Metode Pencarian (Searching)

KECERDASAN BUATAN

Minggu Ke-3

Dr. Sayuti Rahman, ST, M.Kom

Masalah, Ruang Keadaan, dan Pencarian

Untuk membangun sebuah sistem yang digunakan untuk menyelesaikan suatu problem, dibutuhkan tiga komponen utama yang saling berkaitan:

1

Definisi Ruang Masalah

Menspesifikasikan kondisi awal (**initial state**) dan solusi yang diharapkan (**goal state**) secara jelas dan formal.

2

Aturan Produksi

Mendefinisikan aturan yang digunakan untuk mengubah satu **state** ke **state** lainnya dalam ruang masalah.

3

Metode Pencarian

Memilih metode pencarian yang tepat sehingga solusi terbaik dapat ditemukan dengan **usaha yang minimal**.

Metode Pencarian

Secara umum, metode pencarian dalam kecerdasan buatan dibagi menjadi dua kategori besar berdasarkan ketersediaan informasi tentang ruang masalah:

🔍 Pencarian Buta (Blind / Un-informed Search)

Pencarian yang dilakukan **tanpa informasi tambahan** tentang seberapa dekat suatu state dengan goal. Algoritma hanya mengandalkan struktur ruang masalah.

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Uniform Cost Search

🧭 Pencarian Heuristik (Informed Search)

Pencarian yang menggunakan **informasi tambahan (heuristik)** untuk memandu proses pencarian ke arah solusi yang lebih menjanjikan, sehingga lebih efisien.

- Best-First Search
- A* Search
- Hill Climbing

Ruang Masalah

Masalah utama dalam membangun sistem berbasis AI adalah bagaimana **mengkonversikan situasi yang diberikan** ke dalam situasi lain yang diinginkan menggunakan sekumpulan informasi dan aturan tertentu.

State (Keadaan)

Representasi kondisi sistem pada suatu titik waktu tertentu. Setiap state menggambarkan konfigurasi lengkap dari masalah yang sedang diselesaikan.

State Space (Ruang Keadaan)

Himpunan semua state yang dapat dicapai dari initial state melalui penerapan aturan produksi. Pencarian dilakukan di dalam ruang ini.

Goal State (Kondisi Tujuan)

State yang menjadi target pencarian. Solusi adalah jalur dari initial state menuju goal state melalui serangkaian aksi yang valid.

Contoh Permasalahan: Ember Air

Deskripsi Masalah: Diberikan dua ember air berkapasitas **8 liter** dan **6 liter**. Kita dapat mengisi satu ember dari ember lainnya dan proses penakaran hanya menggunakan kedua ember tersebut. Asumsikan tidak ada air yang hilang dalam proses penakaran.

Pertanyaan: Bagaimana cara mengisikan **tepat 4 liter** air ke dalam ember berkapasitas 8 liter?

Ember A

Kapasitas: **8 Liter**Target akhir: **4 Liter**

Ember B

Kapasitas: **6 Liter**Digunakan sebagai alat ukur

Batasan

Tidak ada gelas ukur. Tidak ada air yang boleh terbuang selama proses penakaran berlangsung.

Langkah Penyelesaian Masalah (Bagian 1)

Langkah pertama adalah mendefinisikan **problem space** — yaitu seluruh aksi yang mungkin dapat mengubah kondisi pada kedua ember. Aksi-aksi ini kemudian direpresentasikan dalam bentuk *rule* atau *tree-diagram*.

1

Isi Ember 8 Liter

Mengisi penuh ember berkapasitas 8 liter dari sumber air.

2

Isi Ember 6 Liter

Mengisi penuh ember berkapasitas 6 liter dari sumber air.

3

Kosongkan Ember 8 Liter

Membuang seluruh isi air dari ember 8 liter.

4

Kosongkan Ember 6 Liter

Membuang seluruh isi air dari ember 6 liter.

Langkah Penyelesaian Masalah (Bagian 2)

Selain operasi mengisi dan mengosongkan, terdapat aksi **pemindahan air antar-ember** yang menjadi kunci dalam proses penakaran. Aksi-aksi ini memungkinkan perpindahan state yang lebih beragam:

1

Pindah 8 → 6

Isikan seluruh air dalam ember 8 liter ke ember 6 liter (jika kapasitas mencukupi).

