

LAPORAN STRUKTUR DATA DAN ALGORITMA
APPLICATION (DYNAMIC PROGRAMMING, MINIMUM SPANING TREE,
DISJOINT SETS)



Anggota Kelompok 5 :

- | | |
|--------------------------|-------------|
| 1. Atikah Putri Utami | (G1A024027) |
| 2. Agief Vemas A | (G1A024037) |
| 3. Nayla Viona Azahra | (G1A024045) |
| 4. Rofid Fadhil | (G1A024063) |
| 5. R. Noor Fikhri | (G1A023075) |
| 6. Okta Kurnia Ariansyah | (G1A023099) |

Dosen Pengampu :

Arie Vatresia, S.T., M.TI, Ph.D.

PROGRAM STUDI INFORMATIKA
FAKULTAS TEKNIK
UNIVERSITAS BENGKULU

2025

DAFTAR ISI

BAB I PENDAHULUAN	2
A. LATAR BELAKANG.....	2
B. TUJUAN	3
BAB II PEMBAHASAN.....	4
A. DYNAMIC PROGRAMMING	4
a. Konsep Dasar Dynamic Programming	4
b. Sifat-Sifat Dynamic Programming	4
c. Teknik yang digunakan dalam Dynamic Programming:	5
B. MINIMUM SPANNING TREE.....	5
a. Apa Itu Spanning Tree?	5
b. Sifat yang Dimiliki Oleh Spanning Tree.....	5
d. Algoritma dari MST (Minimum Spanning Tree)	5
C. DISJOINT SETS	8
a. Konsep Dasar Disjoint Sets.....	8
b. Operasi Utama Disjoint Sets	9
c. Optimisasi Disjoint set (Union Find).....	11
e. Quick Find.....	11
f. Aplikasi Disjoint Sets	13
g. Kompleksitas Waktu	15
BAB III.....	15
SOURCE KODE DAN OUTPUT	16
A. DYNAMIC PRO	16
a. Dynamic Programming Memoization (Top Down)	16
b. Dynamic Programming Tabulation (Bottom-Up)	17
B. MINIMUM SPANNING TREE.....	19
a. Algoritma Prim.....	19
b. Algoritma Krusal.....	23
C. Disjoint Sets	26
a. Quick Find.....	26
BAB IV PENUTUP	30
A. KESIMPULAN	30
B. SARAN.....	30

BAB I

PENDAHULUAN

A. LATAR BELAKANG

Struktur data dan algoritma merupakan salah satu cabang utama dalam ilmu komputer yang berperan penting dalam pengelolaan dan pemrosesan data secara efisien. Dengan menggunakan berbagai struktur data yang tepat, kita dapat mengoptimalkan kinerja program, baik dalam hal waktu maupun ruang memori. Dalam konteks ini, teknik-teknik algoritma yang efisien sangat dibutuhkan untuk menyelesaikan masalah komputasi yang kompleks dengan cara yang lebih efektif dan cepat.

Dynamic Programming (DP) adalah salah satu teknik pemrograman yang digunakan untuk menyelesaikan masalah yang dapat dibagi menjadi submasalah yang lebih kecil dan tumpang tindih. Teknik ini dapat mengurangi kompleksitas waktu dengan cara menyimpan hasil perhitungan submasalah yang sudah dihitung sebelumnya. Aplikasi dari DP sangat luas, termasuk dalam permasalahan optimisasi seperti perencanaan produksi, pencocokan pola, dan lain-lain.

Minimum Spanning Tree (MST) adalah suatu pohon dalam graf yang menghubungkan semua simpul dengan total bobot yang minimal. MST sering digunakan dalam berbagai aplikasi yang melibatkan jaringan, seperti perencanaan rute jaringan komputer, perencanaan jalur distribusi listrik, dan optimisasi sistem komunikasi.

Disjoint Sets atau Union-Find merupakan struktur data yang digunakan untuk menangani kumpulan elemen yang dibagi menjadi beberapa kelompok, dan berfungsi untuk memeriksa apakah dua elemen berada dalam kelompok yang sama atau tidak. Algoritma ini sangat berguna dalam berbagai aplikasi graf, seperti dalam pencarian komponen terhubung pada graf atau dalam algoritma Kruskal untuk mencari Minimum Spanning Tree.

Melalui laporan ini, kami akan membahas penerapan dan implementasi dari teknik-teknik algoritma tersebut dalam menyelesaikan berbagai permasalahan komputasi. Pembahasan akan mencakup penjelasan teori dasar dari masing-masing teknik serta studi kasus penerapannya untuk memberikan pemahaman yang lebih mendalam mengenai relevansi dan kegunaan algoritma ini dalam kehidupan sehari-hari, baik di bidang teknologi maupun industri.

Pentingnya pemahaman dan penerapan struktur data dan algoritma yang tepat juga sejalan dengan kebutuhan dunia teknologi informasi yang terus berkembang pesat, di mana efisiensi pemrosesan data dan waktu komputasi menjadi faktor yang sangat menentukan.

B. TUJUAN

Tujuan dari laporan ini adalah untuk menganalisis dan memahami penerapan teknik-teknik algoritma seperti Dynamic Programming, Minimum Spanning Tree, dan Disjoint Sets dalam menyelesaikan berbagai permasalahan komputasi yang kompleks. Melalui laporan ini, diharapkan pembaca dapat mengidentifikasi kelebihan dan kekurangan dari masing-masing algoritma serta memahami kondisi atau masalah yang paling tepat untuk menggunakan teknik tersebut. Selain itu, laporan ini juga bertujuan untuk memberikan contoh penerapan praktis melalui studi kasus yang relevan, sehingga memperdalam pemahaman tentang aplikasinya dalam kehidupan nyata. Laporan ini juga bertujuan untuk meningkatkan pemahaman mengenai pemilihan struktur data yang tepat dalam implementasi algoritma, guna meningkatkan efisiensi dan kinerja program. Selain itu, laporan ini diharapkan dapat mengembangkan keterampilan pemrograman dan kemampuan problem-solving, dengan memberikan pemahaman yang lebih dalam tentang cara mengimplementasikan solusi berbasis algoritma untuk berbagai masalah. Akhirnya, tujuan laporan ini adalah untuk memberikan wawasan mengenai aplikasi algoritma dalam berbagai bidang teknologi informasi dan industri, serta menunjukkan relevansinya dalam dunia profesional.

BAB II

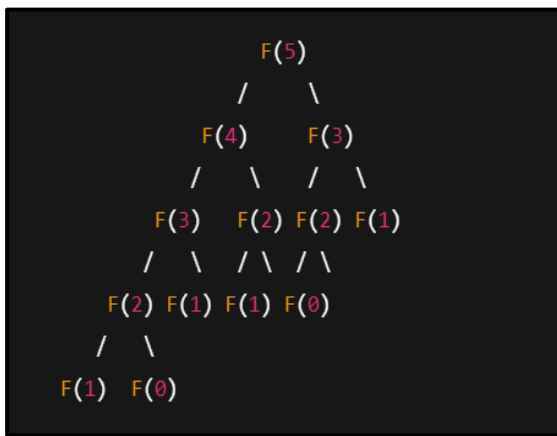
PEMBAHASAN

C. DYNAMIC PROGRAMMING

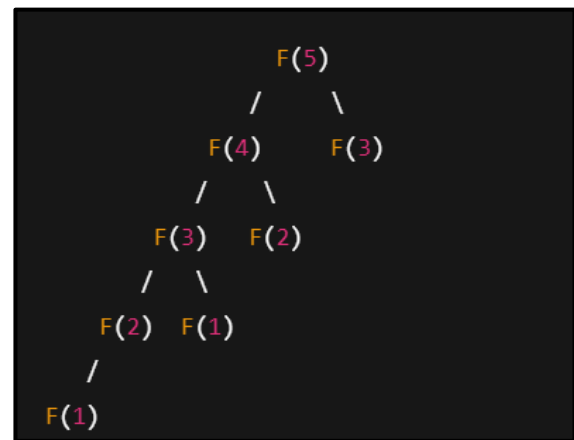
a. Konsep Dasar Dynamic Programming

Dynamic Programming (DP) adalah sebuah teknik dalam ilmu komputer dan matematika untuk menyelesaikan masalah kompleks dengan memecahnya menjadi submasalah yang lebih kecil, lalu menyelesaikan masing-masing submasalah satu kali dan menyimpan hasilnya (biasanya dalam tabel), sehingga jika submasalah yang sama muncul kembali, tidak perlu dihitung ulang.

Contoh:



Tanpa Dynamic Programming



Menggunakan Dynamic Programming

- Tanpa Dynamic Programming: Menghitung hal yang sama berulang-ulang = lambat dan boros waktu.
- Dengan Dynamic Programming: Setiap hasil perhitungan disimpan, tidak perlu menghitung ulang = lebih cepat dan efisien.

Dynamic Programming merupakan metode yang menyimpan hasil perhitungan submasalah sehingga ketika menghadapi permasalahan serupa, kita tidak perlu menghitung ulang, melainkan cukup mengambil hasil yang telah disimpan. Teknik ini sangat efektif untuk menyelesaikan permasalahan kompleks dan berskala besar, selama permasalahan tersebut dapat dipecah menjadi submasalah yang lebih kecil dan memiliki pola yang serupa.

b. Sifat-Sifat Dynamic Programming

1. Overlapping Subproblems :

Masalah besar bisa dipecah menjadi submasalah yang lebih kecil, dan submasalah ini muncul berulang-ulang.

