Aplikasi Graf dan *Disjoint Set Union* dalam Keterhubungan Bangunan Kampus Ganesha ITB

Muhammad Atpur Rafif - 13522086[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13522086@std.stei.itb.ac.id

***Abstrak*—Kampus ITB memiliki ukuran lahan yang relatif kecil dibandingkan dengan kampus lainnya. Dengan ketatnya peraturan kendaraan, satu-satunya moda transportasi bagi mahasiswa dalam kampus adalah berjalan kaki. Untuk memudahkan mobilisasi, terdapat jalan bagi pejalan kaki yang memiliki atap untuk melindungi dari cuaca terik maupun hujan. Makalah ini membahas mengenai keterhubungan bangunan dalam Kampus Ganesha ITB menggunakan struktur data graf dan *Disjoint Set Union*. Hasilnya masih terdapat 12 bangunan yang masih belum memiliki akses jalan beratap untuk pejalan kaki.**

***Kata Kunci*—Graph, Disjoint Set Union, Peta, Rust**

I. Pendahuluan

Institut Teknologi Bandung merupakan salah satu kampus ternama di Indonesia. Meskipun begitu, hal ini bukan berarti kampus tersebut memiliki luas lahan yang besar, dengan luas 28 hektar, kampus ini terbilang cukup kecil. Salah satu aturan yang merupakan imbas dari ukuran lahan adalah ketatnya akses kendaraan untuk keluar masuk Kampus Ganesha. Kendaraan yang diperbolehkan masuk hanyalah kendaraan yang memiliki kartu akses, yang tidak mahasiswa dapatkan. Oleh karena itu, satu-satunya moda transportasi dalam kampus yang bisa digunakan mahasiswa hanyalah dengan berjalan kaki.

Gambar 1 Kampus ITB
(Sumber: dokumentasi resmi ITB, diakses pada 11 Desember 2023)

Terdapat sebuah permasalahan ketika cuaca tidak mendukung, misalkan hujan. Seorang mahasiswa bisa kehujanan saat mereka sedang mobilisasi dari sebuah kelas ke kelas berikutnya apabila tidak terdapat sebuah jalan dengan atap diantara dua bangunan tempat kelas dilaksanakan.

Oleh karena itu, penulis makalah ingin mencari tahu apakah seluruh bangunan dalam Kampus ITB Ganesha terhubung dengan jalan tertutup bagi pejalan kaki. Teori yang digunakan untuk menjawab rasa keingintahuan ini berasal dari sebuah materi dalam Matematika Diskrit, yaitu graf.

II. Landasan Teori

A. Graf

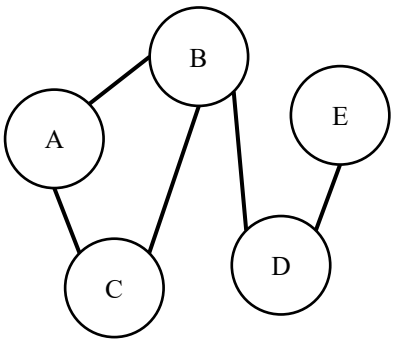
Graf merupakan sebuah konsep dalam matematika diskrit yang terdiri dari *node* dan *edge*. Pendefinisian graph dituliskan dengan *tuple* yang berisi dua komponen seperti berikut [1]:

$$G = (V, E)$$

Nilai dari V adalah kumpulan dari *node*, sedangkan E merupakan kumpulan dari *edge*. Setiap *edge* terdiri dari dua buah *node* yang disebut dengan *endpoint*. Graf ini dapat merepresentasikan hubungan dari objek. Misalkan hubungan pertemanan dapat direpresentasikan menggunakan definisi graf berikut:

$$G = (\{A, B, C, D, E\}, \{(A, B), (A, C), (B, C), (B, D), (D, E)\})$$

Pada kasus ini, *node* merepresentasikan orang, sedangkan *edge* memiliki arti hubungan pertemanan. Sebuah *edge* yang berupa (x, y) memiliki arti x berteman dengan y, begitu pula sebaliknya. Terdapat banyak variasi dari graf, seperti graf berganda, dan berarah.



Terdapat beberapa terminologi dalam graf selain *node* dan *edge* diatas, berikut merupakan penjelasan masing-masing:

1. Ketetanggan
   Dua buah *node* bertetangga apabila kedua *node* tersebut merupakan *endpoint* dari sebuah *edge*.
2. Bersisian
   Sebuah *node* dengan *edge* bersisian apabila *node* tersebut merupakan salah satu *endpoint* dari *edge* tersebut.

