraint*, solusi tersebut dianggap tidak mungkin memenuhi solusi total yang valid, sehingga cabang-cabang solusi tersebut dieliminasi dan tidak diproses sama sekali (*pruning*), lalu traversal dilanjut ke saudara cabang berikutnya. Apabila eksekusi telah mencapai simpul terakhir dari simpul-simpul himpunan cabang milik suatu solusi parsial, dan eksekusi simpul tersebut tidak memenuhi *constraint*, maka dilakukan proses *backtrack*, yaitu kembali ke solusi parsial sebelumnya dan menyatakan bahwa solusi parsial tersebut gagal mendatangkan solusi total yang valid, sehingga dilanjut ke simpul saudara dari solusi parsial tersebut[4].

Berdsarkan jenisnya, algoritma *backtracking* dapat dibagi menjadi dua jenis, yaitu *backtracking* eksplisit dan *backtracking* implisit. *Backtracking* eksplisit merupakan algoritma yang secar eksplisit melakukan *backtracking*, sedangkan *backtracking* implisit melakukan *backtracking* secara alami ketika menjelajahi graf. Algoritma *backtracking* dapat digunakan untuk berbagai masalah kombinatorik seperti *n-queens problem*, *puzzle*, *route planning*, dan *scheduling*. Meskipun dapat menyelesaikan berbagai masalah, algoritma *backtracking* masih memiliki kelemahan. Kompleksitas waktu algoritma ini dapat mencapai O(n!) pada kasus terburuk, yaitu apabila traversal gagal melakukan *pruning* sama sekali. Untuk mengatasi hal ini, diperlukan heuristik yang kuat sehingga dapat memaksimalkan proses *pruning* dan *backtrack*-nya.

D. Algoritma VF2

Algoritma VF2 merupakan algoritma yang digunakan untuk mengetahui apakah dua buah graf isomorifk, serta dapat pula untuk mengetahui apakah suatu graf merupakan subgraf dari suatu graf lain. Algoritma VF2 bekerja berdasarkan algoritma *backtracking* dalam melakukan traversal graf berlabel (*labelled graph*) pada kedua graf yang ingin dicek. Graf berlabel (*labelled graph*) merupakan graf dengan simpul beratribut yang dapat diidentifikasi dan dibandingkan. Algoritma VF2 dapat diterapkan baik untuk graf berarah, tidak berarah, graf sederhana, maupun graf yang tidak sederhana[5].

Secara formal, algoritma VF2 dinotasikan sebagai berikut. Misalkan graf pola G dan graf target H

$$G = (V_G, E_G, L_G)$$

$$H = (V_H, E_H, L_H)$$

dengan V adalah simpul, E adalah sisi, dan L adalah fungsi pelabelan simpul/sisi. Pemetaan dilakukan berdasarkan

$$M \subseteq V_G \times V_H$$

dimana M(n) = m berarti simpul $n \in V_G$ dipetakan ke $m \in V_H$. Pemetaan M adalah isomorfisme jika M adalah korespondensi satu-satu (bijeksi) dan

$$\forall((n1, n2) \in E_G \wedge \exists(M(n_1), M(n_2)) \in E_H) \text{ dan}$$

$$L_G(n_1, n_2) = L_H(M(n_1), M(n_2))$$

Pemetaan M adalah subgraf jika M adalah injeksi satu-satu dan

$$\forall((n1, n2) \in E_G \wedge \exists(M(n_1), M(n_2)) \in E_H) \text{ dan}$$

$$L_G(n) = L_H(m)$$

Karena berdasar pada algoritma *backtracking*, algoritma VF2 masih memiliki kelemahan berupa kompleksitas waktu yang dapat mencapai $O(n! \cdot n)$ pada kasus terburuknya dan kompleksitas ruang O(n) bila diimplementasikan dengan rekursif[5].

E. Code Property Graph

Untuk dapat melakukan pengecekan similaritas dua buah kode C menggunakan algoritma VF2 yang berbasis algoritma *backtracking,* kode C harus terlebih dahulu diekstrak menjadi graf. Program C yang sekuensial merupakan runtutan alur program yang memiliki flow tertentu dan dapat dimodelkan menjadi graf. Selain itu, dependensi antar variabel juga dapat dimodelkan menjadi graf. Hasil ekstraksi menjadi graf ini disebut sebagai *code property graph*.