2. Optimal Substructure (Substruktur Optimal)

Solusi dari masalah besar bisa dibangun dari solusi optimal submasalah-submasalahnya.

c. Teknik yang digunakan dalam Dynamic Programming:

1. Memoization (Top-Down):

Teknik untuk menyimpan hasil submasalah yang sudah dihitung, sehingga tidak perlu dihitung ulang. Dengan pendekatan top-down, program memecah masalah besar menjadi submasalah kecil, menyimpan hasilnya, dan memeriksa terlebih dahulu apakah hasil tersebut sudah ada sebelum menghitungnya lagi.

2. Tabulation (Bottom-Up):

Teknik dalam Dynamic Programming yang menyimpan solusi submasalah dalam sebuah tabel (array), mulai dari submasalah yang paling dasar. Pendekatan ini bottom-up, artinya kita menyelesaikan submasalah terkecil terlebih dahulu, lalu membangun solusi dari bawah ke atas. Berbeda dengan memoization, tabulation tidak menggunakan rekursi, melainkan iterasi untuk mengisi tabel dengan solusi yang sudah dihitung.

D. MINIMUM SPANNING TREE

D. Apa Itu Spanning Tree?

Spanning Tree Terdiri dari beberapa kata:

1. Tree: Graf (jaringan) yang terhubung tanpa siklus (cycle).
2. Spanning tree: Sub-graf dari graf terhubung yang mencakup semua simpul (node) tanpa membentuk siklus.
3. Minimal spanning tree (mst): Graf tak-berarah terhubung yang bukan pohon, yang berarti di g terdapat beberapa sirkuit

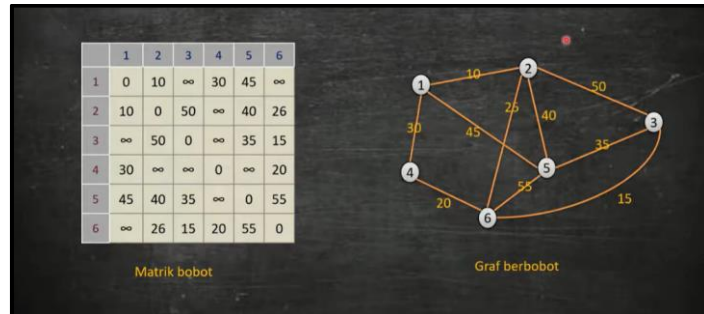
e. Sifat yang Dimiliki Oleh Spanning Tree

1. Jumlah Sisi = Jumlah Simpul – 1
2. Tidak Ada Siklus (Cycle)
3. Bobot Minimum
4. Graf Harus Terhubung
5. MST bisa tidak unik (jika beberapa sisi memiliki bobot yang sama, mungkin ada lebih dari satu mst yang valid)

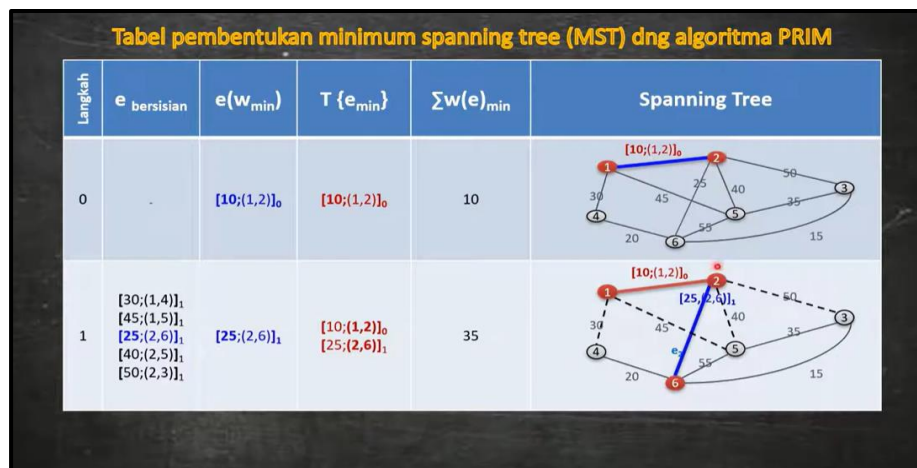
f. Algoritma dari MST (Minimum Spanning Tree)

1. Algoritma Prim

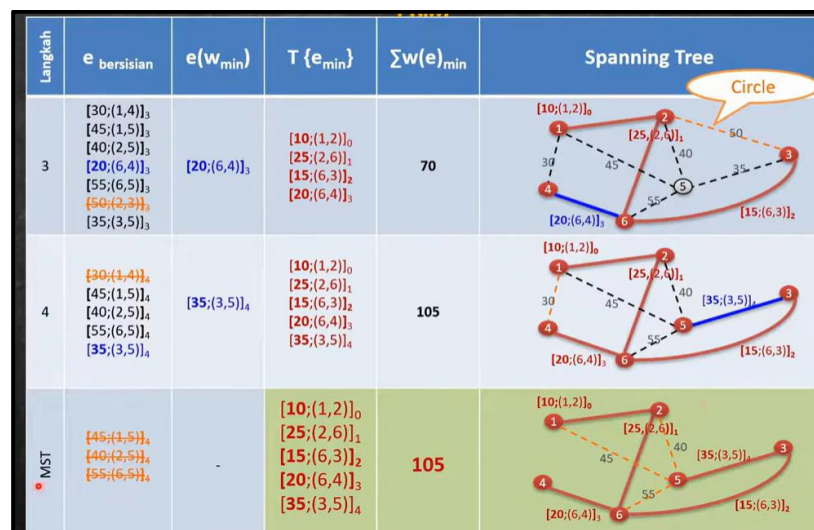
- Memilih sisi yang paling kecil pada graf
- Memilih sisi yang bersisian dengan sisi yang sudah dipilih
- Tidak membentuk cycle (sirkuit)



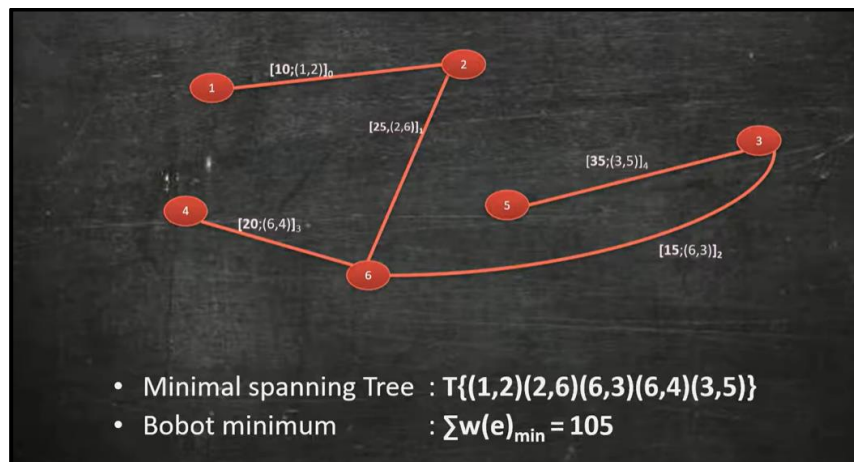
Contoh diatas adalah Graf yang saya ambil dari youtube, Pada graf diatas terdapat matrik bobot, lalu dilanjutkan dengan graf yang sudah terbentuk melalui matrik bobot.



Pada gambar diatas, cara kita untuk memilih sisi yaitu dengan cara memilih sisi yang terkecil terlebih dahulu, lalu dilanjutkan dengan memilih sisi yang bersisian dengan sisi yang sudah kita pilih tadi.



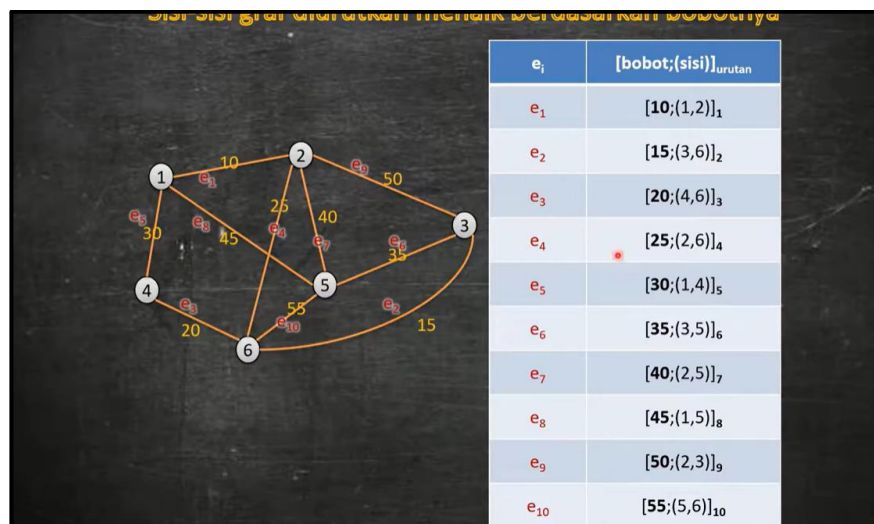
Lalu Pada Gambar diatas kita menemukan sisi yang membentuk cycle, apabila ada sisi yang membentuk cycle kita harus mencoretnya atau tidak menghubungkan sisinya, lalu lanjutkan seperti cara yang tadi.



Lalu Terbentuklah Spanning tree dengan bobot minimum 105, Yang berasal dari sisi $T\{(1,2)(2,6)(6,3)(6,4)(3,5)\}$ jumlah 105 didapatkan dari menjumlahkan setiap bobot pada sisi pada minimal spanning tree.