2

Pindah 6 → 8

Isikan seluruh air dalam ember 6 liter ke ember 8 liter (jika kapasitas mencukupi).

3

Penuhi 8 dari 6

Tuangkan air dari ember 6 liter ke ember 8 liter hingga ember 8 liter penuh.

4

Penuhi 6 dari 8

Tuangkan air dari ember 8 liter ke ember 6 liter hingga ember 6 liter penuh.

Langkah Penyelesaian Masalah (Bagian 3)

Setelah aksi-aksi didefinisikan, langkah berikutnya adalah **menentukan urutan aksi** yang menghasilkan solusi. Berikut adalah salah satu jalur solusi yang dapat ditemukan melalui proses pencarian, mulai dari state **(0,0)** hingga mencapai goal state **(4,0)**:

Jalur solusi: **(0,0) →b (0,6) →f (6,0) →b (6,6) →g (8,4) →c (0,4) →f (4,0)**

Keterangan: setiap huruf (b, f, g, c) pada panah menunjukkan aturan produksi yang diterapkan pada setiap transisi state.

Permasalahan Jirigen Air

Deskripsi Masalah: Dua buah jirigen air **tanpa skala ukuran** dengan kapasitas masing-masing **4 galon** dan **3 galon**. Tersedia sebuah kran dengan pasokan air tidak terbatas.

Pertanyaan: Bagaimana langkah-langkah untuk mendapatkan **tepat 2 galon** air di dalam jirigen berkapasitas 3 galon?

Jirigen 4 Galon

Kapasitas maksimum 4 galon. Tidak memiliki tanda ukur.

Jirigen 3 Galon

Kapasitas maksimum 3 galon. Target: berisi tepat **2 galon**.

Sumber Air

Kran dengan aliran air tidak terbatas untuk mengisi salah satu jirigen kapan saja.

Solusi: Definisi Ruang Masalah

Langkah pertama penyelesaian adalah mendefinisikan ruang masalah secara formal menggunakan representasi state. State direpresentasikan sebagai pasangan **(x, y)**:

Variabel x

Jumlah air (dalam galon) yang ada di dalam **jirigen 4 galon**. Nilai x berkisar antara 0 hingga 4.

Variabel y

Jumlah air (dalam galon) yang ada di dalam **jirigen 3 galon**. Nilai y berkisar antara 0 hingga 3.

Initial State

Kondisi awal: kedua jirigen kosong → **(0, 0)**

Goal State

Kondisi tujuan: jirigen 3 galon berisi tepat 2 galon → **(n, 2)**, di mana n adalah nilai x sembarang.

Solusi: Aturan Produksi

Langkah kedua adalah mendefinisikan **aturan produksi** — yaitu operasi-operasi yang mengubah suatu state ke state lainnya. Setiap aturan memiliki **kondisi prasyarat** yang harus dipenuhi sebelum aturan tersebut dapat diaplikasikan.

Isi Jirigen

Mengisi jirigen 4 galon atau 3 galon hingga penuh dari sumber kran air.

Kosongkan Jirigen

Membuang atau mengosongkan seluruh atau sebagian isi salah satu jirigen.

Tuang Antar-Jirigen

Menuangkan air dari satu jirigen ke jirigen lainnya dengan kondisi kapasitas tertentu.

Total terdapat **12 aturan produksi** yang mungkin berlaku untuk masalah ini, masing-masing dengan kondisi dan efek yang berbeda pada pasangan state (x, y).

Daftar Aturan Produksi

Berikut adalah 12 aturan produksi lengkap untuk permasalahan jirigen air:

No.	Kondisi (if)	Transformasi State	Keterangan
1	x < 4	(x,y) → (4,y)	Isi penuh jirigen 4 galon
2	y < 3	(x,y) → (x,3)	Isi penuh jirigen 3 galon
3	x > 0	(x,y) → (x-d,y)	Buang sebagian air dari jirigen 4 galon
4	y > 0	(x,y) → (x,y-d)	Buang sebagian air dari jirigen 3 galon
5	x > 0	(x,y) → (0,y)	Kosongkan jirigen 4 galon
6	y > 0	(x,y) → (x,0)	Kosongkan jirigen 3 galon
7	x+y ≥ 4, y > 0	(x,y) → (4, y-(4-x))	Tuang dari jirigen 3 ke 4 hingga jirigen 4 penuh
8	x+y ≥ 3, x > 0	(x,y) → (x-(3-y), 3)	Tuang dari jirigen 4 ke 3 hingga jirigen 3 penuh
9	x+y ≤ 4, y > 0	(x,y) → (x+y, 0)	Tuang seluruh isi jirigen 3 ke jirigen 4
10	x+y ≤ 3, x > 0	(x,y) → (0, x+y)	Tuang seluruh isi jirigen 4 ke jirigen 3
11	—	(0,2) → (2,0)	Tuang 2 galon dari jirigen 3 ke jirigen 4
12	—	(2,y) → (0,y)	Buang 2 galon dalam jirigen 4 galon

