TEKNIK INFORMATIKA
FAKULTAS TEKNIK UNIVERSITAS MATARAM

Search

Ramaditia D – rama@unram.ac.id

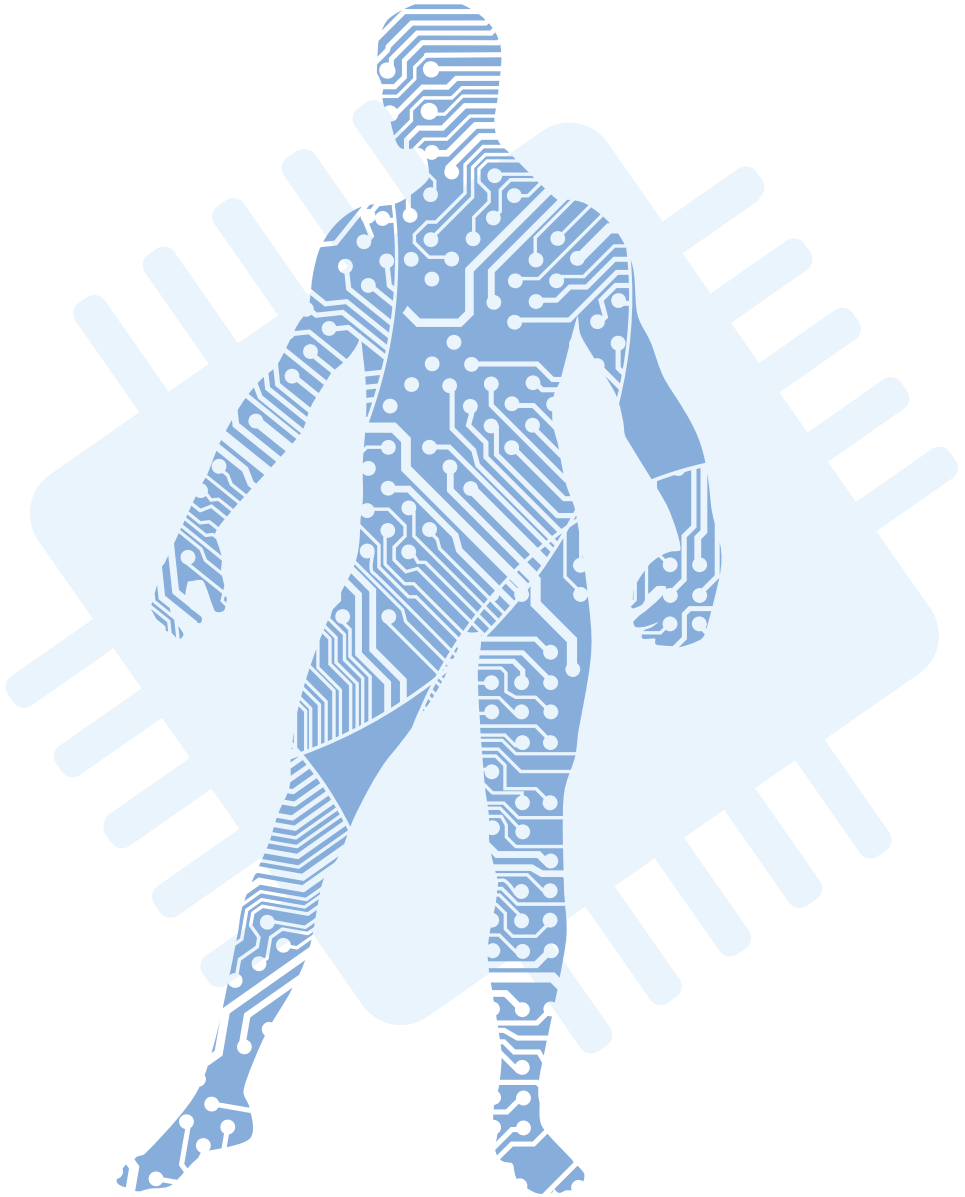
v1.1 update 06-09-20

Outline

01 **Problem**

02 **State Representation**

03 **Uninformed Search**

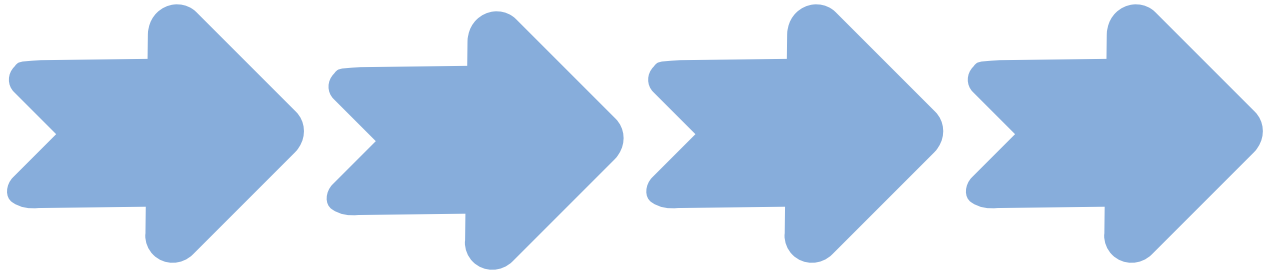


Problem

Problem-Solving Agent

- **Goal-based agent** ➠ mempertimbangkan tindakan-tindakan yang akan datang dan hasil yang ingin dicapai
- **Agent problem solving** ➠ Menemukan rangkaian tindakan (sequence action) untuk mencapai tujuannya
- **Algoritma Uninformed** ➠ Tidak ada informasi untuk problem, hanya deskripsi pada masalah tersebut

Pertimbangan Membangun Sistem AI



01 Mendefinikasikan masalah dengan tepat

02 Menganalisis masalah serta mencari beberapa Teknik penyelesaian

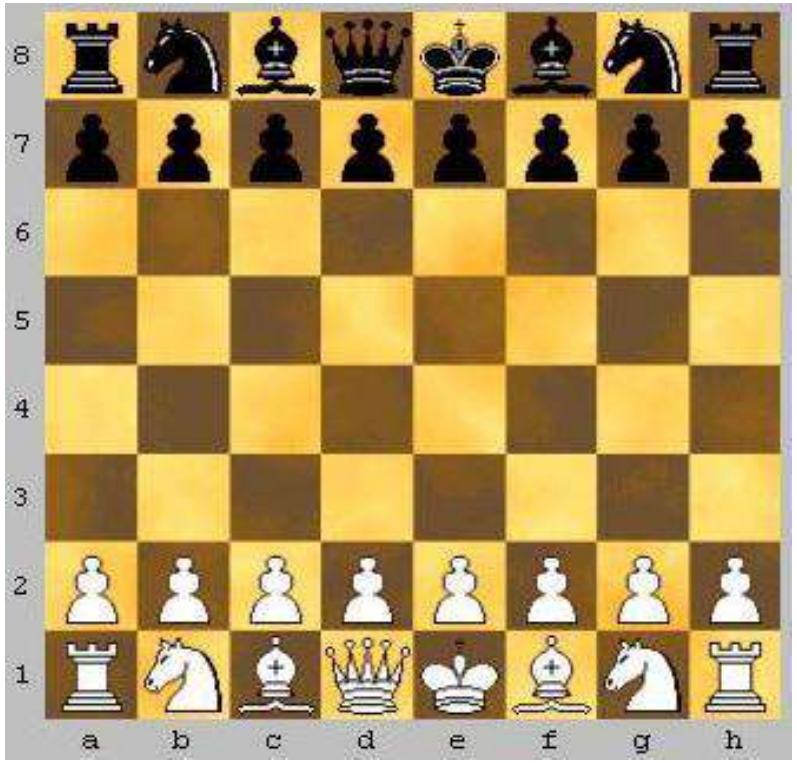
03 Merepresentasikan pengetahuan untuk penyelesaian masalah

04 Memilih Teknik Penyelesaian Terbaik

Contoh Masalah 1

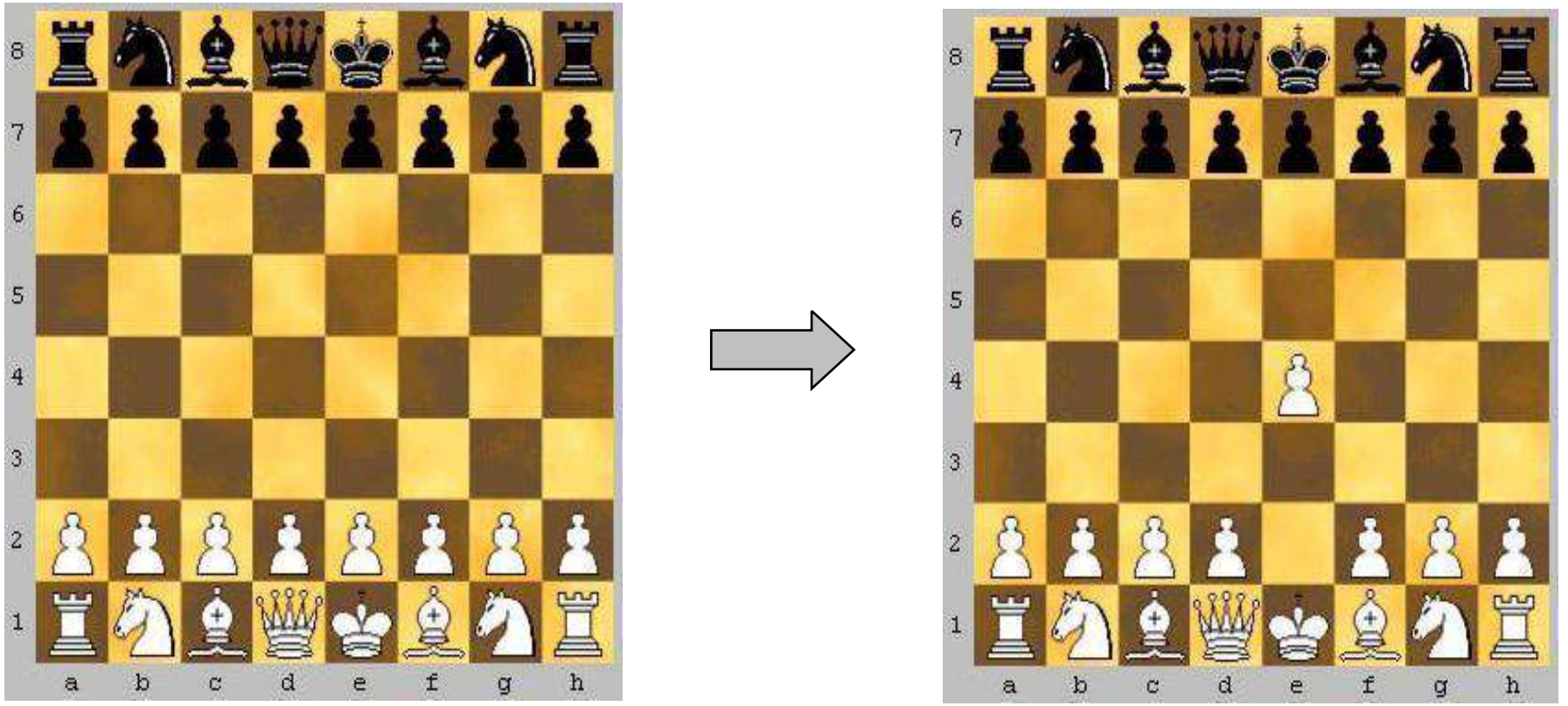
Misalkan permasalahan yang dihadapi adalah “**Permainan Catur**”

1. Posisi Awal Permainan Catur



Masalah Permainan Catur

2. Aturan-aturan untuk melakukan gerakan secara legal.



Contoh Implementasi Aturan

IF

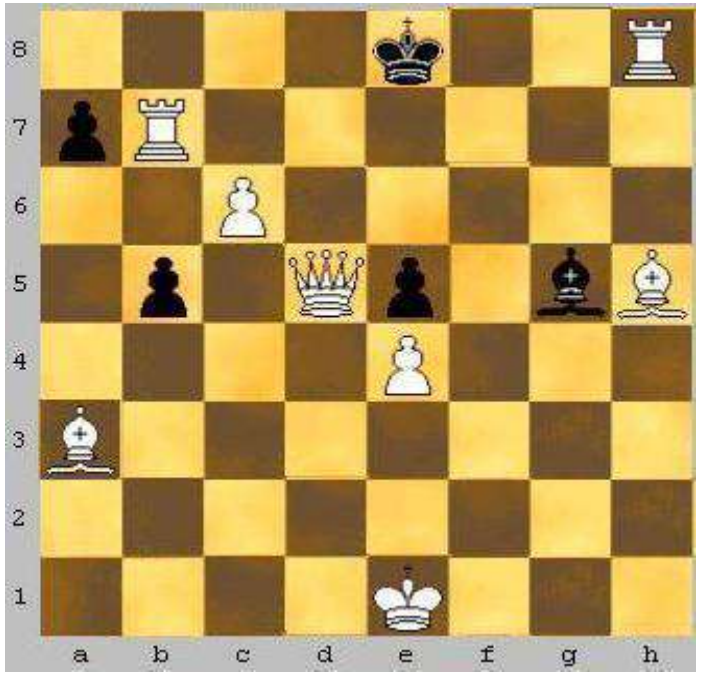
Bidak putih pada Kotak(e,2),
And Kotak(E,3) Kosong,
And Kotak(E,4) Kosong

Then

Gerakkan bidak dari (E,2) ke (E,4

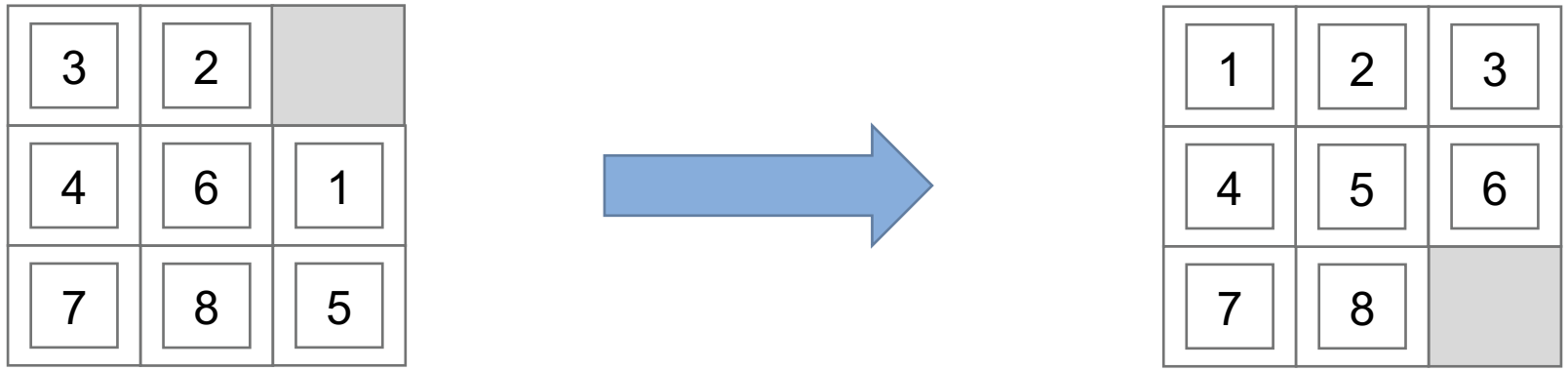
Permasalahan Permainan Catur

- Tujuan yang ingin dicapai (goal) adalah posisi pada papat catur yang menunjukkan kemenangan seseorang terhadap lawannya.
- Kemenangan ini ditandai dengan posisi “Raja” yang sudah tidak dapat bergerak lagi.



Contoh Masalah 2

Permasalahan dalam permainan “**8-Puzzle**”

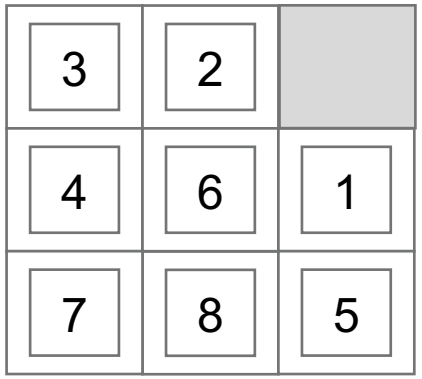


Sembarang Posisi Awal

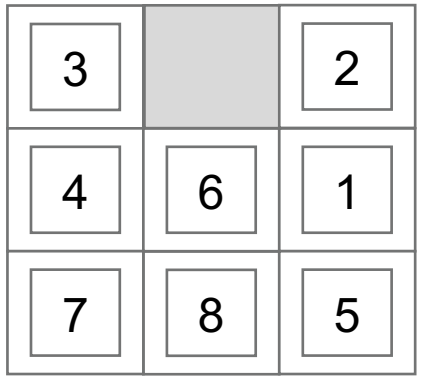
Keadaan Akhir

Representasi Ruang Keadaan

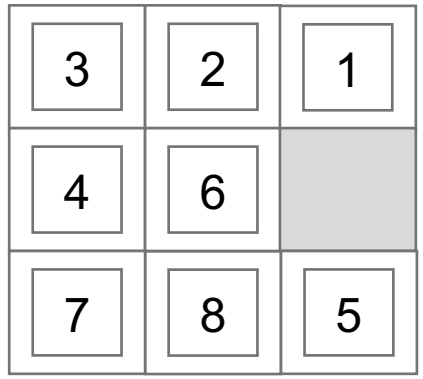
State (Keadaan)



Sebuah State

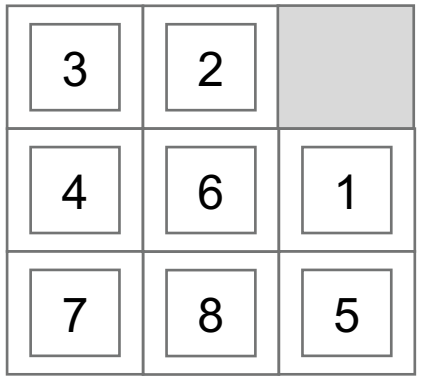


Sebuah state

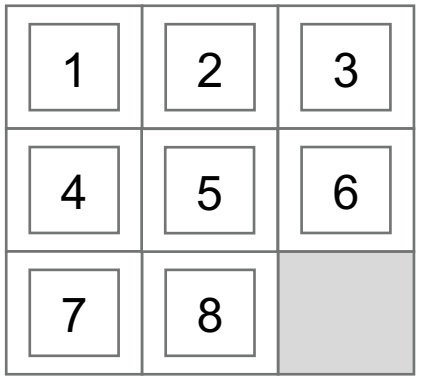


Sebuah state

Initial dan Goal State



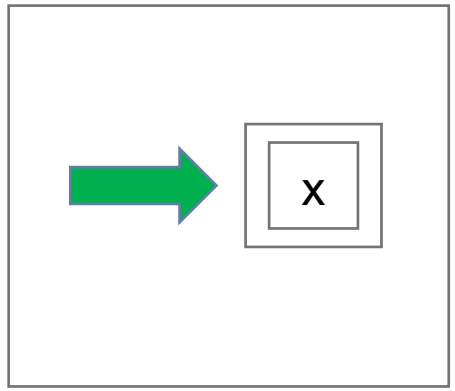
Initial State



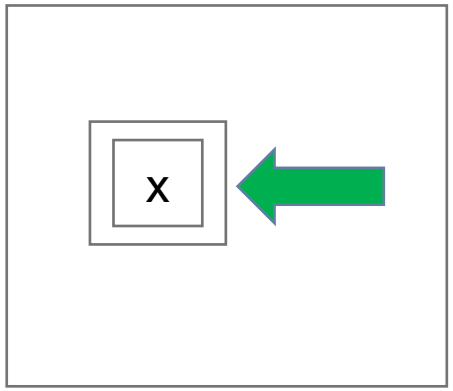
Goal State

Actions (Rule)

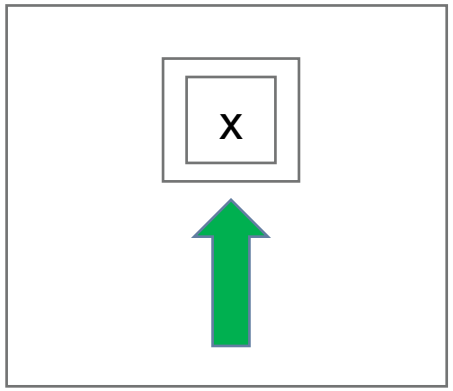
- Action (atau aturan penyelesaian masalah) adalah pilihan tindakan yang dapat dilakukan pada sebuat state.



