PENERAPAN GRAF UNTUK STRUKTUR DATA HIMPUNAN SALING LEPAS

Andrew Pratomo Budianto - NIM : 13505046

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail: if15046@students.if.itb.ac.id

Abstrak

Dalam menyelesaikan suatu permasalahan, sering kali diperlukan struktur data himpunan yang tidak saling tumpuk (suatu elemen hanya terdapat pada sebuah himpunan, sering disebut sebagai himpunan saling lepas). Disamping itu, struktur data himpunan saling lepas seringkali dibutuhkan untuk menjawab pertanyaan: apakah elemen a dan b berada pada sebuah himpunan yang sama?, dan juga harus dapat menggabungkan isi dari beberapa himpunan menjadi sebuah himpunan (*union*) dalam waktu yang singkat. Makalah ini menjelaskan bagaimana graf dapat digunakan untuk menyelesaikan masalah struktur data himpunan seperti yang disebutkan di atas (himpunan saling lepas).

Kata-kata kunci: Graf, struktur data, himpunan saling lepas, *disjoint set, union*, *find.*

1. PENDAHULUAN

Diberikan sejumlah elemen, sering dijumpai kasus dimana kita harus membagi elemen-elemen tersebut, atau mempartisi mereka menjadi sejumlah bagian terpisah, yang tak saling tumpuk [1].

Struktur data himpunan saling lepas adalah sebuah algoritma yang dapat menyelesaikan dua buah operasi yang sangat berguna yaitu: [1]

- **Find**: Menentukan dalam himpunan manakah sebuah elemen berada. Berguna juga dalam menentukan apakah dua buah elemen berada dalam himpunan yang sama.
- **Union**: Menggabungkan (*merge*) dua buah himpunan menjadi sebuah himpunan.

Penggunaan dari struktur data tingkat lanjut ini berkembang secara alami dalam berbagai macam aplikasi, umumnya dimana partisi dan hubungan ekivalensi dibutuhkan, salah satu contoh permasalahan dalam kehidupan sehari-hari yaitu ketika seorang administrator jaringan yang menyimpan catatan akses (*log*) komputer-komputer pada jaringannya, ingin mengetahui apakah sebuah koneksi yang berada pada catatan aksesnya (mungkin merupakan koneksi yang sangat penting dan menyangkut keamanan dalam jaringan yang diawasinya) merupakan koneksi antar komputer yang terkoneksi secara langsung atau tidak langsung [2]. Dalam contoh kecil permasalahan di atas, struktur data himpunan saling lepas akan sangat berguna dalam menemukan solusi. Dimana elemen-elemen tiap himpunan dapat diasumsikan sebagai komputer-komputer yang berada pada jaringan, dan tiap komputer tersebut terpasangkan tepat pada sebuah sub jaringan (himpunan) tertentu. Pada kasus ini, algoritma *find* dapat digunakan untuk menjawab pertanyaan administrator tersebut dengan sangat mangkus. Masih terdapat contoh-contoh lain yang lebih penting dan fundamental dalam kehidupan kita daripada contoh di atas, berbagai contoh aplikasi tersebut akan dibahas lebih mendalam di Bab 6 tentang aplikasi struktur data himpunan saling lepas.

Bagian selanjutnya dari makalah ini diorganisasikan sebagai berikut: Bab 2 akan menerangkan tentang graf, yaitu definisi graf secara singkat, istilah-istilah umum tentang graf, dan bahasan-bahasan graf secara spesifik yang berkaitan dengan tema makalah ini. Berbagai macam bentuk dan macam dari struktur data akan dijelaskan dalam Bab 3 dan pengertian himpunan, bentuk, sifat, dan juga definisi dari himpunan saling lepas akan dibahas pada Bab 4. Pada bab 5 akan dijelaskan bagaimana sebenarnya struktur data himpunan saling lepas yang dibentuk dengan memanfaatkan ilmu graf, beserta algoritma-algoritma yang umum digunakan dalam memproses struktur data himpunan saling lepas tersebut. Selanjutnya pada Bab 6 akan dibahas lebih lanjut mengenai aplikasi-aplikasi yang memanfaatkan kelebihan dari struktur data himpunan saling lepas ini dalam skala luas yang fundamental. Dan sebagai penutup bahasan, pada Bab 7 akan diberikan kesimpulan dari seluruh bahasan yang terdapat pada makalah ini

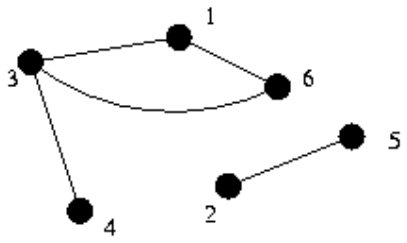
2. GRAF

2.1 Pengertian Graf

Graf didefinisikan sebagai sekelompok simpul-simpul (*nodes/vertices*) V, dan sekelompok sisi (*edges*) E yang menghubungkan sepasang simpul. Bayangkan simpul-simpul tersebut sebagai lokasi-

lokasi, maka himpunan dari simpul-simpul tersebut adalah himpunan lokasi-lokasi yang ada. Dengan analogi ini, maka sisi merepresentasikan jalan yang menghubungkan pasangan lokasi-lokasi tersebut. Himpunan E merupakan semua jalan diantara lokasi-lokasi tersebut [3].

Graf pada umumnya direpresentasikan sebagai berikut: *node*nya digambarkan dengan titik atau lingkaran, dan sisinya digambarkan dengan garis yang menghubungkannya.



Gambar 1. Contoh representasi graf

Pada gambar 1 terlihat bahwa:

- V = {1, 2, 3, 4, 5, 6}
- E = {(1,3),(1,6),(2,5),(3,4),(3,6)}.

Tiap simpul merupakan anggota dari himpunan V, dan tiap sisi merupakan anggota dari himpunan E, perhatikan bahwa terdapat simpul yang bukan merupakan ujung dari sisi manapun, simpul yang demikian disebut "terisolasi" [3].