B. Pohon

Pohon merupakan graf yang memiliki batasan tambahan. Sebuah pohon merupakan graf terhubung yang tidak memiliki sirkuit [2]. Graf terhubung memiliki arti terdapat sebuah lintasan untuk dua *node* mana saja. Sedangkan sirkut merupakan lintasan dari graf yang kembali ke titik awal lintasan. Lalu lintasan adalah kumpulan *node* atau *edge* yang dilewati secara kontinu, misalkan pada contoh pertemanan diatas, A → B → C merupakan lintasan. Apabila kita kembali ke A, maka hal tersebut adalah sirkuit. Definisi dari graf tidaklah memiliki sirkuit, sehingga pada contoh pertemanan diatas bukanlah sebuah pohon.

Sebuah graf yang memenuhi syarat tidak memiliki sirkuit, namun merupakan graf tidak terhubung, maka kita bisa melihatnya sebagai kumpulan dari pohon. Kumpulan ini disebut dengan *forest* atau hutan.

C. Disjoint Set Union

Sebuah struktur data yang dapat menyatakan kelompok terpisah dari kumpulan objek [3], biasanya disingkat DSU. Misalkan kita memiliki kumpulan permen berwarna, kita bisa mengelompokannya secara terpisah berdasarkan warna setiap permen. Pada implementasinya, terdapat dua buah operasi yang bisa dilakukan, yaitu *find* dan *join*. Berikut penjelasan masing-masing operasi:

1. *find*
   Sebuah fungsi yang mengembalikan sebuah objek yang menyatakan kelompok sebuah objek. Pada permisalan diatas, fungsi ini akan mengembalikan permen dengan ukuran terkecil yang memiliki warna sama dengan permen yang ditanyakan/masukan sebagai argumen fungsi.
2. *join*
   Sebuah prosedur yang menggabungkan dua kelompok yang dianggap berwarna sama. Pada permisalan diatas, contohnya kita telah memiliki kelompok merah muda dan tua. Kemudian kita berpikir bahwa kedua merah tersebut bisa dinyatakan sama, karena tidak ada batas eksplisit diantara dua merah tersebut. Kita bisa menggabungkan dua kelompok tersebut menggunakan prosedur *join* ini.

Penjelasan mungkin lebih mudah dipahami dengan menggunakan contoh secara konkrit. Kita menggunakan contoh permen diatas, dengan diberikan informasi mengenai kemiripan warna permen dituliskan sebagai *edge* sebuah graph, lebih tepatnya edge (x, y) memiliki arti permen x memiliki warna yang mirip dengan permen y. Sedangkan permen itu sendiri merupakan *node*. Angka 0-9 merupakan *node* permen, dan berikut merupakan *edge* dari representasi graf tersebut:

(1, 2); (4, 5); (4, 6); (8, 9); (1, 7); (8, 3); (0, 4)

Kemudian dari data yang diberikan, ditanyakan kelompok warna dari seluruh permen. Struktur data DSU dapat digunakan untuk menyelesaikan permasalahan ini. Asumsi awal yang digunakan adalah seluruh permen memiliki warna yang berbeda. Kemudian sebuah kelompok permen direpresentasikan oleh permen dengan indeks paling kecil, karena tidak diberikan informasi mengenai warna setiap permen secara pasti, namun hanya diberikan informasi hubungan antara warna dari dua permen. Hal ini berarti, kumpulan permen (1, 3, 4) dapat dituliskan memiliki warna 1, atau agar lebih mudah dituliskan sebagai kelompok 1.

Dengan menggunakan asumsi dan informasi diatas, awalnya kita bisa menuliskan seluruh permen memiliki warna yang berbeda. Lalu satu-persatu data dari *edge* kita gunakan untuk menggabungkan dua kelompok permen. Kita bisa menggunakan *array* untuk mengimplementasi struktur data ini, dengan *array* awal sebagai berikut:

Kelompok, awal	0	1	2	3	4	5	6	7	8	9
Permen (indeks)	0	1	2	3	4	5	6	7	8	9

Kita mulai membaca data *edge* pertama, yaitu (1, 2). Menggunakan data ini kita tahu bahwa kelompok permen 1 sama dengan kelompok permen 2. Penggabungan dua kelompok ini bisa dilakukan menggunakan prosedur *join*. Pertama prosedur ini mencari tahu kelompok permen 1 (yaitu 1 juga), dan kelompok permen 2 (yaitu 2 juga). Sekarang kita bisa melakukan penggabungan dengan mengubah kelompok yang lebih besar menjadi yang lebih kecil, sehingga *array* akan berbentuk seperti berikut ini:

Kelompok, (1,2)	0	1	1	3	4	5	6	7	8	9
Permen (indeks)	0	1	2	3	4	5	6	7	8	9

Selagi prosedur *join* dilakukan, terkadang nilai dari kelompok permen secara tidak langsung juga dilakukan pembaruan kelompok. Hal ini untuk mempercepat fungsi *find* yang bertujuan mendapatkan kelompok dari sebuah permen. Terdapat kejadian dimana sebuah kelompok tidak merepresentasikan kelompok asli dari sebuah permen. Misalkan permen 5 memiliki kelompok 3, padahal permen 3 sendiri merupakan kelompok 1. Singkatnya dibutuhkan dua kali pencarian untuk mendapatkan kelompok sebenarnya (5 → 3 → 1). Permasalahan ini dapat diselesaikan dengan mengubah nilai kelompok untuk permen 5 langsung menjadi satu, sehingga pencarian kelompok hanya perlu dilakukan sekali (5 → 1). Salah satu kasus hal ini dilakukan terlihat pada tahap ketika memproses *edge* (0, 4) dibawah ini. Awalnya permen (4,5,6) merupakan kelompok 4, namun setelah *edge* diproses, permen (0, 4, 5, 6) memiliki kelompok 0.

Dengan menggunakan seluruh aturan diatas, kita bisa melihat seluruh tahap pemrosesan *edge* dan prosedur *join* yang menggunakan *find* menghasilkan tahapan sebagai berikut:

Permen (indeks)	0	1	2	3	4	5	6	7	8	9
Kelompok, (4,5)	0	1	1	3	4	4	6	7	8	9
Kelompok, (4,6)	0	1	1	3	4	4	4	7	8	9
Kelompok, (8, 9)	0	1	1	3	4	4	4	7	8	8
Kelompok, (1, 7)	0	1	1	3	4	4	4	1	8	8
Kelompok, (8, 3)	0	1	1	3	4	4	4	1	3	3
Kelompok, (0, 4)	0	1	1	3	0	0	0	1	3	3
Permen (indeks)	0	1	2	3	4	5	6	7	8	9

Setelah melakukan seluruh tahapan diatas, kita memiliki tiga buah kelompok bernama 0, 1, dan 3. Masing-masing kelompok memiliki permen dengan indeks sebagai berikut:

0: 0, 4, 5, 6
1: 1, 2, 7
3: 3, 8, 9

Disini kita tidak mempedulikan warna sebenarnya dari setiap permen, namun hanya kelompok warna permen.

D. Rust

Sebuah bahasa pemrograman yang dekat dengan bahasa mesin, atau biasanya disebut dengan *low-level langguage*. Karenanya, program yang ditulis menggunakan bahasa ini memiliki performa lebih baik dibandingkan *high-level langguage* seperti *python*. Namun terdapat *trade-off*, yaitu penulisan yang lebih susah dibandingkan bahasa *high-level*.

Salah satu contoh kekurangannya, memori dari sebuah variable dikelola secara manual oleh penulis program. Biasanya, terdapat dua cara general untuk mengelola variable. Pada bahasa *high-level*, mereka menggunakan *garbage-collector* secara otomatis. Hal ini dilakukan dengan melakukan *reference-counting* untuk mengetahui apakah sebuah variable sudah tidak mungkin digunakan lagi. Apabila sudah tidak digunakan, maka program akan menghapus memori yang digunakan dan dapat dipakai oleh bagian program yang lain. Kontrasnya, pada bahasa *low-level*, memori dikelola secara manual menggunakan *low-level API* seperti *malloc*, atau *free* pada bahasa C.

Pendekatan yang dilakukan oleh Rust berbeda dari dua cara general yang disebutkan diatas. Pada bahasa ini menggunakan sebuah konsep *ownership* atau kepemilikan [4]. Sebuah nilai yang bukan tipe dasar seperti integer, disimpan pada *heap*. Setiap nilai memiliki tepat satu *owner*, yang merupakan sebuah *variable*. Apabila *owner* telah keluar dari *scope*, misalkan ketika fungsi berakhir, maka Rust akan membersihkan memori tersebut sehingga dapat digunakan oleh bagian program yang lain. Pembersihan ini sama seperti memanggil *free* pada nilai ketika *owner* keluar dari *scope*.

