Analisis Kompleksitas Algoritma Minimum Spanning Tree dan Alternatifnya

Ubaidillah Ariq Prathama - 13520085
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13520085@std.stei.itb.ac.id

***Abstract*—Minimum spanning tree merupakan sebuah tree yang didapatkan dari subgraf suatu graf. Subgraf ini harus tidak memiliki cycle, memiliki sisi sejumlah simpul dikurang satu, dan penjumlahan beban minimum. Terdapat beberapa cara untuk mengimplementasikan minimum spanning tree diantaranya adalah algoritma Prim dan algoritma Kruskal. Tentunya program yang dibuat harus efektif. Oleh karena itu, algoritma ini perlu dianalisis kompleksitasnya.**

***Keywords*—Tree, Kompleksitas, Algoritma Prim, Algoritma Kruskal**

I. PENDAHULUAN

Minimum spanning tree adalah himpunan bagian dari himpunan garis-garis (edge) suatu graf berbobot tak berarah yang menghubungkan semua titik tanpa membentuk siklus dan dengan total bobot minimum. Minimum spanning tree memiliki banyak kegunaan, seperti mencari jalan minimum untuk menghubungkan beberapa kota dan perutean pesan pada jaringan komputer. Selain itu, minimum spanning tree juga bisa dilakukan pada matriks RGB sebuah gambar untuk segmentasi citra.

Dapat dilihat bahwa banyak sekali aplikasi dari minimum spanning tree. Oleh sebab itu, diperlukan algoritma minimum spanning tree yang optimal. Terdapat banyak cara mengoptimasi sebuah program, tetapi yang paling signifikan adalah kompleksitas waktu dan ruang. Tentunya permasalahan yang dapat diselesaikan dengan minimum spanning tree ini bisa saja berukuran besar. Jika program tidak efisien bisa saja melewati constraint memori dan waktu.

Terdapat dua algoritma yang umum digunakan untuk mencari minimum spanning tree, yaitu algoritma Prim dan algoritma Kruskal. Kedua algoritma ini sebenarnya sederhana dan akan dibahas kemudian. Akan tetapi, implementasi kedua algoritma ini bisa dilakukan dengan berbagai cara dan struktur data yang berbeda. Perbedaan ini menyebabkan adanya perbedaan kompleksitas yang harus kita analisis mana yang lebih efektif.

II. TEORI DASAR

A. Graf

Graf digunakan untuk merepresentasikan objek-objek diskrit dan hubungan antara objek tersebut. Representasi visual dari graf adalah dengan menyatakan objek sebagai noktah, bulatan, atau titik (node), sedangkan hubungan antara objek tersebut dinyatakan dengan garis atau sisi (edge). Menurut catatan sejarah, masalah jembatan Konigsberg adalah masalah yang pertama kali menggunakan graf, masalah ini kemudian diselesaikan oleh seorang ahli bernama Euler.

Graf G merupakan pasangan himpunan (V, E) dimana: V = himpunan tidak kosong dari simpul-simpul (node) = $\{v_1, v_2, v_3, \ldots, v_n\}$, E = himpunan yang mungkin kosong dari sisi (edge) yang menghubungkan sepasang simpul = $\{e_1, e_2, e_3, \ldots, e_n\}$ Graf G dapat ditulis dengan menggunakan notasi $G = \{V, E\}$. Simpul-simpul pada graf tidak boleh merupakan himpunan kosong, sedangkan sisi-sisi pada graf boleh merupakan himpunan kosong. Graf yang hanya mempunyai satu buah simpul tanpa sisi dinamakan graf trivial. Simpul pada graf dapat dinomori dengan huruf, angka, atau gabungan keduanya. Sedangkan, sisi yang menghubungkan dua buah simpul pada graf dapat dinyatakan dengan pasangan (v_i, v_j) atau dengan lambang e_1, e_2, dan seterusnya.

Graf dapat dikelompokkan ke dalam beberapa jenis tergantung pada sudut pandang pengelompokkannya, baik dari segi ada tidaknya sisi ganda atau kalang, jumlah simpul, maupun dari segi orientasi arah pada sisi. Berdasarkan ada tidaknya sisi ganda atau kalang, graf dikelompokkan menjadi:

1. Graf Sederhana
   Graf sederhana merupakan graf yang tidak memiliki sisi ganda maupun gelang.
2. Graf Tak Sederhana
   Graf tak sederhana merupakan graf yang mengandung sisi ganda atau kalang. Graf tak sederhana ini dibagi lagi menjadi dua, yaitu: graf ganda dan graf semu. Sesuai dengan namanya, graf ganda adalah graf yang mengandung sisi ganda (berjumlah dua atau lebih dari dua) yang menghubungkan dua buah simpul. Graf semu adalah graf yang mengandung gelang. Graf semu juga dapat memiliki sisi ganda.