2. Algoritma Kruskal

- Mengurutkan bobot dari minimum ke terbesar
- Pada awal sisi pada graf sudah harus terbentuk terlebih dahulu
- Sisi yang dipilih tidak harus bersisian dengan sisi yang sudah dipilih sebelumnya
- Tidak boleh membuat cycle(sirkuit)



Graf diatas adalah graf yang dipakai pada algoritma prim, bedanya matrix bobot di buat dengan dari terkecil ke terbesar menurun kebawah.

Langkah	$E(e_i)$	e_i	$T(e_{min})$	$\sum w(e)_{min}$	Forest - Himpunan Spanning Tree
0	$[10;(1,2)]_1$ $[15;(3,6)]_2$ $[20;(4,6)]_3$ $[25;(2,6)]_4$ $[30;(1,4)]_5$ $[35;(3,5)]_6$ $[40;(2,5)]_7$ $[45;(1,5)]_8$ $[50;(2,3)]_9$ $[55;(5,6)]_{10}$			0	
1	$[15;(3,6)]_2$ $[20;(4,6)]_3$ $[25;(2,6)]_4$ $[30;(1,4)]_5$ $[35;(3,5)]_6$ $[40;(2,5)]_7$ $[45;(1,5)]_8$ $[50;(2,3)]_9$ $[55;(5,6)]_{10}$	$[10;(1,2)]_1$	$[10;(1,2)]_1$	10	
2	$[20;(4,6)]_3$ $[25;(2,6)]_4$ $[30;(1,4)]_5$ $[35;(3,5)]_6$ $[40;(2,5)]_7$ $[45;(1,5)]_8$ $[50;(2,3)]_9$ $[55;(5,6)]_{10}$	$[15;(3,6)]_2$	$[10;(1,2)]_1$ $[15;(3,6)]_2$	25	

Pada algoritma kruskal kita menggunakan cara yaitu mngurutkan mulai dari bobot terkecil hingga terbesar, pada algoritma ini kita tidak perlu memilih sisi yang bersisian dengan sisi yang sudah dipilih, jadi kalian bebas selama nilai bobot nya kecil maka itu yang harus duluan, dan tentunya tidak boleh membuat cycle atau sirkuit, apabila terjadi maka di coret.

Langkah	$E(e_i)$	e_i	$T(e_{min})$	$\sum w(e)_{min}$	Forest - Himpunan Spanning Tree
MST	$[40;(2,5)]_7$ $[45;(1,5)]_8$ $[50;(2,3)]_9$ $[30;(1,4)]_5$ $[55;(5,6)]_{10}$		$[10;(1,2)]_1$ $[15;(3,6)]_2$ $[20;(4,6)]_3$ $[25;(2,6)]_4$ $[35;(3,5)]_6$	105	

Minimal spanning Tree :
 $T\{(1,2)(3,6)(4,6)(2,6)(3,5)\}$

Bobot minimum :
 $\sum w(e)_{min} = 105$

Hasilnya Memang sama seperti Algoritma prim tadi, karena ini berasal dari graf yang sama, tetapi sangat mungkin bentuk spanning tree nya berbeda.

E. DISJOINT SETS

g. Konsep Dasar Disjoint Sets

Disjoint Sets atau union find adalah struktur data yang digunakan untuk mengelola kumpulan elemen yang terbagi dalam sejumlah himpunan yang saling tidak beririsan (tidak ada elemen yang berada di lebih dari satu set).

Disjoint Set merupakan struktur data yang melacak sekumpulan elemen yang dipartisi menjadi sejumlah subset disjoint (non-overlapping). Disjoint Set dapat digunakan

untuk menentukan siklus dalam grafik, yang membantu dalam menemukan minimum spanning tree of a graph.

Contoh disjoint sets:

Himpunan A : {27, 37, 45}

Himpunan B: {63, 75, 99}

Jadi, A dan B adalah himpunan terpisah karena tidak ada elemen yang sama antara keduanya. Irisan A dan B adalah himpunan kosong $\{\}$. Maka dari itu disjoint sets digunakan untuk mengelola kumpulan elemen tersebut dan melakukan operasi gabungan (union) dan pencarian (find) dengan efisien.

h. Operasi Utama Disjoint Sets

1. makeSet(x)

- makeSet(x) merupakan operasi untuk membuat sebuah set baru dengan anggota hanya terdiri dari satu elemen.

Contoh: makeSet(Atikah), makeSet(Agief) maka set yang dihasilkan:



Gambar Quick Find

Jadi, bisa dilihat pada gambar, kita menghasilkan 2 set yang berbeda dan masing-masing set hanya memiliki 1 elemen.

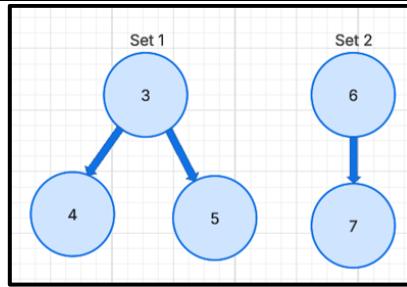
2. findSet(x)

- findSet(x) merupakan operasi untuk mencari di himpunan/set mana suatu elemen berada.
- Setiap set memiliki representatif. Representatif adalah elemen yang mewakili seluruh himpunan tersebut.

Contoh: Kita memiliki 2 set berbeda, yaitu:

Set 1 = {3, 4, 5}

Set 2 = {6, 7}



Pada contoh ini, representatif dari set akan diberi warna merah. Gunakan operasi $\text{findSet}(x)$ untuk mengetahui di himpunan mana elemen 5 berada.

$\text{findSet}(5) = 3$

Jadi, elemen 5 berada di set 1. Pada operasi ditulis $\text{findSet}(5) = 3$ karena representatif set tersebut adalah 3.

3. $\text{union}(x, y)$

- $\text{union}(x, y)$ adalah proses menggabungkan dua himpunan yang terpisah menjadi satu himpunan baru, sekaligus menghilangkan keberadaan himpunan masing-masing elemen sebelumnya.
- Tujuan dari operasi ini adalah menjadikan pohon yang ukurannya lebih kecil sebagai bagian dari pohon yang ukurannya lebih besar.
- Jika implementasi ini tidak digabungkan dengan optimalisasi maka operasi union akan lambat. Optimalisasi bisa dilakukan dengan path compression atau weighted union.

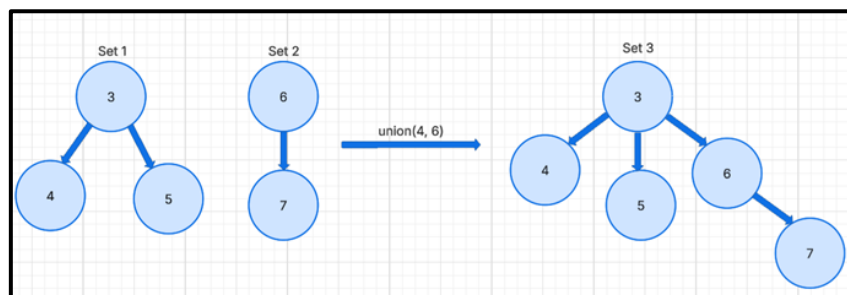
Contoh: Kita memiliki 2 set berbeda, yaitu:

Set 1 = {6, 7}

Set 2 = {3, 4, 5}

Kita gunakan operasi $\text{union}(x, y)$ pada set 1 dan set 2 untuk menggabungkan elemen 4 dan 6.

$\text{union}(4, 6) = \{3, 4, 5, 6, 7\}$



Karena kita menggabungkan elemen 4 & 6 maka set 1 & 2 akan tergabung menjadi 1 set baru, yaitu set 3 dengan elemen 3 sebagai representatifnya.

i. Optimisasi Disjoint set (Union Find)

Terdapat 2 cara agar disjoint set dapat berjalan dengan optimal;

1. Union by rank/size

- Union by rank adalah rank seperti tinggi pohon yang mewakili set yang berbeda. Kita menggunakan array int tambahan yang disebut `rank[]`. Ukuran array ini sama dengan array induk `Parent[]`. Jika `i` adalah representasi dari suatu set, `rank[i]` adalah rank dari elemen `i`. Rank sama dengan tinggi jika kompresi jalur tidak digunakan. Dengan kompresi jalur, rank bisa lebih dari tinggi sebenarnya. Tujuannya untuk menjaga agar pohon tetap seimbang. Setiap node menyimpan rank (perkiraan tinggi). Saat union, sambungkan pohon dengan rank lebih kecil ke yang lebih besar.
- Union by size adalah cara menggabungkan 2 himpunan dengan memilih dua himpunan yang paling kecil disambungkan ke himpunan yang lebih besar.

2. Path Compression adalah teknik yang digunakan untuk meningkatkan efisiensi operasi find (untuk meningkatkan kinerja operasi find, setiap node harus memiliki pointer langsung ke root).

Algoritma bekerja dengan memeriksa apakah node sudah di-root, maka tidak perlu dilakukan. Jika tidak, pointer harus diperbarui untuk menunjuk ke root. Tujuannya untuk mempercepat operasi `findSet`. Saat `findSet(x)` berjalan, ubah parent dari semua node dalam jalur ke root langsung.

j. Quick Find

Quick find adalah algoritma untuk menentukan apakah dua elemen saling terhubung/berada dalam himpunan yang sama. Elemen dikatakan terhubung jika dan hanya jika mereka memiliki parent yang sama.