Ada beberapa jenis *code property graph* dari hasil ekstraksi kode program, beberapa diantaranya adalah AST (*Abstract Syntax Tree*), CFG (*Control Flow Graph*), dan PDG (*Program Dependence Graph*). Singkatnnya, AST merupakan struktur graf yang memodelkan struktur sintaksis program. Pada umumnya, AST digunakan untuk kompilasi. Adapun CFG merupakan struktur graf yang memodelkan alur eksekusi program dengan sisi-sisinya merepresentasikan kontrol. Pada umumnya, CFG digunakan untuk analisis dan optimasi alur program. Adapun PDG merupakan struktur graf yang memodelkan dependensi program. Biasanya digunakan untuk deteksi eror dan analisis optimasi[6].

III. Implementasi

A. Metodologi

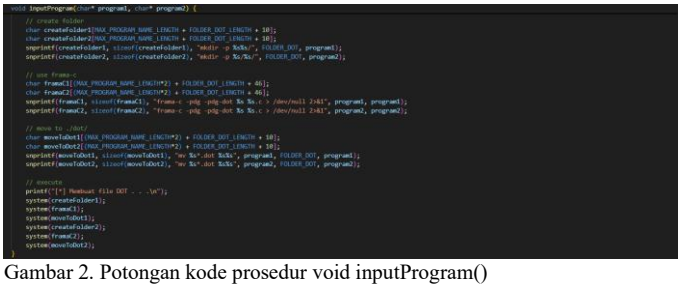
Implementasi pengecekan similaritas kode bahasa C menggunakan algoritma VF2 berbasis *backtracking* memerlukan langkah-langkah berikut.

1. Ekstraksi file kode C menjadi graf berbentuk file dot dengan format graf AST, CFG, atau PDG. Dalam implementasi ini, dipilih PDG sebagai format grafnya.
2. Proses *parsing* file dot hasil ekstraksi menjadi struktur data graf dalam program.
3. Penerapan algoritma VF2 pada kedua struktur data graf, yaitu dengan melakukan traversal graf pada keduanya menggunakan algoritma *backtracking*.
4. Output hasil pengecekan similaritas.

Metode implementasi ini lebih dititik-beratkan pada proses penerapan algoritma VF2, sehingga proses-proses lain diimplementasikan menggunakan *tools* yang dapat langsung digunakan. Adapun struktur data graf dan algoritma VF2 diimplementasi secara mandiri untuk dapat memahami traversal graf dan algoritma VF2 lebih dalam.

B. Kode Sumber

Program ditulis dalam bahasa C dengan bantuan *tools* eksternal untuk mengekstrak kode C menjadi graf PDG. *Tool* eksternal yang digunakan adalah Frama-C. *Tool* ini memungkinkan mengekstrak fungsi-fungsi pada sebuah kode C menjadi file-file dot. Sintaks untuk mengekstrak kode C ini di-*wrap* ke sebuah fungsi yang dapat dijalankan.



Gambar 2. Potongan kode prosedur void inputProgram()

Sumber: Dokumen Penulis

Prosedur ini menghasilkan file-file dot dari dua buah input program. Misalkan salah satu input program adalah test.c yang memiliki fungsi atau prosedur fooA(), fooB(), dan main(). Hasil ekstraksinya adalah file-file dengan nama program.fooA.dot, program.fooB.dot, program.main.dot yang disimpan dalam folder ‘./dot/test/’. Pada proses ini, kode C yang diinput harus memiliki fungsi int main() sebagai *entrypoint*.