Berdasarkan jumlah simpul pada suatu graf, graf dapat digolongkan menjadi dua jenis:

1. Graf Berhingga
   Graf berhingga merupakan graf yang jumlah simpulnya berhingga (dapat dihitung).
2. Graf Tak Berhingga
   Graf tak berhingga merupakan graf yang jumlah simpulnya tidak berhingga (tidak dapat dihitung).

Sisi pada graf dapat mempunyai orientasi arah. Berdasarkan orientasi arah pada sisi, graf dapat digolongkan menjadi dua jenis, yaitu:

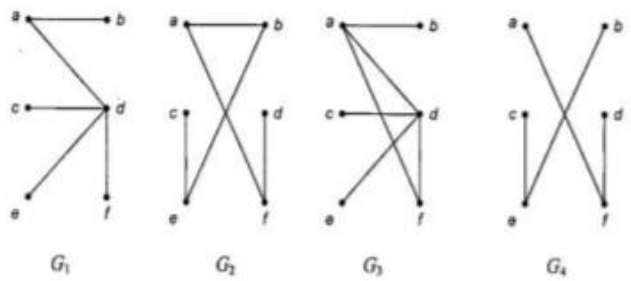
1. Graf Tak Berarah
   Graf tak berarah merupakan graf yang sisinya tidak mempunyai orientasi arah. Pada graf ini, urutan pasangan simpul yang dihubungkan oleh sisi tidak diperhatikan. Jadi, $(v_i, v_j) = (v_j, v_i)$ adalah sisi yang sama.
2. Graf Berarah
   Graf berarah merupakan graf yang setiap sisinya diberikan orientasi arah. Sisi yang berarah disebut juga busur (arc). Pada graf berarah, (v_i, v_j) dan (v_j, v_i) merupakan dua busur yang berbeda. Busur (v_i, v_j) titik pangkalnya berada pada v_i (simpul asal) dan titik ujungnya berada pada v_j (simpul terminal). Graf berarah yang mempunyai sisi ganda atau gelang disebut graf ganda berarah.

Terdapat beberapa terminologi (istilah) yang berkaitan dengan graf. Berikut ini adalah beberapa terminologi tersebut:

1. Bertetangga
   Dua simpul pada graf tak berarah dikatakan bertetangga jika keduanya dihubungkan oleh sebuah sisi. Pada graf berarah, dua buah simpul dikatakan bertetangga jika dihubungkan oleh sebuah busur.
2. Bersisian
   Untuk sembarang sisi $e = (v_i, v_j)$, sisi e dikatakan bersisian dengan simpul v_i dan v_j .
3. Simpul Terpencil
   Simpul terpencil adalah simpul yang tidak mempunyai sisi yang bersisian dengannya.
4. Graf Kosong
   Graf kosong adalah graf yang himpunan sisinya merupakan himpunan kosong, dinyatakan dengan Nn dimana n merupakan jumlah simpul.
5. Derajat
   Derajat suatu simpul pada graf tak berarah adalah jumlah sisi yang bersisian pada simpul. Pada graf berarah, derajat simpul v dinyatakan dengan $d_{in}(v)$ dan $d_{out}(v)$. $d_{in}(v)$ = derajat masuk = jumlah busur yang masuk ke simpul v. $d_{out}(v)$ = derajat keluar = jumlah busur yang keluar dari simpul v. $d(v) = d_{in}(v) + d_{out}(v)$
6. Lintasan
   Lintasan yang panjangnya n dari simpul awal ke simpul tujuan di dalam graf G adalah barisan berselang-seling simpul-simpul dan sisi-sisi yang berbentuk $v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n$ sehingga $e_1 = (v_0, v_1)$, $e_2 = (v_1, v_2)$, …, $e_n = (v_{n-1}, v_n)$ adalah sisi-sisi dari graf. Pada graf sederhana, cukup dituliskan lintasan sebagai barisan dari simpul-simpul.
7. Sirkuit
   Lintasan yang berawal dan berakhir pada simpul yang sama disebut sirkuit atau siklus.
8. Terhubung
   Dua buah simpul v_i dan v_j dikatakan terhubung jika terdapat lintasan dari v_i ke v_j. Apabila setiap pasang simpul di dalam graf terhubung maka graf dikatakan graf terhubung. Graf berarah G dikatakan terhubung jika graf tak berarahnya terhubung.
9. Graf Berbobot
   Graf berbobot merupakan graf yang sering digunakan dalam pemecahan permasalahan. Graf berbobot memiliki suatu nilai pada setiap sisinya. Graf berbobot juga dapat memiliki arah yang disebut sebagai graf berarah berbobot.