Solusi: Jalur Pencarian Terpilih

Dengan menerapkan metode pencarian yang tepat, berikut adalah salah satu jalur solusi yang ditemukan — mulai dari **initial state (0,0)** hingga mencapai **goal state (2, 2)**:

Jirigen 4 Galon (x)	Jirigen 3 Galon (y)	Aturan Produksi yang Diaplikasikan
0	0	— (Initial State)
0	3	Aturan 2: Isi penuh jirigen 3 galon
3	0	Aturan 9: Tuang seluruh jirigen 3 ke jirigen 4
3	3	Aturan 2: Isi penuh jirigen 3 galon
4	2	Aturan 7: Tuang dari jirigen 3 ke jirigen 4 hingga penuh

Goal State Tercapai! Pada state terakhir **(4, 2)**, jirigen 3 galon berisi tepat **2 galon** air sesuai yang diminta. Ini merupakan salah satu dari beberapa jalur solusi yang valid.

Kriteria Performasi Metode Pencarian

Untuk mengukur kualitas dan efisiensi suatu metode pencarian, terdapat **empat kriteria** standar yang umumnya digunakan dalam kajian kecerdasan buatan:

Completeness

Apakah algoritma **selalu menemukan solusi** jika solusi tersebut memang ada?

Time Complexity

Berapa lama waktu yang dibutuhkan untuk menemukan solusi? Diukur dalam jumlah node yang dieksplorasi.

Space Complexity

Seberapa besar **memori** yang dibutuhkan selama proses pencarian berlangsung?

Optimality

Apakah solusi yang ditemukan merupakan **solusi terbaik** (misalnya jalur terpendek atau berbiaya terendah)?

BLIND / UN-INFORMED SEARCH

Kategori: Pencarian Buta

Pencarian buta (*blind search*) bekerja tanpa informasi tambahan mengenai seberapa dekat suatu state dengan tujuan. Berikut adalah algoritma-algoritma yang termasuk dalam kategori ini:

Breadth-First Search (BFS)

Menjelajahi semua node per level sebelum melanjutkan ke level berikutnya.

Uniform Cost Search

Mengembangkan node dengan biaya jalur terendah terlebih dahulu.

Depth-First Search (DFS)

Menjelajahi sedalam mungkin sebelum melakukan backtrack.

Depth Limited Search

DFS dengan batas kedalaman tertentu untuk mencegah loop tak terbatas.

Iterative Deepening Search

Menggabungkan keunggulan BFS dan DFS dengan meningkatkan batas kedalaman secara bertahap.

Bi-directional Search

Mencari dari kedua arah — dari initial state dan goal state — secara bersamaan.

Breadth-First Search (BFS)

Breadth-First Search (BFS) melakukan proses pencarian pada semua node yang berada pada **level atau hirarki yang sama** terlebih dahulu, sebelum melanjutkan ke level berikutnya. Pendekatan ini menjamin bahwa solusi dengan jalur terpendek akan selalu ditemukan terlebih dahulu.

Cara Kerja BFS

BFS menggunakan struktur data **queue (antrian)** — First In, First Out (FIFO). Setiap node yang dikunjungi akan menghasilkan anak-anaknya yang langsung dimasukkan ke akhir antrian.

Urutan Eksplorasi

Pada pohon pencarian, BFS akan mengunjungi node secara berurutan:

- Level 0: **A**
- Level 1: **B, C, D**
- Level 2: **E, F, G, H, I, J, K, L, M**

Diagram Pohon BFS

Diagram pohon di atas mengilustrasikan urutan eksplorasi BFS. Garis putus-putus menunjukkan kelompok node yang dijelajahi bersama pada **level yang sama**. BFS bergerak dari atas ke bawah, memastikan semua node pada satu level selesai diproses sebelum turun ke level berikutnya.