2.2 Representasi Graf Dalam Sistim Komputer

Terdapat berbagai macam cara untuk menyimpan graf dalam sistim komputer. Data struktur yang digunakan bergantung pada struktur graf dan algoritma yang digunakan untuk memanipulasi graf tersebut. Secara teoritis, dapat terlihat dengan jelas perbedaan antara representasi graf dengan struktur data list dan matriks, namun dalam aplikasi konkret, struktur data terbaik adalah gabungan dari keduanya. Struktur data list sering digunakan pada *sparse* graf (graf dengan jumlah simpul yang jarang) karena struktur data ini membutuhkan memori dalam jumlah yang lebih sedikit. Struktur data matriks di sisi lain menjajikan akses yang lebih cepat dalam aplikasi tertentu namun memakan memori yang jauh lebih besar apabila grafnya sangat besar [4].

Terdapat dua buah cara yang umum diketahui dalam merepresentasikan list, yaitu:

- *Incidence list*: Sisi-sisi direpresentasikan dengan senarai yang mengandung pasangan (terurut apabila grafnya berarah) simpul (yang dihubungkan oleh sisi yang bersangkutan), dan bobot dari sisi jika sisinya memiliki bobot.
- Senarai ketetanggaan: Hampir sama dengan *incidence list*, tiap simpul memiliki list dengan simpul tetangganya. Hal ini menimbulkan redudansi pada graf tak berarah, sebagai contoh apabila simpul A dan B bertetangga, maka list ketetanggaan dari A akan mengandung B, demikian pula sebaiknya, list ketetanggaan dari B akan mengandung A. Struktur data ini cenderung lebih cepat daripada *incidence list*, namun dibayar dengan ukuran memori yang lebih besar.

Berbeda dengan list, varian struktur data graf dengan matriks lebih banyak, yaitu:

- *Incidence* matriks: Graf direpresentasikan dengan matriks sisi dengan simpul, dimana [sisi, simpul] mengandung informasi sisi tersebut (kasus termudah: 1 – terhubung, 0 – tak terhubung).
- Matriks ketetanggaan: Struktur data ini merupakan struktur data yang cukup banyak digunakan dalam merepresentasikan sebuah graf, apabila terdapat sebuah sisi yang menghubungkan simpul x dan y, maka $M_{i,j}$ berisi 1, selain itu 0, representasi demikian memberikan kemudahan untuk mencari subgraf dan membalik (*reverse*) graf jika dibutuhkan.
- Matriks Laplacian: Didefinisikan sebagai matriks derajat - matriks ketetanggaan, terkandung didalamnya informasi ketetanggaan dan informasi derajat sebuah simpul.
- Matriks jarak: Berupa matriks n x n, dimana $M_{i,j}$ berisi informasi jarak terdekat antara i dan j, jika tidak terdapat jalur yang menghubungkan i dan j maka $M_{i,j} = \sim$.

3. STRUKTUR DATA

Pada tiap algoritma, terdapat kebutuhan untuk menyimpan data. Mulai dari menyimpan sebuah nilai pada sebuah variable, sampai pada struktur data yang lebih kompleks. Dalam lomba pemrograman, terdapat berbagai macam aspek yang harus diteliti dari struktur data ketika akan memilih struktur data yang akan digunakan dalam merepresentasikan data dari sebuah permasalahan [5].

Dalam istilah ilmu komputer, sebuah struktur data adalah cara penyimpanan, pengorganisasian dan pengaturan data di dalam media penyimpanan komputer sehingga data tersebut dapat digunakan secara mangkus [6].

Dalam tehnik pemrograman, struktur data berarti tata letak data yang berisi kolom-kolom data, baik itu kolom yang tampak oleh pengguna (user) ataupun kolom yang hanya digunakan untuk keperluan pemrograman yang tidak tampak oleh pengguna. Setiap baris dari kumpulan kolom-kolom tersebut dinamakan catatan (record). Lebar kolom untuk data dapat berubah dan bervariasi. Ada kolom yang lebarnya berubah secara dinamis sesuai masukan dari pengguna, dan juga ada kolom yang lebarnya tetap. Dengan sifatnya ini, sebuah struktur

data dapat diterapkan untuk pengolahan database (misalnya untuk keperluan data keuangan) atau untuk pengolahan kata (word processor) yang kolomnya berubah secara dinamis. Contoh struktur data dapat dilihat pada file-file spreadsheet, database, pengolahan kata, gambar yang dikompres, juga pemampatan (kompres) file dengan teknik tertentu yang memanfaatkan struktur data [6].
Berbagai macam stuktur data digunakan pada berbagai macam aplikasi, dan beberapa jenis struktur data sangat spesifik untuk masalah tertentu. Sebagai contoh struktur data Pohon-B digunakan secara spesifik pada implementasi database, dengan *routing tables* bergantung pada jaringan mesin untuk dapat berfungsi.

Dalam mendesain bermacam macam program, pemilihan struktur data yang tepat merupakan pertimbangan desain utama, pengalaman para ahli membangun sistem dalam skala besar membuktikan bahwa tingkat kesulitan dalam inplementasi, kualitas, dan performa dari sebuah sistem sangat bergantung pada struktur data yang digunakan. Setelah struktur data yang tepat dipilih, algoritma yang digunakan sering kali sudah dapat dilihat dengan jelas [7].

Karena alasan di atas, maka banyak metoda desain dan bahasa pemrograman lebih menekankan organisasi pada struktur data, bukan algoritmanya. Sebagian besar bahasa pemrograman memberikan fitur *module system*, yang memungkinkan struktur data dapat digunakan pada aplikasi yang berbeda dengan menyembunyikan detail implementasinya dibalik layar. Pemrograman berorientasi objek seperti C++ dan Java menggunakan *classes* untuk menangani masalah ini [7].