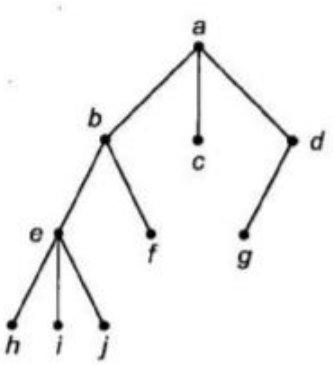
B. Pohon

Pohon adalah graf tak-berarah terhubung yang tidak mengandung sirkuit. Istilah pohon pertama kali ditemukan oleh seorang matematikawan Inggris, Arthur Cayley, pada tahun 1857. Gambar 1 adalah contoh graf yang merupakan pohon dan graf yang bukan pohon. G2 bukan pohon, sedangkan sisanya merupakan pohon.



Gambar 1. Perbandingan Graf dan Pohon

Pada kebanyakan aplikasi pohon, simpul tertentu pada pohon diperlakukan sebagai akar (root) dan sisi-sisi pohon yang mengikuti akar tersebut diberi arah menjauhi akar tersebut. Pohon seperti ini dinamakan pohon berakar (rooted tree). Sebuah pohon tak-berakar dapat diubah menjadi pohon berakar dengan cara memilih sebuah simpul sembarang pada pohon tersebut sebagai akar sehingga dari sebuah pohon tak-berakar dapat dibuat berbagai pohon berakar yang berbeda tergantung dapat simpul yang dipilih sebagai akar. Gambar 2 adalah contoh pohon berakar.



Gambar 2. Contoh Pohon

Pohon berakar memiliki beberapa terminologi sebagai berikut :

1. Anak (child atau children) dan Orangtua (parent)
   Simpul y pada suatu pohon berakar dikatakan anak dari simpul x pada pohon yang sama jika terdapat sisi dari simpul x ke y dan x disebut sebagai orangtua dari y. Pada Gambar 2, b, c, dan d adalah anak-anak dari simpul a, dan a adalah orangtua dari dari b, c, dan d.
2. Lintasan (path)
   Lintasan adalah runtunan simpul-simpul yang dilalui dari suatu simpul ke simpul lainnya sedemikian sehingga setiap simpul yang dilalui merupakan orangtua dari simpul yang berikutnya. Pada Gambar 2, lintasan dari a ke j adalah a, b, e, j.

3. Keturunan (descendant) dan Leluhur (ancestor)
   Jika terdapat lintasan dari simpul x ke simpul y di dalam pohon, maka x adalah leluhur dari simpul y, dan y adalah keturunan dari simpul x. Pada Gambar 2, b adalah leluhur dari simpul h, dan simpul h adalah keturunan dari simpul b.
4. Saudara kandung (sibling)
   Dua buah simpul dikatakan sebagai saudara kandung satu sama lain jika kedua simpul tersebut memiliki orangtua yang sama. Pada Gambar 2, simpul h adalah saudara kandung simpul i.
5. Upapohon (subtree)
   Upapohon adalah upagraf dari suatu pohon berakar yang mengandung suatu simpul pada pohon berakar sebagai akar dan semua keturunannya beserta semua sisi dalam semua lintasan yang bersasal dari simpul tersebut.
6. Derajat (degree)
   Derajat sebuah simpul pada pohon berakar adalah jumlah anak atau upapohon pada simpul tersebut. Pada Gambar 2 derajat simpul e adalah tiga, dan derajat simpul g adalah nol.
7. Daun (leaf)
   Daun adalah simpul yang berderajat nol (atau tidak mempunyai anak). Pada Gambar 2, simpul h, i, j, f, c, g adalah daun.
8. Simpul Dalam (internal nodes)
   Simpul dalam adalah simpul yang mempunyai anak. Pada Gambar 2, simpul e, b, d adalah simpul dalam.
9. Aras (level) atau Tingkat
   Akar beraras nol, sedangkan aras simpul lainnya adalah satu ditambah panjang lintasan dari akar ke simpul tersebut.
10. Tinggi (height) atau Kedalaman (depth)
    Tinggi atau kedalaman adalah aras maksimum dari suatu pohon berakar. Pohon pada Gambar 3 mempunyai tinggi empat.