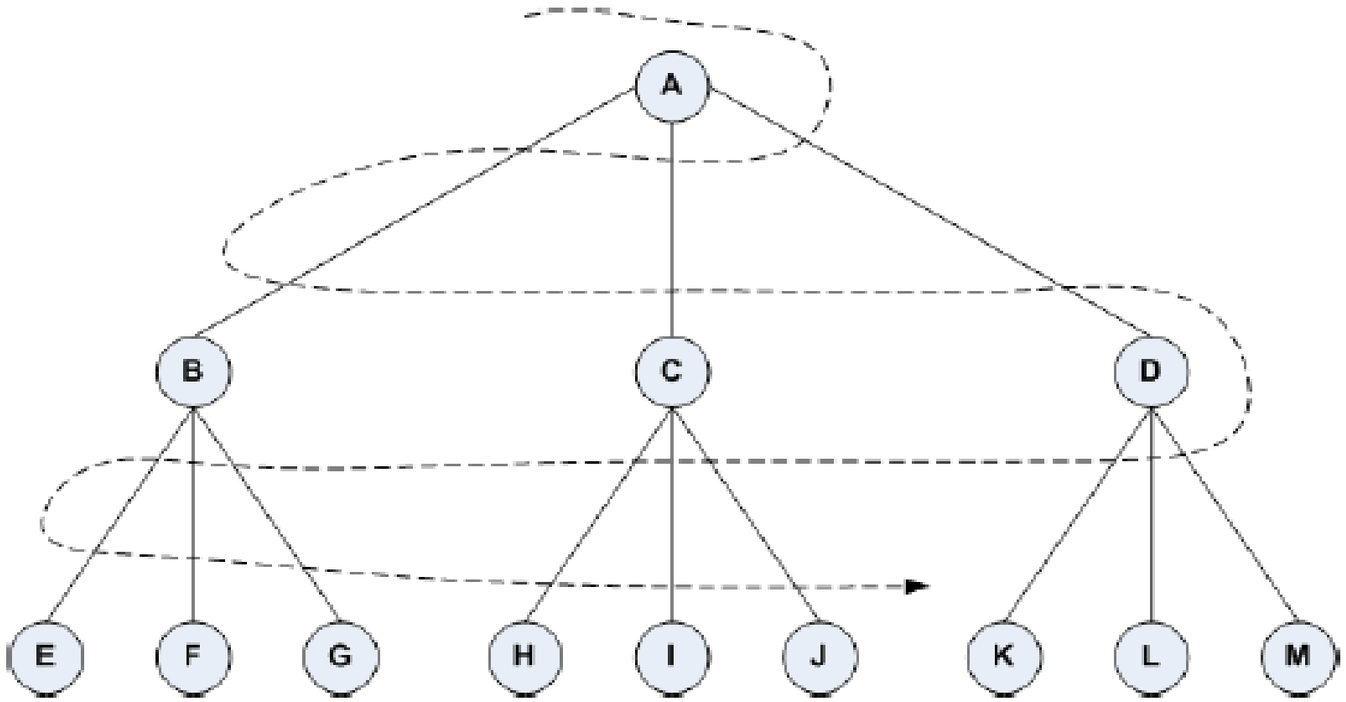


Figure 1.6: Diagram pohon dari BFS.

Algoritma BFS

Berikut adalah pseudocode algoritma **Breadth-First Search** yang menggunakan struktur antrian (*queue*) bernama `NODE-LIST`:

1 **Inisialisasi**

Buat variabel `NODE-LIST` (queue) dan set ke **initial state** sebagai elemen pertama.

2 **Iterasi Utama**

Ulangi proses berikut selama **goal state belum ditemukan** atau `NODE-LIST` tidak kosong: Ambil elemen pertama dari `NODE-LIST`, sebut sebagai **E**. Jika `NODE-LIST` kosong, hentikan proses — solusi tidak ditemukan.

3 **Ekspansi Node**

Untuk setiap aturan yang cocok dengan state **E**: terapkan aturan untuk menghasilkan state baru. Jika state baru adalah **goal state**, kembalikan solusi. Jika bukan, tambahkan state baru ke **akhir** `NODE-LIST`.

Karakteristik BFS

BFS memiliki sejumlah sifat khas yang perlu dipahami sebelum memilihnya sebagai metode pencarian untuk suatu masalah:

✅ Complete

Jika solusi ada, BFS **pasti akan menemukannya**. Tidak ada solusi yang terlewat selama ruang pencarian berhingga.