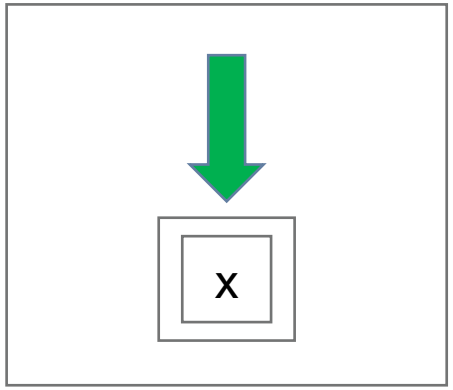
Action 1



Action 2

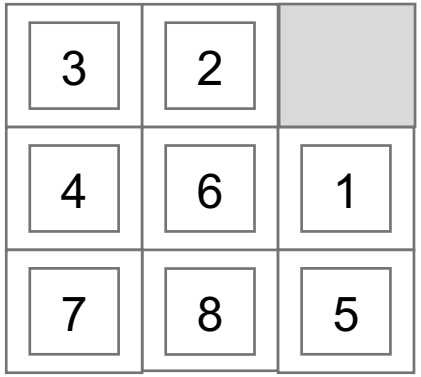


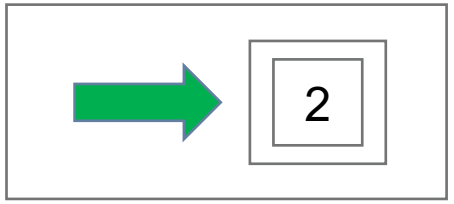
Action 3



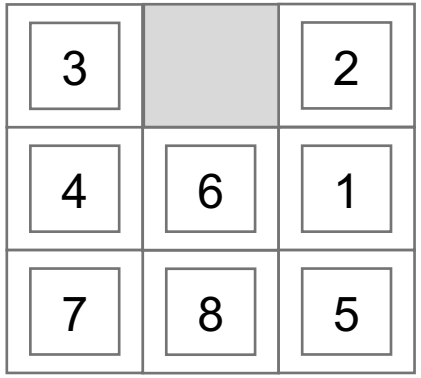
Action 4

Contoh:



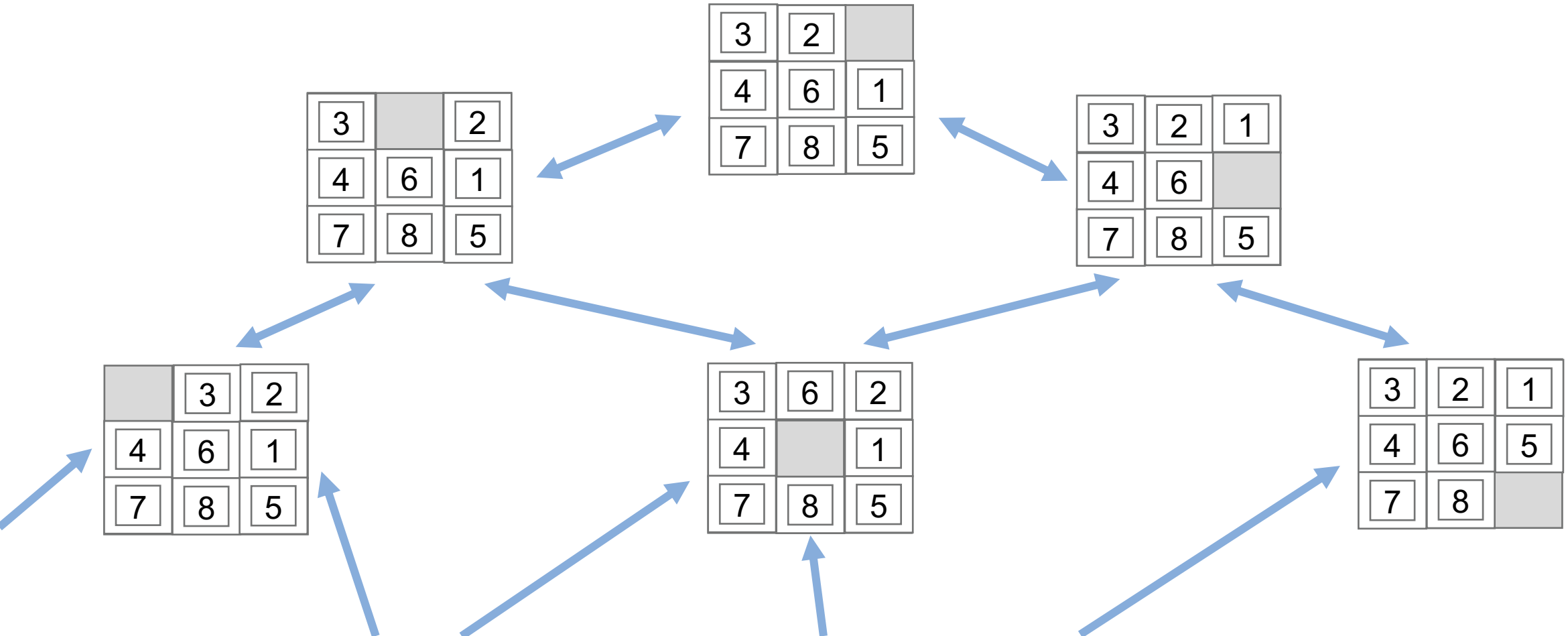


Action 1



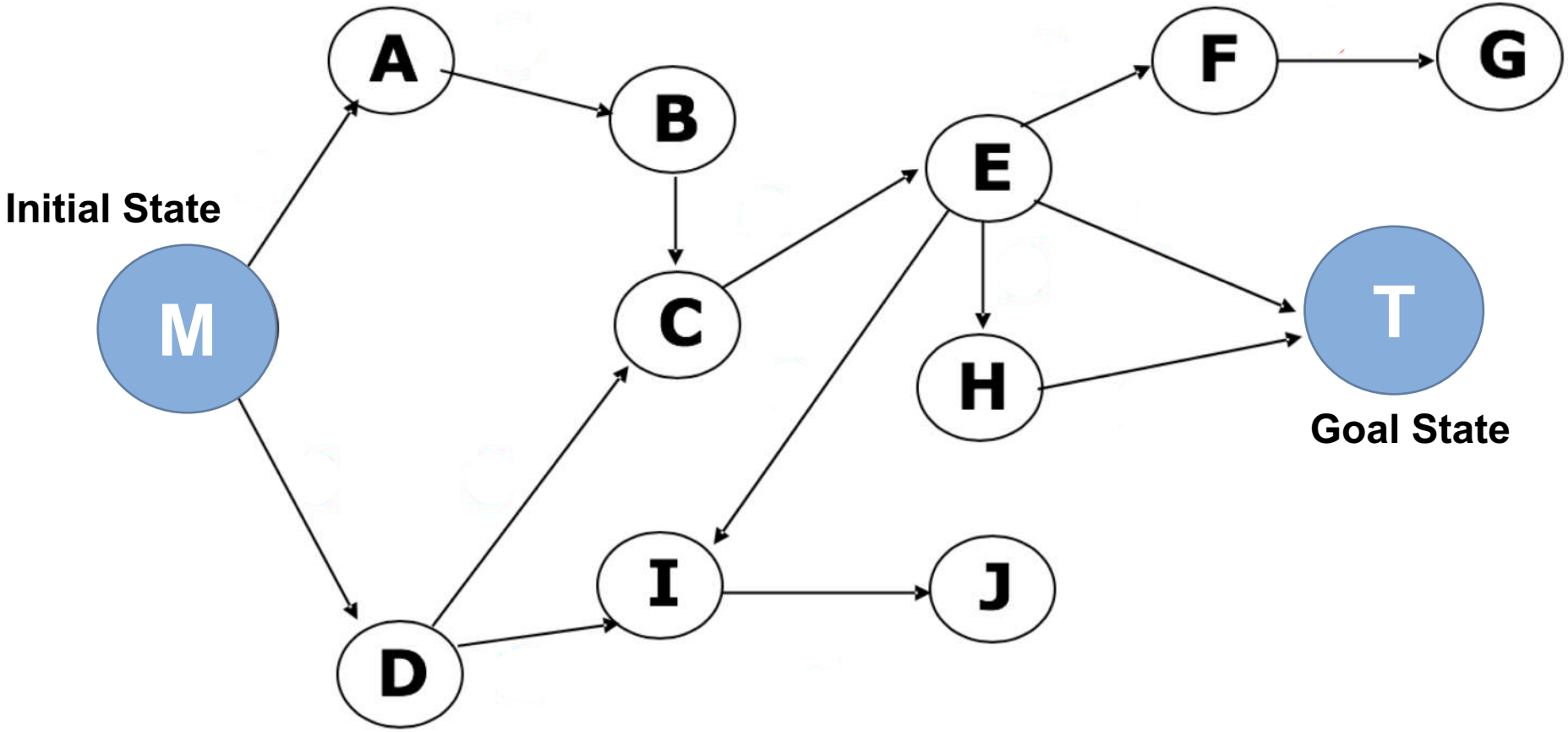
State Space (Ruang Keadaan)

- Himpunan semua state yang dapat dijangkau dari status awal dengan urutuan aturan apapun



Representasi Ruang Keadaan

1. Graph Keadaan

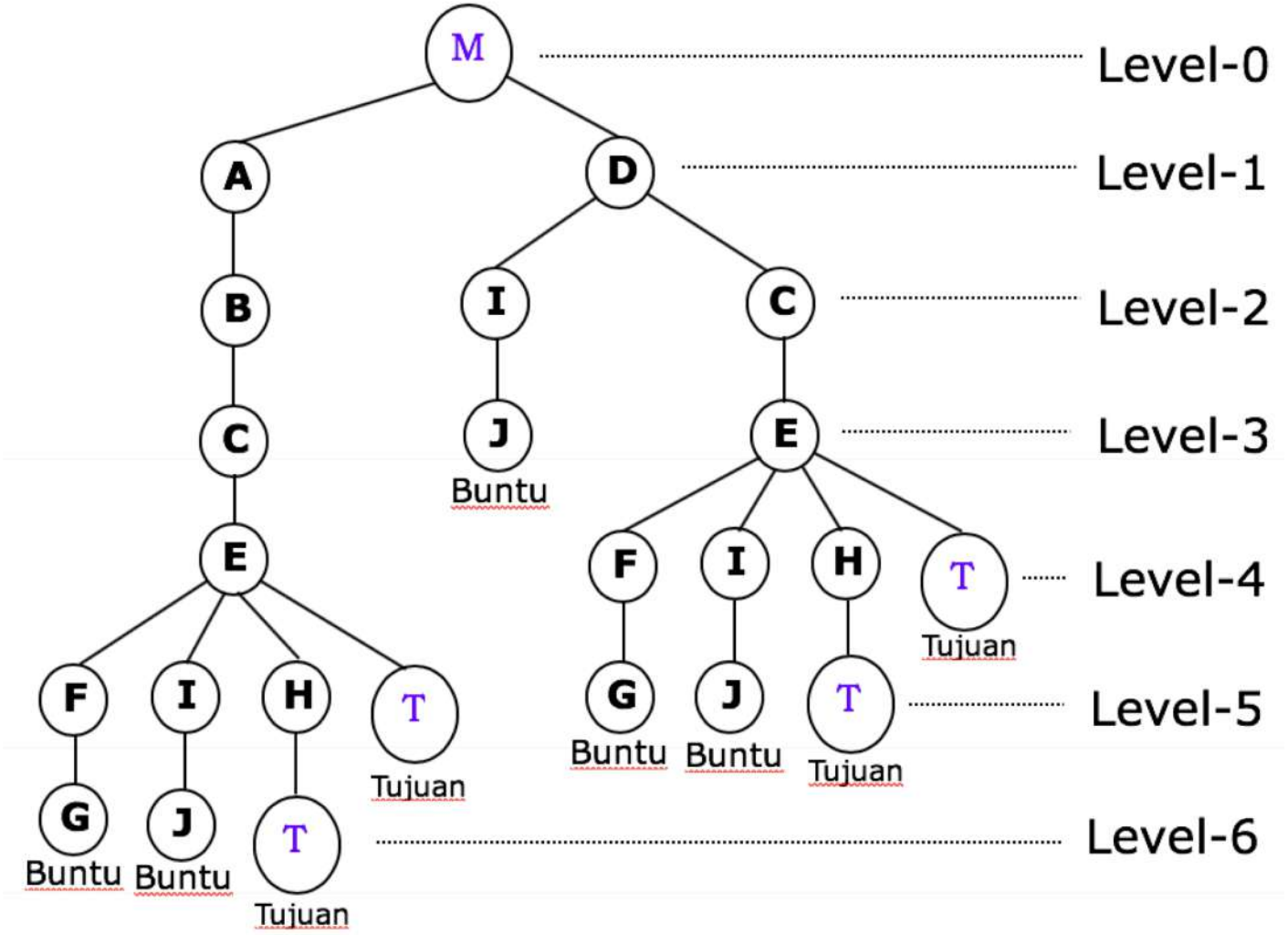


Solusi: Lintasan pada Graph Keadaan

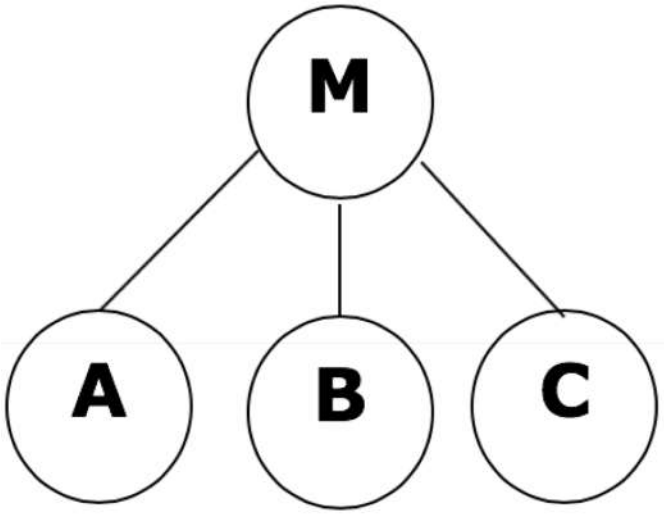
- Lintasan dari M ke T:
  - M-A-B-C-E-T
  - M-A-B-C-E-H-T
  - M-D-C-E-T
  - M-D-C-E-H-T

- Lintasan yang menemui jalan buntu (tidak sampai ke T)
  - M-A-B-C-E-F-G
  - M-A-B-C-E-I-J
  - M-D-C-E-F-G
  - M-D-C-E-I-J
  - M-D-I-J

2. Pohon Pelacakan

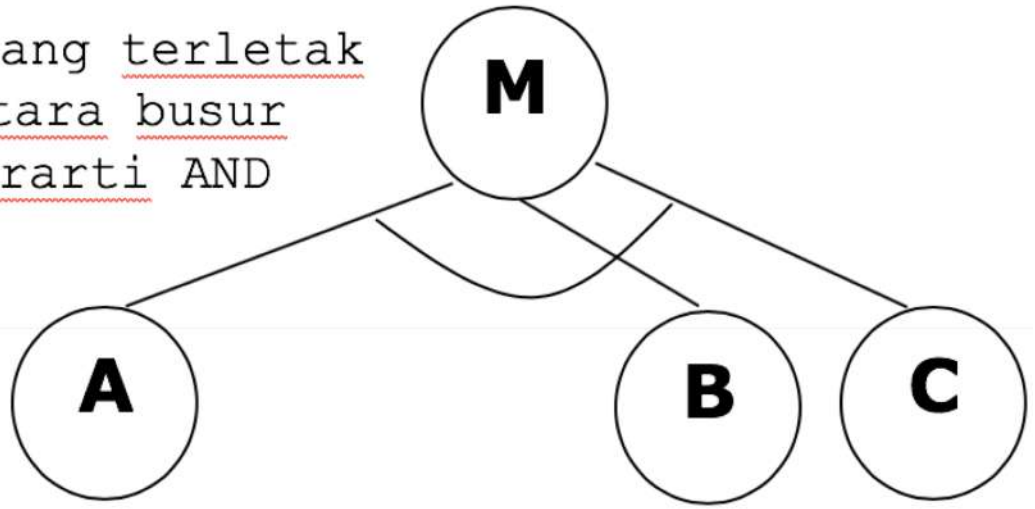


3. Pohon AND / OR



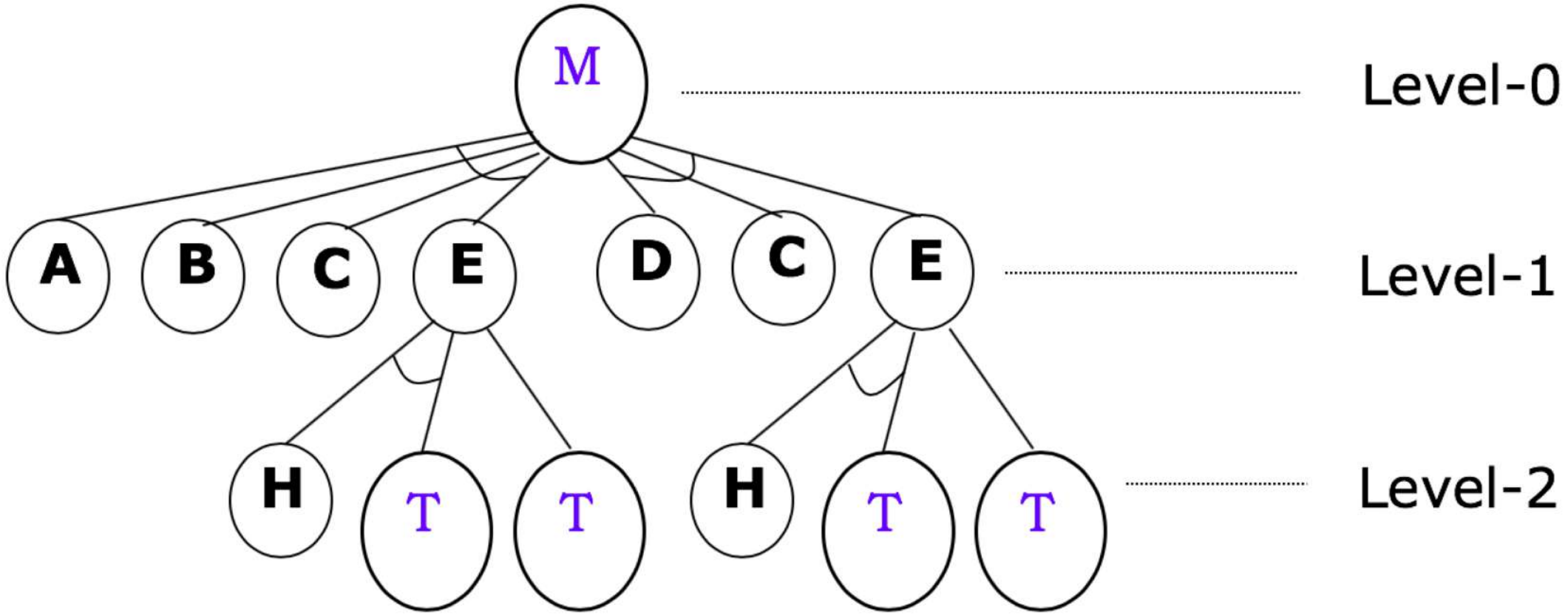
(a)

arc yang terletak antara busur berarti AND



(b)