Pemindahan *owner* adalah hal yang biasa terjadi dalam bahasa ini. Hal ini biasanya disebut dengan *move*, dengan berikut merupakan contohnya. Awalnya *string* dimiliki oleh variabel `a`, namun selanjutnya *string* akan berpindah kepemilikan menjadi milik b. Pada titik ini, nilai a tidak valid lagi, karena nilai *string* telah berpindah ke `b`. Contoh lain pemindahan *ownership* ketika sebuah fungsi menerima *string* langsung sebagai parameter. Pemilik nilai string akan berpindah ke parameter tersebut, sehingga variabel `b` tidak akan valid setelah pemanggilan fungsi. Pemindahan *ownership* juga terjadi ketika mengembalikan nilai secara langsung dari sebuah fungsi. Pada kasus ini, kepemilikan nilai akan diberikan kepada variabel pemanggil fungsi.

```
let a = String::from("Hello, world");
let b = a;
some_function(b);
```

Terkadang kita tidak menginginkan sebuah fungsi untuk mengambil *ownership* dari sebuah nilai. Kita hanya ingin sebuah fungsi hanya "meminjam" sebuah nilai dan mengembalikan nilai tersebut ke pemanggil. Hal ini sudah menjadi sebuah fitur dalam bahasa ini. Fitur tersebut adalah *reference*. Contohnya fungsi `greet` dibawah ini menggunakan `&str` sebagai parameter input. Dalam hal ini, fungsi tidak akan mengambil *ownership* dari variabel `a`, namun hanya meminjam nilainya. Sehingga variabel a masih valid setelah pemanggilan fungsi.

```
fn greet(name: &str) {
  println!("Hello {name}!");
}

fn main() {
  let a = String::from("Afif");
  greet(&a);
  // Variable a still valid
}
```

Seluruh variabel yang telah dijelaskan sebelumnya memiliki sifat *immutable*, yang artinya nilai internal mereka tidak dapat diubah. Perhatikan disini bahwa *ownership* dari nilai dapat berubah, seperti variabel yang menampungnya, tapi nilai internal sendiri tidak dapat berubah, seperti string "Afif" tidak bisa ditambahkan dengan string lain. Dalam spektrum berlawanan, terdapat sifat *mutable* yang artinya sebuah nilai dapat diubah secara internal. Pada bahasa Rust, hal ini dilakukan dengan menambahkan *keyword* `mut` pada deklarasi variabel. Dengan begitu, *string* dibawah ini bisa ditambahkan dengan *string* lainnya, dan nilai tersebut berubah secara internal, yang artinya hasil penggabungan dua *string* tersebut akan tersimpan pada variabel `a`.

```
let mut a = String::from("Hello");
a.push_str(", world!");
```

Pada bahasa ini, untuk membuat tipe data bentukan dapat menggunakan `struct`. Tidak terdapat *object-oriented paradigm* secara eksplisit menggunakan *class* dalam bahasa ini, namun hal ini dapat diraih menggunakan *method* yang hanya berlaku pada tipe data tertentu. Pendefinisian *method* dilakukan dengan *keyword* `impl`. Nilai dari `self` pada *method* merujuk pada konteks fungsi tersebut dipanggil. Sedangkan *method* yang tidak memiliki parameter `self` sama seperti *static method* pada *class*.

```rust
struct Frac {
  numerator: i32,
  denominator: i32
}

impl Frac {
  fn from(numerator: i32, denominator:i32) -> Frac {
    Frac {
      numerator,
      denominator
    }
  }

  fn multiply(self, other: &Frac) -> Frac {
    Frac {
      numerator: self.numerator * other.numerator,
      denominator: self.denominator * other.denominator
    }
  }
}

fn main() {
  let a = Frac::from(1, 2);
  let b = Frac::from(2, 3);
  let c = a.multiply(&b);
}
```