✅ Optimal (Jalur Terpendek)

BFS selalu menemukan solusi dengan **panjang jalur minimal** — yaitu solusi yang memerlukan jumlah langkah paling sedikit.

✅ Bebas Siklus

BFS tidak akan terjebak dalam **siklus (cycle)** karena setiap node hanya dikunjungi satu kali dalam antrian.

⚠️ Konsumsi Memori Besar

Kelemahan utama BFS: membutuhkan **memori yang sangat besar** karena harus menyimpan semua node pada level aktif sekaligus.

Depth-First Search (DFS)

Depth-First Search (DFS) mengeksplorasi satu cabang secara mendalam terlebih dahulu sebelum mencoba cabang lainnya. Pencarian dilakukan pada suatu simpul dalam setiap level — jika solusi belum ditemukan di level terdalam, pencarian dilanjutkan ke simpul sebelah kanan dan simpul kiri dapat dihapus dari memori.

Strategi Eksplorasi

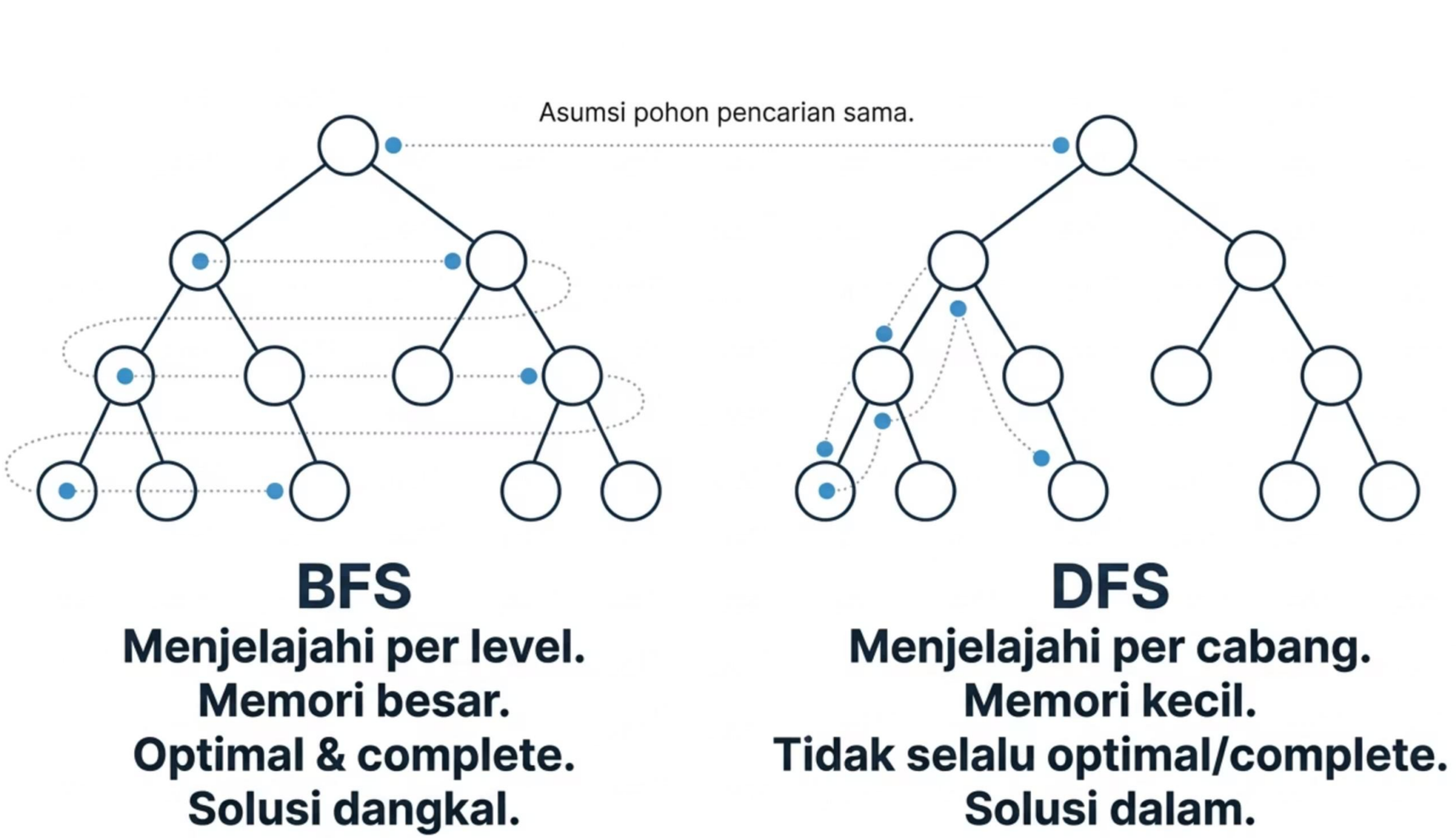
DFS menggunakan struktur data **stack (tumpukan)** — Last In, First Out (LIFO). Algoritma selalu mengembangkan node terdalam yang belum dieksplorasi.

