**LAPORAN PRAKTIKUM**

**MATA KULIAH STRUKTUR DATA – GRAPH**

Laporan Ini Dibuat Untuk Memenuhi Tugas Mata Kuliah Struktur Data

Dosen Pengampu : Asep Jamaludin, S.Si., M.Kom.



|  |  |
| --- | --- |
| Nama | : Muhammad Rizky Dermawan |
| NPM | : 2410631170038 |
| Kelas | : 2B – Informatika |

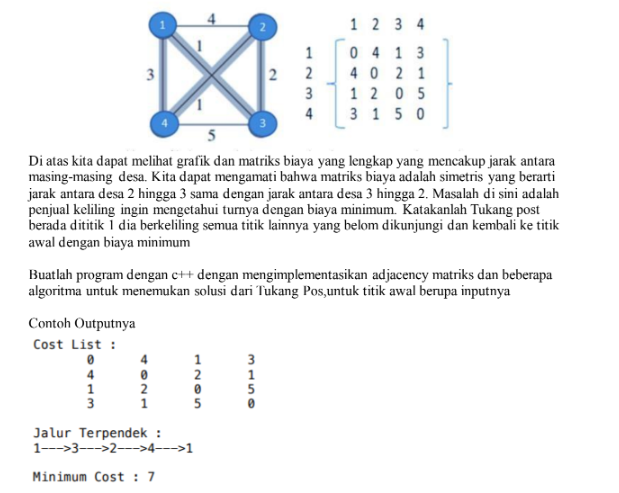
**PROGRAM STUDI INFORMATIKA**

**FAKULTAS ILMU KOMPUTER**

**UNIVERSITAS SINGAPERBANGSA KARAWANG**

**2025**

**DIBERIKAN SOAL**

**1.**

**JAWAB**



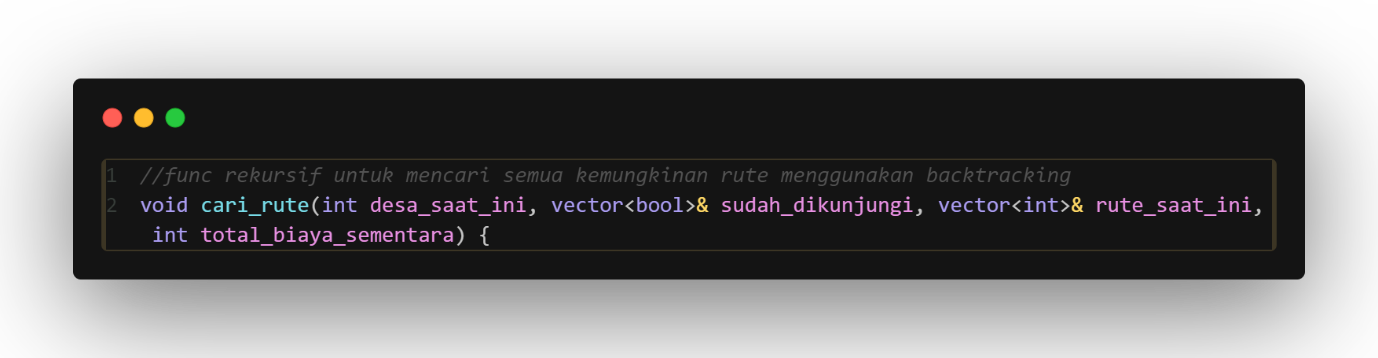
Pertama-tama, dimulai dari bagian awal program ini. Program ini dibuat untuk menyelesaikan persoalan rute terpendek antar desa, yang mirip banget sama masalah Traveling Salesman Problem (TSP). Tujuannya utamanya adalah mencari jalur paling hemat buat keliling semua desa dan balik lagi ke awal.

Pertama definisikan **jumlah total desa** yang bakal kunjungi dalam perjalanan.  
Nilainya 4, maksudnya adalah punya **empat desa** yang harus disambangi satu per satu.