Untuk memudahkan eksekusi, file-file dot yang merepresentasikan satu program ini di-*parse* dan dimasukkan menjadi dua buah file txt, yaitu nodes.txt dan edges.txt. pada file txt, informasi graf yang disimpan lebih sederhana dan sesuai dengan apa yang dibutuhkan program. Proses *parsing* ini memanfaatkan *library* luar bernama libcgraph dari Graphiz.

```c
void extract_dot(char* dotdir, char* nodesfile, char* edgesfile) {
    DIR *dir;
    struct dirent *entry;
    struct stat fileStat;
    char fullPath[1024]; // Buffer untuk menyimpan path lengkap file .dot
    const char *dotExtension = ".dot";

    // Buka direktori
    dir = opendir(dotdir);
    if (dir == NULL) {
        perror("Gagal membuka direktori DOT");
        fprintf(stderr, "Pastikan folder '%s' ada dan dapat diakses.\n", dotdir);
        return;
    }

    // printf("Mulai memproses file DOT di folder: %s\n", dotdir);

    // Baca setiap entri di direktori
    while ((entry = readdir(dir)) != NULL) {
        // Abaikan entri "." dan ".."
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
            continue;
        }

        // Buat path lengkap file
        // snprintf lebih aman daripada sprintf karena mencegah buffer overflow
        snprintf(fullPath, sizeof(fullPath), "%s/%s", dotdir, entry->d_name);

        // Periksa apakah ini file reguler dan apakah berakhir dengan ".dot"
        if (stat(fullPath, &fileStat) == 0) {
            if (S_ISREG(fileStat.st_mode)) { // Pastikan itu adalah file biasa
                if (endsWith(entry->d_name, dotExtension)) {
                    // printf("  - Ditemukan file DOT: %s\n", entry->d_name);
                    // Panggil fungsi convert_to_txt untuk memproses file ini
                    convert_to_txt(fullPath, nodesfile, edgesfile);
                }
            }
        } else {
            // Menangani kasus di mana stat gagal (misal: permission denied, broken link)
            // perror("Gagal mendapatkan status file");
            // fprintf(stderr, "Tidak dapat memproses: %s\n", fullPath);
        }
    }
```

Gambar 3. Prosedur extract_dot()
Sumber: Dokumen penulis

File txt di-*parse* untuk mengisi struktur data graf yang akan dibandingkan. Struktur data Graph dan proses *parsing* ini diimplementasi mandiri tanpa *library* eksternal.

```c
void extract_txt(char* nodesfile, char* edgesfile, Graph* result) {
    // Inisialisasi graph
    CreateGraph(result);

    // Baca file nodes.txt
    FILE* fnodes = fopen(nodesfile, "r");
    if (fnodes == NULL) {
        perror("Error opening nodes file");
        return;
    }

    char line[256];
    while (fgets(line, sizeof(line), fnodes)) {
        // Parse setiap baris node
        Node n = CreateNode();

        // Format: id<double space>label_full<double space>label<double space>fillcolor<double space>shape
        char* token = strtok(line, "~~");  // Double space sebagai delimiter
        if (token == NULL) continue;

        n.id = atoi(token);

        token = strtok(NULL, "~~");
        if (token) {
            strncpy(n.label_full, token, MAX_LABEL_FULL_LENGTH-1);
            n.label_full[MAX_LABEL_FULL_LENGTH-1] = '\0';
        } else {
            n.label_full[0] = '\0';
        }

        token = strtok(NULL, "~~");
        if (token) {
            strncpy(n.label, token, MAX_LABEL_LENGTH-1);
            n.label[MAX_LABEL_LENGTH-1] = '\0';
        } else {
            n.label[0] = '\0';
        }

        token = strtok(NULL, "~~");
        if (token) {
            strncpy(n.fillcolor, token, FILLCOLOR_LENGTH-1);
            n.fillcolor[FILLCOLOR_LENGTH-1] = '\0';
        } else {
            n.fillcolor[0] = '\0';
        }
```