Backtracking

Jika pada level paling dalam tidak ditemukan solusi, DFS melakukan **backtrack** ke level sebelumnya dan mencoba cabang yang belum dieksplorasi.

Perbandingan BFS vs DFS

Kedua algoritma pencarian buta ini memiliki karakteristik yang sangat berbeda. Memahami perbedaan ini krusial untuk memilih algoritma yang tepat sesuai kebutuhan masalah:



Pilihan antara BFS dan DFS bergantung pada karakteristik ruang masalah: kedalaman solusi yang diperkirakan, keterbatasan memori, dan apakah optimalitas diperlukan.

Algoritma DFS

Algoritma DFS menggunakan pendekatan **rekursif**. Berbeda dengan BFS yang iteratif dengan queue, DFS secara alami cocok diimplementasikan secara rekursif karena sifat penelusuran mendalam yang dimilikinya:

1

Kondisi Basis (Base Case)

Jika **initial state adalah goal state**, hentikan pencarian dan kembalikan **sukses**. Ini adalah kondisi terminasi yang paling diharapkan.

2

Ekspansi Rekursif

Jika bukan goal state, lakukan iterasi: bangkitkan **successor E** dari initial state. Jika tidak ada successor lagi, kembalikan **gagal**. Panggil DFS secara rekursif dengan E sebagai initial state baru.

3

Propagasi Hasil

Jika pemanggilan rekursif mengembalikan **sukses**, propagasikan sukses ke atas. Jika gagal, lanjutkan loop ke successor berikutnya hingga semua cabang dieksplorasi.

Keuntungan DFS

Meskipun memiliki beberapa keterbatasan, DFS menawarkan keunggulan signifikan terutama dalam hal efisiensi penggunaan sumber daya komputasi:

Hemat Memori

DFS hanya menyimpan node-node yang berada pada **jalur aktif** dari root ke node yang sedang dieksplorasi. Berbeda jauh dengan BFS yang harus menyimpan seluruh level sekaligus. Kebutuhan memori DFS bersifat **linier** terhadap kedalaman pohon.

Cepat untuk Solusi Dalam

Jika solusi yang dicari berada pada **level yang dalam** dan posisinya paling kiri dalam pohon pencarian, maka DFS dapat menemukannya dengan sangat cepat — jauh lebih cepat dibandingkan BFS yang harus menjelajahi semua node di atas level tersebut.

Kelemahan DFS

DFS memiliki dua kelemahan fundamental yang membuatnya tidak selalu menjadi pilihan terbaik, terutama untuk masalah dengan ruang pencarian yang dalam atau memiliki banyak solusi:

⚠️ Tidak Complete

Jika pohon pencarian memiliki **level yang sangat dalam atau tak terbatas**, DFS tidak memberikan jaminan untuk menemukan solusi. Algoritma dapat terjebak menjelajahi cabang yang sangat panjang tanpa pernah menemukan goal state, bahkan ketika solusi sebenarnya ada di tempat lain.

⚠️ Tidak Optimal

Jika terdapat **lebih dari satu solusi** yang ekivalen tetapi berada pada level kedalaman yang berbeda, DFS tidak menjamin untuk menemukan solusi yang paling baik. DFS akan mengembalikan solusi pertama yang ditemukan — yang belum tentu merupakan solusi dengan jalur terpendek atau berbiaya minimum.

Kesimpulan: Gunakan DFS ketika memori terbatas dan solusi diperkirakan berada di level yang dalam. Gunakan BFS ketika optimalitas dan completeness diprioritaskan.