Beberapa contoh bentuk struktur data yang umum dan berkaitan dengan graf antara lain [8]:

- Senarai ketetangggaan
- Struktur data himpunan saling lepas
- Graf-*stack* terstruktur
- *Scene Graph*
- Struktur data pohon
  - Pohon M-Way
  - Pohon B
  - Pohon Biner
  - Trie
  - Heap
  - *Parse tree*
  - *Space partitioning*

4. HIMPUNAN

4.1 Pengertian Himpunan

Dalam matematika, himpunan dapat didefinisikan sebagai kumpulan elemen-elemen yang unik, meskipun diperoleh dari sebuah definisi yang sangat singkat, himpunan merupakan salah satu konsep yang sangat penting dan fundamental dalam matematika modern, dan studi tentang struktur himpunan (*set theory*), merupakan studi yang berkembang dengan sangat pesat akhir-akhir ini [9].

Studi tentang himpunan dimulai sejak akhir abad ke 19, yang sekarang menjadi bagian dari pendidikan matematika, dan diajarkan sebagai pendidikan elementer. Teori himpunan dapat dipandang sebagai fondasi dari hampir seluruh ilmu matematika dibangun dan sumber dari hampir seluruh persamaan matematika.

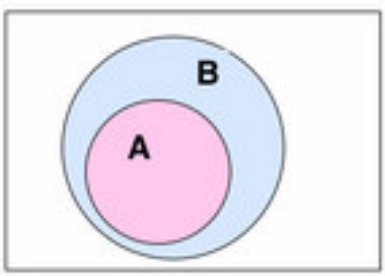
Menurut kesepakatan, sebuah himpunan dapat dibentuk dengan menuliskan isi dari elemen-elemen himpunan tersebut dan ditulis di dalam kurung kurawal, contoh: A = {3, 1, 2}, B = {merah, putih, biru}, C = {1, 2, 3, …, 100}, D = {n^2 + 1 : n bilangan bulat; dan $0 \leq n \leq 100$}.

Keanggotaan dari himpunan juga merupakan pertanyaan mendasar yang berkaitan dengan himpunan. Apakah sebuah elemen merupakan atau bukan merupakan anggota dari sebuah himpunan tertentu dapat disimbolkan dengan $\in$ dan $\notin$.

Jumlah anggota dari sebuah himpunan (kardinal) yang dijelaskan di atas selalu terbatas, misalnya, himpunan A terdiri dari 4 anggota, himpunan B terdiri dari 10 anggota. Himpunan juga dapat terdiri dari 0 anggota, yang sering disebut sebagai himpunan kosong, dan dinotasikan dengan symbol $\varnothing$, sebagai contoh, A = $\varnothing$. Namun, anggota dari sebuah himpunan juga dapat tidak terbatas, misalnya himpunan dari bilangan bulat.

4.2 Sub Himpunan

Apabila tiap anggota himpunan A juga merupakan anggota himpunan B, maka A dikatakan sub himpunan dari himpunan B, dituliskan sebagai A $\subseteq$ B atau B $\supseteq$ A. Himpunan kosong merupakan sub himpunan dari semua himpunan dan semua himpunan merupakan sub himpunan dari dirinya sendiri ($\varnothing \subseteq$ A dan A $\subseteq$ A).



Gambar 2. A merupakan sub himpunan dari B

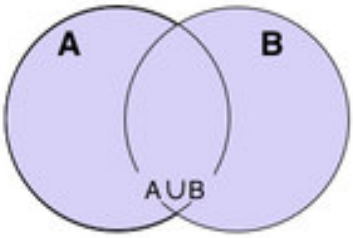
4.3 Penjumlahan, Pengurangan, dan Komplemen Himpunan

Terdapat berbagai macam cara untuk membentuk himpunan baru dari himpunan yang sudah ada. Dua buah himpunan dapat "dijumlahkan" (union), yang dinotasikan dengan A ∪ B. Hasil dari penjumlahan ini berupa himpunan dari semua elemen yang merupakan anggota A atau B. Contoh:

- {1, 2} ∪ {merah, putih} = {1, 2, merah, putih}
- {1, 2, hijau} ∪ {merah, putih, hijau} = {1, 2, merah, putih, hijau}
- {1, 2} ∪ {1, 2} = {1, 2}

Sifat-sifat dasar dari penjumlahan himpunan:

- A ∪ B = B ∪ A
- A merupakan sub himpunan dari A ∪ B
- A ∪ A = A
- A ∪ ∅ = A



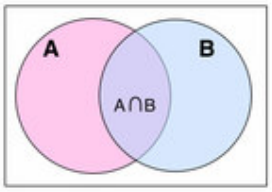
Gambar 3. Penjumlahan himpunan A dan B

Himpunan baru dapat juga dibentuk dari elemen-elemen yang dimiliki oleh dua buah himpunan secara bersama-sama (*intersection*). Perpotongan dari A dan B, dinotasikan dengan A ∩ B, merupakan himpunan dari semua elemen yang merupakan anggota dari A sekaligus anggota dari B. Jika A ∩ B = ∅, maka A dan B dikatakan saling lepas. Contohnya:

- {1, 2} ∩ {merah, putih} = ∅
- {1, 2, hijau} ∩ {merah, putih, hijau} = {hijau}
- {1, 2} ∩ {1, 2} = {1, 2}

Sifat-sifat dasar dari perpotongan himpunan:

- A ∩ B = B ∩ A
- A ∩ B merupakan sub himpunan dari A
- A ∩ A = A
- A ∩ ∅ = ∅



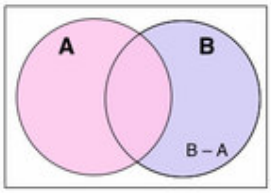
Gambar 4. Perpotongan dari A dan B

Dua buah set dapat pula "dikurangkan". Komplemen relative A dalam B, dinotasikan dengan B – A, adalah himpunan dari semua elemen yang merupakan anggota B namun bukan merupakan anggota dari A. Catatlah bahwa mengurangkan elemen yang tidak terdapat dalam sebuah himpunan adalah valid, contohnya, mengurangkan hijau dari {1, 2, 3}, hasil dari perhitungan tersebut adalah {1, 2, 3}. Pada kasus tertentu, semua himpunan dapat dikatakan sebagai sub himpunan dari himpunan universal U. U – A disebut komplemen absolute dari A, dan dinotasikan dengan A'. Contohnya:

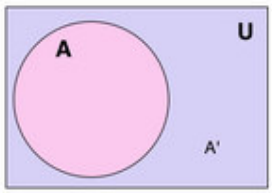
- {1, 2} - {merah, putih} = {1, 2}
- {1, 2, hijau} - {merah, putih, hijau} = {1, 2}
- {1, 2} ∩ {1, 2} = ∅

Sifat-sifat dasar dari komplemen himpunan:

- A ∪ A' = U
- A ∩ A' = ∅
- (A')' = A
- A - A = ∅
- A - B = A ∩ B'



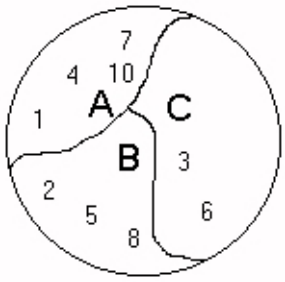
Gambar 5. Himpunan B – A



Gambar 6. Komplemen dari himpunan A

4.4 Himpunan Saling lepas (Disjoint Set)

Dalam matematika, dua himpunan dikatakan saling lepas (*disjoint*) apabila kedua himpunan tersebut tidak memiliki elemen yang sama, himpunan yang saling lepas dinotasikan dengan A // B. Contoh: {1, 2, 3} dan {4, 5, 6} adalah himpunan saling lepas.



Gambar 7. Himpunan A, B, dan C saling lepas

Pada dasarnya, himpunan A dan B saling lepas apabila perpotongan kedua himpunan tersebut

menghasilkan himpunan kosong, yaitu bila A ∩ B = ∅. Definisi ini dapat diperluas untuk semua bentuk himpunan.

5. STRUKTUR DATA GRAF UNTUK HIMPUNAN SALING LEPAS

Struktur data himpunan saling lepas, yang juga dikenal dengan *union-find set*, adalah struktur data yang terus menyimpan partisi dari himpunan elemen selama selang waktu tertentu ketika pemrosesan masalah dilakukan. Terdapat 3 operasi utama yaitu [10]:

- *MakeSet*: Membuat sebuah partisi baru yang berisi sebuah elemen.
- *Find*: Mencari tahu pada partisi manakah sebuah elemen berada.
- *Merge/Union*: Menggabungkan dua buah partisi menjadi sebuah partisi.

Terdapat beberapa solusi untuk masalah ini, menggunakan struktur data senarai berantai, atau struktur data pohon himpunan saling lepas. Sampai saat ini, solusi yang paling mangkus dan banyak digunakan adalah struktur data pohon himpunan saling lepas dengan *path compression* dan *union rank heuristic*, dengan performa O(α(*n*)), untuk α(*n*) merupakan fungsi yang bertumbuh sangat lambat dan pada aplikasinya hampir tidak pernah melebihi 5 [10].

5.1 Senarai Berantai Himpunan Saling Lepas

Mungkin cara termudah dalam membentuk struktur data himpunan saling lepas adalah dengan membuat senarai berantai untuk tiap himpunan [1].

Algoritma *MakeSet* sudah cukup jelas, membuat senarai dengan satu elemen. *Union* hanya menggabungkan dua buah senarai. Sayangnya, dengan implementasi demikian, *Find* memiliki kompleksitas O(*n*) (linear).

Kita dapat mencoba mengurangi waktu eksekusi tersebut dengan menggabungkan senarai yang lebih pendek pada senarai yang lebih panjang, yang biasa disebut *weighted union heuristic*. Metoda ini juga perlu menyimpan panjang tiap senarai ketika sebuah operasi dilakukan untuk menjaga kemangkusan algoritmanya. Menggunakan metoda ini, sederetan *m* buah operasi *MakeSet, Union,* dan *Find* dengan *n* buah elemen membutuhkan O(*m* + *n*log *n*) waktu. Untuk memperoleh hasil yang lebih baik, kita harus memulai lagi dengan struktur data yang berbeda [1].

5.2 Pohon Himpunan Saling Lepas

Pada struktur data ini, tiap himpunan direpresentasikan dengan data struktur pohon dimana tiap simpulnya memiliki referensi ke simpul orang tuanya (*parent node*). Pohon himpunan saling lepas ini semula ditemukan oleh Bernard A. Galler dan Michael J. Fisher pada tahun 1964, meskipun analisanya memakan waktu sampai bertahun-tahun kemudian [10]

Untuk menggambarkan struktur data ini dengan lebih jelas, akan dibahas penjelasannya dalam bahasa *C*.

Di bawah ini adalah representasi dari simpulnya dalam bahasa *C*, simpul tidak memiliki *pointer* ke simpul anaknya namun memiliki *pointer* yang menunjuk ke orang tuanya.

Algoritma 1. Simpul struktur data pohon himpunan saling lepas

```
typedef struct
forest_node_simple_t {
    void* value;
    struct forest_node_simple_t*
parent;
} forest_node_simple;
```

Ketika menggunakan struktur data ini dalam aplikasi sebenarnya, terdapat pilihan cara lain yaitu memasukan properti dari simpul pohon langsung pada struktur yang merepresentasikan tiap elemen (biasa disebut dengan *internal storage*) [10].

Operasi find cukup singkat, diberikan simpul yang terdapat pada sebuah pohon, maka akar pohon dapat dicari dengan mengikuti *pointer* orang tuanya, akar dapat diasumsikan sebagai simpul yang merepresentasikan seluruh partisi tersebut. Dalam bahasa C, algoritma find dapat dituliskan sebagai berikut:

Algoritma 2. Algoritma untuk mencari informasi pada himpunan mana sebuah elemen berada

```
forest_node_simple*
FindSimple(forest_node_simple* node) {
    while (node->parent != NULL) {
        node = node->parent;
    }
    return node;
}
```

Menggabungkan dua buah pohon menjadi sebuah pohon juga merupakan hal yang mudah, kita hanya perlu mencari akar dari salah satu pohon dan menggabungkan akar tersebut pada akar pohon berikutnya.