Contoh: Kita mempunyai 5 elemen: {3, 4, 5, 6, 7} dan masing-masing elemen merupakan `parent[id[i]]` untuk dirinya sendiri.

Index	3	4	5	6	7
Parent	3	4	5	6	7

1. Union. Gabungkan elemen 4 dan 6.

Pertama, cari parent dari masing-masing elemen. Pada tabel diatas kita dapat

mengetahui `parent[4] = 4` dan `parent[6] = 6`. Gabungkan elemen 4 & 6 menjadi 1 set dan

tentukan parent dari set tersebut. Gunakan 4 sebagai parent dari elemen 4 & 6. Maka parent akan berubah seperti tabel dibawah ini.

Index	3	4	5	6	7
Parent	3	4	5	4	7

Selanjutnya, kita akan melakukan operasi union sekali lagi untuk elemen 4 & 3. Kita cari parent kedua elemen, yaitu $\text{parent}[4] = 4$ dan $\text{parent}[3] = 3$. Gabungkan kedua elemen menjadi 1 set, jadi elemen 3, 4, 6 akan menjadi 1 set yang sama. Lalu kita tentukan parent baru dari ketiga elemen tersebut.

Index	3	4	5	6	7
Parent	3	3	5	3	7

Kita gunakan elemen 3 menjadi parent dari set tersebut. Jadi, saat kita melakukan operasi union dengan set yang memiliki lebih dari satu elemen dan parentnya diubah, maka semua parent dari set tersebut juga harus diubah.

2. Find. Cek apakah elemen 3 dan 4 memiliki $\text{parent}/\text{id}[i]$ yang sama.

Dari tabel kita dapat mengetahui $\text{parent}[3] = 3$ dan $\text{parent}[4] = 3$. Jadi, 3 dan 4 terhubung satu sama lain. Selanjutnya, kita cek apakah elemen 6 dan 7 memiliki parent yang sama. Dapat dilihat dari tabel bahwa $\text{parent}[6] = 3$ dan $\text{parent}[7] = 7$. Jadi, elemen 6 & 7 tidak terhubung satu sama lain.

Pada quick find, kompleksitas waktu dalam skenario terbaik adalah $O(1)$, yang berarti operasinya berlangsung dalam waktu konstan. Namun, pada skenario terburuk, kompleksitas waktunya menjadi $O(N)$, menunjukkan bahwa waktu yang dibutuhkan akan sebanding dengan jumlah elemen (N) dalam input.

3. Kelemahan Quick Find

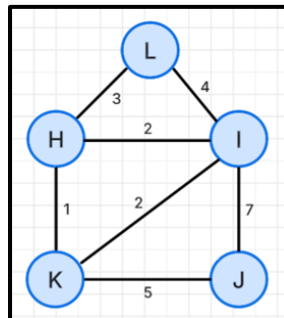
- Operasi union pada quick find terlalu lambat karena setiap kali kita melakukan operasi $\text{union}(p, q)$, kita harus meng-scan seluruh array $\text{id}[i]$. Satu operasi union membutuhkan $O(N)$ akses ke array $\text{id}[i]$. Jadi, jika ada N operasi union yang harus dilakukan, kita membutuhkan total akses array sekitar $N \times N = N^2$.

- Quick find memiliki kompleksitas ruang $O(\log n)$. Jadi, membutuhkan cukup banyak memori dan bisa menjadi masalah jika kita memiliki memori yang terbatas.

k. Aplikasi Disjoint Sets

1. Algoritma Kruskal

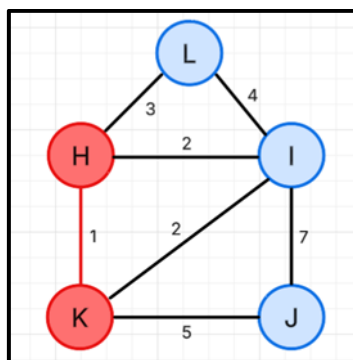
Disjoint sets dapat membantu algoritma Kruskal untuk mengecek apakah penambahan sudut membentuk siklus. Contoh:



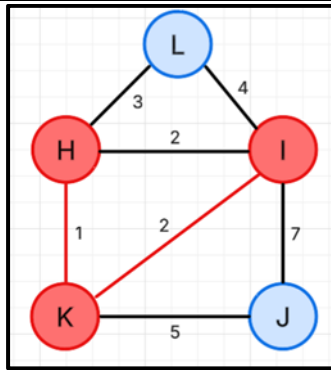
Sudut	(H, K)	(I, K)	(H, I)	(H, L)	(I, L)	(J, K)	(I, J)
Bobot	1	2	2	3	4	5	7

Kita memiliki graf serta tabel untuk mengecek apakah penambahan sudut akan membentuk siklus. Kita akan mulai mengecek sudut satu per satu dimulai dari bobot yang paling rendah.

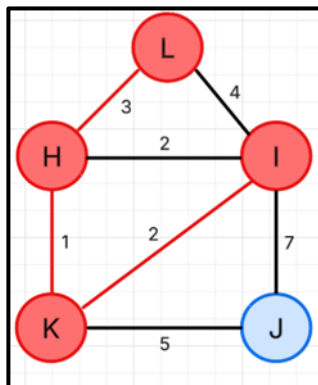
- 1) Setiap node di set sendiri-sendiri, jadi set awal: $\{H\}$, $\{I\}$, $\{J\}$, $\{K\}$, $\{L\}$.
- 2) Mulai proses union pada sudut dimulai dari bobot terkecil. Jadi, kita mulai dari bobot 1, yaitu H dan K. Karena H dan K tidak membentuk siklus maka H dan K menjadi 1 set. Sudut yang tidak membentuk siklus akan ditandai dengan warna merah pada gambar.



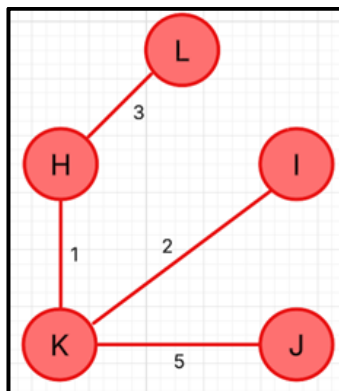
- 3) Union sudut I dan K.



- 4) Union sudut H dan I. Karena union H dan I akan membentuk siklus, maka kita skip proses union pada sudut ini.
- 5) Union sudut H dan L.



- 6) Union sudut I dan L. Karena union I dan L akan membentuk siklus, maka kita skip proses union pada sudut ini.
- 7) Union sudut J dan K.



Karena semua sudut sudah terhubung, maka proses union selesai. MST sudah terbentuk dengan sudut: (H, K), (I, K), (H, L), dan (K, J).

Berikut tabel langkah per langkah dari proses union yang sudah dilakukan:

Sudut diproses	Apakah membentuk siklus?	Tindakan	Disjoint set sekarang
(H, K)	Tidak	Union (H, K)	{H, K}, {I}, {J}, {L}
(I, K)	Tidak	Union (I, K)	{H, I, K}, {J}, {L}
(H, I)	Ya	Lewati	{H, I, K}, {J}, {L}
(H, L)	Tidak	Union (H, L)	{H, I, K, L}, {J}
(I, L)	Ya	Lewati	{H, I, K, L}, {J}
(J, K)	Tidak	Union (J, K)	{H, I, J, K, L}

Aplikasi lain dari disjoint sets:

- 1) Disjoint set dapat digunakan untuk merepresentasikan kumpulan kota di sebuah peta, di mana masing-masing set berisi kota-kota yang saling terhubung. Struktur ini menggambarkan kelompok kota yang memiliki hubungan lewat jaringan jalan raya. Jika dua kota berada dalam set yang sama, berarti terdapat jalur yang menghubungkan keduanya, baik secara langsung maupun melalui kota-kota lain di dalam set tersebut.
- 2) Komponen Listrik pada chip. Di dalam chip elektronik, komponen bisa dihubungkan dengan jalur listrik. Disjoint Set dapat melacak kelompok komponen mana saja yang terhubung langsung.

1. Kompleksitas Waktu

Disjoint set atau union find yang memakai union by rank/size dan path compression memiliki kompleksitas waktu $O(\log n)$ /logaritmik, sedangkan disjoint set tanpa optimisasi memiliki kompleksitas waktu $O(n)$ /linear.

Disjoin set yang memakai union by rank/size memiliki kompleksitas waktu $O(\log n)$ karena metode ini, operasi find dan union akan sangat cepat. Cepat nya bahkan disebut hampir konstan = $O(\log n)$ /logaritmik. ($O(\log n)$) artinya apabila terdapat 1000 elemen, hanya membutuhkan sekitar 10 langkah.

Sedangkan, disjoint set tanpa optimisasi memiliki kompleksitas waktu $O(n)$ /linear. Karena metode ini, harus mengecek satu satu elemen yang ada hingga selesai, maka akan sangat menguras waktu. waktu yang diperlukan sebanding dengan banyaknya elemen. Linear time ($O(n)$) artinya apabila terdapat 1000 elemen, bisa butuh sampai 1000 langkah.