Selanjutnya definisikan matriks biaya yang berisi ongkos perjalanan antar desa. Matriks ini berukuran 4x4 karena ada 4 desa, dan biaya[i][j] berarti ongkos dari desa ke-i ke desa ke-j.

Selanjutnya membut variabel **int biaya\_minimum = INT\_MAX;** yang dipake untuk menyimpan total biaya minimum dari semua kemungkinan rute yang dicoba. Nilai awalnya diset ke INT\_MAX, yaitu nilai terbesar yang bisa ditampung tipe int. Tujuannya itu rute pertama yang ditemukan langsung dianggap sebagai yang paling murah, lalu dibandingkan dengan rute-rute berikutnya. Dan jika ditemukan rute dengan total biaya yang lebih rendah, nilainya akan diperbarui.

**vector<int> rute\_terbaik;** ini adalah **vektor** (array dinamis) yang isinya adalah urutan desa yang dilalui dalam **rute terbaik** sejauh ini, berdasarkan biaya paling rendah yang telah dihitung. Jadi akan disimpan kedalam array rute\_terbaik.



Function cari\_rute adalah inti dari pencarian rute terbaik dan function program ini. Fungsi ini bekerja secara **rekursif**, maskudnya itu dia akan memanggil dirinya sendiri terus-menerus untuk menjelajahi semua kemungkinan urutan kunjungan desa. Pendekatan ini disebut **backtracking**, karena setiap kali selesai mencoba satu rute, program akan “mundur” lagi untuk mencoba pilihan rute lain yang belum dicoba. Func ini menerima 4 paramter, yaitu:

* desa\_saat\_ini: posisi saat ini (lagi di desa mana)
* sudah\_dikunjungi: vektor yang menandai desa mana saja yang sudah dikunjungi
* rute\_saat\_ini: menyimpan rute saat ini yang sedang dicoba
* total\_biaya\_sementara: jumlah biaya dari rute yang sedang berjalan



Selanmjutnya adalah pengecekan untuk semua desa sudah dikunjungi. Ini dicek dengan memeriksa ukuran dari rute\_saat\_ini. Kalau ukurannya sudah sama dengan jumlah\_desa, berarti rute sudah lengkap (seluruh desa sudah disinggahi sekali). Tapi belum selesai di situ, karena harus **kembali ke desa awal** agar rutenya melingkar. Jadi, biaya dari desa\_saat\_ini ke 0 (desa awal) ditambahkan ke total\_biaya\_sementara, hasilnya jadi total\_biaya. Kalau ternyata total biaya ini lebih kecil dari biaya\_minimum yang tersimpan sebelumnya, maka artinya rute ini adalah yang terbaik sejauh ini. Maka biaya\_minimum diperbarui, dan rute\_saat\_ini disimpan ke rute\_terbaik.



Selanjutnya adalah perulangan for untuk literasi atayu mencoba semua kemungkinan desa yang bisa dikunjungi berikutnya. Loop berjalan dari 0 sampai jumlah\_desa-1, dan di setiap iterasi dicek:

* apakah desa tersebut belum dikunjungi (!sudah\_dikunjungi[desa\_tujuan])
* dan desa itu bukan desa yang sedang disinggahi (desa\_tujuan != desa\_saat\_ini)

Nah kalau semisalkan dua kondisi itu terpenuhi, artinya desa itu sah untuk dikunjungi berikutnya. Maka:

* Desa tersebut ditandai sudah dikunjungi (true)
* Ditambahkan ke rute\_saat\_ini
* Fungsi cari\_rute dipanggil lagi secara rekursif dengan posisi baru (desa\_tujuan) dan total biaya yang diperbarui sesuai biaya dari desa\_saat\_ini ke desa\_tujuan.