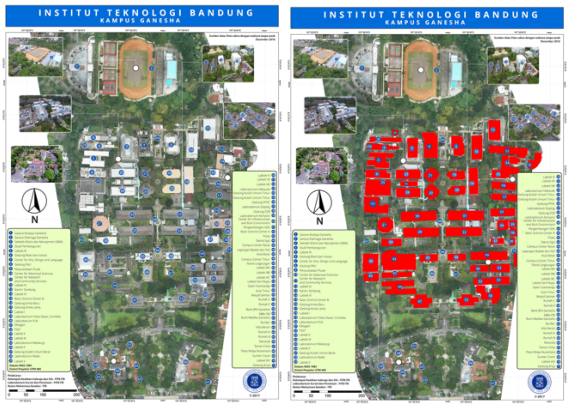
Pada spesifikasi fungsi `multiply`, parameter `self` adalah nilai secara langsung, tanpa menggunakan *reference*. Prinsip *ownersip* berlaku disini. Hal ini berarti variabel `a` tidak valid setelah melakukan perkalian. Apabila kita menggunakan `&self`, maka variable a masih akan tetap valid setelah perkalian. Sedangkan pada parameter `other`, kita menggunakan *reference*, sehingga variable `b` masih valid setelah operasi perkalian. Kevalidan variable sendiri memiliki arti, apabila kita mengakses variable tersebut ketika tidak valid, maka *compiler* tidak akan mengkompilasi program kita, dan mengeluarkan error. Terdapat masih banyak fitur Rust yang tidak sempat dibahas dalam makalah ini.

III. Eksperimen

A. Pencarian Data

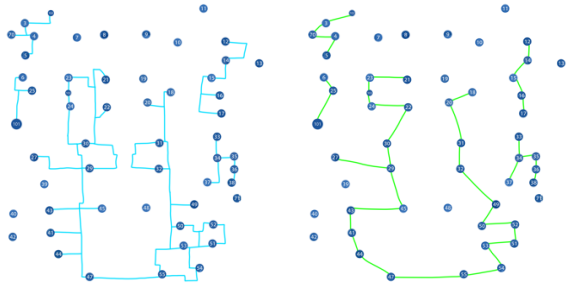
Data pertama yang diperlukan untuk mencari tahu hubungan antar bangunan berdasarakan jalan tertutup agar bisa dibentuk representasi graf adalah peta dari Kampus ITB Ganesha. Peta tersebut sudah dipublikasikan oleh pihak ITB dan dapat diakses pada link ini. Peta yang digunakan berasal dari tahun 2016, seperti yang informasi yang diberikan pada peta. Gambar dari peta ini telah menggunakan proyeksi sedemikian rupa sehingga sisi samping gedung tidak terlihat dan menutupi hal yang ada dibawahnya. Hal ini kontras dengan foto satelit yang didapatkan dari Google Maps.

Setelah gambar peta didapatkan, dilakukan identifikasi untuk bangunan yang dapat digunakan sebagai tempat kelas dilaksanakan. Hal ini dilakukan menggunakan aplikasi pemrosesan gambar secara manual dengan memberikan bentuk berwarna merah untuk bangunan yang dimaksud. Tidak lupa juga membuat *overlay* untuk seluruh label angka agar tidak tertutupi oleh bentuk identifikasi. Berikut merupakan gambar yang telah didapatkan hingga titik ini:



Gambar 2 Peta asli ITB beserta identifikasi bangunan pada lingkungan kampus (Sumber gambar kiri: website resmi ITB, diakses pada 11 Desember 2023)

Identifikasi bangunan digunakan sebagai *node* dalam representasi graf, kemudian kita gambarkan seluruh jalan dengan atap yang terlihat pada peta. Sebuah *edge* (x, y) memiliki arti terdapat jalan beratap dari bangunan x menuju bangunan y. Penulis sadar bahwa terdapat jalan beratap baru yang menghubungan gedung CAS dan Labek 1. Namun hal ini tidak ditambahkan, seluruh jalan yang diidentifikasi hanyalah jalan yang telihat pada peta. Kemudian dilakukan simplifikasi dari grah ganda, menjadi hutan. Hal ini dilakukan dengan menghapus beberapa *edge* dengan tujuan untuk mempermudah penulisan *edge* yang ada. Dalam hal ini, kita tidak peduli banyaknya jalan yang ada, namun hanya mempedulikan adanya jalan diantara dua bangunan yang ada. Identifikasi jalan dan simplifikasinya dapat dilihat pada dua gambar berikut:



Gambar 3 Representasi node sebagai bangunan dan edge yang merupakan jalan penghubung tertutup beserta simplifikasinya menjadi pohon merentang

Total banyaknya *node* yang akan dimasukan kedalam program adalah 55, sedangkan untuk banyaknya *edge* setelah simplifikasi menjadi hutan adalah 38. Nantinya seluruh konfigurasi graf akan dituliskan pada file, *node* dan *edge* memiliki filenya masing-masing untuk mempermudah pembacaan oleh program.