Contoh lain Pohon AND



Definisi Penyelesaian Masalah

- Mendefinisikan suatu **Ruang Keadaan**, yaitu suatu ruang yang berisi semua keadaaan yang mungkin.
- Menetapkan satu atau lebih **Initial State (keadaan awal)**
- Menetapkan satu atau lebih **Goal State (keadaan tujuan)**
- Menetapkan kumpulan **actions (aturan)**

Contoh Masalah: The Water Jug Problem

3 liter gallon

4 liter gallon

There's

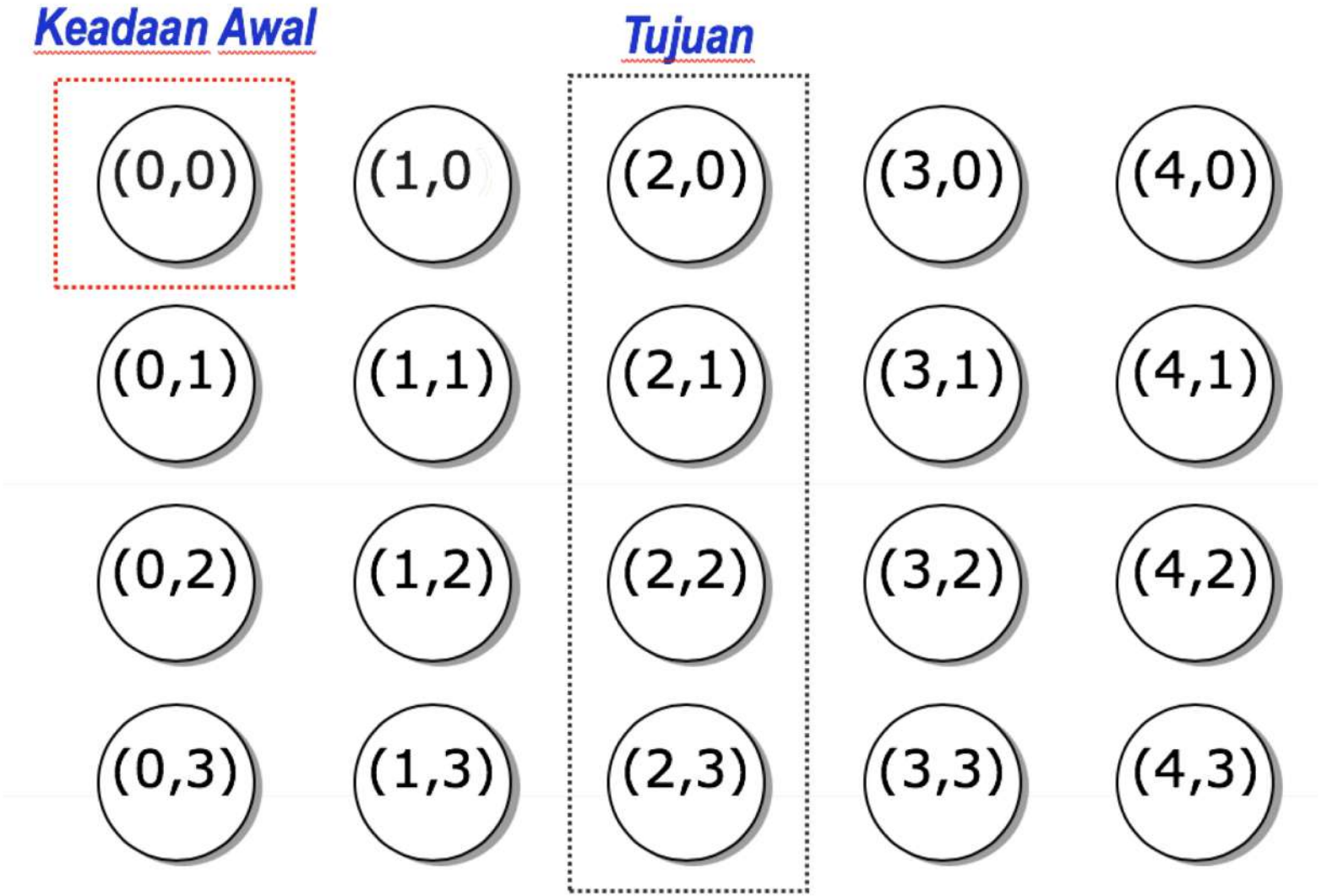
- A **4-gallon** (Gallon A) and a **3-gallon** (Gallon B) water container
- An infinite water fountain

- Fill **exactly 2 liter** of water inside the **4-gallon (Gallon A)** container

Penyelesaian

- Identifikasi ruang keadaan
  - Permasalahan ini dapat direpresentasikan dengan 2 bilangan integer, yaitu x dan y.
    - **x** = air yang diisikan pada Galon A
    - **y** = air yang diisikan pada Galon B
  - Ruang keadaan: (x, y) sedemikian sehingga x = {0,1,2,3,4} dan y = {0,1,2,3}
- Keadaan awal dan tujuan
  - Keadaan awal, kedua galon dalam keadaan kosong: (0,0)
  - Tujuan, keadaan dimana pada Galon A berisi tepat 2 liter air: (2,n) untuk sembarang n.

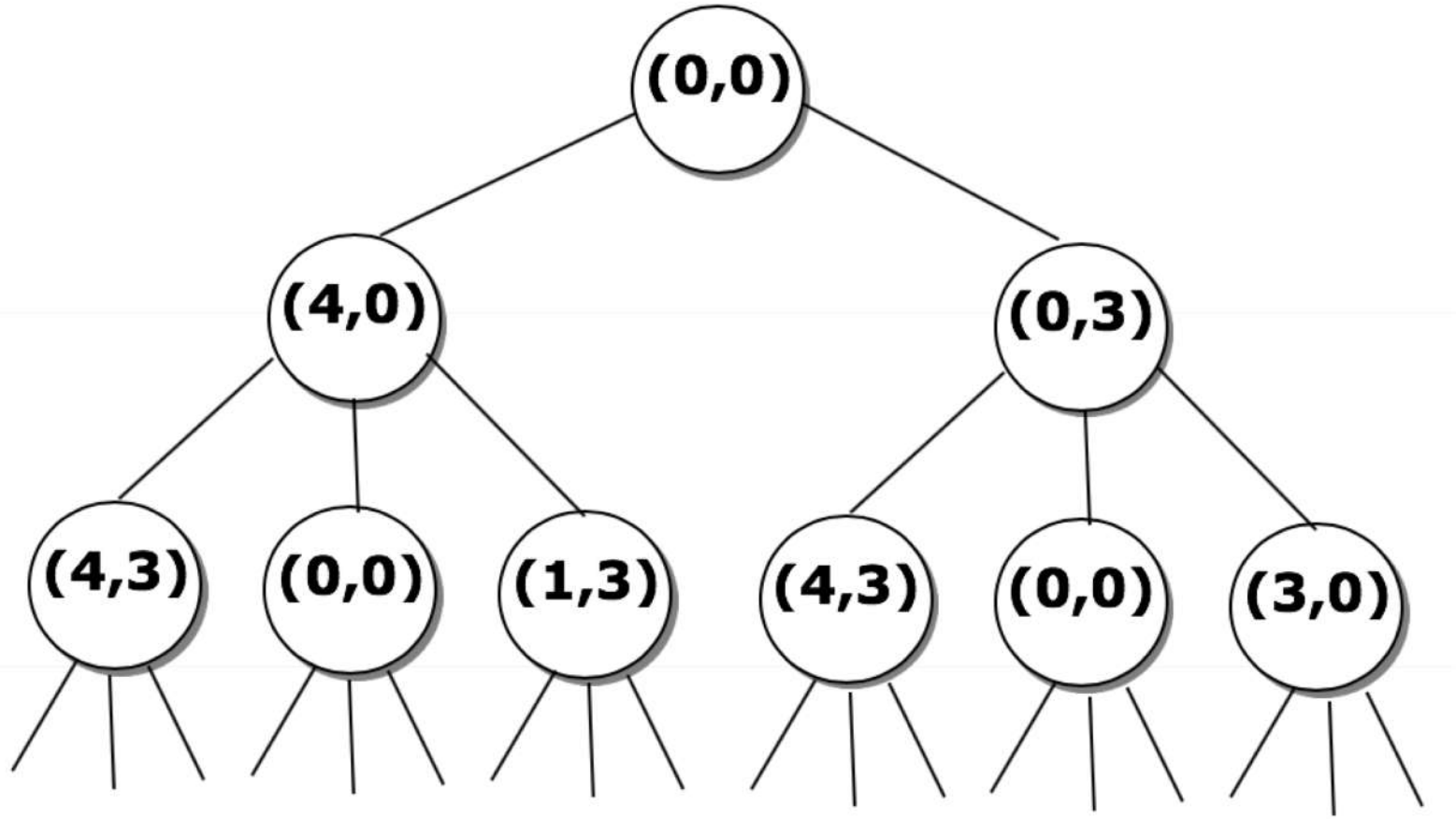
Penyelesaian (2)



Penyelesaian (3)

Rule	Jika	Maka
1	(x,y), x < 4	(4,y). Isi galon A.
2	(x,y), y < 3	(x,3). Isi galon B.
3	(x,y), x > 0	(x-d,y). Tuangkan sebagian air keluar dari galon A.
4	(x,y), y > 0	(x,y-d). Tuangkan sebagian air keluar dari galon B.
5	(x,y), x > 0	(0,y). Kosongkan galon A dengan membuang airnya ke tanah.
6	(x,y), y > 0	(x,0). Kosongkan galon B dengan membuang airnya ke tanah.
7	(x,y), x+y ≤ 4 dan y > 0	(4,y-(4-x)). Tuangkan air dari galon B ke galon A sampai galon A penuh.
8	(x,y), x+y ≤ 3 dan x > 0	(x-(3-y),3). Tuangkan air dari galon A ke galon B sampai galon A penuh.
9	(x,y), x+y ≤ 4 dan y > 0	(x+y,0). Tuangkan seluruh air dari galon B ke galon A
10	(x,y), x+y ≤ 3 dan x > 0	(0,x+y). Tuangkan seluruh air dari galon A ke galon B
11	(0,2)	(2,0). Tuangkan 2 galon air dari galon B ke galon A.
12	(2,y)	(0,y). Kosongkan 2 galon air di galon A dengan membuang airnya ke tanah.

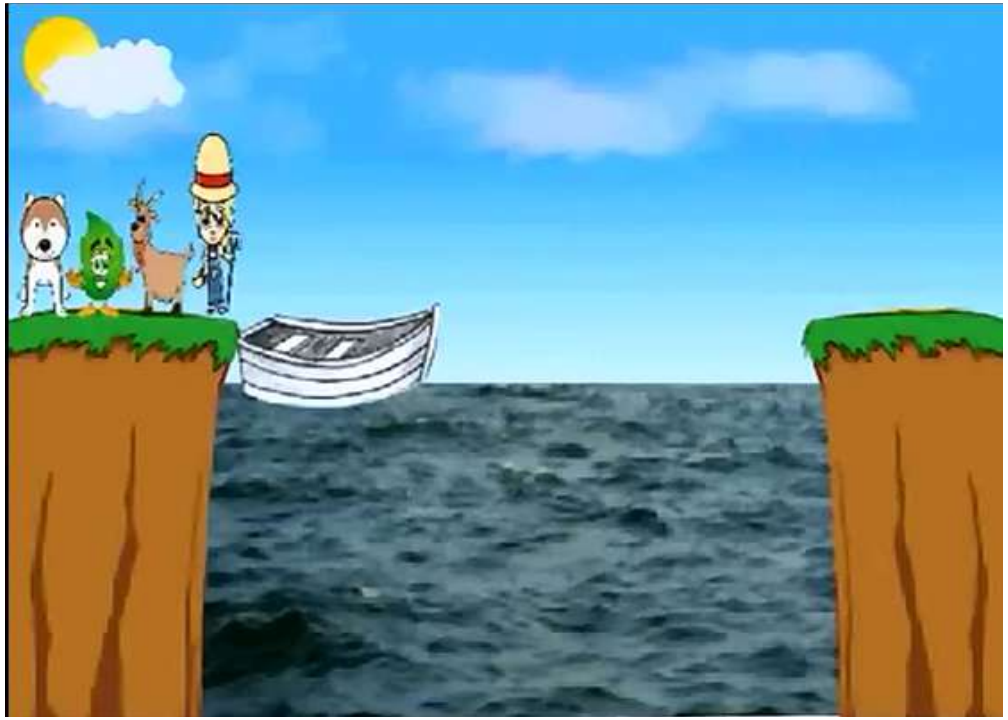
Representasi ruang keadaan



Salah Satu Solusi

Isi Galon A	Isi Galon B	Aturan yang dipakai
0	0	2
0	3	9
3	0	2
3	3	7
4	2	5
0	2	9
2	0	solusi

Contoh Permasalahan Lain



- Seorang petani akan menyeberangkan seekor kambing, seekor serigala, dan sayur-sayuran dengan sebuah boat yang melalui sungai.
- Boat hanya bisa memuat petani dan satu penumpang yang lain (kambing, serigala atau sayur-sayuran).
- Jika ditinggalkan oleh petani tersebut, maka sayur-sayuran akan dimakan oleh kambing, dan kambing akan dimakan oleh serigala.

Penyelesaian

- Identifikasi ruang keadaan
  - Permasalahan ini dapat dilambangkan dengan (JumlahKambing, JumlahSerigala, JumlahSayuran, JumlahBoat).
  - Sebagai contoh: Daerah asal (0,1,1,1) berarti pada daerah asal tidak ada kambing, ada serigala, ada sayuran, dan ada boat.

- Keadaan Awal (Jumlah obyek)
  - Daerah asal: (1,1,1,1)
  - Daerah seberang: (0,0,0,0)

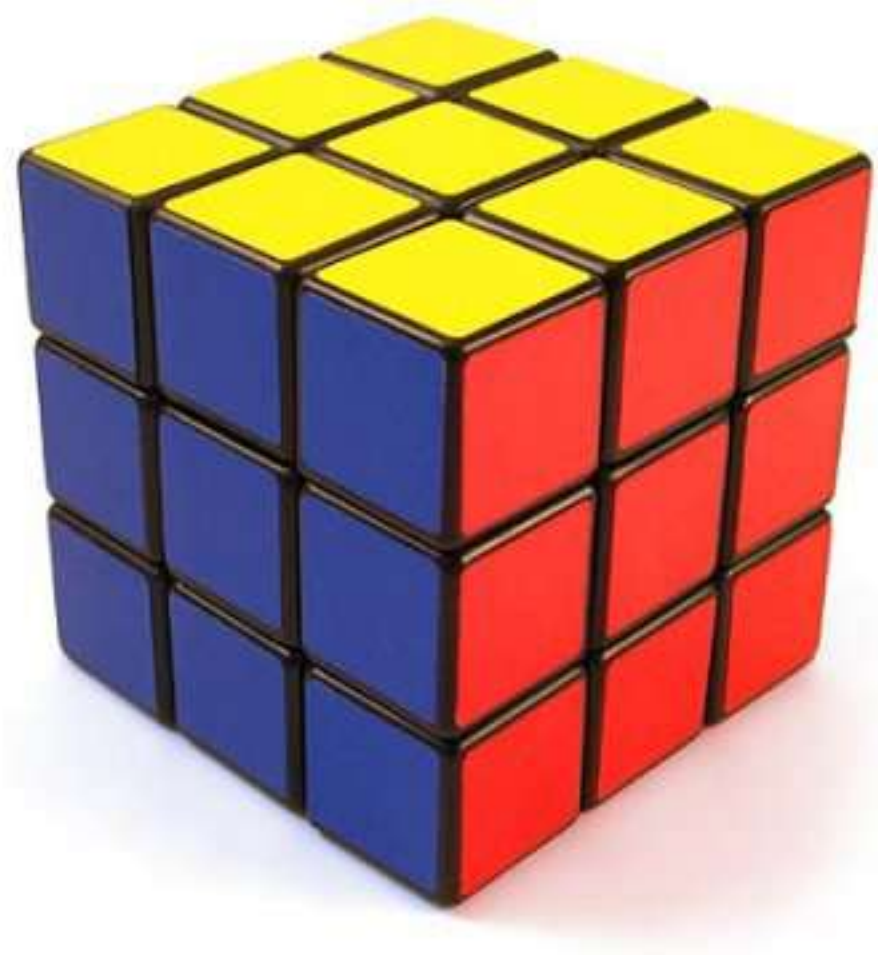
- Tujuan
  - Daerah asal: (0,0,0,0)
  - Daerah seberang: (1,1,1,1)