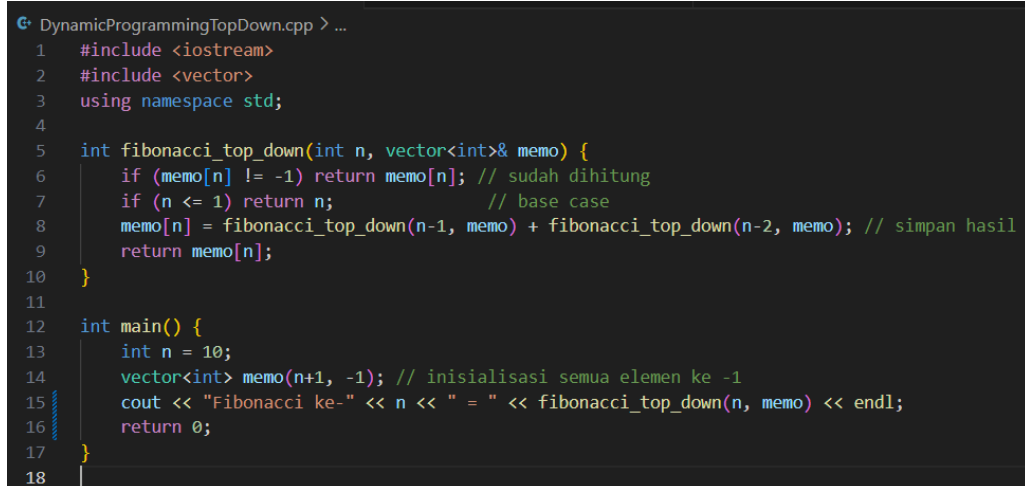
BAB III

SOURCE KODE DAN OUTPUT

F. DYNAMIC PRO

m. Dynamic Programming Memoization (Top Down)

Printscreen Source Code:



```
DynamicProgrammingTopDown.cpp > ...
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int fibonacci_top_down(int n, vector<int>& memo) {
6      if (memo[n] != -1) return memo[n]; // sudah dihitung
7      if (n <= 1) return n; // base case
8      memo[n] = fibonacci_top_down(n-1, memo) + fibonacci_top_down(n-2, memo); // simpan hasil
9      return memo[n];
10 }
11
12 int main() {
13     int n = 10;
14     vector<int> memo(n+1, -1); // inisialisasi semua elemen ke -1
15     cout << "Fibonacci ke-" << n << " = " << fibonacci_top_down(n, memo) << endl;
16     return 0;
17 }
18
```

Gambar 1.1 Source Code

Pembahasan Source Code:

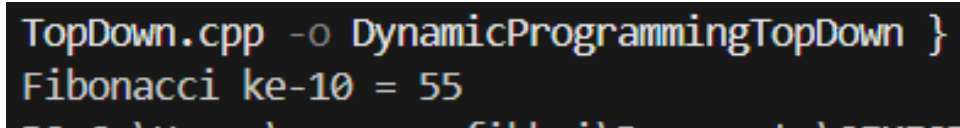
Penejelasan untuk gambar 1.1, kode di atas adalah implementasi dari algoritma pencarian bilangan Fibonacci menggunakan metode Dynamic Programming top-down atau memoization. Pertama, file program meng-include header <iostream> untuk operasi input-output dan <vector> untuk menggunakan struktur data vektor di C++. Dengan perintah using namespace std;, program ini dapat menggunakan elemen dalam namespace std tanpa harus menuliskan awalan std:: setiap saat.

Fungsi fibonacci_top_down menerima dua parameter, yaitu sebuah bilangan bulat n dan referensi ke vektor memo yang menyimpan hasil perhitungan sebelumnya. Fungsi ini pertama-tama mengecek apakah nilai Fibonacci ke-n sudah pernah dihitung dengan memeriksa apakah memo[n] tidak sama dengan -1. Jika sudah dihitung, fungsi langsung mengembalikan nilai yang disimpan di memo[n]. Jika n bernilai 0 atau 1, fungsi mengembalikan n karena itu adalah kasus dasar dalam deret Fibonacci. Jika belum pernah dihitung, fungsi akan memanggil dirinya sendiri secara rekursif untuk n-1 dan n-2, menjumlahkan hasilnya, dan menyimpannya di memo[n] agar tidak perlu dihitung ulang di masa depan.

Pada fungsi main, variabel n diatur ke 10, yang berarti program akan mencari nilai Fibonacci ke-10. Kemudian dibuat vektor memo dengan ukuran n+1, dan semua elemennya diinisialisasi ke -1 untuk menandai bahwa belum ada nilai yang dihitung. Program lalu mencetak hasil perhitungan menggunakan cout, yaitu nilai Fibonacci ke-10

yang diperoleh dari pemanggilan fungsi `fibonacci_top_down`. Terakhir, fungsi `main` mengembalikan 0 yang menandakan program berakhir dengan sukses.

Printscreen Output:



```
TopDown.cpp -o DynamicProgrammingTopDown }
Fibonacci ke-10 = 55
```

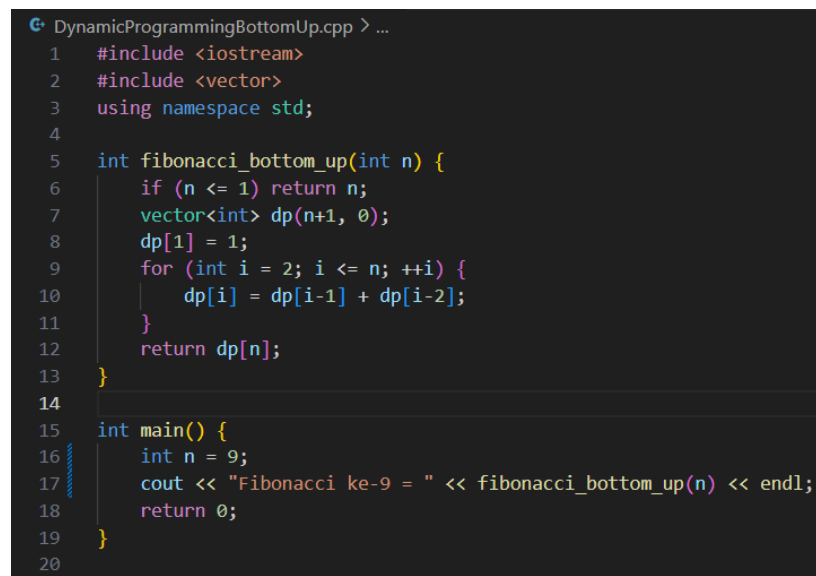
Gambar 1.2 Output

Penjelasan Output:

Output yang ditampilkan pada gambar 1.2 menunjukkan hasil kompilasi dan eksekusi program C++ yang menghitung nilai Fibonacci ke-10 menggunakan metode *dynamic programming* dengan pendekatan *top-down* (memoization). Setelah program berhasil dikompilasi tanpa error, file eksekusi yang dihasilkan dijalankan, dan program menampilkan hasil Fibonacci ke-10 = 55. Ini berarti program telah menghitung bahwa angka ke-10 dalam deret Fibonacci adalah 55, yang memang benar sesuai dengan urutan bilangan Fibonacci, yaitu 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, dan 55. Program ini menggunakan teknik penyimpanan hasil sementara (memoization) untuk menghindari perhitungan ulang pada sub-masalah yang sama, sehingga proses komputasi menjadi lebih efisien dibandingkan metode rekursif biasa. Secara keseluruhan, output yang muncul menandakan bahwa program berjalan dengan baik dan algoritma yang diterapkan bekerja sesuai dengan tujuan

n. Dynamic Programming Tabulation (Bottom-Up)

Printscreen Source Code:



```
DynamicProgrammingBottomUp.cpp > ...
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4
5  int fibonacci_bottom_up(int n) {
6      if (n <= 1) return n;
7      vector<int> dp(n+1, 0);
8      dp[1] = 1;
9      for (int i = 2; i <= n; ++i) {
10         dp[i] = dp[i-1] + dp[i-2];
11     }
12     return dp[n];
13 }
14
15 int main() {
16     int n = 9;
17     cout << "Fibonacci ke-9 = " << fibonacci_bottom_up(n) << endl;
18     return 0;
19 }
20
```

Gambar 1.3 Source Code

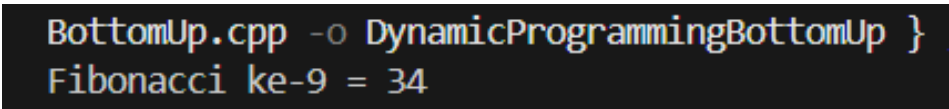
Pembahasan Source Code:

Penjelasan untuk gambar 1.3, kode di atas merupakan implementasi perhitungan bilangan Fibonacci menggunakan metode bottom-up dengan pendekatan dynamic programming di dalam bahasa C++. Pertama, program meng-include header <iostream> untuk menangani proses input-output dan <vector> untuk menggunakan struktur data vektor. Kata kunci using namespace std; digunakan agar penulisan fungsi standar seperti cout atau vector menjadi lebih ringkas tanpa perlu menuliskan awalan std::.

Fungsi fibonacci_bottom_up didefinisikan untuk menerima sebuah parameter integer n, yang merupakan indeks bilangan Fibonacci yang ingin dicari. Jika nilai n kurang dari atau sama dengan 1, fungsi langsung mengembalikan nilai n karena untuk Fibonacci ke-0 dan ke-1, hasilnya memang sama dengan indeks itu sendiri. Apabila n lebih besar dari 1, program membuat sebuah vektor dp dengan ukuran n+1 dan semua elemennya diinisialisasi dengan nilai 0. Kemudian, dp[1] diatur menjadi 1 sebagai dasar perhitungan. Setelah itu, program menjalankan perulangan for dari i = 2 sampai i = n, dan di setiap iterasi, dp[i] diisi dengan penjumlahan dp[i-1] dan dp[i-2], sesuai dengan sifat dasar bilangan Fibonacci.