Begitu rekursi selesai, proses **backtracking** dilakukan:

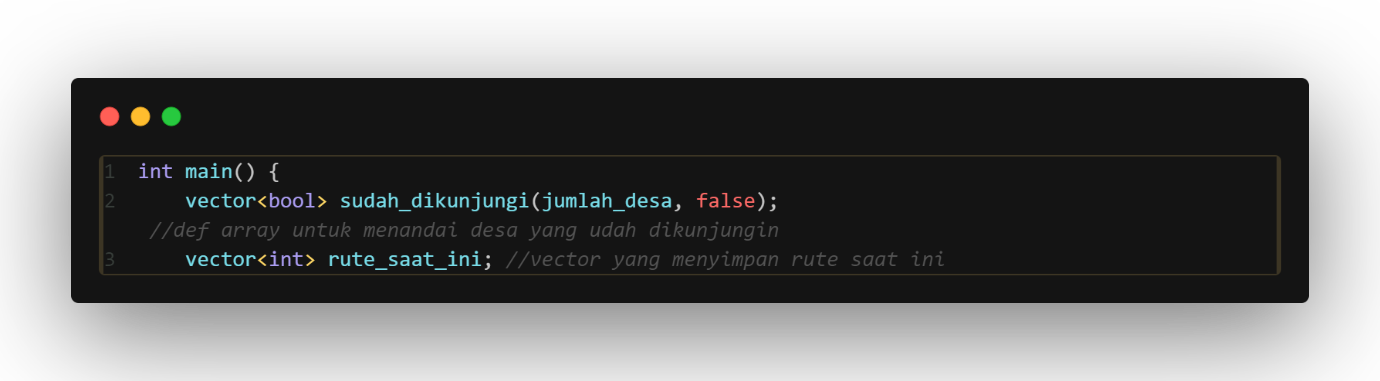
* Desa terakhir dihapus dari rute
* Tandanya juga di-reset jadi belum dikunjungi (false)

**ALUR KERJA UMUM NYA GINI**

1. **Mulai dari desa 0**, ditandai sudah dikunjungi, dimasukkan ke rute awal.
2. Masuk ke fungsi cari\_rute, semua kemungkinan desa tujuan dicek satu per satu.
3. Untuk setiap pilihan:

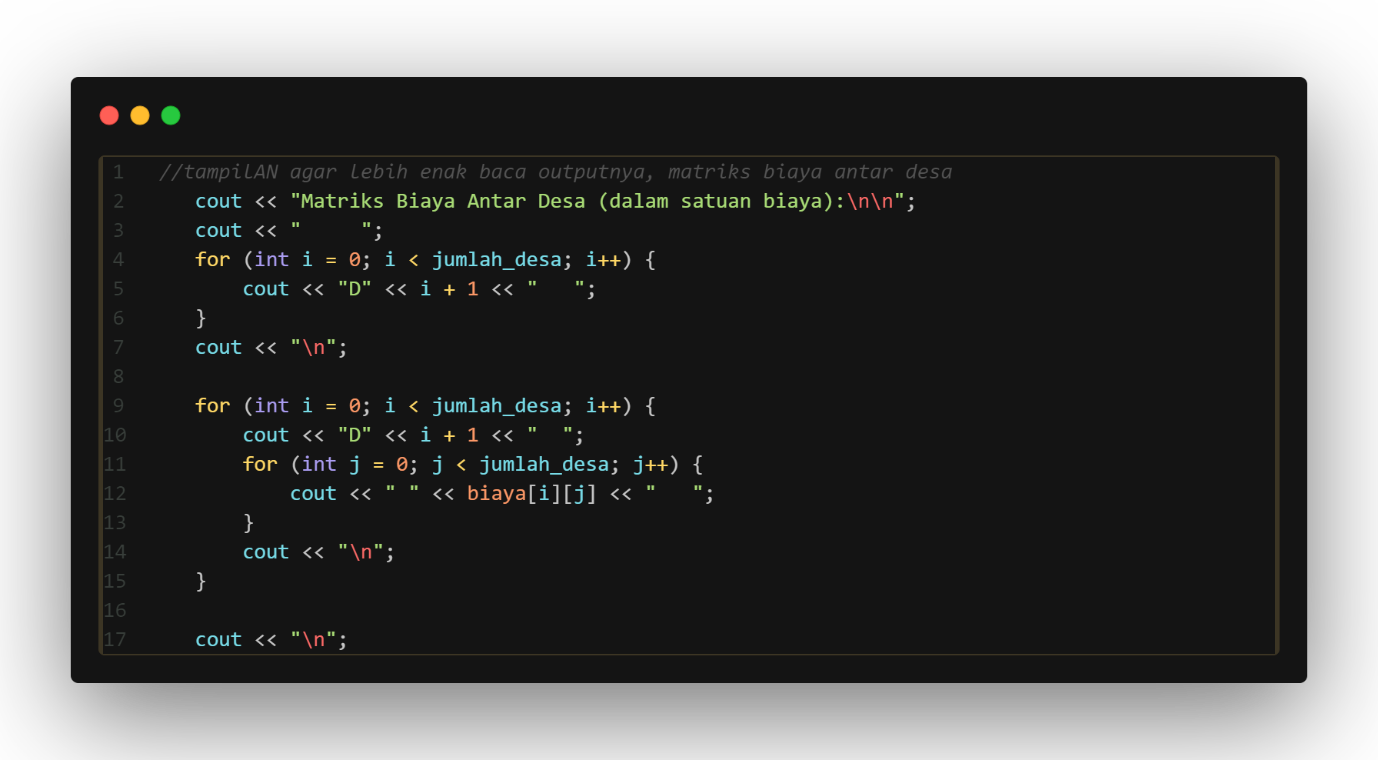
* Desa ditandai sudah dikunjungi
* Masuk rekursi lebih dalam
* Proses terus berlangsung sampai rute sudah menyentuh semua desa

1. Begitu sampai ke titik di mana semua desa sudah dikunjungi, ditambahkan biaya kembali ke awal, dan totalnya dibandingkan dengan biaya minimum sejauh ini.
2. Setelah selesai mengevaluasi satu rute lengkap, kembali ke level sebelumnya dan lanjut coba pilihan rute yang lain.
3. Proses terus berlangsung sampai semua kombinasi rute dijelajahi.
4. Di akhir, akan tersimpan rute terbaik (biaya paling kecil) dalam variabel rute\_terbaik.



Lanjut ke dalam program utamanya. Pertama-tama, membuat sebuah vektorboolean bernama sudah\_dikunjungi, dengan ukuran sebanyak jumlah desa (jumlah\_desa), dan seluruh elemennya diisi false. Nah maksudnya ini, belum ada desa yang dikunjungi. Vektor ini akan berguna saat proses pencarian rute nanti, untuk melacak desa mana yang sudah atau belum disinggahi.