Algoritma 3. Algoritma menggabungkan dua buah himpunan menjadi sebuah himpunan

```
/* Given the root elements of two
trees, merge the trees into one tree
*/

void UnionSimple
(forest_node_simple* node1,
forest_node_simple* node2) {
    node2->parent = node1;
    /* or node1->parent = node2; */
}
```

Pada akhirnya, *MakeSet* mengalokasikan tiap simpul baru tanpa orang tua dan memasukkan nilai yang diberikan pada simpul tersebut. Perlu diperhatikan bahwa *stdlib.h* harus di*include* untuk menggunakan fungsi *malloc()*.

Algoritma 4. Algoritma membentuk himpunan baru berisi hanya sebuah elemen

```
forest_node_simple*

MakeSetSimple(void* value) {
    forest_node_simple* node =
malloc(sizeof(forest_node_simple));
    node->value = value;
    node->parent = NULL;
    return node;
}
```

Di bawah ini merupakan contoh penggunaan dari struktur data himpunan saling lepas seperti yang telah dijelaskan di atas.

Algoritma 5. Contoh penggunaan (mesin) dari struktur data himpunan saling lepas

```
#include <assert.h>
forest node simple
simple MakeSet operation
simple find operation
simple union operation
int main() {
        int i1=1, i2=2, i3=3;
        forest_node_simple* s1 = MakeSetSimple(&i1);
        forest_node_simple* s2 = MakeSetSimple(&i2);
        forest_node_simple* s3 = MakeSetSimple(&i3);
        assert(FindSimple(s1) == s1);
        UnionSimple(s1, s2);
        assert(FindSimple(s1) == FindSimple(s2));
        assert(FindSimple(s1) != FindSimple(s3));
        UnionSimple(s2, s3);
        assert(FindSimple(s1) == FindSimple(s2) &&
               FindSimple(s1) == FindSimple(s3));
        return 0;
}
```

Algoritma 6. Simpul pohon himpunan saling lepas yang disempurnakan

```
typedef struct forest_node_t {
    void* value;
    struct forest_node_t* parent;
    int rank;
} forest_node
```

Algoritma 7. Isi dari prosedur *MakeSet* yang telah disempurnakan

```
<<declarations>>=
forest_node* MakeSet(void* value);

<<MakeSet operation>>=
forest_node* MakeSet(void* value) {
    forest_node* node =
malloc(sizeof(forest_node));
    node->value = value;
    node->parent = NULL;
    node->rank = 0;
    return node;
}
```

5.3 Meningkatkan Efisiensifitas Struktur Data Pohon Himpunan Saling Lepas

Meskipun sederhana, algoritma yang dijelaskan sejauh ini memiliki masalah sama seperti masalah yang dimiliki pohon pada umumnya, setelah dilakukan banyak operasi penggabungan, ketinggian pohon menjadi bertambah besar. Karena operasi *find* memakan waktu sebanding dengan ketinggian pohon, maka hal ini dapat mengarah pada kompleksitas linier. Pada penmasangan akar sebuah pohon pada akar pohon yang lain, idealnya kita memasangkan pohon dengan ketinggian lebih rendah pada pohon yang memiliki ketinggian lebih tinggi. Sayangnya, ketinggian merupakan hal yang membutuhkan banyak kalkulasi dalam perhitungannya, terlebih tanpa pointer anak. Oleh karena itu, kita akan menggunakan perhitungan kasar melalui tinggi kira-kira yang disebut dengan *rank*, yang disimpan pada akar pohon (**Algoritma 6**).

Rank dapat kita definisikan sebagai berikut [10]:

- Prosedur *MakeSet* selalu menghasilkan pohon dengan *rank* 0 (**Algoritma 7**).
- Jika *rank*(s) ≠ *rank*(t), maka *rank*(*Union*(s, t)) merupakan *rank* terbesar antara *rank*(s) dan *rank*(t). Dalam kasus ini, kita memasangkan akar pohon

dengan *rank* yang lebih kecil pada akar pohon dengan *rank* yang lebih besar.

- Jika *rank*(s) = *rank*(t), maka *rank*(*Union*(s, t)) = *rank*(s) + 1 = *rank*(t) + 1 (**Algoritma 8**).

Tehnik seperti yang dijelaskan di atas biasa dikenal dengan nama *union find heuristic,* menjamin ketinggian logaritmik, yang tentu lebih mangkus. Namun kita bahkan dapat lebih mengefiseienkannya dengan tehnik yang dikenal dengan nama *path compression*. Idenya adalah semua simpul yang dilewati ketika men-*traverse* dari sebuah simpul sampai ke akar berada pada pohon yang sama, oleh karena itu, simpul-simpul tersebut akan lebih baik apabila langsung menunjuk (*pointing*) pada akar yang terdapat pada pohon yang sama. Kita membuat kalang kedua dan memperbaharui *pointer* orang tua mereka, mempercepat pencarian berikutnya yang berkaitan dengan simpul yang bersangkutan dengan dramatis (**Algoritma 9**) [10].