C. Minimum Spanning Tree

Minimum spanning tree adalah himpunan bagian dari himpunan garis-garis (edge) suatu graf berbobot tak berarah yang menghubungkan semua titik tanpa membentuk sirkuit dan dengan total bobot minimum. Minimum spanning tree didapatkan dari memutus sirkuit-sirkuit dari sebuah graf. Setiap graf terhubung pasti memiliki sebuah minimum spanning tree. Akan tetapi, minimum spanning tree ini belum tentu tunggal. Terdapat dua algoritma minimum spanning tree yang umum, yaitu algoritma Prim dam algoritma Kruskal. Berikut adalah langkah-langkah membuat minimum spanning tree dengan algoritma Prim :

1. Ambil sisi dari graf G yang berbobot minimum, masukkan ke dalam T.
2. Pilih sisi (u, v) yang mempunyai bobot minimum dan bersisian dengan simpul di T, tetapi (u, v) tidak membentuk sirkuit di T. Masukkan (u, v) ke dalam T.
3. Ulangi langkah 2 sebanyak $n - 2$ kali.

Berikut adalah langkah-langkah membuat minimum spanning tree dengan algoritma Prim :

1. Urutkan sisi-sisi dari graf secara menaik berdasarkan bobotnya.
2. Inisialisasi T kosong.
3. Pilih sisi (u, v) dengan bobot minimum yang tidak membentuk sirkuit di T. Tambahkan (u, v) ke dalam T.
4. Ulangi langkah 3 sebanyak $n - 1$ kali.

D. Kompleksitas Algoritma

Kompleksitas algoritma dinilai dari beberapa tinjauan. Tinjauan kompleksitas yang mungkin dilakukan terhadap suatu algoritma adalah kompleksitas ruang dan kompleksitas waktu . Kommpleksitas waktu $T(n)$ merupakan pengurukran terhadap jumlah tahapan komputasi yang dibutuhkan untuk menjalankan algoritma sebagai fungsi dari ukuran masukan n. Kompleksitas ruang $S(n)$ dapat diukur melalui memori yang digunakan oleh struktur data yang terdapat di dalam algoritma sebagai fungsi dari ukuran masukan n.

Kompleksitas waktu dihitung berdasarkan berapa kali suatu operasi dilaksanakan dalam sebuah algoritma sebagai fungsi dengan ukuran masukan n. Operasi yang diperhitungkan dalam kompleksitas waktu suatu algoritma hanya operasi-operasi khas yang mendasari algoritma tersebut. Adapun operasi-operasi tersebut adalah operasi aritmatika sederhana dan operasi perbadingan.

Kompleksitas waktu digolongkan menjadi 3 macam, yaitu $T_{max}(n)$ atau kompleksitas waktu untuk kasus terburuk, $T_{avg}(n)$ atau kompleksitas waktu untuk kasus rata-rata, dan $T_{min}(n)$ atau kompleksitas waktu untuk kasus terbaik. Nilai masing-masing besaran tersebut sangat beragam untuk tiap-tiap algoritma. Sebagai contoh, pada algoritma sequential search, $T_{min}(n) = 1$, $T_{max}(n) = n$, dan $T_{avg}(n) = 0.5(n + 1)$.

Notasi untuk kompleksitas waktu asimptotik adalah "O besar" (Big-O). Dalam hal ini "$T(n) = O(f(n))$" bermakna bahwa orde maksimum $T(n)$ adalah $f(n)$, atau dengan ekspresi lain dapat dinyatakan sebagai $T(n) \leq C(f(n))$ untuk $n \geq n_0$. Dalam hal ini $f(n)$ adalah batas lebih atas dari $T(n)$ untuk n yang besar. Urutan spektrum kompleksitas waktu algoritma dari kecil ke besar adalah sebagai berikut.