B. Program

Terdapat dua buah struktur data bentukan dalam program yang dibuat dalam bahasa Rust. Pertama adalah graf, dan yang kedua adalah DSU. Struktur data graf dinamakan dengan `Graph`. Sesuai dengan definisi sebelumnya, tipe data ini memiliki dua *field*, yaitu *node* dan *edge* yang masing-masing menyimpan data sesuai namanya. Digunakan tipe data `Vec` untuk menampung kumpulan dari data.

Dalam definisinya, terdapat sebuah *method* bernama `from_file` yang berfungsi untuk membaca konfigurasi sebuah graf dari file. Fungsi ini akan mengembalikan graf secara

langsung, tanpa menggunakan *reference*. Sehingga pemanggil fungsi akan menjadi *owner* dari nilai yang dikembalikan oleh fungsi tersebut. Selain itu, pada definsi tipe datanya, menggunakan *string* untuk mempermudah pembacaan dari *file*. Sehingga tidak diperlukan konversi dari *string* menjadi *integer*. Pada kode dibawah, hanya dituliskan spesifikasi fungsi, tidak dengan implementasinya. Link untuk seluruh kode akan diberikan pada bagian akhir makalah. Selain itu, pada kode tampilan dibawah tidaklah valid, terdapat detail implementasi yang dihapus (seperti *module definition*) untuk mempermudah penjelasan.

```
struct Graph {
    node: Vec<String>,
    edge: Vec<(String, String)>,
}

impl Graph {
    fn from_file(node_path: &str, edge_path: &str) -> Graph;
}
```

Selanjutnya terdapat tipe data bentukan yang bernama DSU yang berfungsi sebagai *Disjoint Set Union*. Pastinya terdapat dua *method* yang telah dijelaskan sebelumnya, yaitu `find` dan `join`. Kemudian terdapat tambahan *method* yang bernama `insert` yang berfungsi untuk menambahkan elemen baru pada data tersebut. Selain itu, terdapat *method* `from_graph` yang menerima graf sebagai argumen dan menghasilkan DSU seperti pada penjelasan di landasan teori. Kebanyakan *method* pada tipe ini menggunakan *keyword* mut dalam parameternya, karena dalam operasi fungsi tersebut, kita mengubah nilai secara internal. Selain itu, kita menggunakan `HashMap` dibandingkan dengan array pada penjelasan, dikarenakan indeks yang digunakan sekarang adalah *string* [5].

```
struct DSU {
    data: HashMap<String, String>,
}

impl DSU {
    fn find(&mut self, key: &String) -> String;
    fn join(&mut self, a: &String, b: &String);
    fn insert(&mut self, a: &String);
    fn from_graph(graph: &Graph) -> DSU;
    fn group(&self) -> Vec<Vec<String>>;
}
```

Pada fungsi utama program akan memanggil `Graph::from_file` lalu mengubahnya menjadi DSU dengan `DSU::from_graph`, dan nantinya akan ditampilkan banyaknya *node* pada setiap kelompok beserta seluruh *node* pada sebuah kelompok pada *terminal output* setelah melakukan *grouping* dan *sorting*.

IV. HASIL

Program yang telah dijelaskan sebelumnya dijalankan dan menghasilkan output sebagai berikut:

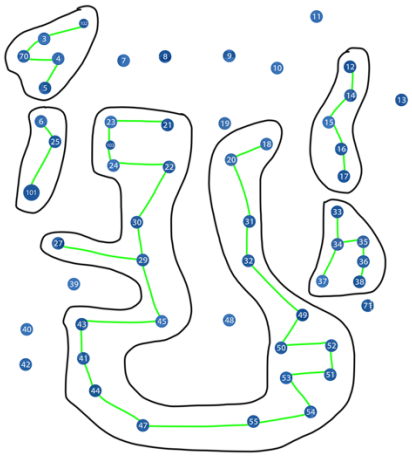
```
24      [22, 32, 44, 51, 27, 45, 18, 49, 23, 53, 24, 52, 29, 41, 30, 43, 47, 21, 55, 50, 31, 54, 103, 20]
6       [33, 35, 38, 34, 36, 37]
5       [3, 70, 5, 4, 102]
5       [17, 15, 12, 14, 16]
3       [101, 25, 6]
1       [8]
1       [48]
1       [9]
1       [42]
1       [7]
1       [13]
1       [39]
1       [40]
1       [71]
1       [19]
1       [11]
1       [10]
```