Penyelesaian (2)

- Aturan-aturan (Actions)

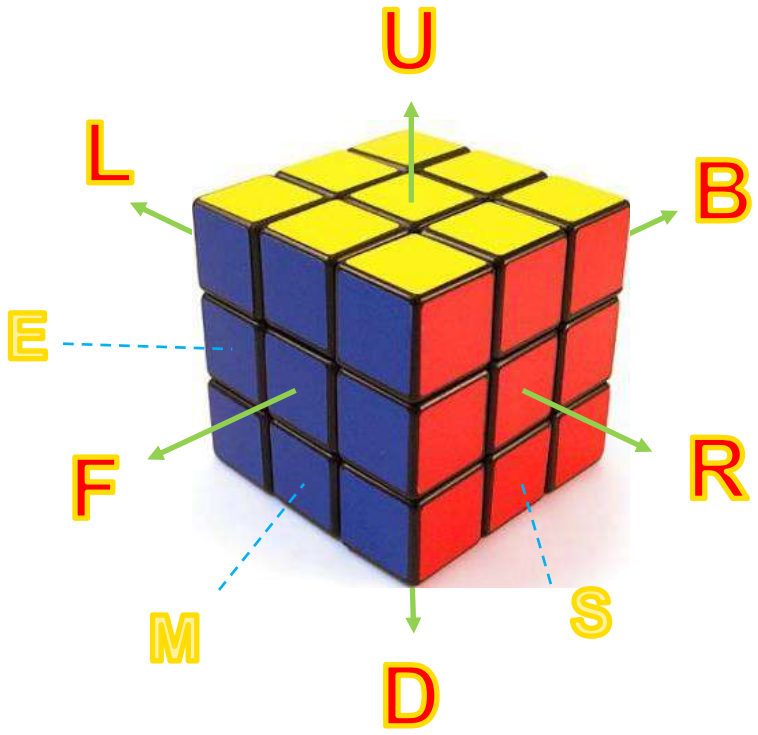
Aturan ke-	Aturan
1	Kambing menyeberang
2	Sayuran menyeberang
3	Serigala menyeberang
4	Kambing kembali
5	Sayuran kembali
6	Serigala kembali
7	Boat kembali

Rubik's Cube Problem



Aturan – 3x3 Rubik's Cube

#	Aturan
1	1L
2	2L
3	3L
4	1M
5	2M
6	3M
7	1R
8	2R
…	
…	
…	
27	3D



Search

Search (Pencarian)

Blind Search /Un-informed Search (Pencarian Buta)

- **Breadth Fist Search /BFS (pencarian melebar pertama)**
- **Depth First Search /DFS (pencarian kedalam pertama)**
- **Depth-Limited Search (DLS)**
- **Uniform Cost Search (UCS)**
- **Iterative-Deepening Search (IDS)**
- **Bi-Directional Search (BDS)**

Heuristic Search (Pencarian Terbimbing)

- Greedy Best First Search
- A*
- Generate And Test
- Hill Climbing
- Simulated Annealing

Perbedaan Mendasar Blind Search vs Heuristic

- Blind Search merupakan pendekatan yang lempang (*straightforward*) untuk memecahkan suatu persoalan

- Pendekatan ini memecahkan persoalan dengan
  - Cukup sederhana,
  - langsung,
  - jelas (*obvious way*).

- *Just do it!* atau *Just Solve it!*

Karakteristik Blind Search

- Perbedaan antara beragam metode pencarian menggunakan metode Blind Search adalah **prioritas** pengecekan dan ekspansi suatu node.

- Karakteristik umum Blind Search adalah mencoba semua kemungkinan yang ada selama solusi belum ditemukan.

- Meskipun ada kalkulasi perkiraan (seperti UCS), prinsip Blind Search adalah tetap mencoba semua kemungkinan **tanpa mengabaikan** satu alternatif pun (jika solusi belum ditemukan).

- Algoritma yang mendasari karakteristik Blind Search adalah **Algoritma Brute Force** atau **Exhaustive Search**

Contoh Persoalan Algoritma Brute Force

1. Mencari Suatu Elemen dalam Larik

Persoalan: Diberikan sebuah senarai (array) yang beranggotakan *n* buah bilangan bulat (a_1, a_2, …, a_n). Carilah elemen dengan nilai x di dalam senarai tersebut. Jika *x* ditemukan, maka keluarannya adalah indeks elemen senarai, jika *x* tidak ditemukan, maka keluarannya adalah -1.

Algoritma *brute force*: bandingkan setiap elemen senarai dengan nilai x yang dicari

2. Penentuan apakah suatu bilangan merupakan bilangan prima

Persoalan: Diberikan sebuah bilangan bulat positif n. Ujilah apakah bilangan tersebut merupakan bilangan prima atau bukan.

Algoritma *brute force*: bagi *n* dengan 2 sampai *n-1*. Jika semuanya tidak habis membagi *n*, maka *n* adalah bilangan prima.

Karakteristik Algoritma Brute Force

1. Algoritma *brute force* umumnya tidak “cerdas”, karena ia membutuhkan jumlah komputasi yang besar dan waktu yang lama dalam penyelesaiannya.

2. Kata **“force”** mengindikasikan **“tenaga”** ketimbang **“otak”**

3. Algoritma *brute force* lebih cocok untuk persoalan yang berukuran kecil. Pertimbangannya: sederhana, implementasinya mudah.

4. Meskipun bukan metode yang efisien, hampir semua persoalan dapat diselesaikan dengan algoritma *brute force.* Bahkan, ada persoalan yang hanya dapat diselesaikan dengan metode *brute force*. Contoh: mencari elemen terbesar di dalam senarai.

5. “*When in doubt, use brute force*” (Ken Thompson, penemu Sistem Operasi UNIX).

Penerapan Brute Force Dalam Sudoku

- **Sudoku** adalah adalah permainan teka-teki (*puzzle*) logik yang berasal dari Jepang. Permainan ini sangat populer di seluruh dunia.

- Contoh sebuah Sudoku:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Aturan Permainan Sudoku

- Kotak-kotak di dalam Sudoku harus diisi dengan angka 1 sampai 9 sedemikian sehingga:
  1. tidak ada angka yang sama (berulang) pada setiap baris;
  2. tidak ada angka yang sama (berulang) pada setiap kolom;
  3. tidak ada angka yang sama (berulang) pada setiap bujursangkar (persegi) yang lebih kecil.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Algoritma Brute Force untuk Sudoku

1. Tempatkan angka “1” pada sel pertama. Periksa apakah penempatan “1” dibolehkan (dengan memeriksa baris, kolom, dan kotak).

2. Jika tidak ada pelanggaran, maju ke sel berikutnya. Tempatkan “1” pada sel tersebut dan periksa apakah ada pelanggaran.

3. Jika pada pemeriksaan ditemukan pelanggaran, yaitu penempatan “1” tidak dibolehkan, maka coba dengan menempatkan “2”.

4. Jika pada proses penempatan ditemukan bahwa tidak satupun dari 9 angka diperbolehkan, maka tinggalkan sel tersebut dalam keadaan kosong, lalu mundur satu langkah ke sel sebelumnya. Nilai di sel tersebut dinaikkan 1.

5. Ulangi sampai 81 buah sel sudah terisi solusi yang benar.

Kelemahan Algoritma Brute Force

1. Metode *brute force* jarang menghasilkan algoritma **yang efisien**.

2. Beberapa algoritma *brute force* **lambat** sehingga tidak dapat diterima.

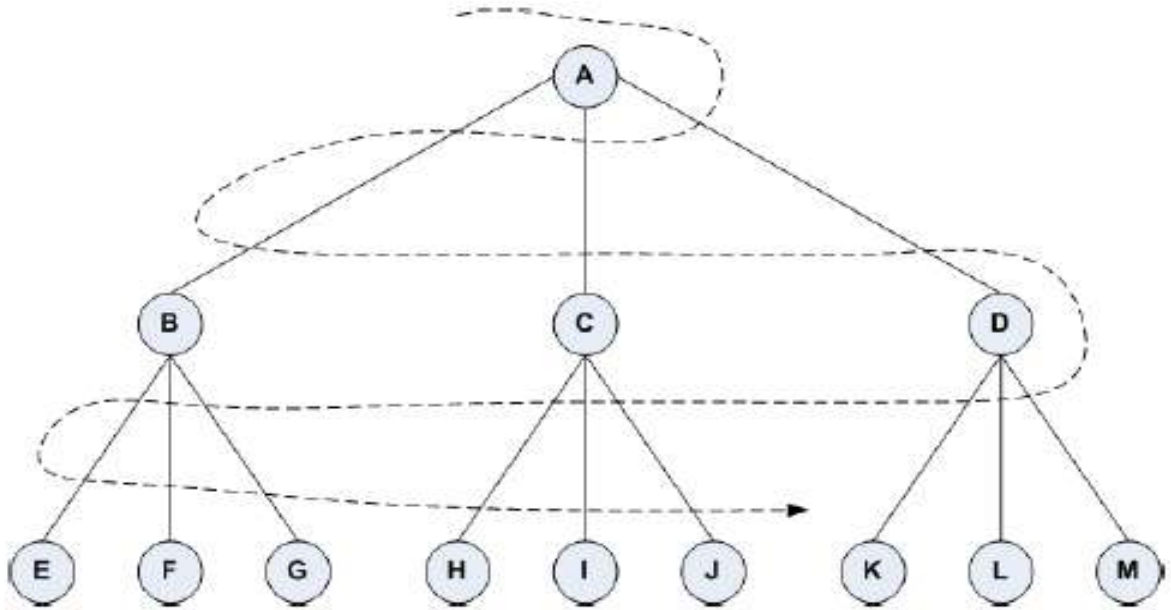
3. Tidak sekontruktif/sekreatif teknik pemecahan masalah lainnya.

Uninformed Search

1. Breadth-First Search (BFS)

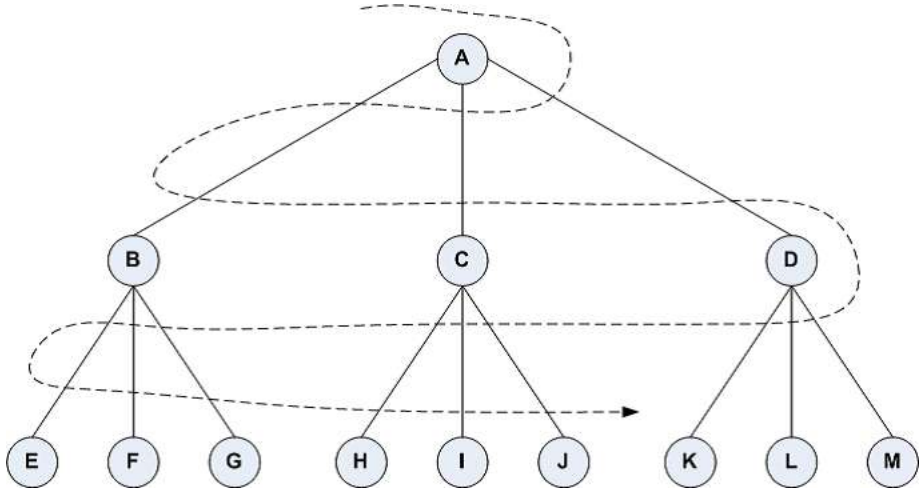
- *Breadth-first search (BFS)* melakukan proses searching pada semua node yang berada pada level atau hierarki yang sama terlebih dahulu sebelum melanjutkan proses searching pada node di level berikutnya.

BFS →
ABCD
EFG
HIJ
KLM



Penerapan BFS dalam Algoritma

- Pendekatan BFS adalah Queue ➔ FIFO
- Misalkan kita hendak mencari lintasan ke node “E”
  - A →
  - B_A, C_A, D_A →
  - C_A, D_A , E_{AB}, F_{AB}, G_{AB} ➔
  - D_A , E_{AB}, F_{AB}, G_{AB} , H_{AC}, I_{AC}, J_{AC} ➔
  - E_{AB}, F_{AB}, G_{AB} , H_{AC}, I_{AC}, J_{AC} , K_{AD}, L_{AD}, M_{AD} ➔
  - E ditemukan ➔ **Lintasan ke E adalah ABE**
  - **Jalur pencarian: ABCDE**



Penerapan BFS dalam Algoritma

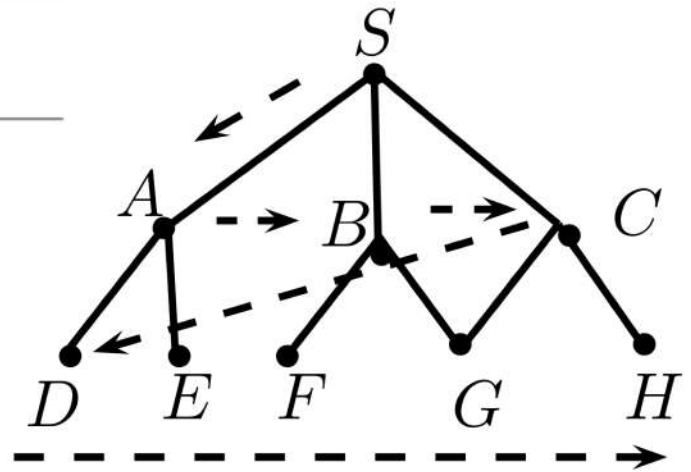
Open-list	Closed-list
$[(S, 0)]$	[]
$[(A, S), (B, S), (C, S)]$	$[(S, 0)]$
$[(B, S), (C, S), (D, A), (E, A)]$	$[(S, 0), (A, S)]$
$[(C, S), (D, A), (E, A), (F, B), (G, B)]$	$[(S, 0), (A, S), (B, S)]$
$[(D, A), (E, A), (F, B), (G, B), (G, C), (H, C)]$	$[(S, 0), (A, S), (B, S), (C, S)]$
$[(E, A), (F, B), (G, B), (G, C), (H, C)]$	$[(S, 0), (A, S), (B, S), (C, S), (D, A)]$
$[(F, B), (G, B), (G, C), (H, C)]$	$[(S, 0), (A, S), (B, S), (C, S), (D, A), (E, A)]$
$[(G, B), (G, C), (H, C)]$	$[(S, 0), (A, S), (B, S), (C, S), (D, A), (E, A), (F, B)]$

Source: Chowdhary, Fundamentals of Artificial Intelligence. 2020

Mencari lintasan dari “S” ke node “G”

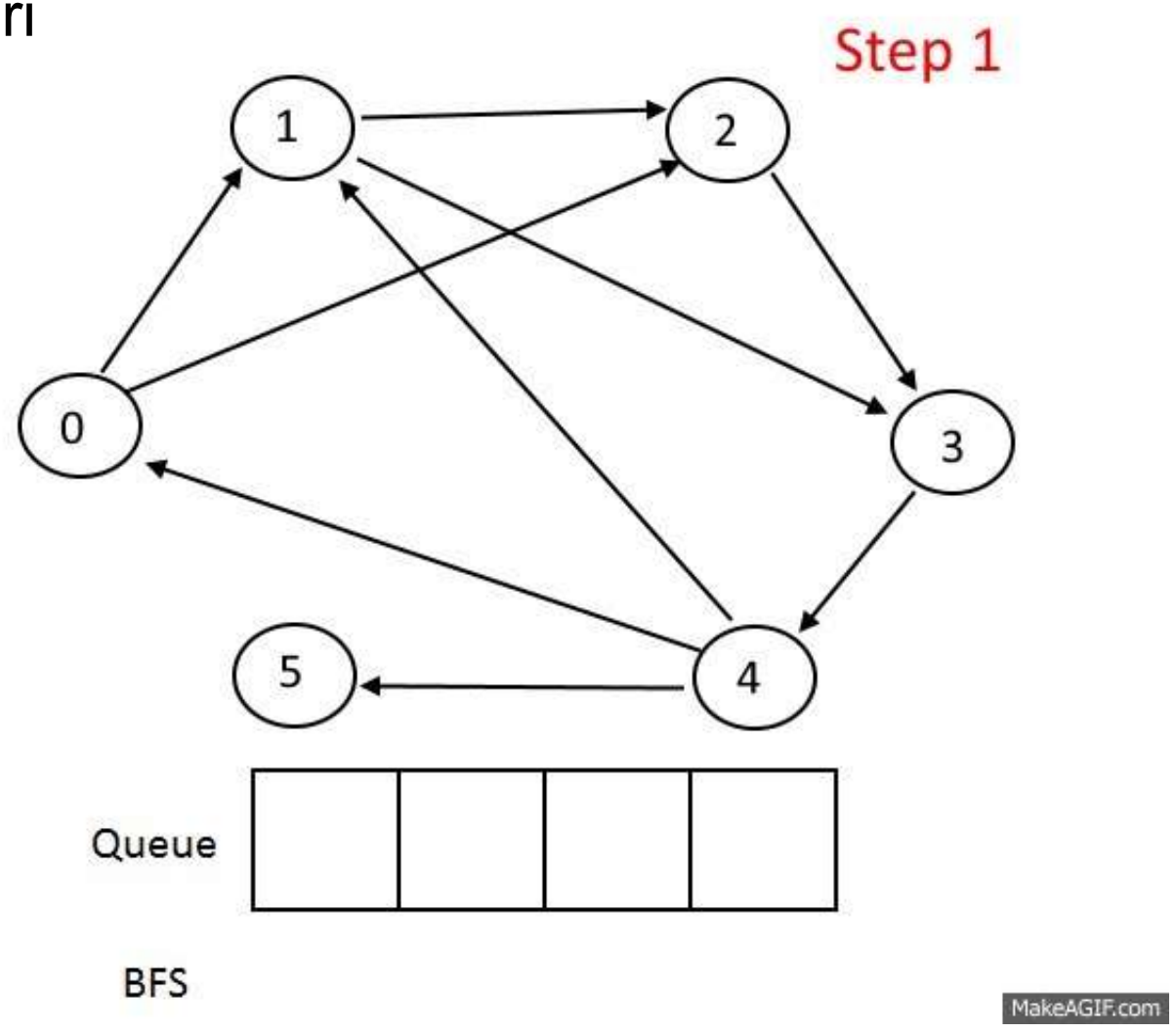
Lintasan ke “G” adalah SBG

Jalur pencarian ke “G” adalah **SABCDEFG**



Contoh Animasi BFS

Lintasan yang dibentuk dari node “0” ke node “5” ?