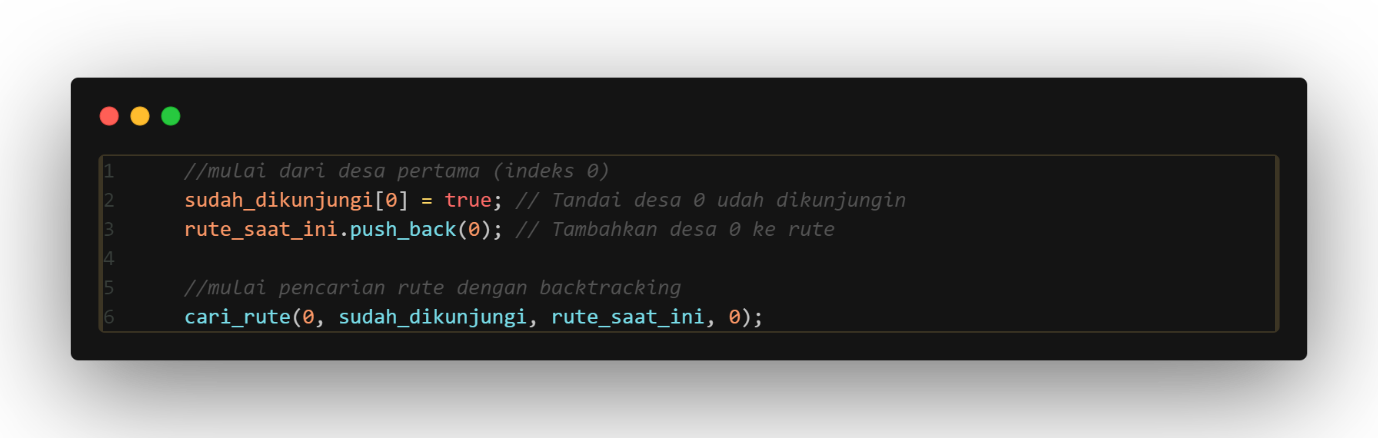
Setelah itu, dibuat juga rute\_saat\_ini, yaitu sebuah vektor integer kosong yang akan menyimpan urutan desa yang sedang dijelajahi. Vektor ini akan terus diisi dan dikosongkan selama proses rekursif berlangsung, dan pada akhirnya bisa menjadi rute terbaik kalau kombinasi tersebut cocok menghasilkan biaya paling kecil. Dua vektor ini (sudah\_dikunjungi dan rute\_saat\_ini).



Selanjutnya pada bagian ini cuma berfungsi sebagai pemanis tampilan output biar pengguna bisa lihat isi matriks biaya antar desa dengan lebih rapi dan gampang dibaca. Pertama, ditampilkan header kolom (D1, D2, dst) yang artinya desa satu sampai desa empat, lalu diikuti isi dari setiap baris dan kolom yang menunjukkan biaya perjalanan dari satu desa ke desa lainnya. Ini engga mempengaruhi logika program, hanya bantu visualisasi data biar lebih jelas.

Outputnya begini:

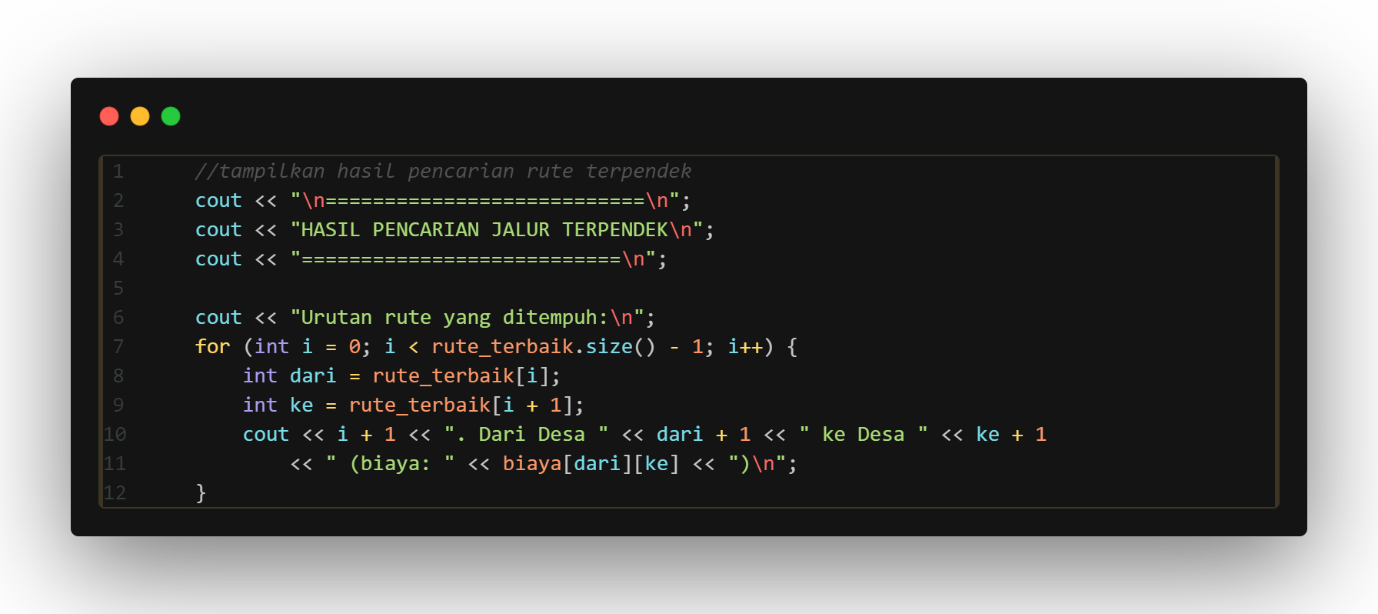




Selanjutnya adalah memanggil func cari\_rute() untuk memulai proses pencarian semua kemungkinan rute perjalanan antar desa. Tapi sebelum function itu dijalankan, terlebih dahulu desa awal (dalam hal ini desa dengan indeks 0 atau *Desa 1*) ditandai sebagai sudah dikunjungi dengan sudah\_dikunjungi[0] = true, dan juga ditambahkan ke dalam rute awal lewat rute\_saat\_ini.push\_back(0). Jadi desa 1 ditambahkan pertama kalinya karena dari desa 1 lah rute pertama kali dimulai.

Selanjutnya adalah **memanggil func** cari\_rute() yang menjadi inti dari proses pencarian rute terbaik. Function ini dipanggil dengan empat parameter:

* 0, maksudnya pencarian dimulai dari **desa dengan indeks 0** (alias Desa 1),
* sudah\_dikunjungi, adalah vektor yang nyimpan status desa mana aja yang udah dikunjungi,
* rute\_saat\_ini, adalah vektor yang nyimpan urutan desa yang sedang ditempuh,
* 0, yang maksudnya adalah **total biaya sementara**,yang masih nol karena belum jalan ke mana-mana.



Selanjutnya adalah membuat kode untuk menampilkan output di terminal, program menuliskan **urutan rute yang ditempuh**, berdasarkan isi dari rute\_terbaik, yaitu vektor yang udah nyimpan jalur dengan total biaya paling rendah.

Setelah program menemukan rute\_terbaik (hasil dari proses backtracking pada func sebelumnya), program harus mencetak rute itu satu per satu ke terminal. Jadi menggunakan perulangan for yang jalan dari i = 0 sampai rute\_terbaik.size() - 2. Nah pertanyannya kenapa -2? Karena kita ambil dua desa sekaligus di tiap iterasi, satu sebagai **desa asal** (rute\_terbaik[i]), dan satu lagi sebagai **desa tujuan** (rute\_terbaik[i + 1]). Jadi kalau total desa ada 4, indeks terakhir yang boleh dipakai sebagai asal adalah indeks ke-2 (i = 2), supaya i + 1 masih valid.

Setiap iterasi mencetak satu langkah perjalanan, dari desa ke-i menuju desa ke-i + 1, lengkap dengan biaya perjalanannya yang diambil dari matriks biaya.

Misalnya gini, kalau rute\_terbaik berisi: {0, 2, 1, 3} looping akan mencetak:

1. Dari Desa 1 ke Desa 3
2. Dari Desa 3 ke Desa 2
3. Dari Desa 2 ke Desa 4

Lalu setelah perulangan ini selesai, program keluar dari loop.