Algoritma 8. *rank*(*Union*(s, t)) = *rank*(s) + 1 = *rank*(t) + 1

```
<<declarations>>=
void Union(forest_node* node1, forest_node* node2);
<<union operation>>=
void Union(forest_node* node1, forest_node* node2) {
    if (node1->rank > node2->rank) {
        node2->parent = node1;
    } else if (node2->rank > node1->rank) {
        node1->parent = node2;
    } else { /* they are equal */
        node2->parent = node1;
        node1->rank++;
    }
}
```

Algoritma 9. Pointer tiap simpul diperbaharui ketika melakukan *traversal*

```
<<declarations>>=
forest_node* Find(forest_node* node);
<<find operation>>=
forest_node* Find(forest_node* node) {
    /* Find the root */
    forest_node* root = node;
    while (root->parent != NULL) {
        root = root->parent;
    }
    /* Update the parent pointers */
    forest_node* temp;
    while (node->parent != NULL) {
        temp = node->parent;
        node->parent = temp;
        node = temp;
    }
    return root;
}
```

Algoritma 10. Struktur data himpunan saling lepas secara keseluruhan

```
<<union_find.h>>=
#ifndef _UNION_FIND_H_
#define _UNION_FIND_H_

forest node
declarations

#endif /* #ifndef _UNION_FIND_H_ */

<<union_find.c>>=
header files
#include "union_find.h"

MakeSet operation
union operation
find operation
```

Masih terdapat beragam algoritma lain yang tidak memerlukan kalang kedua, namun algoritma itu jauh lebih rumit dan akan melenceng terlalu jauh dari bahasan makalah ini.

Secara keseluruhan, struktur data di atas dapat digambarkan seperti pada **Algoritma 10** di atas.

Di bawah ini diberikan contoh penggunaan dari struktur data pohon himpunan saling lepas yang telah disempurnakan, kurang lebih tidak jauh berbeda dengan struktur data pohon himpunan saling lepas yang belum disempurnakan:

Algoritma 11. Struktur data himpunan saling lepas secara keseluruhan

```
<<union_find_example.c>>=
#include <assert.h>
#include "union_find.h"

int main() {
    int i1=1, i2=2, i3=3;
    forest_node *s1=MakeSet(&i1),
                *s2=MakeSet(&i2),
                *s3=MakeSet(&i3);
    assert(Find(s1) == s1);
    Union(s1, s2);
    assert(Find(s1) == Find(s2));
    assert(Find(s1) != Find(s3));
    Union(s2, s3);
    assert(Find(s1) == Find(s2) &&
           Find(s1) == Find(s3));
    return 0;
}
```

6. APLIKASI STRUKTUR DATA HIMPUNAN SALING LEPAS

6.1 Mengoptimalkan Algoritma *Kruskal* dalam mencari *Minimum Spanning Tree*

Analisis membuktikan bahwa algoritma Kruskal untuk *Minimum Spanning Tree* memiliki kompleksitas $O(m \log m)$ dengan m adalah jumlah sisi-sisinya. Algoritma ini membutuhkan cara yang lebih cepat untuk mengetes apakah sisi yang menghubungkan dua buah simpul berada pada komponen terkoneksi yang sama.

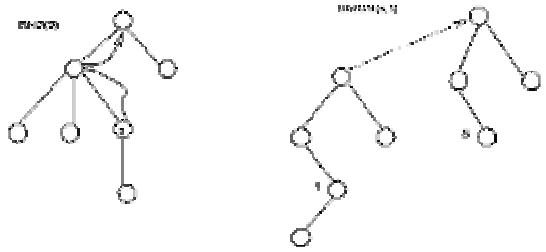
Untuk mencari tahu jawabannya, kita membutuhkan struktur data untuk memanipulasi himpunan-himpunan yang dapat mengetes apakah dua buah elemen berada pada himpunan yang sama dan menggabungkan dua buah himpunan tersebut. Masalah ini dapat diimplementasikan dengan operasi *union* dan *find.*

Algoritma 11. Berikut di bawah ini adalah contoh pseudocode dari implementasi *union find* pada *kruskal*

```
Is s_i ≡ s_j
        t = Find (s_i)
        u = Find (s_j)
        Return (Is t=u?)

Make s_i ≡ s_j
        t = d(s_i)
        u = d(s_j)
        Union (t, u)
```

Implementasi sederhana adalah dengan merepresentasikan tiap himpunan dengan pohon, dengan *pointer* dari simpul ke orang tuanya. Tiap elemen terkandung dalam simpul, dan nama dari himpunan adalah elemen dari akar.



Gambar 8. *Find* dan *Union*

6.2 Menelusuri Komponen yang Saling Terkoneksi pada Graf Tak Berarah

Asumsikan kita memiliki graf yang tak berarah dan ingin secara efisien menjawab pertanyaan yang berkaitan dengan koneksi antar komponen dari graf tersebut seperti [1]:

- Apakah dua buah simpul dalam graf yang sama saling terkoneksi?
- Tampilkan semua simpul dalam graf sesuai dengan komponennya (keterhubungannya)
- Berapa banyak komponen yang saling terkoneksi?

Apabila grafnya statis (tidak berubah), maka kita dapat dengan singkat menggunakan *breadth-first search* untuk mengasosiasikan sebuah komponen dengan tiap simpul. Bagaimanapun, jika kita ingin menelusuri komponen-komponen dalam graf selama menambah simpul dan sisi ke dalamnya, struktur data himpunan saling lepas akan sangat efisien [1].

Asumsikan graf pada mulanya kosong. Setiap kali kita menambahkan simpul, kita menggunakan *MakeSet* untuk membuat himpunan yang mengandung hanya simpul tersebut. Setiap kali kita menambahkan sebuah sisi, kita menggunakan *Union* untuk menggabungkan himpunan dari dua simpul yang terhubung pada sisi tersebut. Sekarang, tiap himpunan akan mengandung simpul dari komponen yang saling terkoneksi, dan kita dapat menggunakan *Find* untuk menentukan pada komponen terkoneksi manakah sebuah simpul berada, atau apakah dua buah simpul terdapat pada komponen yang sama [1].

Tehnik ini digunakan oleh *Boost Graph Library*, untuk mengimplementasikan fungsi *Incremental Connected Components*.

Perlu diperhatikan bahwa metoda ini tidak memperbolehkan penghapusan sisi-sisi, meskipun tanpa *path compression* atau *rank heuristic*, hal ini tidaklah mudah, namun, saat ini metoda yang lebih rumit telah didesain untuk dapat menangani masalah ini [1].