Karakteristik BFS

❖ **Keuntungan**

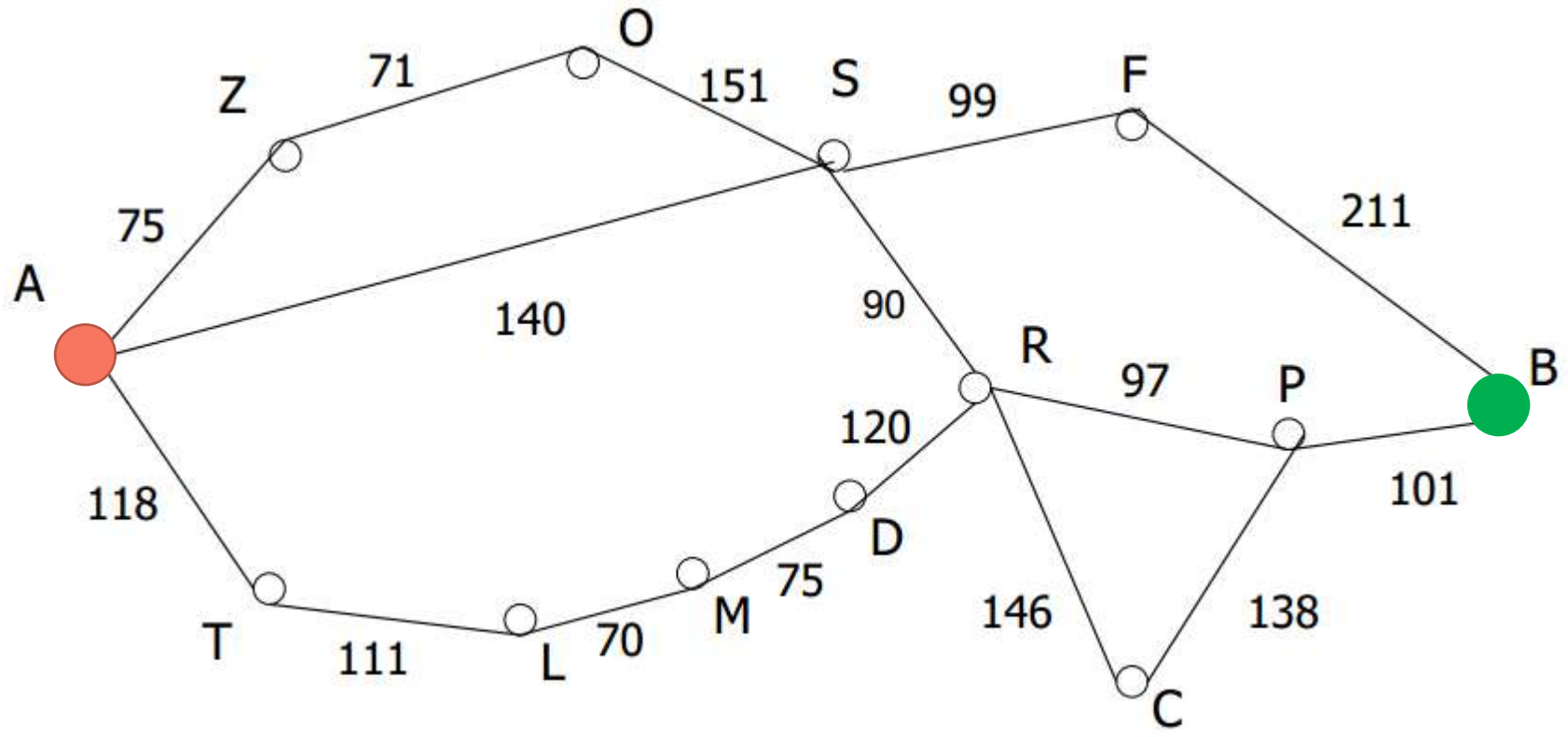
- Tidak akan menemui jalan buntu
- Jika ada satu solusi, maka breadth-first search akan menemukannya. Jika ada lebih dari satu solusi, maka solusi minimum akan ditemukan

❖ **Kerugian**

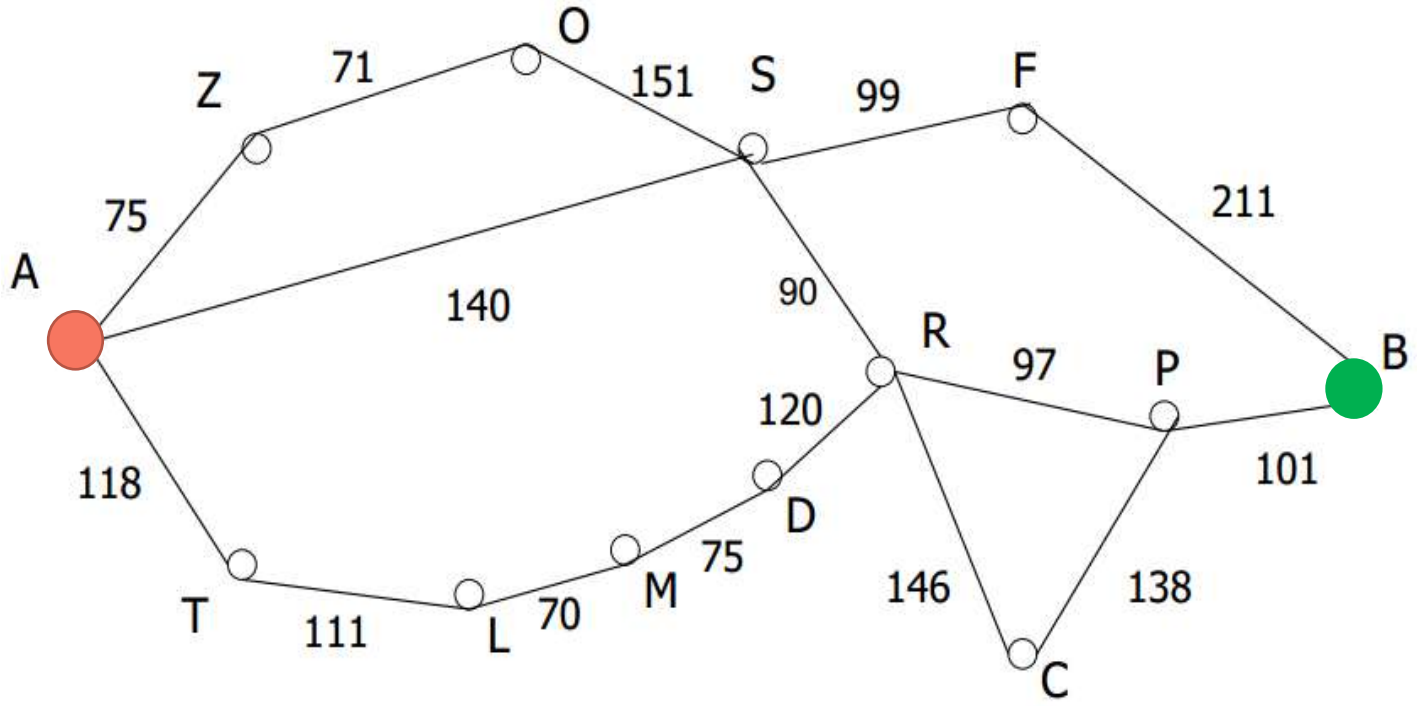
- Membutuhkan memori yang cukup banyak, karena menyimpan semua node dalam satu pohon
- Membutuhkan waktu yang cukup lama

Latihan 1: Penerapan BFS

- Berdasarkan gambar berikut, terapkan metode BFS untuk menentukan rute dan biaya (cost) dari Kota A ke Kota B



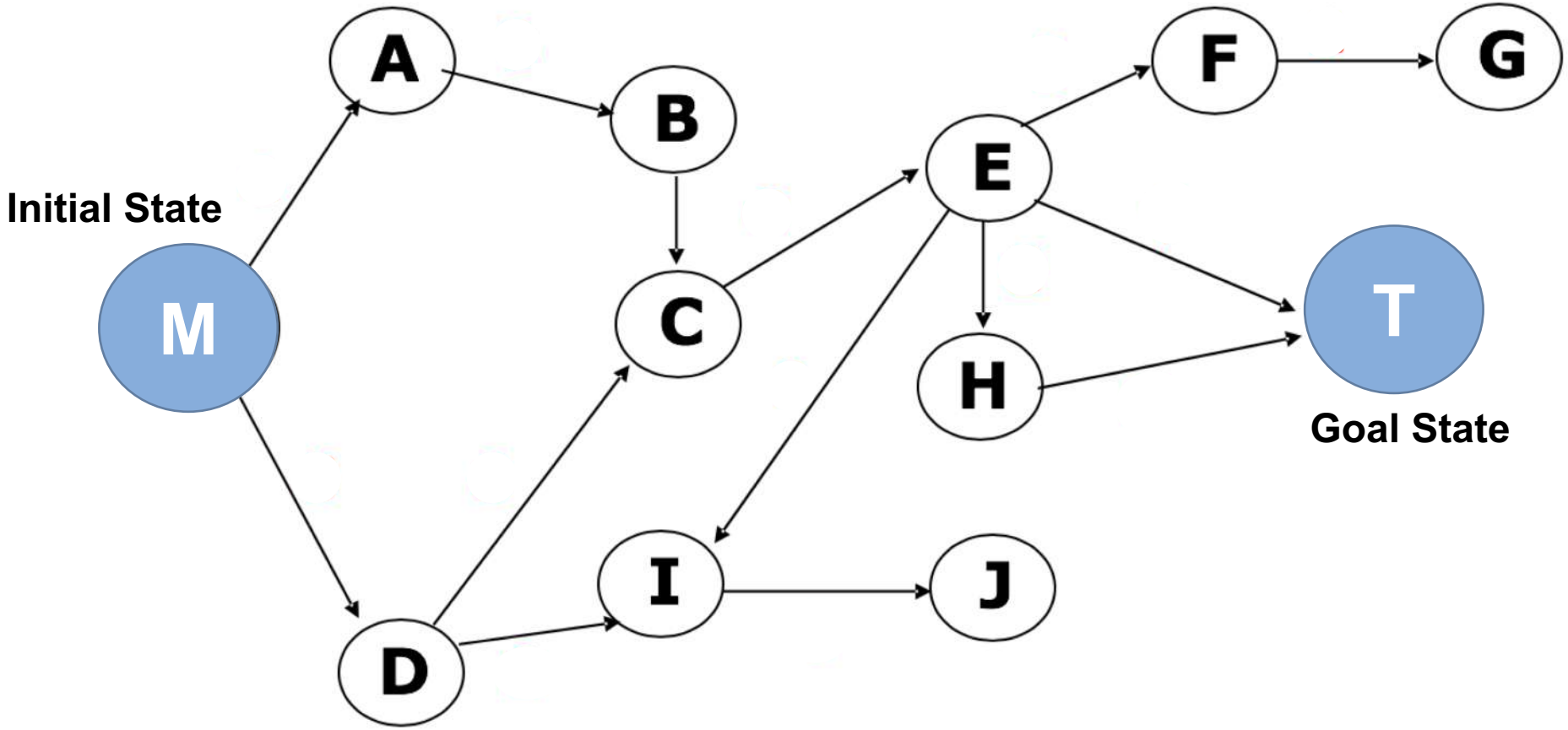
Solusi Latihan 1



A → Z_A,S_A,T_A → S_A,T_A,O_{AZ} → T_A,O_{AZ},O_{AS},F_{AS},R_{AS} →
O_{AZ},O_{AS},F_{AS},R_{AS},L_{AT} → O_{AS},F_{AS},R_{AS},L_{AT} → F_{AS},R_{AS},L_{AT} → R_{AS},L_{AT},B_{ASF}
→ L_{AT},B_{ASF},D_{ASR},C_{ASR},P_{ASR} → B_{ASF},D_{ASR},C_{ASR},P_{ASR},M_{ATL} →
Stop: B=goal, path: A→ S → F → B, path-cost = 450

Latihan 2: Penerapan BFS

- Berdasarkan gambar berikut, terapkan metode BFS untuk menentukan solusi dari **M** ke **T**



2. Depth First Search (DFS)

- *Depth-first search (DFS)* adalah proses pencarian buta yang melakukan ekspansi terhadap sebuah path (jalur) hingga tuntas sebelum melakukan ekplorasi terhadap path yang lain.

- Proses searching mengikuti sebuah path tunggal sampai menemukan goal atau dead-end.

- Apabila proses searching menemukan dead-end, DFS akan melakukan penelusuran balik (back-track) ke node terakhir untuk melihat apakah node tersebut memiliki path cabang yang belum dieksplorasi.

Prinsip Kerja DFS

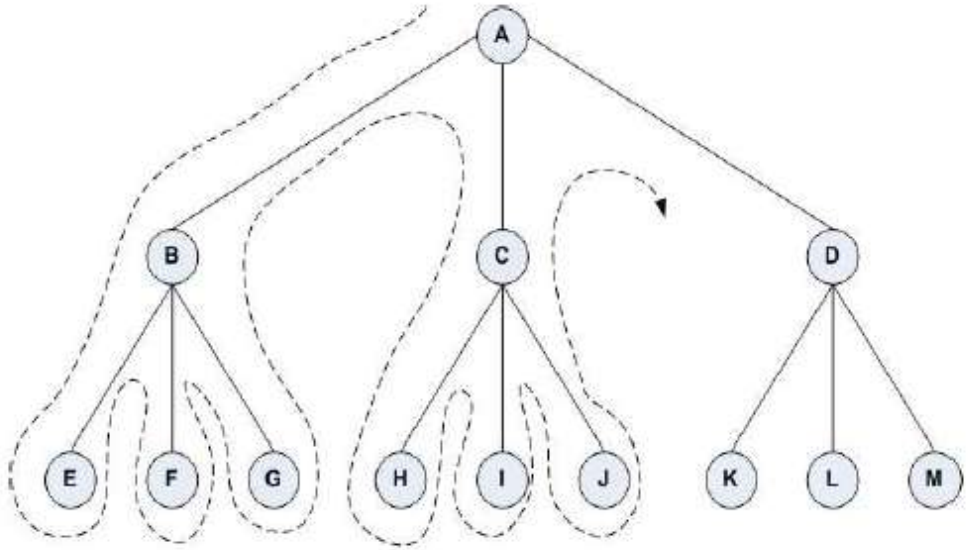
- Apabila cabang ditemukan, DFS akan melakukan eksplorasi terhadap cabang tersebut.

- Apabila sudah tidak ada lagi cabang yang dapat dieksplorasi, DFS akan kembali ke node parent (back-track) dan melakukan proses searching terhadap cabang yang belum dieksplorasi dari node parent sampai menemukan penyelesaian masalah.

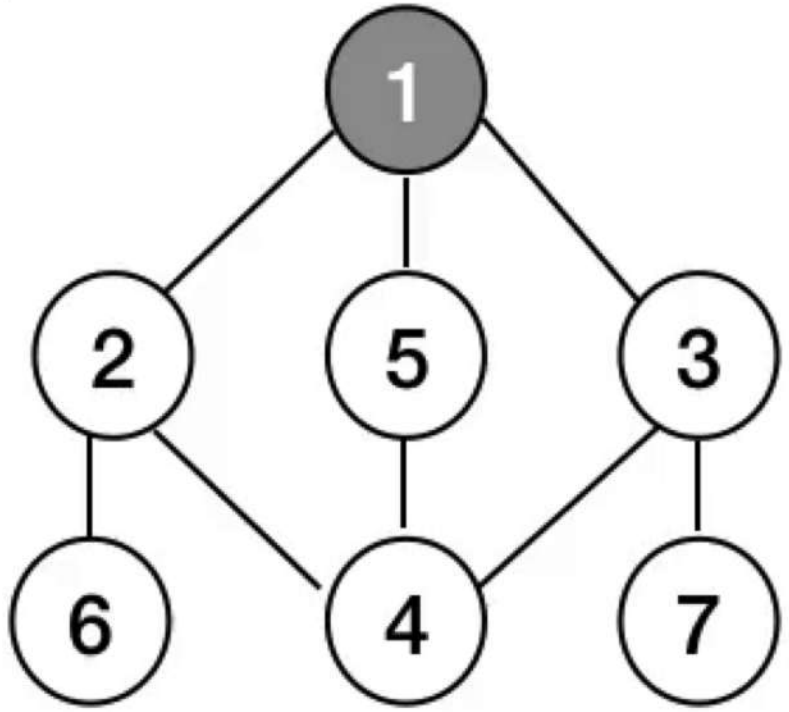
- Urutan proses searching DFS ditunjukkan dalam Gambar adalah: A, B, E, F, G, C, ...

Penerapan DFS dalam Algoritma

- Pendekatan DFS adalah Stack ➔ LIFO
- Misalkan kita hendak mencari lintasan ke node “I”
  - A →
  - B_A, C_A, D_A (Perhatikan Cara Memaknainya) ➔
  - E_{AB}, F_{AB}, G_{AB} , C_A, D_A ➔
  - F_{AB}, G_{AB} , C_A, D_A ➔ G_{AB} , C_A, D_A ➔ C_A, D_A ➔
  - H_{AC}, I_{AC}, J_{AC} , D_A ➔ I_{AC}, J_{AC} , D_A ➔
  - I ditemukan ➔ Lintasan ke I adalah ACI



Animasi DFS dengan Stack



Karakteristik DFS

❖ **Keuntungan**

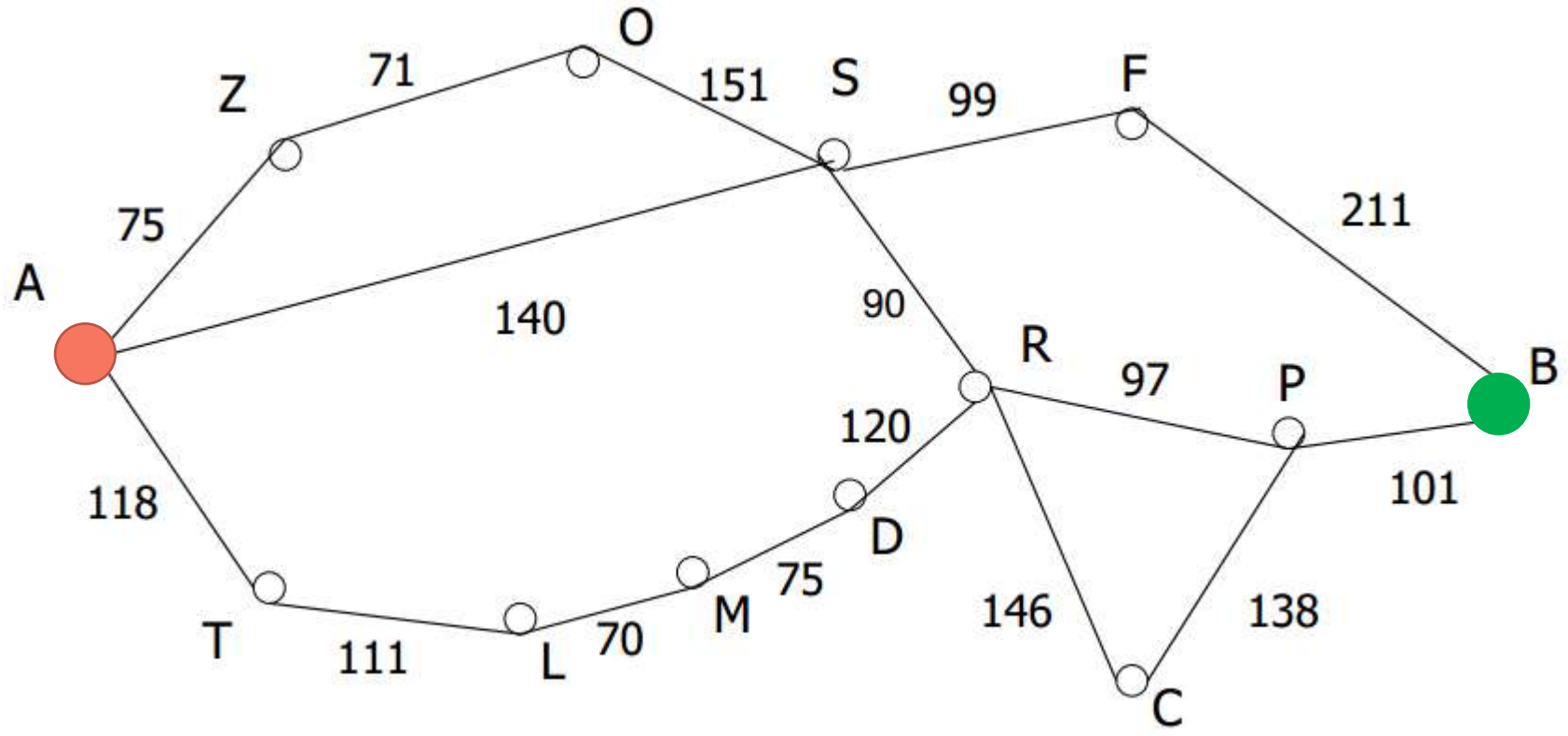
- ❑ Pemakaian memori hanya sedikit, berbeda jauh dengan BFS yang harus menyimpan semua node yang pernah dibangkitkan.
- ❑ Jika solusi yang dicari berada pada level yang dalam dan paling kiri, maka DFS akan menemukannya secara cepat.