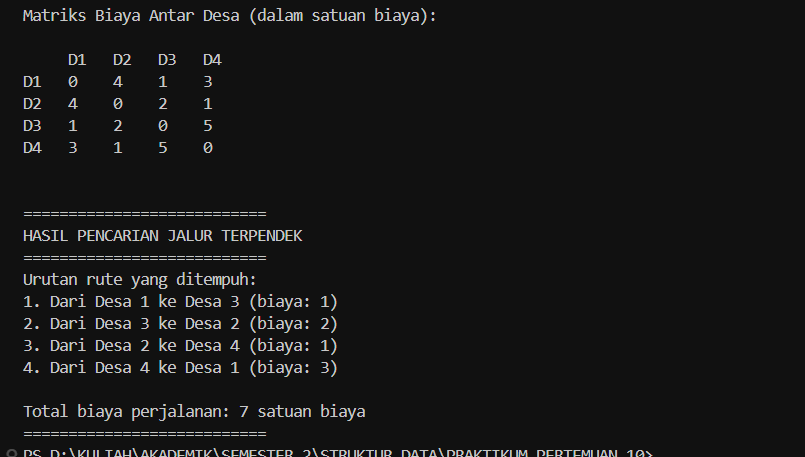
Selanjutnya adalah bagian terakhir dari program, dimana proses untuk menambahkan langkah kembali ke desa awal, menampilkan total biaya perjalanan.

Setelah seluruh rute dicetak lewat for loop sebelumnya, program masih perlu menyelesaikan satu langkah terakhir buat perjalanan membentuk rute melingkar. Nah ini dipake dengan mengambil desa terakhir dari rute\_terbaik menggunakan .back(), dan menyambungkannya kembali ke desa pertama (rute\_terbaik[0]).

Langkah ini ditampilkan di output sebagai langkah keempat (kalau ada 4 desa), misalnya:  
4. Dari Desa 4 ke Desa 1 (biaya: 3)

Lalu jika semua langkah rute dicetak, program mencetak total biaya perjalanan yang dah disimpan sebelumnya di variabel biaya\_minimum. Nilai ini adalah hasil dari pencarian jalur paling efisien (biaya paling kecil) lewat fungsi rekursif backtracking yang sudah dibuat diatas tadi.

OUTPUT PROGRAM

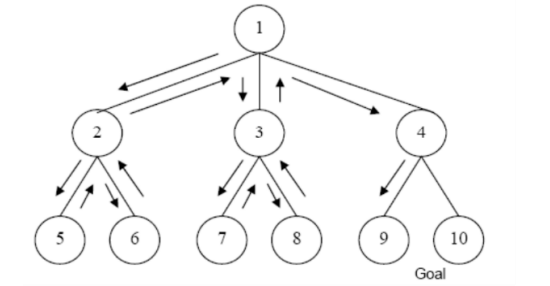


ACCESS FULL CODE ON MY GITHUB

<https://github.com/dermawannnn99/tspmanual.cpp.git>

**PENGERTIAN, PERBEDAAN DFS (DEPTH FIRST SEARCH) DAN BFS (BREADTH FIRST SEARCH)**

DFS singkatan dari *Depth First Search* algoritma pencarian pada sebuah pohon atau tree. Pencarian DFS ini adalah dengan menelusuri satu cabang sebuah pohon sampai ke bawah (menemukan solusi) sebelum melakukan backtracking (Putri and Jarti, 2023).

****

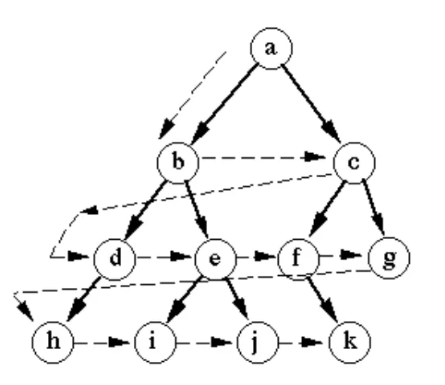
**Gambar 1.** Ilustrasi pencarian dengan DFS

Algoritma *Depth First Search* (DFS) Pencarian dilakukan pada satu *node* dalam setiap level dari yang paling kiri dan dilanjutkan pada node sebelah kanan. Jika solusi ditemukan maka tidak diperlukan proses backtracking yaitu penelusuran balik untuk mendapatkan jalur yang diinginkan. Pada metode DFS pemakaian memori tidak banyak karena hanya *node-node* pada lintasan yang aktif saja yang disimpan. Selain itu, jika solusi yang dicari berada pada level yang dalam dan paling kiri, maka DFS akan menemukannya secara cepat (Putri and Jarti, 2023).