No.	Kelompok Algoritma	Nama
1.	$O(1)$	Konstan
2.	$O(\log n)$	Logaritmik
3.	$O(n)$	Linear
4.	$O(n \log n)$	N log N
5.	$O(n^2)$	Kuadratik
6.	$O(n^3)$	Kubik
7.	$O(a^n)$	Eksponensial
8.	$O(n!)$	Faktorial

Tabel 1. Kelompok Kompleksitas

III. ALGORITMA PRIM

Akan dianalisis kompleksitas algoritma Prim dengan asumsi, graf yang akan dianalisis memiliki n buah simpul dan m buah sisi. Perhatikan bahwa nilai m terbesar yang mungkin adalah $\frac{n(n-1)}{2}$. Jika kita mencari, panjang sisi yang minimum dengan iterasi semua sisi yang mungkin akan memerlukan kompleksitas waktu $O(m)$. Karena diperlukan $n - 1$ sisi, maka kompleksitas waktu keseluruhannya adalah $O(mn)$. Pada kondisi terburuknya kompleksitasnya dapat menjadi $O(n^3)$ yang mana cukup

lambat. Terdapat cara untuk mengoptimasi algoritma Prim ini. Ide utamanya adalah kita tidak perlu mengiterasi semua sisi yang ada, cukup sisi tertentu saja. Algoritma yang pertama akan lebih efektif pada graf yang padat. Sedangkan, algoritma kedua akan lebih efektif pada graf yang tidak padat.

A. Algoritma Prim pada Graf Padat

Pada algoritma ini, graph direpresentasikan menggunakan adjacency matrix karena lebih efektif pada graf yang padat. Ide algoritma ini adalah setiap simpul yang belum masuk ke dalam tree, simpan sisi minimum yang terhubung dengan simpul yang sudah ada di dalam tree. Hal ini menyebabkan kita hanya perlu $O(n)$ untuk mencari sisi baru yang akan dimasukkan setiap langkahnya. Setelah menambahkan sisi, nilai-nilai sisi minimum harus diperbarui. Perhatikan bahwa nilai ini pasti monoton turun. Memperbarui nilai sisi minimum ini dapat dilakukan dalam $O(n)$ karena kita hanya perlu mengecek nilai dari sisi yang dibentuk oleh simpul baru di tree dengan simpul yang belum ada di tree. Oleh karena itu, kompleksitas waktu algoritma ini adalah $O(n^2)$.

```cpp
int n;
vector<vector<int>> adj;
const int INF = 1000000000;

struct Edge {
    int w = INF, to = -1;
};

void prim() {
    int total_weight = 0;
    vector<bool> selected(n, false);
    vector<Edge> min_e(n);
    min_e[0].w = 0;

    for (int i=0; i<n; ++i) {
        int v = -1;
        for (int j = 0; j < n; ++j) {
            if (!selected[j] && (v == -1 || min_e[j].w < min_e[v].w))
                v = j;
        }

        if (min_e[v].w == INF) {
            cout << "No MST!" << endl;
            exit(0);
        }

        selected[v] = true;
        total_weight += min_e[v].w;
        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (int to = 0; to < n; ++to) {
            if (adj[v][to] < min_e[to].w)
                min_e[to] = {adj[v][to], v};
        }
    }

    cout << total_weight << endl;
}
```

Gambar 3. Implementasi Algoritma Prim Alternatif 1 Dalam Bahasa C++

Adjacency matrix adj[][] menyimpan weight setiap sisi dan bernilai INF jika tidak ada sisi antara dua simpul. Array selected[] berisi boolean yang menandakan apakah simpul sudah ada dalam tree atau belum. Array min_e[] menyimpan weight minimum dari simpul yang belum ada di tree ke simpul yang sudah ada di tree.