❖ **Kerugian**

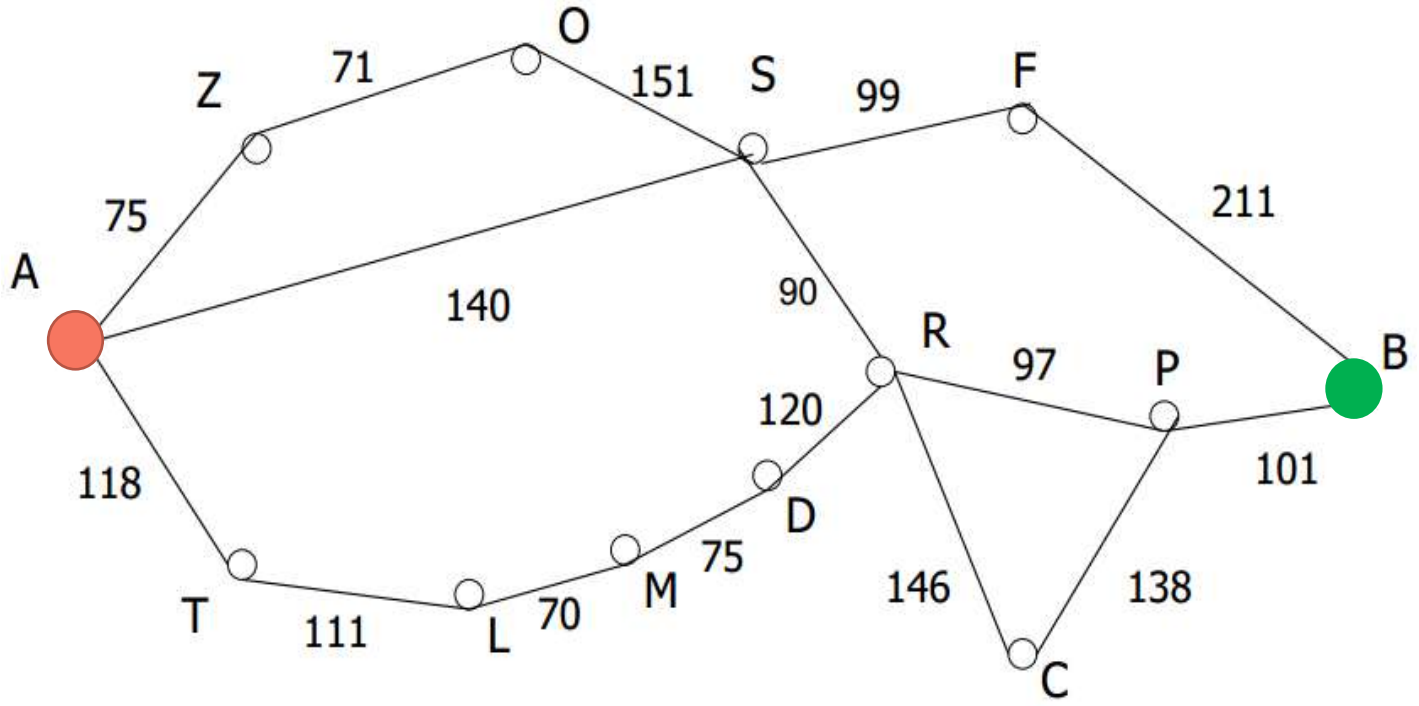
- ❑ Jika pohon yang dibangkitkan mempunyai level yang dalam (tak terhingga), maka tidak ada jaminan untuk menemukan solusi (Tidak Complete).
- ❑ Jika terdapat lebih dari satu solusi yang sama tetapi berada pada level yang berbeda, maka pada DFS tidak ada jaminan untuk menemukan solusi yang paling baik (Tidak Optimal).

Latihan 3: Penerapan DFS

- Berdasarkan gambar berikut, terapkan metode DFS untuk menentukan rute dan biaya (cost) dari Kota A ke Kota B



Solusi Latihan 3

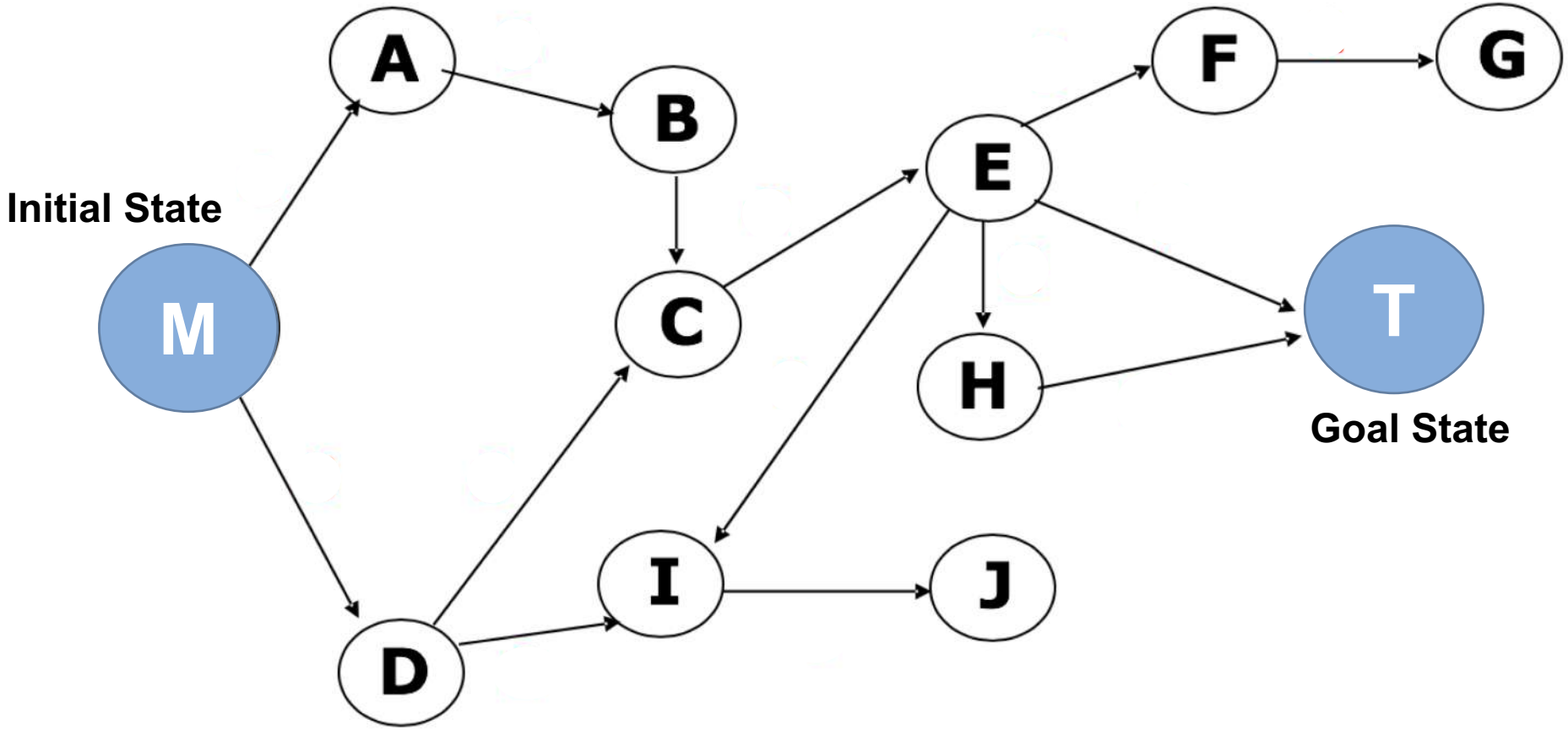


A → Z_A,S_A,T_A → O_{AZ}, S_A,T_A → S_{AZO},S_A,T_A → F_{AZOS}, R_{AZOS},S_A,T_A → B_{AZOSF}, R_{AZOS},S_A,T_A →

Stop: B=goal, path: A→ Z → O → S → F → B, path-cost = 607

Latihan 4: Penerapan DFS

- Berdasarkan gambar berikut, terapkan metode DFS untuk menentukan solusi dari **M** ke **T**

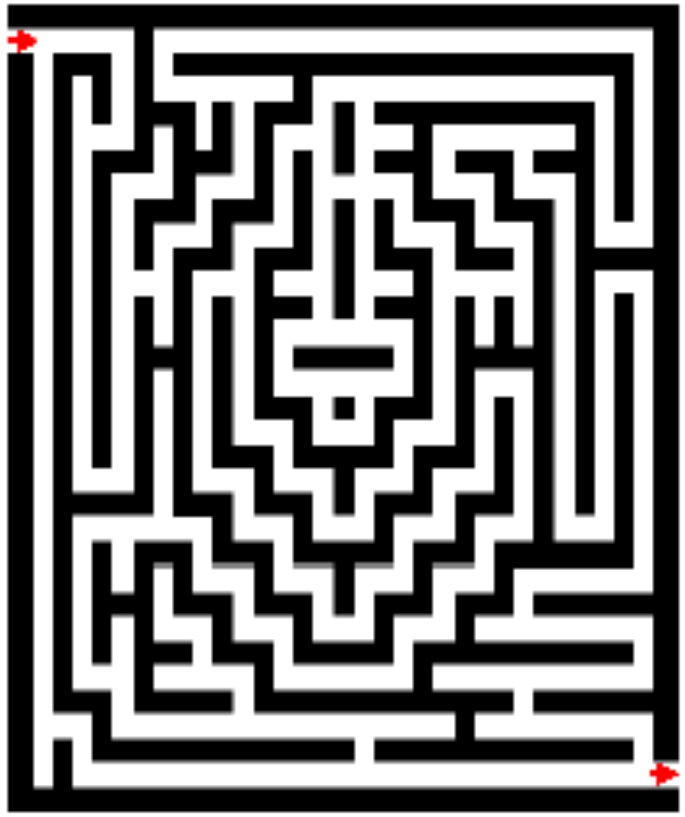


Konsep BackTrack DFS

- *Backtracking* dapat dipandang sebagai salah satu dari dua hal berikut:
  1. Sebagai sebuah fase di dalam algoritma traversal DFS
  2. Sebagai sebuah metode pemecahan masalah yang terstruktur dan sistematis

Contoh Penerapan BackTrack

- **Contoh** ***(Maze problem***): diberikan sebuah labirin (*maze*), temukan lintasan dari titik awal sampai titik akhir



Maze Problem

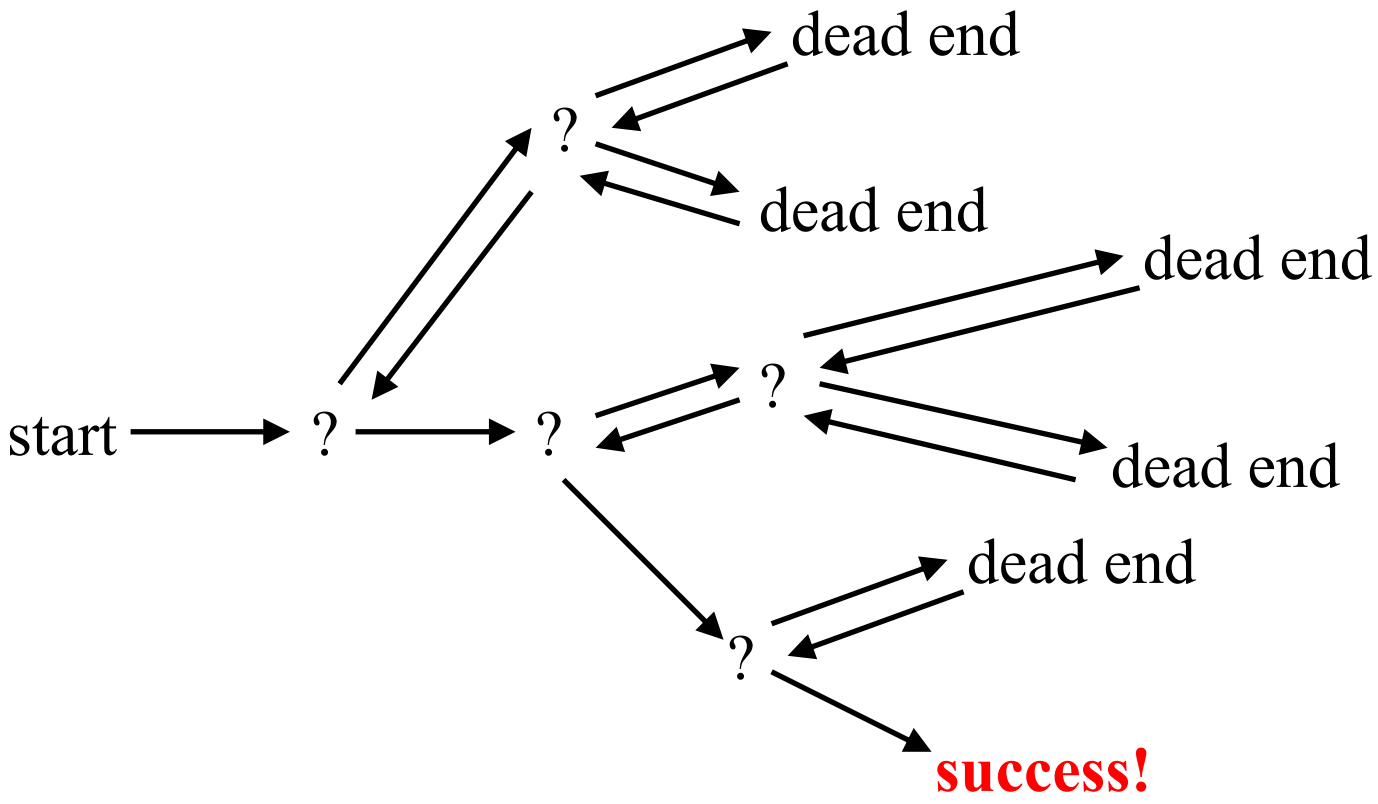
- Pada tiap perpotongan, Anda harus memutuskan satu di antara tiga pilihan:
  - Maju terus
  - Belok kiri
  - Belok kanan

- Anda tidak punya cukup informasi untuk memilih pilihan yang benar (yang mengarah ke titik akhir)
- Tiap pilihan mengarah ke sekumpulan pilihan lain
- Satu atau lebih sekuens pilihan mengarah ke solusi
- *Backtracking* (runut-balik) dapat digunakan untuk persoalan seperti ini

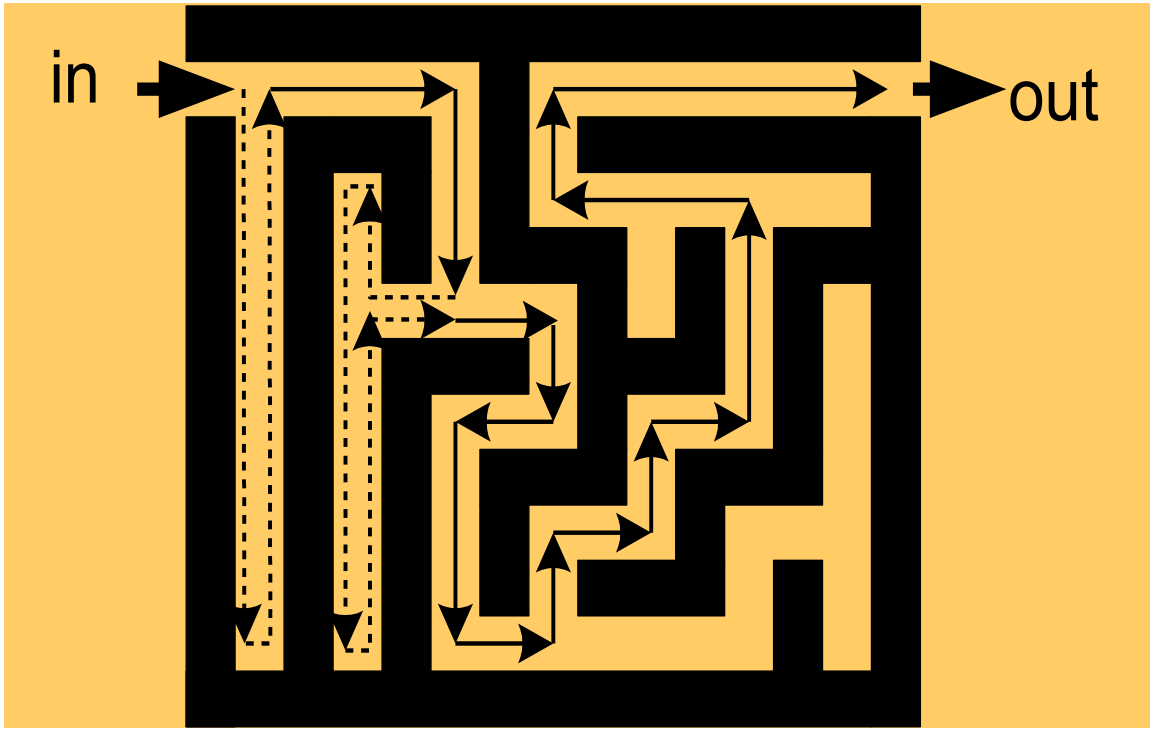
Penyelesaian dengan *Bactracking*:

- Bagi lintasan menjadi sederetan langkah.
- Sebuah langkah terdiri dari pergerakan satu unit sel pada arah tertentu.
- Arah yang mungkin: lurus (*straight*), kiri (*left*), ke kanan (*right*).