Fungsi main bertugas sebagai titik awal program. Di dalamnya, variabel n diatur dengan nilai 9, artinya program akan mencari nilai Fibonacci ke-9. Nilai tersebut kemudian diproses melalui pemanggilan fungsi fibonacci_bottom_up(n) dan hasil akhirnya ditampilkan ke layar menggunakan cout. Setelah itu, program mengembalikan nilai 0 untuk menandakan bahwa program telah selesai dengan normal.

Printscreen Output:



```
BottomUp.cpp -o DynamicProgrammingBottomUp }  
Fibonacci ke-9 = 34
```

Gambar 1.4 Output

Output yang ditampilkan pada gambar 1.4 menunjukkan hasil akhir dari program yang menghitung nilai Fibonacci ke-9 menggunakan pendekatan bottom-up. Setelah program dijalankan, ia menghasilkan output berupa tulisan Fibonacci ke-9 = 34. Ini berarti bahwa program telah berhasil menghitung bahwa angka ke-9 dalam deret Fibonacci adalah 34. Nilai ini sesuai dengan deret Fibonacci standar yang dimulai dari 0, yaitu 0, 1, 1, 2, 3, 5, 8, 13, 21, dan 34. Hasil ini didapatkan melalui proses iterasi, di mana setiap nilai Fibonacci dibangun dari hasil penjumlahan dua nilai sebelumnya, dan hasil-hasil tersebut disimpan ke dalam sebuah vektor untuk menghindari perhitungan berulang. Dengan menggunakan metode bottom-up, program dapat menghitung nilai Fibonacci secara efisien dan langsung mendapatkan hasil akhir yang diinginkan.

G. MINIMUM SPANNING TREE

o. Algoritma Prim

PrintScreen Source Code:

```
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

const int jumlahKota = 5; // Jumlah kota

// Fungsi untuk mencari kota dengan kunci terkecil yang belum masuk MST
int cariKunciMinimum(int kunci[], bool sudahTermasuk[]) {
    int nilaiMinimum = INT_MAX, indeksMinimum = -1;

    for (int i = 0; i < jumlahKota; i++) {
        if (!sudahTermasuk[i] && kunci[i] < nilaiMinimum) {
            nilaiMinimum = kunci[i];
            indeksMinimum = i;
        }
    }

    return indeksMinimum;
}

void bangunMST(int graf[jumlahKota][jumlahKota]) {
    int induk[jumlahKota];
    int kunci[jumlahKota];
    bool sudahTermasuk[jumlahKota];

    for (int i = 0; i < jumlahKota; i++) {
        kunci[i] = INT_MAX;
        sudahTermasuk[i] = false;
    }

    kunci[0] = 0;
    induk[0] = -1; // Node akar tidak punya induk

    for (int langkah = 0; langkah < jumlahKota - 1; langkah++) {
        int u = cariKunciMinimum(kunci, sudahTermasuk);
        sudahTermasuk[u] = true;
```

Gambar 2.1 Source Code

```

// Perbarui nilai tetangga dari simpul terpilih
for (int v = 0; v < jumlahKota; v++) {
    if (graf[u][v] && !sudahTermasuk[v] && graf[u][v] < kunci[v]) {
        induk[v] = u;
        kunci[v] = graf[u][v];
    }
}

// Cetak hasil MST
cout << "Jaringan jalan terbaik (Minimum Spanning Tree):" << endl;
int totalBiaya = 0;
for (int i = 1; i < jumlahKota; i++) {
    cout << "Dari kota " << (char)('A' + induk[i])
        << " ke kota " << (char)('A' + i)
        << " dengan jarak " << graf[i][induk[i]] << endl;
    totalBiaya += graf[i][induk[i]];
}
cout << "Total jarak seluruh jalan: " << totalBiaya << endl;
}

int main() {
    int peta[jumlahKota][jumlahKota] = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };

    bangunMST(peta);

    return 0;
}

```

Gambar 2.2 Source Code

Pembahasan Source Code:

Program ini memiliki kode `#include <iostream>` yang berfungsi untuk mengimpor library `iostream` agar dapat menggunakan operasi input dan output, seperti `std::cout`. Kode `#include <vector>` sebenarnya ikut dimasukkan, tetapi tidak dipakai di program ini. Kode `#include <climits>` berfungsi untuk mengimpor konstanta `INT_MAX`, yang berguna untuk menginisialisasi nilai maksimum pada array kunci. Kode `const int JumlahKota = 5;` berfungsi untuk menetapkan jumlah kota (vertex) yang akan diproses, dalam hal ini ada 5 kota. Kota-kota ini secara logika bisa diibaratkan sebagai A, B, C, D, dan E.

Kode `int cariKunciMinimum(int kunci[], bool sudahTermasuk[])` adalah fungsi untuk membantu memilih kota yang memiliki nilai kunci (kunci) terkecil dan belum termasuk ke dalam MST (Minimum Spanning Tree). Fungsi ini bekerja dengan inisialisasi variabel `nilaiMinimum` dengan nilai `INT_MAX` dan variabel `indeksMinimum` yang akan menyimpan indeks kota dengan kunci terkecil. Lalu dilakukan perulangan dari `i = 0` sampai `i < jumlahKota`. Jika `sudahTermasuk[i]` bernilai `false` (artinya kota tersebut belum masuk MST) dan `kunci[i]` lebih kecil daripada `nilaiMinimum` saat ini, maka `nilaiMinimum` diupdate menjadi `kunci[i]` dan `indeksMinimum` diupdate menjadi `i`. Setelah perulangan selesai, fungsi akan mengembalikan `indeksMinimum`, yaitu indeks kota yang memiliki kunci terkecil dan belum masuk MST.

Kode `void bangunMST(int graf[jumlahKota][jumlahKota])` adalah fungsi utama untuk membangun MST menggunakan algoritma Prim. Di dalam fungsi ini, dideklarasikan tiga array: `induk[jumlahKota]`, `kunci[jumlahKota]`, dan `sudahTermasuk[jumlahKota]`. Array `induk[jumlahKota]` digunakan untuk menyimpan parent (induk) dari masing-masing kota dalam MST, array `kunci[jumlahKota]` untuk menyimpan nilai minimum edge yang menghubungkan ke MST, dan array `sudahTermasuk[jumlahKota]` untuk melacak apakah suatu kota sudah dimasukkan ke MST atau belum.

Pada awalnya, semua elemen kunci diinisialisasi ke `INT_MAX` supaya semua kota dianggap belum memiliki jalur minimum, dan semua elemen `sudahTermasuk` diinisialisasi ke `false` supaya semua kota dianggap belum masuk MST.

Kode `kunci[0] = 0`; berfungsi untuk memulai MST dari kota pertama, yaitu kota A. Ini berarti kota A akan dipilih pertama kali. Kode `induk[0] = -1`; menandakan bahwa kota A adalah root dari MST, sehingga tidak memiliki parent. Selanjutnya dilakukan perulangan sebanyak `jumlahKota - 1` kali (yaitu 4 kali untuk 5 kota). Pada setiap iterasi, fungsi `cariKunciMinimum(kunci, sudahTermasuk)` dipanggil untuk memilih kota dengan kunci terkecil yang belum ada di MST. Setelah mendapatkan kota `u`, kota tersebut dimasukkan ke MST dengan mengatur `sudahTermasuk[u] = true`.

Kemudian dilakukan perulangan kedua untuk semua kota `v`. Jika terdapat edge antara `u` dan `v` (dengan kata lain, `graf[u][v]` tidak 0), `v` belum ada di MST (`!sudahTermasuk[v]`), dan bobot edge `graf[u][v]` lebih kecil dari `kunci[v]`, maka:

- `induk[v]` akan diisi dengan `u`, artinya kota `v` akan dihubungkan dengan `u` di MST.
- `kunci[v]` diupdate dengan bobot dari edge `graf[u][v]`, supaya di iterasi berikutnya, `v` bisa dipilih kalau memang menjadi edge terkecil.

Setelah semua proses perulangan selesai, program akan menampilkan hasil MST.

Kode `cout << "Jaringan jalan terbaik (Minimum Spanning Tree):"` berfungsi untuk mencetak judul kolom. Lalu dilakukan loop dari `i = 1` sampai `i < jumlahKota`, untuk mencetak edge dari induk ke anak dan bobotnya. Kota diubah dari angka ke huruf menggunakan `(char)('A' + induk[i])` dan `(char)('A' + i)`, sehingga yang tercetak di layar adalah huruf A, B, C, D, dan E, bukan angka 0, 1, 2, 3, dan 4.

Selain itu, program juga menghitung `totalBiaya`, yaitu total jarak seluruh jalan yang terbentuk dalam MST, dan mencetak hasilnya di akhir. Di dalam fungsi `main()`, dibuat matriks `peta[jumlahKota][jumlahKota]` yang merepresentasikan bobot antar kota. Contohnya, `peta[0][1] = 2` berarti jarak atau bobot dari kota A ke kota B adalah 2. Jika

peta[i][j] = 0, artinya tidak ada jalan langsung antara kota i dan j. Setelah peta didefinisikan, fungsi bangunMST(peta); dipanggil untuk mencari dan mencetak MST.

Printscreen Output:

```
Jaringan jalan terbaik (Minimum Spanning Tree):  
Dari kota A ke kota B dengan jarak 2  
Dari kota B ke kota C dengan jarak 3  
Dari kota A ke kota D dengan jarak 6  
Dari kota B ke kota E dengan jarak 5  
Total jarak seluruh jalan: 16  
PS D:\Documents\Coding\SDA teori>
```

Gambar 2.3 Output

Penjelasan Output:

Pada program ini, kita menggunakan **algoritma Prim** untuk mencari MST (Minimum Spanning Tree) dari 5 kota (A sampai E). Output program menampilkan koneksi antar kota (edge) beserta bobot (weight) minimum yang membentuk MST.