Gambar 4. Potongan kode prosedur extract_txt()
Sumber: Dokumen penulis

Proses *parsing* dari file txt pada prosedur void extract_txt() menghasilkan struktur data Graph. implementasi struktur data Graph menggunakan *adjacency list* yang dimodifikasi. Model ini dipilih karena traversal graf pada algoritma VF2 banyak memerlukan akses ketetanggaan. *Adjacency list* mampu mengoptimalkan proses ini dengan *locality* yang lebih tinggi dibanding pemodelan graf lainnya.

```c
// Struktur Data Node
typedef int NodeId;
typedef struct Node {
    NodeId id;
    char label_full[MAX_LABEL_FULL_LENGTH];
    char label[MAX_LABEL_LENGTH];
    char fillcolor[FILLCOLOR_LENGTH];
    char shape[SHAPENAME_LENGTH];
} Node;
#define ID(n) (n).id
#define LABEL_FULL(n) (n).label_full
#define LABEL(n) (n).label
#define FILLCOLOR(n) (n).fillcolor
#define SHAPE(n) (n).shape

Node CreateNode();


// Struktur Data Edge
typedef struct Edge {
    int from;   // Node.id asal
    int to;     // Node.id tujuan
} Edge;
#define FROM(e) (e).from
#define TO(e) (e).to

Edge CreateEdge();


// Struktur Data Graph (Adjacency List)
typedef struct Graph {
    Node* nodes;
    NodeId** edges_out;
    NodeId** edges_in;
    int* edge_out_count;
    int* edge_in_count;
    int node_count;
} Graph;
#define NODE(g, id) (g).nodes[id]
#define NODE_OUT(g, id) (g).edges_out[id]
#define NODE_IN(g, id) (g).edges_in[id]
#define COUNT_OUT(g, id) (g).edge_out_count[id]
#define COUNT_IN(g, id) (g).edge_in_count[id]
#define COUNT_NODE(g) (g).node_count

// Main
void CreateGraph(Graph* g); // alokasi
void DeleteGraph(Graph* g); // free memory
```

Gambar 5. Struktur data Graph (beberapa bagian terpotong).
Sumber: Dokumen Penulis

Selanjutnya, dua buah graf dibandingkan dengan algoritma VF2. Penerapan algoritma VF2 menggunakan struktur data Stack yang diimplementasi mandiri. Stack digunakan untuk menyimpan simpul-simpul yang akan dieksekusi seperti pada algoritma DFS. Adapun simpul-simpul yang telah dipetakan disimpan dalam struktur data MappedNode yang merupakan *wrapper* untuk Stack dengan penambahan boolean is_mapped untuk menghemat waktu eksekusi.

```c
#ifndef STACK_H
#define STACK_H

#include "../graph/graph.h"

#define MAX_STACK_SIZE MAX_NODES

// Struktur Data Stack
typedef int StackType;
typedef struct Stack {
    StackType* data;
    int top;
} Stack;

// Akses
#define TOP(s) (s).top

// Alokasi dan Free
void CreateStack(Stack* s);
void DeleteStack(Stack* s);

// Operasi
void push(StackType el, Stack* s);
int pop(Stack* s);
int peek(const Stack* s);

// Cek
bool isStackEmpty(const Stack* s);
bool isStackFull(Stack* s);

// Size
int getSize(const Stack* s);

// Printf
void printStack(const Stack* s);

#endif
```

Gambar 6. Struktur data Stack (dibuat mandiri)
Sumber: Dokumen penulis

```c
typedef struct MappedNode {
    Stack nodes;
    bool* is_mapped;
} MappedNode;
#define STACK_NODES(mn) (mn).nodes
#define IS_MAPPED(mn, id) (mn).is_mapped[id]

void CreateMappedNode(MappedNode* mn);
void DeleteMappedNode(MappedNode* mn);
void mapped_push(MappedNode* mn, StackType id);
StackType mapped_pop(MappedNode* mn);
```