Animasi *Backtracking*



Contoh Lain Penerapan BackTrack



Contoh runut-balik pada sebuah labirin.
Runut-balik diperlihatkan dengan garis putus-putus.

Garis besar algoritma runut-baliknya:


```
while belum sampai pada tujuan do
  if terdapat arah yang benar sedemikian sehingga kita belum pernah
     berpindah ke sel pada arah tersebut
  then
     pindah satu langkah ke arah tersebut
  else
     backtrack langkah sampai terdapat arah seperti yang disebutkan
     di atas
  endif
endwhile
```

Bagaimana mengetahui langkah mana yang perlu dijejaki kembali?

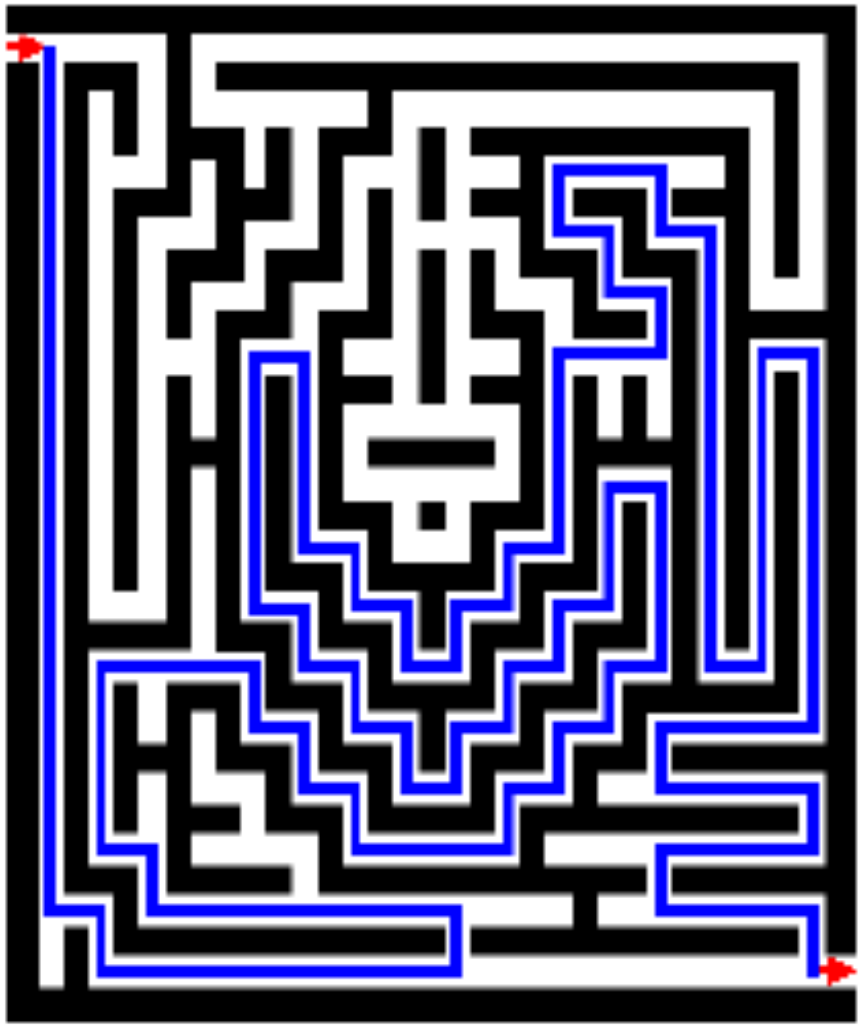
Ada dua solusi untuk masalah ini:

1. Simpan semua langkah yang pernah dilakukan, atau
2. Gunakan rekursi (yang secara implisit menyimpan semua langkah).

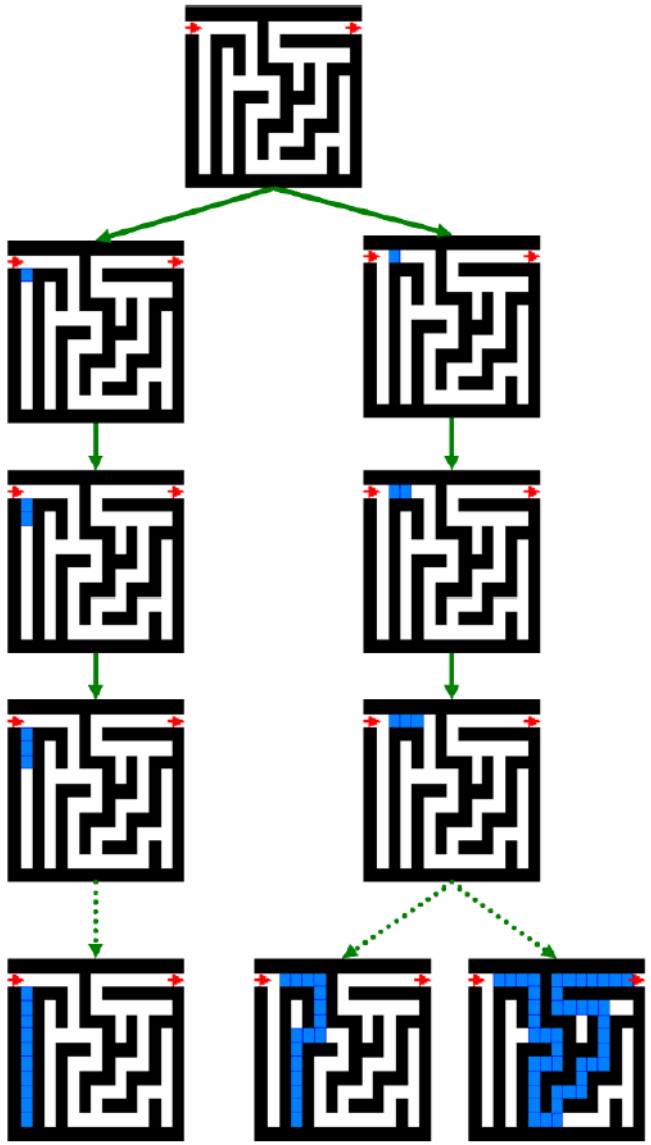
Rekursi adalah solusi yang lebih mudah.

```
function SolveMaze(input M : labirin)→boolean
{ true jika pilihan mengarah ke solusi }

Deklarasi
   arah : integer  { up = 1, down, 2, left = 3, right = 4
}

Algoritma:
   if pilihan arah merupakan solusi then
     return true
   else
     for tiap arah gerakan (lurus, kiri, kanan) do
        move(M, arah)  { pindah satu langkah (satu sel)
                         sesuai arah  tersebut }
        if SolveMaze(M) then
          return true
        else
          unmove(M, arah)  { backtrack }
        endif
     endfor
     return false     { semua arah sudah dicoba, tetapi
                        tetap buntu,  maka
                        kesimpulannya: bukan solusi }
   endif
```

Contoh Solusi :



Jika kita menggambarkan sekuens pilihan yang kita lakukan, maka diagram berbentuk seperti pohon.



Simpul daun merupakan:

1. Titik *backtrack*, atau
2. Simpul *goal*

Pada titik *backtrack*, simpul tersebut menjadi mati (tidak bisa diekspansi lagi)

Aturan pembentukan simpul: DFS

3. Depth-Limited Search (DLS)

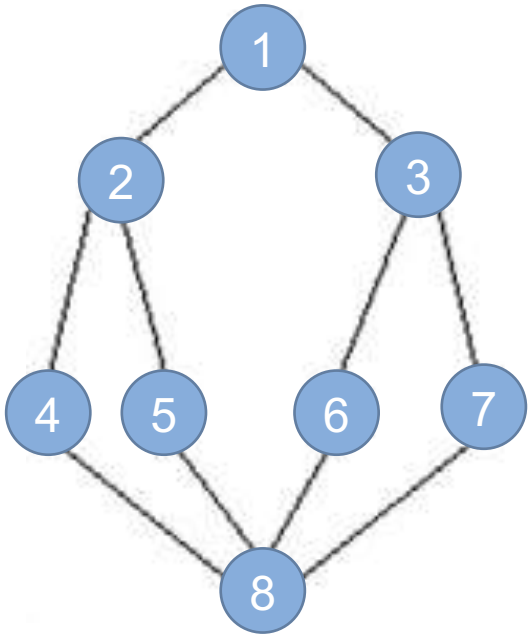
- Algoritma Depth-Limited Search (DLS), adalah salah satu jenis algoritma pencarian solusi. Algoritma ini dijalankan dengan cara membangkitkan pohon pencarian secara dinamis. Pencarian solusi dilakukan secara mendalam.

- Pada dasarnya, algoritma DLS sama dengan algoritma DFS, hanya saja dalam permasalahan penelusuran graf, sebelumnya ditentukan terlebih dahulu batas maksimum level yang dikunjungi.

- Sederhananya, DLS merupakan **DFS** yang memiliki **batas maksimum** kedalaman. Batas maksimum ini ditentukan di awal pencarian.

Contoh Penerapan DLS

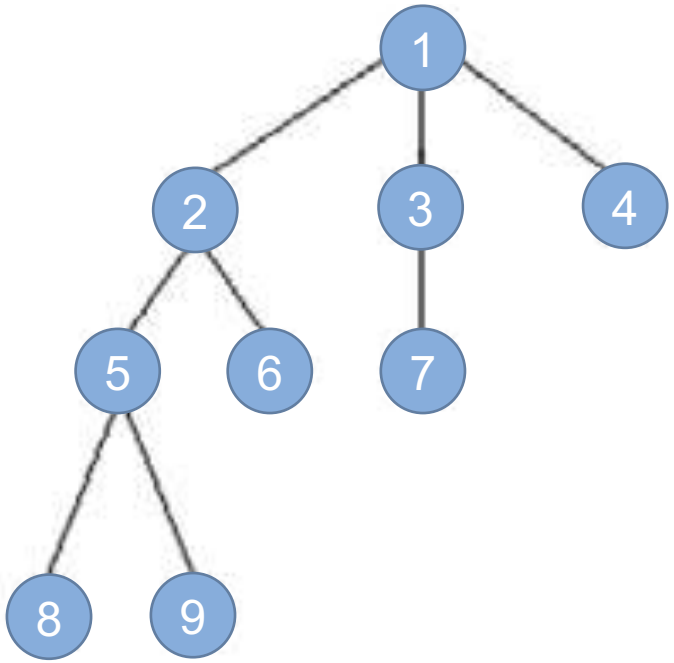
Asumsi Konvensi :
Node Paling Awal → Level 1



Bila simpul awal adalah 1 dan batas kedalaman adalah 3, maka urutan dikunjunginya adalah 1, 2, 4, 5, 3, 6, 7.

Contoh Penerapan DLS

Asumsi Konvensi :
Node Paling Awal → Level 1

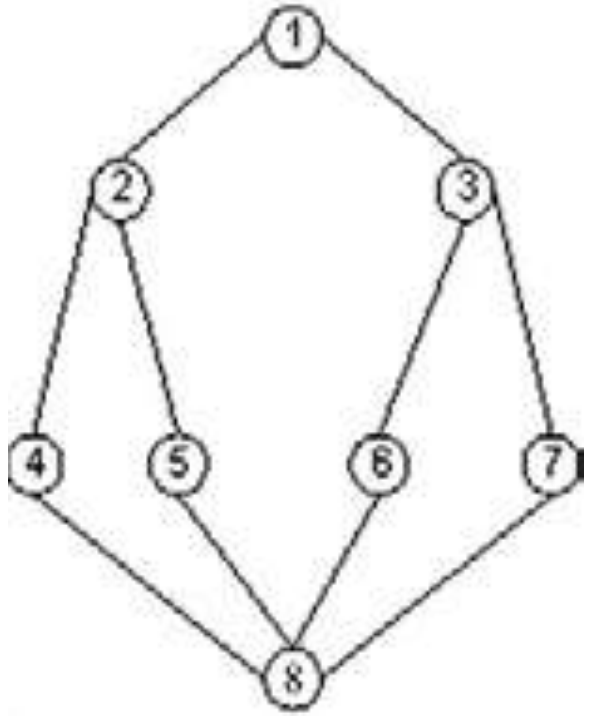


Bila simpul awal juga 1 dan batas kedalaman adalah 3, maka urutan dikunjunginya adalah 1, 2, 5, 6, 3, 7, 4

Karakteristik DLS

- DLS lahir untuk mengatasi kelemahan DFS (tidak complete) dengan membatasi kedalaman maksimum dari suatu jalur solusi.

- Pada DLS, harus diketahui atau ada batasan dari sistem tentang level maksimum. Jika batasan kedalaman terlalu kecil, DLS tidak complete.



4. Iterative-Deepening Search (IDS)

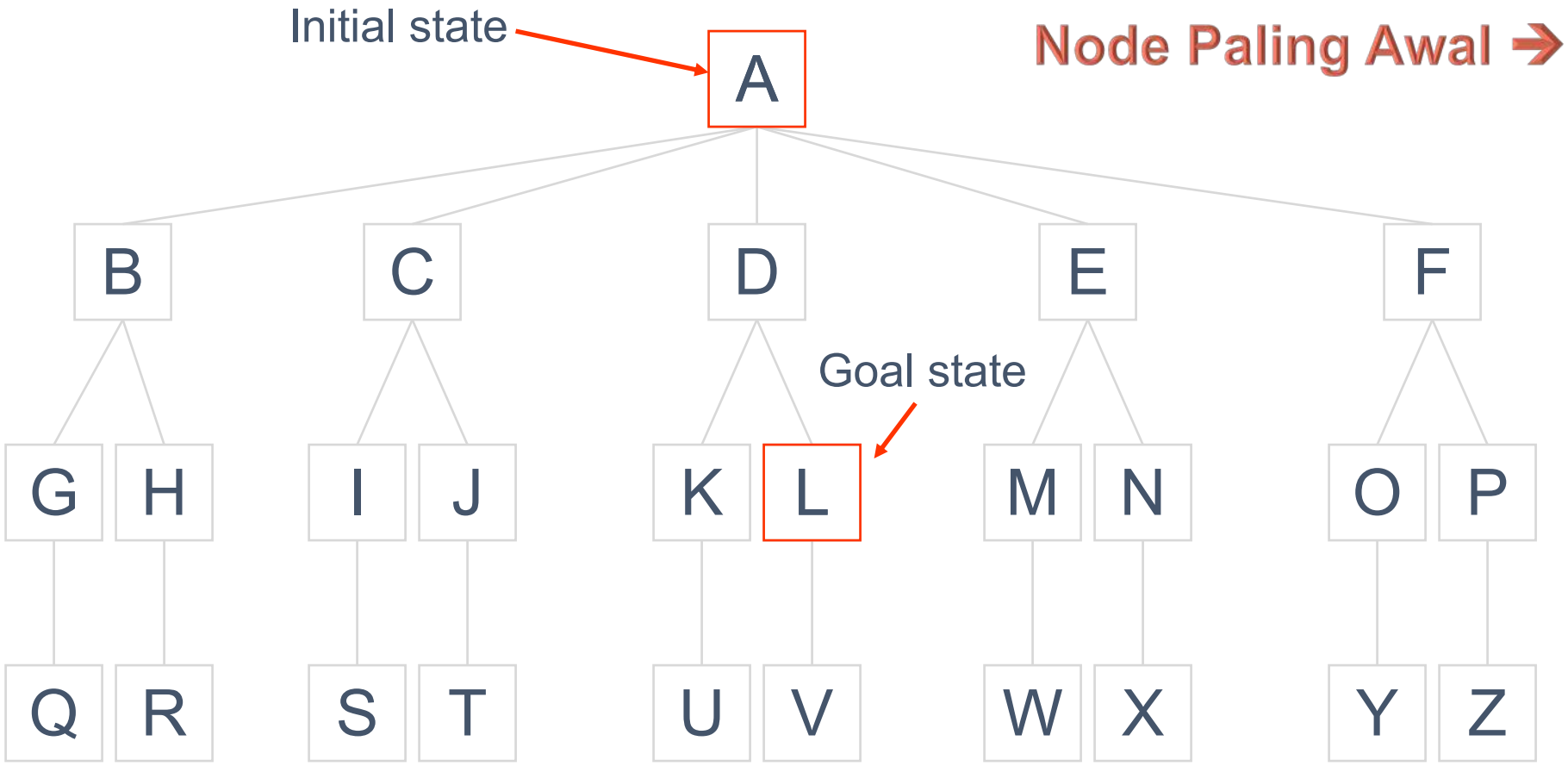
- IDS merupakan metode yang menggabungkan kelebihan BFS (Complete dan Optimal) dengan kelebihan DFS (space complexity rendah atau membutuhkan sedikit memori)

- IDS merupakan *Incremental* DLS (Depth Limited Search)

- Pada IDS, dilakukan pencarian DLS untuk level maksimum 0, jika tidak ditemukan solusi, berlanjut ke level maksimum 1, 2, 3, dst.