Proses pencarian dilakukan dengan mengunjungi cabang terlebih dahulu hingga tiba di simpul terakhir. Jika tujuan yang diinginkan belum tercapai maka pencarian dilanjutkan ke cabang sebelumnya, turun ke bawah jika memang masih ada cabangnya. Begitu seterusnya hingga diperoleh tujuan (goal). Operasi semacam ini dikenal dengan sebutan backtracking (Muhardono, 2023).

*Breadth First Search* adalah algoritma pencarian secara menyeluruh dengan cara mengunjungi setiap simpul secara preorder. Algoritma ini memerlukan sebuah antrian untuk menyimpan simpul yang telah dikunjungi. Simpul diperlukan sebagai acuan untuk menentukan solusi yang tepat. Dalam algoritma BFS, simpul anak yang telah dikunjungi disimpan dalam suatu antrian. Antrian ini digunakan untuk mengacu simpul-simpul yang bertetangga dengan yang akan dikunjungi kemudian sesuai urutan antrian (Chandra, Ramaningtyas and Hakim, 2021).

BFS juga dapat diartikan suatu metode yang melakukan pencarian secara melebar yang mengunjungi simpul secara preorder yaitu mengunjungi suatu simpul kemudian mengunjungi semua simpul yang bertetangga dengan simpul tersebut dahulu. Selanjutnya, simpul yang belum dikunjungi dan bertetangga dengan simpul-simpul yang tadi dikunjungi, demikian seterusnya (Prasetiyo, 2018).

****

**Gambar 2.** Ilustrasi pencarian dengan BFS

Untuk perbedaan dari kedua algortima berikut adalah. BFS bermula dari suatu akar dan akan mengunjungi simpul tujuan dengan urutan bergantung pada jarak mereka dari akar. Tentu akibatnya, simpul terdekat akan dikunjungi terlebih dahulu.

Sedangkan, DFS lebih memilih untuk mengunjungi langsung, simpul yang belum ditemukan, sehingga pohon pencarian (search tree) hasil traversal akan lebih dalam, dibandingkan dengan BFS, yang akan lebih seimbang. DFS mengandung tiga lintasan Hamiltonian, satu di setiap komponen – dan sebaliknya, pohon BFS, lebih jauh, memiliki simpul 3 derajat yang merefleksikan keseimbangan (Inggiantowi, 2018).

**REFERENSI**

Chandra, D., Ramaningtyas, A.T. and Hakim, L. (2021) ‘Penerapan Breadth First Search Untuk Mengelola Keuangaan Dengan Menentukan Karakteristik Investasi Individu’, *Jurnal Teknik Informatika UNIKA Santo Thomas*, 06, pp. 395–402. Available at: https://doi.org/10.54367/jtiust.v6i2.1561.

Inggiantowi, H. (2008) ‘Perbandingan Algoritma Penelusuran Depth First Search dan Breadth First Search pada Graf serta Aplikasinya’, pp. 1–9.

Muhardono, A. (2023) ‘Penerapan Algoritma Breadth First Search dan Depth First Search pada Game Angka’, *Jurnal Minfo Polgan*, 12(1), pp. 171–182. Available at: https://doi.org/10.33395/jmp.v12i1.12340.

Prasetiyo, M. (2018) ‘Ipi355205’, 1(2), pp. 161–167.

Putri, W.L. and Jarti, N. (2023) ‘Algoritma General and Test Menggunakan Metode Depth First Search Dalam Penentuan Jalur Rute Terpendek’, *Brahmana: Jurnal Penerapan Kecerdasan …*, 4(2), pp. 154–163.