Gambar 7. Struktur data MappedNode
Sumber: Dokumen penulis

Algoritma VF2 diimplementasi secara iteratif menggunakan dua buah struktur data Stack yang merepresentasikan dua buah pohon status traversal kedua buah graf. Dibuat pula dua buah struktur data Stack lainnya yang di-*wrap* dengan MappedNode untuk menyimpan simpul-simpul yang sudah dipetakan. Proses rekursif hanya digunakan pada backtrack_pop() yang akan melakukan rekursif hanya untuk backtrack. Adapun implementasi iteratif dipilih atas pertimbangan jumlah simpul yang cukup banyak.

```c
int vf2_subgraph(Graph* g1, Graph* g2, char* matcher) {
    // 3. State Awal: push g1 ke stack
    // printf("[VF2] 3. State Awal: push g1 ke stack\n");
    NodeId* nodes_n1 = getNodesByLesserDegree(g1);
    for (int i = 0; i < COUNT_NODE(*g1); i++) {
        push(nodes_n1[i], &stack_g1);
    }
    free(nodes_n1);

    // 4. Iterasi 1: pohon NodeId pada G1
    // printf("[VF2] 4. Iterasi 1: pohon NodeId pada G1\n");
    while (!isStackEmpty(&stack_g1)) {

        // 4.1. State Awal: push g2 ke stack
        // printf("[VF2] 4.1. State Awal: push g2 ke stack\n");
        NodeId* nodes_n2 = getNodesByLesserDegree(g2);
        for (int i = 0; i < COUNT_NODE(*g2); i++) {
            push(nodes_n2[i], &stack_g2);
        }
        free(nodes_n2);

        // 4.2. Iterasi 2: pohon NodeId pada G2
        // printf("[VF2] 4.2. Iterasi 2: pohon NodeId pada G2\n");
        while (!isStackEmpty(&stack_g2)) {

            // 4.2.1. Proses TOP
            // printf("[VF2] 4.2.1. Proses TOP\n");
            bool match =  isNodeMatch(g1, peek(&stack_g1), g2, peek(&stack_g2), matcher);
            NodeId last_g1 = peek(&STACK_NODES(mapped_g1));
            NodeId last_g2 = peek(&STACK_NODES(mapped_g2));
            if (match) {
                result = max(result, getSize(&STACK_NODES(mapped_g1)));
                mapped_push(&mapped_g1, peek(&stack_g1));
                mapped_push(&mapped_g2, peek(&stack_g2));
            }

            // 4.2.2. Proses next branch
            // printf("[VF2] 4.2.2. Proses next branch\n");
            if (match) {
                addBranches(g1, peek(&stack_g1), &stack_g1, &mapped_g1);
                addBranches(g2, peek(&stack_g2), &stack_g2, &mapped_g2);
            } else {
                backtrack_pop(&stack_g1, &stack_g2, '2', &mapped_g1, &mapped_g2, g2);
            }

            // 4.2.3 Proses pengecekan leave
            // printf("4.2.3 Proses pengecekan leave\n");
            if (match && (peek(&STACK_NODES(mapped_g2)) == last_g2)) {
                backtrack_pop(&stack_g1, &stack_g2, '2', &mapped_g1, &mapped_g2, g2);
            } else if (match && (peek(&STACK_NODES(mapped_g1)) == last_g1)) {
                backtrack_pop(&stack_g1, &stack_g2, '1', &mapped_g1, &mapped_g2, g2);
            }

            // Pengecekan similaritas untuk efisiensi
            if (result >= COUNT_NODE(*g1)) {
                break;
            }

            // printf("is stack empty? %d\n", isStackEmpty(&stack_g2));
            // printStackMonitor(g1, g2, &stack_g1, &stack_g2, &mapped_g1, &mapped_g2);
        }

        // Pengecekan similaritas untuk efisiensi
        if (result >= COUNT_NODE(*g1)) {
            break;
        }
```

Gambar 8. Potongan kode fungsi vf2_subgraph()
Sumber: Dokumen penulis

Keluaran dari fungsi vf2_subgraph() adalah banyaknya simpul dari graph yang merupakan subgraf (atau isomorf jika isomorfik) yang dapat dipetakan pada graf lainnya. Similaritas dihitung berdasarkan persentase jumlah simpul yang berhasil dipetakan ini.