6.3 Hubungan Ekivalensi dan Kelas Ekivalensi

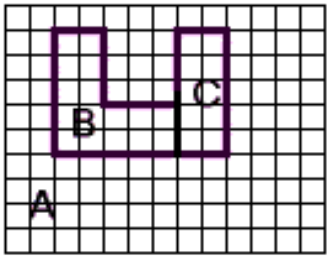
Hubungan relasi pada himpunan A disebut relasi ekivalensi apabila refleksif, simetris, dan transitif. Operasi kongruen modulo n merupakan relasi ekivalensi. Pernyataan a ~ b mengindikasikan bahwa elemen a dan b berada pada himpunan yang sama. Hal ini mengikuti aturan bahwa untuk membentuk sebuah kelas ekivalensi dari sekelompok relasi ekivalen kita harus melakukan langkah-langkah seperti di bawah ini:

$\forall$ *x, MakeSet*(x)
$\forall$ relasi ekivalen (x, y)
    if *FindSet*(x) $\neq$ *FindSet*(y)
        Union(x, y)

Contoh yang sangat bagus untuk aplikasi ini adalah relasi ekivalen kongruen modulo 5. Dalam masalah ini, elemen {0, 5, 10, ...} berada pada kelas

ekivalensi yang sama. Representasi umum dari kelas adalah 0. Elemen 7 tidak berada pada himpunan yang memiliki representasi 0.

Gambar di bawah adalah contoh lain dari relasi ekivalen yang mengasumsikan *pixel* sebagai elemen.

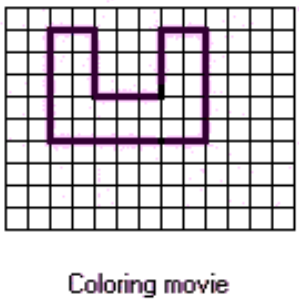


Gambar 9. Relasi ekivalensi

Pada gambar di atas, A, B, dan C, adalah tiga kelas ekivalensi. Sebuah *pixel* adalah elemen kelas apabila *pixel* tersebut berada pada wilayah kelas tersebut. Selanjutnya, dua buah *pixel* adalah ekivalen apabila mereka berdua berada pada himpunan yang sama.

6.4 Pewarnaan Film Tua

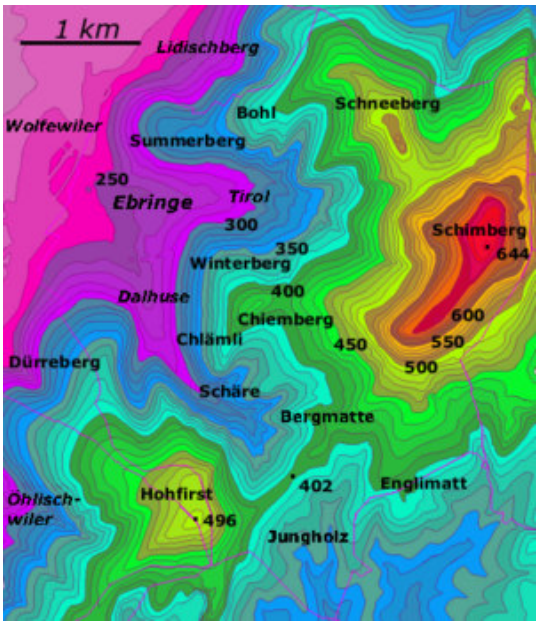
Setiap frame pada film hitam putih yang tua terdiri atas *pixel-pixel*, untuk mewarnai film ini, kita harus mewarnai tiap *pixel* nya dengan aturan tertentu sehingga tiap *pixel* yang ekivalen memiliki warna yang sama. *Pixel* yang ekivalen adalah *pixel* yang secara logika berada pada daerah yang sama, sebagai contoh ketika seorang aktor mengenakan jaket. Pada frame, *pixel* yang bertetangga dengan *gray levels* yang sama atau sedikit berbeda dapat diasumsikan ekivalen. Oleh karena itu, sebuah *gray level* frame *pixel* mendefinisikan sekitar ribuan atau jutaan hubungan ekivalensi antara *pixel* tetangganya. Untuk memisahkan kelas ekivalensi dari hubungan ini, metoda *scan* 4-*pixel square* digunakan untuk menganalisis tetangga dari sebuah *pixel x*. Proses *scan* ini akan memeriksa hubungan ekivalensi antar *pixel* dari kiri atas sampai ke kanan bawah, baris demi baris.



Gambar 10. Pewarnaan film tua

6.5 Menghitung Garis Pantai Sebuah Daratan

Ketika menghitung kontur permukaan 3-dimensi, salah satu langkah pertama yang harus dilakukan adalah menghitung "garis pantai". Bayangkan kita sedang men-*sweeping* sebuah model daratan, menggunakan bidang yang sejajar dengan "permukaan air laut", mulai dari dasar daratan hingga ke puncaknya. Kita akan mendapatkan sekumpulan garis kontur seiring bergerak ke puncak daratan tersebut, dilabelkan dengan solusi lokal yang dimiliki tiap kontur (ketinggian). Pada akhirnya, kita akan mendapatkan sebuah kontur yang mengandung seluruh solusi lokal.



Gambar 11. Contoh kontur ketinggian

Ketika permukaan air laut meningkat sampai di atas minimum lokal daratan, akan terbentuk sebuah "danau", yaitu garis kontur yang mengelilingi minimum lokal. Masalah ini dapat diselesaikan dengan operasi *MakeSet*.

Seiring dengan meningkatnya air laut, permukaannya akan menyentuh sela gunung (*saddle point*). Ketika itu, kita mengikuti tanjakan bukit dari tiap sisinya sampai pada minimum lokal. Kita menggunakan *Find* untuk menentukan kontur mana yang mengelilingi dua buah solusi lokal ini, lalu menggunakan *Union* untuk menggabungkan keduanya. Pada akhirnya, seluruh kontur akan tergabung menjadi satu, dan masalah ini selesai.