Data kemudian dirapihkan dan seluruh kelompok yang hanya terdiri dari satu bangunan digabungkan untuk menghemat tempat. Berikut merupakan seluruh kelompok bangunan yang dihubungkan oleh jalan beratap:

24	22, 32, 44, 51, 27, 45, 18, 49, 23, 53, 24, 52, 29, 41, 30, 43, 47, 21, 55, 50, 31, 54, 103, 20
6	33, 35, 38, 34, 36, 37
5	3, 70, 5, 4, 102
5	17, 15, 12, 14, 16
3	101, 25, 6
1	8; 48; 9; 42; 7; 13; 39; 40; 71; 19; 11; 10

Tabel 1 Data hasil pengguna program

Terdapat total 17 kelompok bangunan terpisah pada Kampus Ganesha ITB, 12 diantaranya tidak terhubung dengan bangunan lainnya. Sedangkan untuk 5 sisanya setidaknya terhubung dengan sebuah bangunan lain. Kelompok terbesar terbentuk dari 24 bangunan. Berikut merupakan bentuk dari hutan setelah dilakukan pengelompokan menggunakan program yang mengimplementasi struktur data DSU.



Gambar 4 Hutan yang telah dikelompokkan

V. KESIMPULAN

Berdasarkan data yang didapatkan dari hasil eksperimen, masih terdapat banyak bangunan yang tidak dapat diakses melalui jalan beratap, lebih tepatnya ada 12 bangunan. Hal ini dapat diatasi dengan membangun jalan beratap baru. Namun terdapat banyak tantangan untuk menerapkan hal ini, seperti budget kampus. Bahkan jika budget kampus bukan masalah, terdapat permasalahan lain, yaitu menyeimbangkan akses pejalan kaki dengan akses mobil besar seperti truk yang tinggi

VI. Lampiran

Link *Source Code* yang berisi implementasi dari spesifikasi fungsi yang telah dijelaskan dapat diakses pada https://github.com/atpur-rafif/IF2120-graph. Pada folder `img` juga terdapat gambar dari peta dan graf yang digunakan pada makalah ini. Kemudian seluruh gambar merupakan dokumentasi penulis kecuali disebutkan hal lain (termasuk graf dan visualisasi DSU, merupakan buatan penulis). Kode juga dibuat oleh penulis dengan bantuan codeimage untuk melakukan *styling*.

VII. Ucapan Terima Kasih

Saya berterimakasih kepada Allah S.W.T atas izinnya untuk membuat makalah ini. Kemudian kepada orang tua dan keluarga penulis yang telah mendukung penulis sampai titik ini. Lalu tak lupa dosen Matematika Diskrit, untuk Dr. Ir. Rinaldi, M.T., Dr. Fariska Zakhralativa Ruskanda, S.T., M.T., dan Dr. Nur Ulfa Maulidevi, S.T, M.Sc. yang telah mengajarkan dan menyiapkan materi untuk para mahasiswanya sebagai inspirasi isi dari makalah ini.

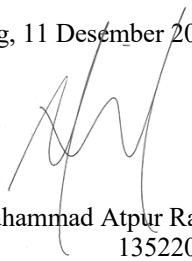
VIII. Referensi

[1] Kenneth H. Rosen, “Discrete Mathematics and Its Applications”. New York: MacGraw-Hill, 2012, pp. 641
[2] Kenneth H. Rosen, “Discrete Mathematics and Its Applications”. New York: MacGraw-Hill, 2012, pp. 745
[3] Thomas H. Cormen, “Introduction to Algorithms, Fourth Edition”. Cambridge: The MIT Press, 2022, pp. 520.
[4] Steve Klabnik, “The Rust Programming Language”. San Francisco: No Starch Press, 2022, ch. 4.0.
[5] Jay Wengrow, “A Common-Sense Guide to Data Structures and Algorithm”. North Carolina: The Pragmatic Bookself, 2020, pp. 113.

IX. 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 Desember 2023



Muhammad Atpur Rafif
13522086