C. Tampilan Program dan Pengujian

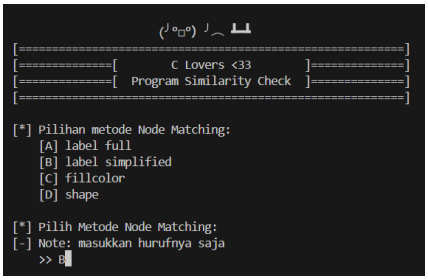
Pertama, program meminta input dua buah file kode C yang hendak dicek similaritasnya.

```
              (╯°□°) ╯︵ ┻━┻
[================================================]
[============[        C Lovers <33       ]============]
[============[  Program Similarity Check  ]============]
[================================================]

[*] Masukkan Program 1 (yang ingin dicek)
[-] Note: tidak perlu menuliskan .c nya
    >> test/test_1_1

[*] Masukkan Program 2 (referensi atau acuan)
[-] Note: tidak perlu menuliskan .c nya
    >> test/test_1_2
```

Gambar 9. Tampilan input program

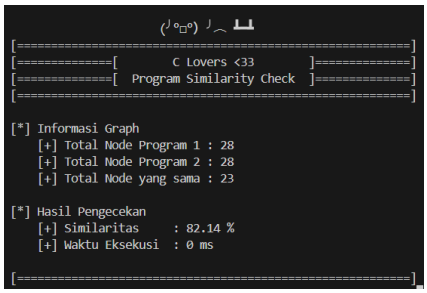
Sumber: Dokumen penulis

Setelah itu, program meminta input berupa jenis pencocokkan label. Terdapat 4 jenis pencocokkan yang didasarkan atribut yang dimiliki simpul pada hasil ekstraksi dot menggunakan Frama-C, yaitu label_full, label (simplified), fillcolor, dan shape.



Gambar 10. Tampilan input metode pencocokkan

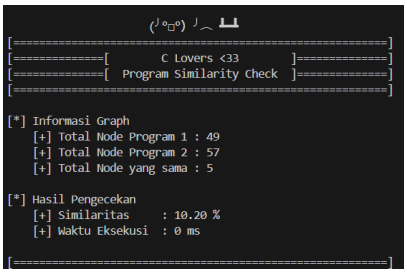
Sumber: Dokumen penulis



Gambar 11. Tampilan hasil

Sumber: Dokumen penulis

Contoh tersebut merupakan hasil pengecekan similaritas dua buah program pencarian GCD menggunakan algoritma Euclidean, salah satu program hanya mengganti penamaan variabel dan beberapa eksekusi di main nya saja. Sekarang, perhatikan contoh berikut.



Gambar 12, Hasil percobaan kedua

Sumber: Dokumen penulis

Hasil ini diperoleh dari input dua buah program pencarian perhitungan faktorial. Perbedaannya, program pertama menggunakan cara rekursif, sementara program kedua menggunakan cara iteratif. Terlihat bahwa hasil memiliki perbedaan yang signifikan dibanding contoh kasus sebelumnya.

IV. Pembahasan

A. *Backtracking pada Algoritma VF2*

Penerapana *backtracking* pada algoritma VF2 terdapat pada transversal kedua buah graf yang dibandingkan. Untuk mengimplementasikannya ke dalam kode, ada dua pilihan cara merepresentasikan struktur dan algoritma VF2.

1. Menggunakan satu buah struktur data pohon dengan simpul berisi dua buah simpul pada kedua graf yang diabandingkan. Dua buah simpul graf yang berada dalam smpul pohon ini merepresentasikan pemetaan korespondensi satu-satu.
2. Menggunakan dua buah struktur struktur data pohon dengan simpul merupakan simpul graf yang dietelusuri. Pohon ini merupakan *spanning tree* dari masing-masing graf.

Pada makalah, ini cara yang digunakan adalah cara nomor 2, yaitu dengan memanfaatkan dua buah pohon yang merupakan *spanning tree* untuk kedua graf. Hanya saja, struktur data pohon tidak benar-benar diimplementasi secara fisik, melainkan secara lojik dengan struktur data Stack dan MappedNode.

Terdapat penggunaan algoritma *backtracking* secara ganda yang digunakan secara saling beriringan. Penggunaan *backtracking* ganda ini direpresentasikan dalam kode sebagai dua level kalang. Kalang pertama, merepresentasikan traversal graf yang pada bentuk lengkapnya menjadi sebuah *spanning tree* dari graf pertama. Graf pertama adalah graf dengan jumlah simpul yang lebih sedikit. Graf ini menjadi graf G yang akan dipetakan menuju graf target H.