6.6 Mengklasifikasikan himpunan atom-atom pada molekul atau fragmen

Dalam perhitungan kimia, tabrakan yang menyangkut pecahan dari molekul berukuran besar

dapat disimulasikan dengan *molecular dynamics*. Hasil dari tabrakan ini berupa data atom-atom beserta posisinya. Pada analisisnya, algoritma *union-find* dapat digunakan untuk mengklasifikasikan atom-atom ini dalam fragmen-fragmen. Tiap atom diinisialisasi sebagai bagian dari fragmen dirinya sendiri. Pada algoritma *Find* umumnya akan mengandung pengecekan jarak antara sepasang atom, meskipun kriteria lain seperti *electronic charge* yang terjadi antara atom-atom dapat digunakan. *Union* menggabungkan dua buah fragmen bersama-sama. Pada akhirnya, ukuran dan karakteristik tiap fragmen dapat dianalisis.

7. KESIMPULAN

Pada dasarnya, terdapat beragam sebuah himpunan direpresentasikan dalam sebuah struktur data. Sebagai salah satu contoh, larik satu dimensi yang memiliki dua informasi, informasi nilai yang dikandung di dalamnya, dan informasi pada himpunan manakah nilai tersebut berada. Bayangkan untuk mencari pada himpunan apakah sebuah nilai berada membutuhkan O(n), untuk menggabungkan dua buah himpunan juga memerlukan waktu O(n) yaitu dengan menelusuri tiap elemen larik dan mensubstitusi identitas salah satu himpunan pada himpunan yang lain. Sekarang kita lihat pada contoh-contoh aplikasi dari bab 6, menghitung garis pantai dimana kita ingin mendapatkan keakuratan setinggi mungkin tentu saja merupakan hal yang sangat sulit dilakukan dengan kompleksitas linear. Bandingkan dengan algoritma dengan menggunakan list linear, algoritma untuk menggabungkan dua buah himpunan telah jauh diminimalisir dari kompleksitas linear menjadi konstan. Setelah dilakukan penelitian lebih lanjut, terlihat bahwa masih ada cara untuk merepresentasikan himpunan ini dengan algoritma yang jauh lebih efisien, struktur data himpunan saling lepas. Makalah ini menjelaskan bagaimanakah cara berpikir dari struktur data dan algoritma untuk himpunan saling lepas menyelesaikan suatu permasalahan himpunan saling lepas dengan pertumbuhan waktu eksekusinya yang sangat lamban (O($\alpha(n)$), untuk $\alpha(n)$ merupakan fungsi yang bertumbuh sangat lambat).

Kesimpulan yang dapat diambil dari makalah ini adalah bahwa struktur data yang dipilih merupakan desain utama dalam merancang berbagai macam program. Struktur data dipilih karena pertanyaan kunci dari suatu permasalahan memiliki algoritma yang bekerja lebih baik pada struktur data yang satu dibanding struktur data yang lain.

Seiring dengan perkembangan jaman, program-program yang dibuat makin bertambah rumit, seiring dengan hal ini pula, prioritas struktur data makin melewati prioritas algoritma yang digunakan itu sendiri. Banyak para programmer profesional mulai menggunakan ekstensi *library* dari bahasa pemrograman modern seperti *Standard Template Library* milik C++, Java API, dan Microsoft .NET Framework.

Bahan yang paling fundamental dalam pembuatan sebagian besar struktur data adalah larik, *records, discriminated unions,* dan *references*. Sebagai contoh *nullable reference*, merupakan *reference* yang dapat berupa *null*, merupakan kombinasi dari *references*, dan *discriminated unions*, dan senarai berantai yang kita kenal, dibangun dari *records* dan *nullable references*.

Selama ini banyak terdapat debat tentang bagaimana struktur data diabstraksikan, implementasi atau *interface*?. Bagaimana mereka diabstraksikan mungkin bergantung pada sudut pandang kita masing-masing. Sebuah struktur data dapat dilihat sebagai *interface* antara dua buah fungsi atau implementasi dari metoda untuk mengakses penyimpanan data yang diorganisasikan sesuai dengan tipe data yang sesuai.

REFERENSI

[1] Struktur data himpunan saling lepas dari: http://en.wikipedia.org/wiki/Disjoint-set_data_structure (Tanggal akses: 22 Desember 2006 pukul 06.00)

[2] Koneksi Jaringan dari: http://acm.uva.es/p/v7/793.html (Tanggal akses: 22 Desember 2006 pukul 06.00)

[3] Pemrograman Graf dari: http://www.comp.nus.edu.sg/~stevenha/programming/prog_graph1.html (Tanggal akses: 23 Desember 2006 pukul 07.00)

[4] Teori Struktur Data Graf dari: http://en.wikipedia.org/wiki/Graph_theory (Tanggal akses: 23 Desember 2006 pukul 07.00)

[5] Pemrograman Struktur Data dari: http://www.comp.nus.edu.sg/~stevenha/programming/prog_datastructures1.html (Tanggal akses: 22 Desember 2006 pukul 06.00)

[6] Struktur Data dari: http://id.wikipedia.org/wiki/Struktur_data (Tanggal akses: 23 Desember 2006 pukul 07.00)

[7] Struktur Data dari: http://en.wikipedia.org/wiki/Data_structure (Tanggal akses: 23 Desember 2006 pukul 07.00)

[8] Macam Struktur Data dari: http://en.wikipedia.org/wiki/List_of_data_structures (Tanggal akses: 23 Desember 2006 pukul 07.00)

[9] Himpunan dari: http://en.wikipedia.org/wiki/Set (Tanggal akses: 23 Desember 2006 pukul 07.00)

[10] Struktur Data Himpunan Saling Lepas dari: http://en.literateprograms.org/Disjoint_set_data_structure_(C) (Tanggal akses: 23 Desember 2006 pukul 07.00)