B. Algoritma Prim pada Graf Tidak Padat

Pada algoritma ini, graf direpresentasikan dengan adjacency list karena pada graf yang tidak padat, adjacency list lebih efisien dalam penggunaan memori. Pada dasarnya, ide yang digunakan pada algoritma ini mirip dengan algoritma sebelumnya. Idenya adalah untuk mencari beban minimum dari sisi yang simpulnya belum ada di tree dengan yang sudah ada di tree. Akan tetapi, pendekatan pencarian minimum dan memperbaruinya sedikit berbeda. Simpul yang belum dipilih akan disimpan ke dalam sebuah priority queue, yang diimplementasikan dengan sebuah set. Oleh karena itu, kita dapat mencari sisi yang memiliki beban minimum dalam $O(\log n)$. Akan tetapi memperbarui nilai minimum memerlukan $O(n \log n)$. Oleh karena itu, kompleksitas waktu totalnya adalah $O(m \log n)$.

```cpp
const int INF = 1000000000;

struct Edge {
    int w = INF, to = -1;
    bool operator<(Edge const& other) const {
        return make_pair(w, to) < make_pair(other.w, other.to);
    }
};

int n;
vector<vector<Edge>> adj;

void prim() {
    int total_weight = 0;
    vector<Edge> min_e(n);
    min_e[0].w = 0;
    set<Edge> q;
    q.insert({0, 0});
    vector<bool> selected(n, false);
    for (int i = 0; i < n; ++i) {
        if (q.empty()) {
            cout << "No MST!" << endl;
            exit(0);
        }

        int v = q.begin()->to;
        selected[v] = true;
        total_weight += q.begin()->w;
        q.erase(q.begin());

        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (Edge e : adj[v]) {
            if (!selected[e.to] && e.w < min_e[e.to].w) {
                q.erase({min_e[e.to].w, e.to});
                min_e[e.to] = {e.w, v};
                q.insert({e.w, e.to});
            }
        }
    }

    cout << total_weight << endl;
}
```

Gambar 4. Implementasi Algoritma Prim Alternatif 2 Dalam Bahasa C++

Adjacency list adj[] menyimpan semua sisi pada graph, adj[v] berisi simpul yang bertetangga dengan v. Array min_e[v] menyimpan sisi dengan beban minimum dari simpul v dengan simpul di dalam tree. Priority queue q yang diimplementasikan dengan set, berisi simpul yang belum terpilih.

IV. ALGORITMA KRUSKAL

Akan dianalisis kompleksitas algoritma Kruskal dengan asumsi, graf memiliki n buah simpul dan m buah sisi. Pada algoritma pertama, akan diimplementasikan secara langsung prosedur Kruskal seperti yang telah dijelaskan pada teori dasar. Pada algoritma kedua, algoritma Kruskal akan dioptimasi dengan menggunakan struktur data disjoint set union.

A. Algoritma Kruskal

Algoritma ini mengikuti prosedur yang sudah dijelaskan di atas. Array berisi sisi harus diurutkan berdasarkan bebannya terlebih dahulu yang membutuhkan kompleksitas waktu $O(m \log m)$. Untuk setiap sisi, mengecek keanggotaan pada sebuah tree dapat dilakukan dalam $O(1)$ dengan bantuan tree_id. Menggabungkan dua tree dapat dilakukan dalam $O(n)$ dengan iterasi tree_id. Karena penggabungan ini dilakukan sebanyak $n - 1$ kali sesuai banyaknya sisi pada tree. Akibatnya kompleksitas keseluruhan algoritma ini adalah $O(m \log m + n^2)$.

```cpp
struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
vector<int> tree_id(n);
vector<Edge> result;
for (int i = 0; i < n; i++)
    tree_id[i] = i;

sort(edges.begin(), edges.end());

for (Edge e : edges) {
    if (tree_id[e.u] != tree_id[e.v]) {
        cost += e.weight;
        result.push_back(e);

        int old_id = tree_id[e.u], new_id = tree_id[e.v];
        for (int i = 0; i < n; i++) {
            if (tree_id[i] == old_id)
                tree_id[i] = new_id;
        }
    }
}
```

Gambar 5. Implementasi Algoritma Kruskal Alternatif 1 Dalam Bahasa C++

Graf direpresentasikan sebagai array edge berisi tipe bentukan Edges yang berisi dua simpul dan weight. Sisi yang sudah dipilih akan dimasukkan ke dalam array result yang juga berisi tipe bentukan. Array tree_id berguna untuk, tree_id[v] menyimpan banyaknya subtree yang terdapat v di dalamnya.