Setiap simpul pada pohon level pertama merupakan simpul pada graf G yang sedang atau telah dijelajahi. Sebuah simpul memiliki cabang berupa simpul-simpul tetangga, baik dengan arah masuk maupun keluar, dengan ketentuan simpul tetangga tersebut belum dipetakan. Pengecekan pemetaan diimplementasi secara hash dengan MappedNode, sehingga kompleksitasnya yang O(1) tidak mempengaruhi algoritma inti. Untuk dapat mengatakan bahwa sebuah simpul memenuhi *constraint*, simpul tersebut harus memenuhi hal berikut.

1. Memiliki kesamaan informasi simpul. Informasi simpul yang dibandingkan dipilih oleh pengguna, yaitu label_full, label (*simplified*), fillcolor, atau shape.
2. Memiliki jumlah tetangga masuk yang sama dan tetangga keluar yang sama.
3. Setiap simpul tetangga yang dimiliki sebuah simpul graf juga harus dapat berkorespondensi satu-satu dengan simpul graf yang lain. Pada tahap pengecekan ini, korespondensi hanya tertuju pada pengecekan kesamaan informasinya saja, tidak secara rekursif mengecek poin (3) lagi.

Sebagaimana yang telah dijelaskan sebelumnya, *backtracking* ganda pada VF2 memiliki dua level kalang, dengan level pertama merupakan graf G yang memiliki lebih sedikit simpul, dan level kedua merupakan graf target H yang memiliki lebih banyak simpul. Algoritma *backtracking* ini memiliki langkah-langkah berikut.

1. Proses satu simpul pada G. Sebelum berlanjut ke cabang-cabangnya, simpul ini harus memenuhi *constraint* terlebih dahulu, yaitu dengan

membandingkan dengan simpul pada H yang memiliki kedalaman pohon yang sama.

2. Proses satu simpul pada H. Cek pemenuhan *constraint*, apakah simpul G memenuhi kesamaan dengan simpul H.
3. Apabila kedua simpul ini memenuhi kesamaan, simpan kedua simpul ini dalam MappedNode mappedG dan mappedH. lanjutkan traversal graf dengan membangun cabang-cabang dari kedua simpul ini, pembentukan cabang tidak boleh mengandung simpul yang sudah ada di MappedNode. Proses simpul cabang pada graf G kembali pada poin (1).
4. Apabila kedua simpul ini gagal dipetakan, lakukan *pruning* pada simpul H, lanjutkan pengecekan untuk simpul alternatif/saudara-nya.
5. Apabila proses *pruning* pada poin (4) merupakan pemrosesan cabang terakhir dari simpul induknya, lakukan *backtrack* pada kedua pohon, baik G maupun H. Di titik ini, simpul yang terakhir kali dipetakan pada G maupun H dianggap gagal mendekati solusi total yang valid, sehingga keduanya di-*pop* dari MappedNode.
6. Proses *backtrack* pada poin (5) membuat simpul G dinyatakan gagal, dan dilanjut ke simpul alternatif/saudaranya seperti pada poin (4) lalu pengecekan dilanjutkan dari poin (1).
7. Apabila dari keenam poin proses diatas ditemukan ada satu yang memenuhi korespondensi satu-satu dengan seluruh simpul pada G telah dipetakan, proses pencarian dihentikan, dan dinyatakan bahwa graf G merupakan subgraf dari graf H, atau isomorfik jika pada pengecekan awal, graf G memiliki jumlah simpul yang sama dengan graf H.
8. Apabila tidak menghasilkan solusi, pencarian dilanjut hingga cabang terakhir kedalaman pertama pada pohon G. Hal ini dapat diketahui dengan mengecek apakah Stack yang berisi urutan pemrosesan simpul graf G telah kosong.
9. Pada titik ini, graf G dinyatakan sebagai bukan subgraf dan bukan isomorf dari graf H.
10. Namun, fungsi vf2-subgraf() tetap mengembalikan sebuah hasil berupa maksimal jumlah simpul pada graf G yang berhasil dipetakan.