Program pertama memilih kota A sebagai titik awal karena sudah di-set kunci[0] = 0. Dari kota A, dipilih edge terkecil, yaitu ke B dengan bobot 2, sehingga tercetak A - B 2.

Selanjutnya, dari semua kota yang sudah terhubung (A dan B), dipilih lagi edge terkecil berikutnya. Dari kota B, ke kota C memiliki bobot 3 yang paling kecil dibandingkan edge lain, sehingga tercetak B - C 3.

Setelah itu, dari kota A, B, dan C yang sudah masuk MST, dicari kembali edge minimum ke kota yang belum terhubung. Didapatkan A ke D dengan bobot 6, maka tercetak

A	-	D	6.
---	---	---	----

Terakhir, dari kota yang sudah terhubung (A, B, C, D), edge minimum berikutnya adalah dari B ke E dengan bobot 5, sehingga tercetak B - E 5.

Dengan begitu, hasil akhir MST berhasil menghubungkan semua kota dengan total bobot minimum. Program mencetak hasil MST dengan menampilkan setiap edge berdasarkan hubungan antara induk dan anak di MST, sesuai dengan struktur yang dibentuk oleh algoritma Prim

p. Algoritma Krusal

PrintScreen Source Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>

using namespace std;

struct Sisi {
    int asal, tujuan, bobot;
};

struct Graf {
    int jumlahKota, jumlahSisi;
    vector<Sisi> daftarSisi;
};

bool bandingkan(Sisi a, Sisi b) {
    return a.bobot < b.bobot;
}

int cariInduk(int induk[], int i) {
    if (induk[i] == -1)
        return i;
    return cariInduk(induk, induk[i]);
}

void gabungkanSet(int induk[], int x, int y) {
    induk[x] = y;
}

void kruskalMST(Graf& graf) {
    // Mengurutkan sisi berdasarkan bobot
    sort(graf.daftarSisi.begin(), graf.daftarSisi.end(), bandingkan);

    int induk[graf.jumlahKota];
    fill_n(induk, graf.jumlahKota, -1);

    vector<Sisi> hasil; // Untuk menyimpan hasil MST

    for (auto sisi : graf.daftarSisi) {
        int x = cariInduk(induk, sisi.asal);
        int y = cariInduk(induk, sisi.tujuan);
```

Gambar 3.1 Source Code


```

// Jika x dan y tidak berada dalam satu set, tambahkan sisi ke MST
if (x != y) {
    hasil.push_back(sisi);
    gabungkanSet(induk, x, y);
}

// Menampilkan MST
cout << "Jaringan jalan terbaik (Minimum Spanning Tree) dengan Kruskal:" << endl;
int totalBiaya = 0;
for (auto sisi : hasil) {
    cout << "Dari kota " << (char)('A' + sisi.asal)
         << " ke kota " << (char)('A' + sisi.tujuan)
         << " dengan jarak " << sisi.bobot << endl;
    totalBiaya += sisi.bobot;
}
cout << "Total jarak seluruh jalan: " << totalBiaya << endl;
}

int main() {
    int jumlahKota = 5; // Jumlah kota
    int jumlahSisi = 8; // Jumlah sisi

    Graf graf;
    graf.jumlahKota = jumlahKota;
    graf.jumlahSisi = jumlahSisi;

    graf.daftarSisi.push_back({0, 1, 2}); // A - B
    graf.daftarSisi.push_back({0, 3, 6}); // A - D
    graf.daftarSisi.push_back({1, 2, 3}); // B - C
    graf.daftarSisi.push_back({1, 4, 1}); // B - E
    graf.daftarSisi.push_back({2, 4, 7}); // C - E
    graf.daftarSisi.push_back({3, 4, 9}); // D - E
    graf.daftarSisi.push_back({3, 1, 8}); // D - B
    graf.daftarSisi.push_back({2, 3, 2}); // C - D

    kruskalMST(graf);

    return 0;
}

```

Gambar 3.2 Source Code

Pembahasan Source Code:

Program ini merupakan implementasi Algoritma Kruskal dalam bahasa C++ untuk membangun Minimum Spanning Tree (MST) dari sebuah graf berbobot. Di bagian awal, program mengimpor beberapa library penting, yaitu `<iostream>` untuk operasi input-output, `<vector>` untuk menggunakan struktur data vector, `<algorithm>` untuk mengurutkan data, dan `<climits>` untuk penggunaan konstanta batas nilai jika diperlukan.

Program ini mendefinisikan sebuah struktur bernama `Sisi` yang berfungsi untuk menyimpan informasi tentang sebuah sisi dalam graf, yaitu kota asal, kota tujuan, dan bobot yang menghubungkan keduanya. Selain itu, ada juga struktur `Graf` yang menyimpan jumlah kota, jumlah sisi, serta daftar sisi dalam bentuk vector. Untuk mengurutkan sisi berdasarkan bobot terkecil, digunakan fungsi `bandingkan(Sisi a, Sisi b)`, sehingga proses pemilihan sisi dengan bobot terendah dalam Kruskal dapat berjalan dengan benar.

Fungsi `cariInduk(int induk[], int i)` berfungsi untuk mencari root parent dari suatu kota dalam struktur union-find. Jika suatu kota belum memiliki induk (`induk[i] == -1`), maka kota tersebut adalah root dari dirinya sendiri. Fungsi ini penting untuk memastikan bahwa saat menambahkan sebuah sisi, kita tidak membentuk siklus dalam MST. Fungsi `gabungkanSet(int induk[], int x, int y)` berfungsi untuk menyatukan dua komponen berbeda dengan menghubungkan root-nya, sehingga memperbarui struktur set dalam union-find.

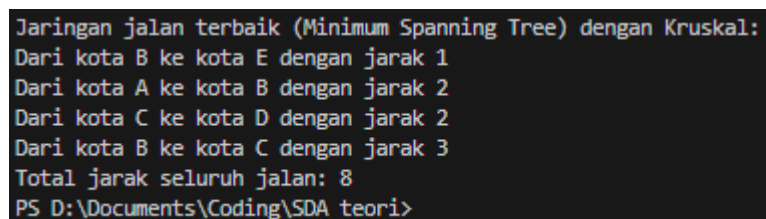
Fungsi utama `kruskalMST(Graf& graf)` menjadi pusat dari algoritma Kruskal. Pada fungsi ini, semua sisi dalam graf terlebih dahulu diurutkan berdasarkan bobotnya dari yang terkecil ke yang terbesar menggunakan fungsi `sort()`. Kemudian dibuat sebuah array induk yang berfungsi untuk mengatur struktur union-find, yang awalnya semua elemennya diisi dengan -1 menandakan masing-masing kota berdiri sendiri. Sebuah vector hasil disiapkan untuk menyimpan sisi-sisi yang akhirnya akan menjadi bagian dari MST.

Dalam prosesnya, program akan memeriksa setiap sisi yang sudah terurut. Untuk setiap sisi, program mencari root parent dari kota asal dan kota tujuan. Jika kedua kota tersebut berada dalam komponen yang berbeda (artinya penambahan sisi ini tidak membuat siklus), maka sisi tersebut dimasukkan ke MST dan kedua komponen tersebut digabungkan menjadi satu. Proses ini akan terus berlangsung sampai MST terbentuk, yaitu ketika jumlah sisi dalam MST adalah satu kurang dari jumlah kota.

Setelah seluruh MST terbentuk, program akan mencetak semua koneksi antar kota beserta bobot jalannya. Nama kota diubah dari indeks angka menjadi huruf, misalnya 0 menjadi 'A', 1 menjadi 'B', dan seterusnya, agar hasil output lebih mudah dipahami. Di akhir, program juga menghitung dan menampilkan total bobot seluruh jalan dalam MST.

Pada fungsi `main()`, graf didefinisikan dengan 5 kota dan 8 sisi yang menghubungkan berbagai kota dengan bobot tertentu. Setiap koneksi antar kota dimasukkan ke dalam daftar sisi graf. Setelah semua data selesai diinput, fungsi `kruskalMST()` dipanggil untuk memproses graf tersebut dan menampilkan hasil jaringan jalan terbaik berdasarkan MST.

Printscreen Output:



```
Jaringan jalan terbaik (Minimum Spanning Tree) dengan Kruskal:  
Dari kota B ke kota E dengan jarak 1  
Dari kota A ke kota B dengan jarak 2  
Dari kota C ke kota D dengan jarak 2  
Dari kota B ke kota C dengan jarak 3  
Total jarak seluruh jalan: 8  
PS D:\Documents\Coding\SDA teori>
```

Gambar 3.3 Output

Penjelasan Output:

Pada program ini, kita menggunakan algoritma Kruskal untuk mencari Minimum Spanning Tree (MST) dari sebuah graf yang terdiri dari 5 kota (A, B, C, D, E) dan 8 jalan. Program pertama-tama mengurutkan seluruh jalan berdasarkan bobot terkecil menggunakan fungsi `sort`, lalu memilih jalan satu per satu dengan syarat jalan tersebut tidak membentuk siklus.

Jalan pertama yang dipilih adalah antara kota B dan E dengan bobot 1. Karena kota B dan E sebelumnya belum tergabung dalam satu komponen, jalan ini langsung masuk ke MST. Selanjutnya, program memilih jalan antara kota A dan B yang memiliki bobot 2. Karena A dan B juga berasal dari komponen berbeda, maka jalan ini pun ditambahkan ke MST. Kemudian program memilih jalan antara kota C dan D dengan bobot 2, karena C dan D belum terhubung, sehingga penggabungan ini valid dan jalan tersebut masuk ke MST. Setelah itu, program memilih jalan antara kota B dan C dengan bobot 3. Walaupun kota B sudah terhubung dengan A dan E, kota C berada di komponen yang berbeda, sehingga jalan B ke C juga bisa dimasukkan ke MST tanpa membentuk siklus.

Pada titik ini, seluruh kota sudah saling terhubung satu sama lain, sehingga MST telah selesai terbentuk. Program kemudian menampilkan hasil MST ke layar dengan format pasangan kota beserta bobotnya. Hasil MST yang dicetak adalah dari kota B ke E dengan bobot 1, A ke B dengan bobot 2, C ke D dengan bobot 2, dan B ke C dengan bobot 3.

Total bobot dari semua jalan yang terpilih untuk MST adalah 8. Karena algoritma Kruskal selalu memilih sisi dengan bobot terkecil dari seluruh graf, tanpa bergantung pada titik awal tertentu, hasil MST yang dihasilkan bisa berbeda dengan MST dari algoritma Prim yang memperluas dari satu titik ke tetangga terdekat. Pada perubahan graf ini, hasil MST dari algoritma Kruskal memang berbeda dari hasil Prim seperti yang diharapkan, namun tetap membentuk koneksi minimum yang menghubungkan semua kota.

H. Disjoint Sets

q. Quick Find

PrintScreen Source Code:

```

1  #include <vector>
2  #include <iostream>
3  class QuickFind {
4
5      std::vector<int> id;
6
7      public:
8      int i;
9
10     QuickFind(int N) {
11         id.resize(N);
12         for (i=0; i < N; ++i){
13             id[i] = i;
14         }
15
16         bool terhubung(int p, int q) {
17             return id[p] == id[q];
18         }
19
20         void gabungElemen(int p, int q){
21             int pid = id[p];
22             int qid = id[q];
23
24             for (i=0; i < id.size(); ++i) {
25                 if (id[i] == pid) {
26                     id[i]=qid;
27                 }
28             }
29         }
30
31         void cetakId() {
32             for (i=0; i < id.size(); ++i){
33                 std::cout << id[i] << " ";
34             }
35             std::cout << std::endl;
36         }
37     };

```

Gambar 4.1 Source Code

```

38 int main() {
39     int N = 10;
40     QuickFind uf(N);
41
42     uf.gabungElemen(4, 9);
43     uf.gabungElemen(8, 4);
44
45     std::cout << std::boolalpha;
46     std::cout << "Terhubung(8, 9): " << uf.terhubung(8, 9) << std::endl;
47     std::cout << "Terhubung(5, 0): " << uf.terhubung(5, 0) << std::endl;
48
49     uf.cetakId();
50     return 0;
51 }

```

Gambar 4.2 Source Code

Pembahasan Source Code:

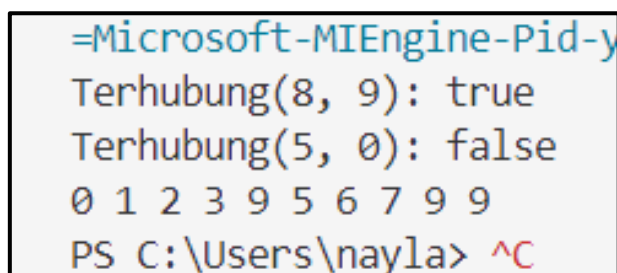
Program ini memiliki kode `#include <vector>` berfungsi untuk mengimpor library vector untuk membuat array dan Kode `#include <iostream>` berfungsi agar kita dapat menggunakan operasi input & output. Kode `class QuickFind { : std::vector<int> id;` berfungsi mendeklarasikan kelas Bernama QuickFind dan menyimpan satu data, yaitu sebuah vector berisi integer yang menyimpan id komponen dari elemen i. Kode `int i;` berfungsi untuk mendeklarasikan variabel i bertipe data integer. Kode `public: QuickFind(int N) {` berfungsi untuk membuat objek bernama QuickFind dengan elemen N. Kode `id.resize(N);` membuat vector id berukuran N. Kode `for (i = 0; i < N; ++i) { id[i] = i;`

membuat setiap elemen `id[i]` mulai dari 0 sampai `N-1` akan diisi dengan masing-masing nilai `i`. Misalnya `N = 5` maka `id = [0, 1, 2, 3, 4]`.

Kode `bool terhubung(int p, int q) { return id[p] == id[q];` berfungsi untuk mengecek apakah elemen `p` dan `q` terhubung, jika terhubung maka fungsi akan mengembalikan nilai `true`, jika tidak maka `false`. Kode `void gabungElemen(int p, int q) { int pid = id[p]; int qid = id[q];` berfungsi untuk menggabungkan dua komponen `p` dan `q`. Caranya dengan menyimpan `id[p]` ke `pid` dan `id[q]` ke `qid`. Kode `for (i = 0; i < id.size(); ++i) { if (id[i] == pid) { id[i] = qid;` berfungsi untuk melakukan loop semua elemen ke dalam `id`, jika ada elemen dengan `id[i] == pid`, artinya dia berada di komponen `p` jadi kita ubah `id[i]` menjadi `qid`, supaya dia pindah ke komponen `q`. Jadi semua elemen yang sekomponen dengan `p` akan dipindahkan ke komponen `q`. Kode `void cetakId() { for (i = 0; i < id.size(); ++i) ; std::cout << id[i] << " "; }` berfungsi untuk menampilkan array `id` ke layar.

Kode `int main() { int N = 10; QuickFind uf(N);` merupakan awal dari main program. Kode ini berfungsi untuk membuat objek `uf` dengan 10 elemen (0-9). Kode `uf.gabungElemen(4, 9);` dan `uf.gabungElemen(8, 4);` berfungsi untuk menggabungkan elemen. Pertama kita gabungkan elemen 4 dan 9, jadi semua elemen yang memiliki `id[i] == id[4]` akan diubah menjadi `id[9]`. Lalu kita gabungkan elemen 8 dan 4. Karena 4 sudah terhubung dengan 9, maka 8 akan digabungkan juga ke 9. Kode `std::cout << std::boolalpha; std::cout << "Terhubung(8, 9): " << uf.terhubung(8, 9) << std::endl; std::cout << "Terhubung(5, 0): " << uf.terhubung(5, 0) << std::endl` berfungsi untuk menggunakan `boolalpha` agar hasil Boolean (`true/false`) dicetak sebagai tulisan, bukan angka 1/0. Disini kita mengecek apakah 8 dan 9, 5 dan 0 terhubung. Kode `uf.cetakId();` berfungsi untuk cetak array `id[i]` untuk melihat hasil akhir dari operasi yang sudah kita lakukan.

Output:



```
=Microsoft-MIEngine-Pid-y
Terhubung(8, 9): true
Terhubung(5, 0): false
0 1 2 3 9 5 6 7 9 9
PS C:\Users\nayla> ^C
```

Gambar 4.3 Output

Penjelasan Output:

Pada program kita sudah melakukan operasi union untuk elemen 8 dan 9 maka output menampilkan `true`, sedangkan kita tidak melakukan operasi union untuk elemen 5

dan 0 maka output yang dihasilkan adalah false. Lalu output program juga menampilkan `id[i]` dari setiap elemen array, kita bisa melihat untuk elemen ke 4, 8, dan 9 memiliki `id[i]` yang sama, yaitu `id[9]`.

BAB IV

PENUTUP

I. KESIMPULAN

Berdasarkan hasil pembahasan, dapat disimpulkan bahwa penggunaan struktur data dan algoritma yang tepat sangat berperan penting dalam meningkatkan efisiensi pemrosesan data dan pemecahan masalah. Dynamic Programming efektif untuk menyelesaikan masalah dengan substruktur optimal dan submasalah yang tumpang tindih, MST membantu mengoptimalkan jaringan dengan biaya minimum melalui algoritma Prim dan Kruskal, sedangkan Disjoint Sets mempercepat operasi pengelompokan elemen dalam berbagai aplikasi graf. Implementasi ketiga algoritma ini membuktikan bahwa pendekatan yang terstruktur dan efisien mampu memberikan solusi optimal untuk berbagai permasalahan di bidang informatika.

J. SARAN

Agar pemahaman dan penerapan algoritma Dynamic Programming, Minimum Spanning Tree, dan Disjoint Sets dapat lebih optimal, disarankan agar mahasiswa memperdalam teknik-teknik optimasi seperti memoization pada Dynamic Programming, serta union by rank/size dan path compression pada Disjoint Sets. Selain itu, latihan penerapan algoritma dalam kasus nyata dengan tingkat kompleksitas yang beragam sangat penting untuk meningkatkan kemampuan problem solving. Penggunaan tools visualisasi algoritma juga dianjurkan untuk membantu memperjelas konsep kerja masing-masing algoritma secara intuitif. Mahasiswa juga perlu mengembangkan keterampilan dalam mengintegrasikan beberapa algoritma sekaligus dalam satu solusi komputasi, agar mampu menghadapi tantangan yang lebih kompleks di dunia industri. Dengan pemahaman mendalam dan praktik yang berkelanjutan, diharapkan mahasiswa dapat menghasilkan solusi komputasi yang lebih efisien, efektif, dan inovatif di masa depan.