B. Algoritma Kruskal Dengan DSU

Seperti halnya pada algoritma Kruskal sebelumnya, array yang berisi sisi dari graph diurutkan berdasarkan bebannya terlebih dahulu. Hal ini dapat dilakukan dalam $O(m \log m)$. Lalu

```cpp
vector<int> parent, rank;

void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}

int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}

struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
vector<Edge> result;
parent.resize(n);
rank.resize(n);
for (int i = 0; i < n; i++)
    make_set(i);

sort(edges.begin(), edges.end());

for (Edge e : edges) {
    if (find_set(e.u) != find_set(e.v)) {
        cost += e.weight;
        result.push_back(e);
        union_sets(e.u, e.v);
    }
}
```

Gambar 6. Implementasi Algoritma Kruska Dengan Disjoint Set Union

masukkan setiap simpul ke dalam tree nya masing-masing, tree di sini diimplementasikan menggunakan set. Memasukkan simpul ini cukup dengan fungsi make_set dari struktur data

disjoint set union yang kita buat. Hal ini bisa dilakukan dalam $O(n)$. Sekarang kita hanya perlu iterasi pada semua sisi, jika kedua simpul berada di set yang berbeda, gabungkan kedua set ini. Pengecekan dapat dilakukan dengan find_set dan penggabungan dapat dilakukan dengan union_set keduanya dapat dilakukan dalam $O(1)$. Iterasi dilakukan sebanyak m kali pada kasus terburuk, tetapi dapat dihentikan saat set sudah berukuran $n-1$. Maka, kompleksitas waktunya adalah $O(m \log m + n + m) = O(m \log m)$.

V. Kesimpulan

Keempat algoritma minimum spanning tree ini pada umumnya dapat diimplementasikan pada berbagai persoalan yang ada. Algoritma Prim yang pertama memiliki kompleksitas waktu $O(n^2)$. Algoritma Prim yang kedua memiliki kompleksitas waktu $O(m \log n)$. Algoritma Kruskal memiliki kompleksitas waktu $O(m \log m + n^2)$. Algoritma Kruskal dengan struktur data disjoint set union memiliki kompleksitas $O(m \log m)$. Untuk kompleksitas ruang, keempat algoritma ini tidak memiliki perbedaan yang signifikan.

Implementasi algoritma Kruskal dengan alternatif pertama cenderung sedikit lebih lambat dibandingkan ketiga algoritma lainnya. Untuk kasus ketika graph tidak padat disarankan menggunakan algoritma Prim kedua atau algoritma Kruskal dengan disjoint set union. Akan tetapi, saat graph padat nilai m akan mendekati nilai n^2 sehingga $O(m \log n) = O(n^2 \log n)$ dan $O(m \log m) = O(n^2 \log n^2) = O(n^2 \log n)$. Oleh karena itu, saat graph padat lebih baik digunakan algoritma Prim yang pertama.

VI. Ucapan Terima Kasih

Pertama penulis ingin mengucapkan puji syukur kepada Tuhan Yang Maha Esa karena dengan rahmat dan karunai-Nya penulis dapat menyelesaikan makalah dengan judul “Analisis Kompleksitas Algoritma Minimum Spanning Tree dan Alternatifnya” ini dengan baik. Penulis juga berterima kasih kepada dosen mata kuliah IF2120 Matematika Diskrit, Dr. Ir. Rinaldi Munir, M.T., Dra. Harlili S., M.Sc., dan Dr. Nur Ulfa Maulidevi, S.T, M.Sc. atas bimbingan beliau selama ini dalam mengajar dan memberikan ilmu pada mata kuliah matematika diskrit sehingga penulis mampu membuat makalah ini. Penulis juga berterima kasih kepada rekan-rekan yang telah memberikan semangat dan dorongan kepada penulis.

Referensi

[1] R. Munir, *Matematika Diskrit*. Bandung : Penerbit Informatika, Palasari
[2] W. Gozali & A.F.Aji, Pemrograman Kompetitif Dasar, Jakarta : Nulis Buku Jendela Dunia, 2015.
[3] https://www.ics.uci.edu/~eppstein/161/960206.html , diakses pada 10 Desember 2021 pukul 21.37
[4] https://www.programiz.com/dsa/prim-algorithm, diakses pada 10 Desember 2021, pukul 21.37
[5] https://www.programiz.com/dsa/kruskal-algorithm, diakses pada 10 Desember 2021 pukul 21.37
[6] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2021-2022/matdis21-22.htm , diakses pada 10 Desember 2021, pukul 22.00
[7] Silde Perkuliahan Algoritma dan Struktur Data, Teknik Informatika, Institut Teknologi Bandung

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, 10 Desember 2021

Ubaidillah Ariq Prathama - 13520085