Hsil ini kemudian dibandingkan dengan jumlah simpul pada graf G untuk merepresentasikan persentase similaritas graf G dengan graf H.

B. Hasil Implementasi dan Pengujian

Implementasi program dapat berjalan dengan baik sesuai dengan apa yang diharapkan. Implementasi program dengan bahasa C membuatnya dapat berjalan dengan ringan dan cepat dan sesuai untuk algoritma *backtracking* yang pada kasus terburuknya dapat sama dengan DFS yang berbasis *brute force*. Pada hasil pengujian, terlihat bahwa program berhasil memberikan similaritas tinggi pada program yang hanya mengganti beberapa variabel, serta memberikan similaritas rendah pada program yang memimiliki prosedur berbeda. Hal ini menandakan bahwa penerapan algoritma VF2 berbasis *backtracking* untuk pengecekan similaritas dua buah kode C telah berhasil memenuhi jawaban.

Namun, penerapan ini masih memiliki kelemahan. Yaitu terletak pada pemilihan metode representasi struktur data pohon pada algoritma VF2 yang dirasa memerlukan implementasi alur kode yang terlalu kompleks dibanding alternatifnya. Meskipun tidak mengganggu kompleksitas waktu, hal ini membuat program cukup sulit untuk dimodifikasi dan dikembangkan lebih lanjut.

V. KESIMPULAN

Maraknya globalisasi dikuti dengan perkembangan teknologi membuat akses informasi menjadi semakin luas. Meskipun memili banyak manfaat, keluasan akses informasi ini juga dapat menjadi tantangan baru, terutama bagi masyarakat yang belum siap mengadopsi teknologi dari sisi etika. Salah satu masalah penting mengenai etika berteknologi yang sering terlupakan adalah kecurangan dalam bentuk plagiarisme. Salah satu bentuk plagiarisme pada profesi *programmer* adalah plagiarisme program.

Untuk mengatasi masalah ini, diterapkanlah algoritma VF2 yang dapat mengecek similaritas dua buah graf yang merepresentasikan sintaksis, alur, dan dependensi dari program. Algoritma VF2 menggunakan algoritma *backtracking* dalam traversal graf yang hendak dibandingkan.

Harapannya, dengan adanya teknologi semacam ini, tingkat plagiarisme dapat ditekan sehingga dapat mengatasi masalah kemerosotan etika dari adopsi teknologi yang tidak bijak.

APPENDIX

Tautan rilis repositori GitHub yang digunakan untuk implementasi program adalah:

https://github.com/nazihstei/makalah-stima-2025/releases/

REFERENCES

[1] A. Dickson, Introduction to Graph Theory. Department of Mathematics, University of Utah, Oct. 2006. [Online]. Available: https://www.math.utah.edu/mathcircle/notes/MC_Graph_Theory.pdf

[2] B. Mitra, Basics of Graph Theory. CSE IIT Kharagpur. [Online]. Available: https://cse.iitkgp.ac.in/~bivasm/cnt_notes/Basic-graph-theory-1.pdf

[3] F. Ruskey, "Chapter 3 Backtracking," Computer Science, University of Victoria, 2001. [Online]. Available: https://webhome.cs.uvic.ca/~ruskey/classes/Knuth4A/BacktrackChpt.pdf

[4] P. van Beek, "Chapter 4 Backtracking Search Algorithms," in Handbook of Constraint Programming, F. Rossi and T. Walsh, Eds. 2006. [Online]. Available: https://cs.uwaterloo.ca/~vanbeek/Publications/survey06.pdf

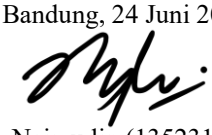
[5] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "An Improved Algorithm for Matching Large Graphs," Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli "Federico II", Jan. 2001. [Online]. Available: https://www.researchgate.net/publication/200034365

[6] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in Proc. 35th IEEE Symposium on Security and Privacy, 2014. [Online]. Available: https://www.ieee-security.org/TC/SP2014/papers/ModelingandDiscoveringVulnerabilitieswithCodePropertyGraphs.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, 24 Juni 2025



Muhamad Nazih Najmudin (13523144)