Contoh Penggunaan IDS

Asumsi Konvensi :
Node Paling Awal → Level 0



Pencarian dimulai dari Node A
Node A di-ekspansi

Karena pada tahap ini batas pencarian adalah hingga level 0, maka pencarian (iterasi) dihentikan. Lanjutkan IDS untuk level maksimum = 1

Ukuran Node: 0	**Jumlah Node: 0**	
Node Ekspansi: 1	**Current Action: Expanding**	**Level Saat Ini: 0**

ITERATIVE DEEPENING SEARCH PATTERN (0^{th} ITERATION)

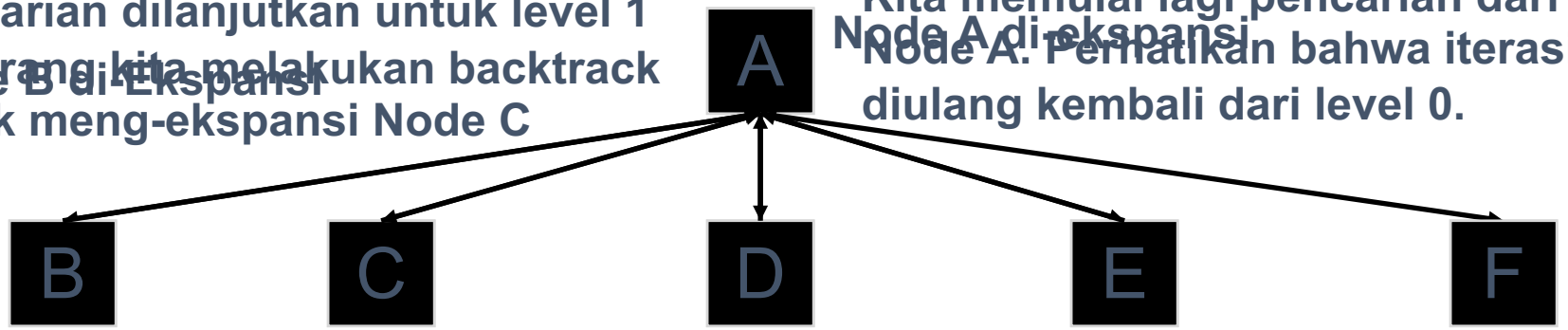
Pencarian dilanjutkan untuk level 1

Sekarang kita melakukan backtrack untuk meng-ekspansi Node C

Node B di Ekspansi

Node A di-ekspansi

Kita memulai lagi pencarian dari Node A. Perhatikan bahwa iterasi diulang kembali dari level 0.



Karena pada tahap ini batas pencarian adalah hingga level 1, maka pencarian (iterasi) dihentikan. Lanjutkan IDS untuk level maksimum = 2

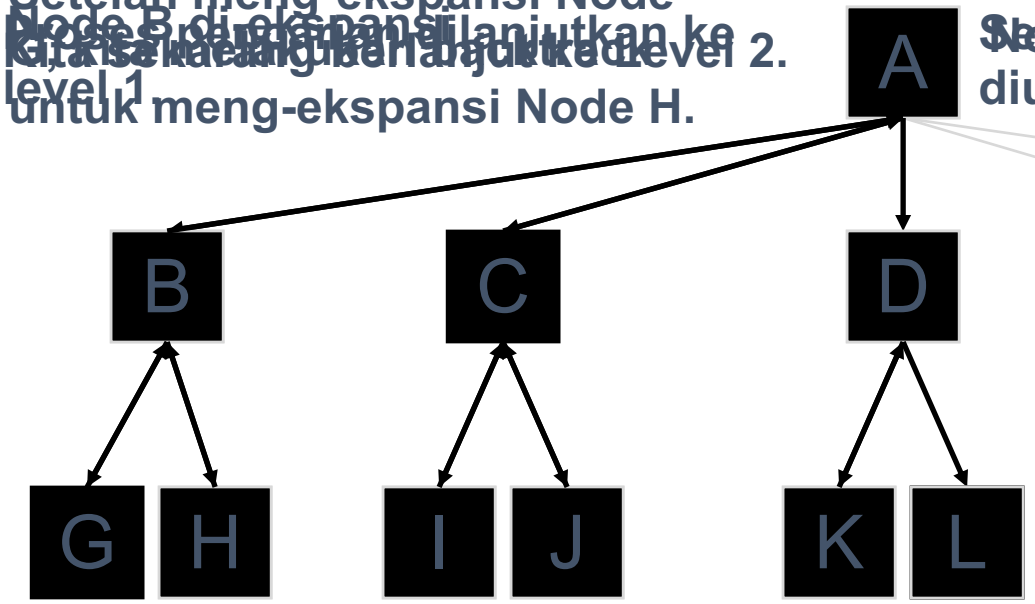
Ukuran Node: 0	Jumlah Node : 1	
Node Ekspansi: 7	Current Action: Expanding	Level Saat Ini : 1

ITERATIVE DEEPENING SEARCH PATTERN (1st ITERATION)

Setelah meng-ekspansi Node B di level 1, kita lanjutkan ke level 2.

Node B di ekspansi dilanjutkan ke level 2. Kita kembali untuk meng-ekspansi Node H.

Seperti tahap sebelumnya, pencarian diulang dari Node A.



Kita memulai lagi pencarian dari Node A. Perhatikan bahwa pencarian dilanjutkan dari tahap sebelumnya. Sehingga node yang sudah di-ekspansi hingga tahap ini adalah 7 (1 + 6)

Node L berada pada level 2. Node L merupakan solusi.

Solusi untuk masalah ini ditemukan pada iterasi kedua.

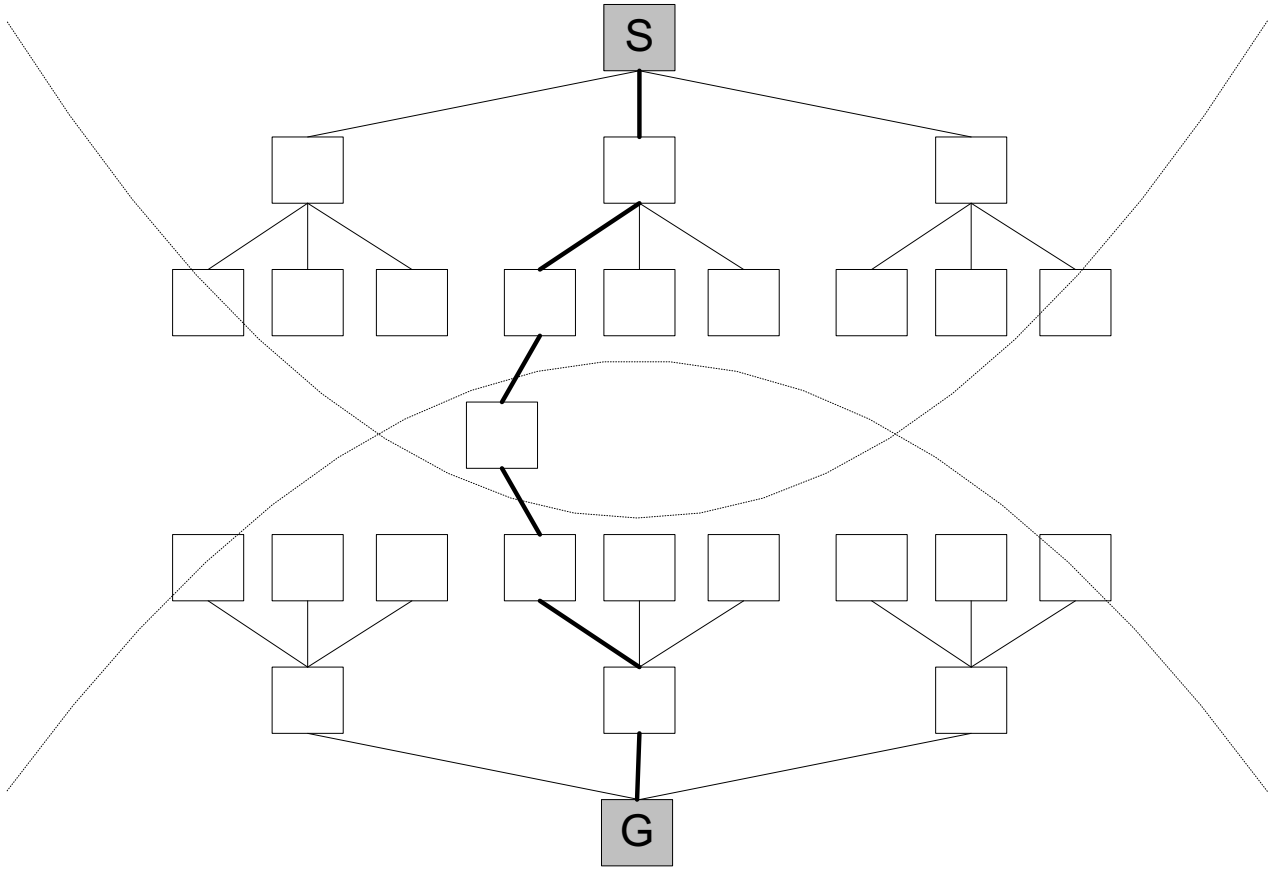
Ukuran Node : 0	Jumlah Node : 0	
Nodes Ekspansi :16	Pencarian Berakhir	Level Saat Ini: 2

ITERATIVE DEEPENING SEARCH PATTERN (2nd ITERATION)

5. Bi-Directional Search (BDS)

- Pencarian dilakukan dari **dua arah** : pencarian maju (dari start ke goal) dan pencarian mundur (dari goal ke start).
- Ketika dua arah pencarian telah membangkitkan simpul yang sama, maka solusi telah ditemukan, yaitu dengan cara menggabungkan kedua jalur yang bertemu.
- Pendekatan Algoritma BDS disesuaikan dengan kasus yang ingin diselesaikan

Contoh BDS



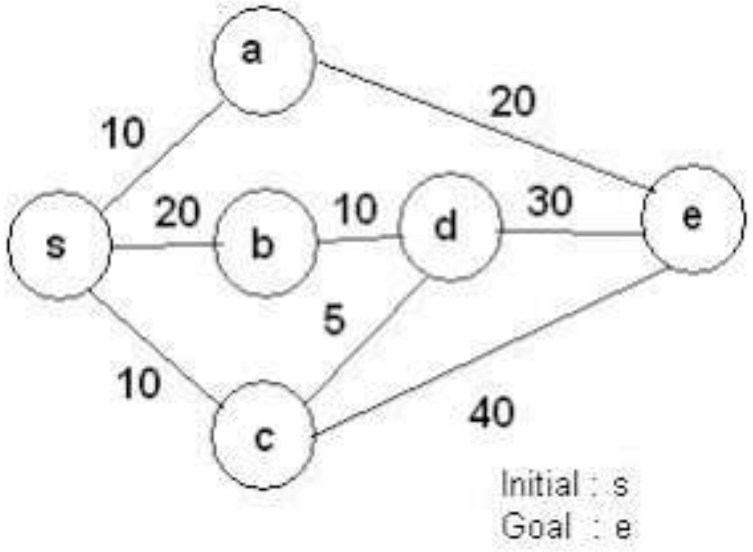
6. Uniform Cost Search (UCS)

- Konsepnya hampir sama dengan BFS, bedanya adalah bahwa BFS menggunakan urutan level yang paling rendah sampai yang paling tinggi, sedangkan UCS menggunakan urutan **biaya** dari yang paling kecil sampai yang terbesar.

- UCS berusaha menemukan solusi dengan total biaya terendah yang dihitung berdasarkan biaya dari simpul asal menuju ke simpul tujuan.

- Pendekatan UCS adalah **Priority Queue**.

Contoh Penggunaan UCS



Start : S

Goal : E

- S ➔
- $A_{S(10)}$, $C_{S(10)}$, $B_{S(20)}$ ➔
- $C_{S(10)}$, $B_{S(20)}$, $E_{SA(30)}$ ➔
- $D_{SC(15)}$, $B_{S(20)}$, $E_{SA(30)}$, $E_{SC(50)}$ ➔
- $B_{S(20)}$, $E_{SA(30)}$, $E_{SCD(45)}$, $E_{SC(50)}$ ➔
- $D_{SB(30)}$, $E_{SA(30)}$, $E_{SCD(45)}$, $E_{SC(50)}$ ➔
- $E_{SA(30)}$, $E_{SCD(45)}$, $E_{SC(50)}$, $E_{SBD(60)}$ ➔

- E ditemukan : Lintasan ➔ SAE, Cost = **30**
- Jalur pencarian: SACDBE

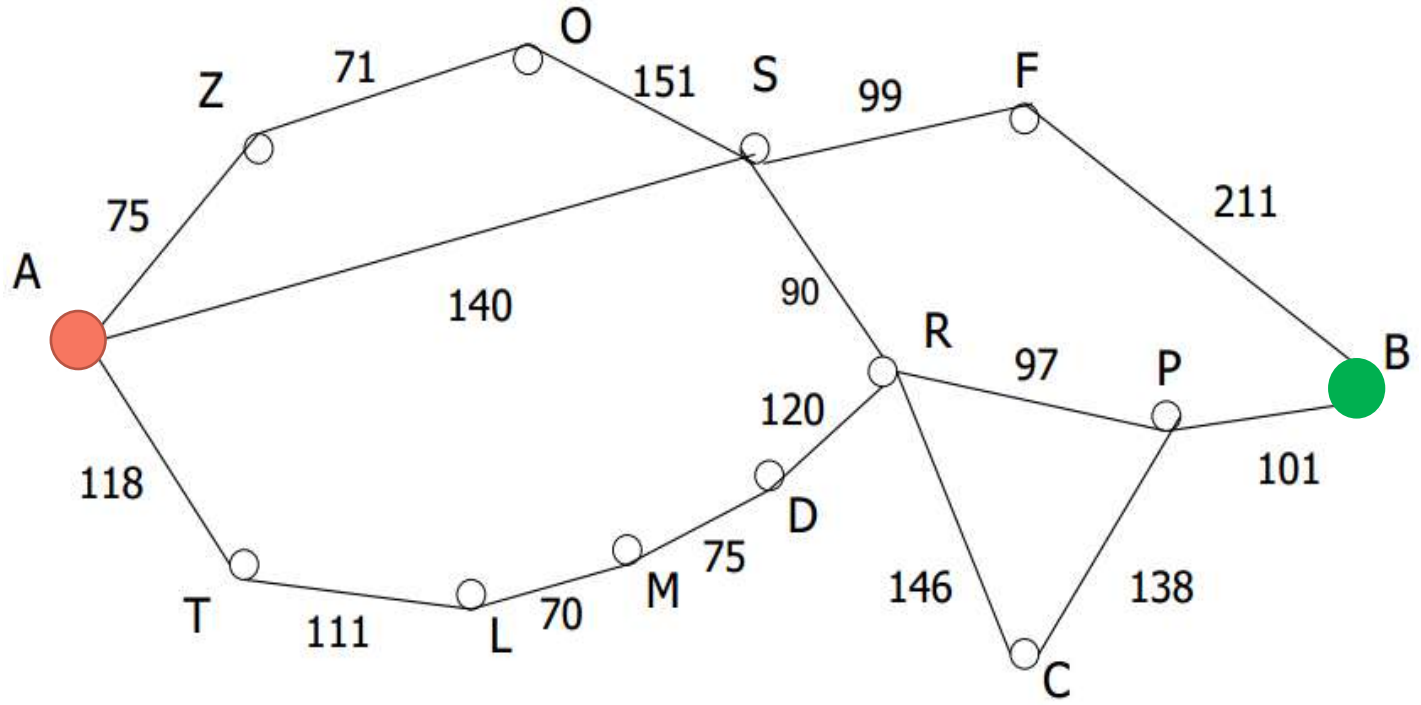
Karakteristik UCS

- Solusi yang dihasilkan dapat menemukan biaya (cost) termurah

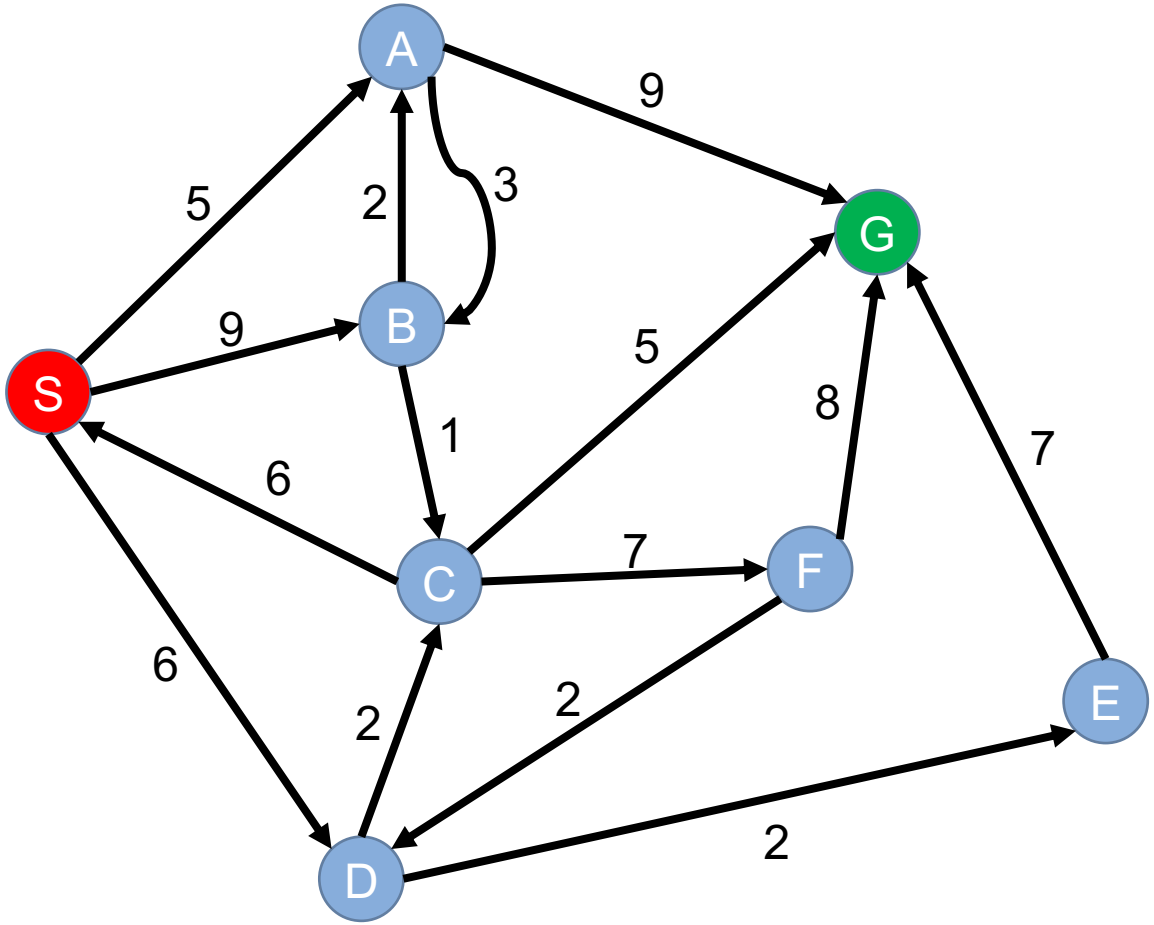
- Kebutuhan komputasi lebih rumit dibandingkan BFS ➔ Membutuhkan memori dan waktu komputasi yang tinggi

Latihan: Penerapan UCS dalam Mencari Lintasan Termurah

- Berdasarkan gambar berikut, terapkan metode UCS untuk menentukan rute dan biaya (cost) dari Kota A ke Kota B



Latihan: Penerapan UCS dalam Mencari Lintasan Termurah



Referensi

- Slide perkuliahan Kecerdasan Buatan oleh Ario Yudo Husodo (Teknik Informatika – Universitas Mataram)
- Chowdhary, K.R. 2020. Fundamentals of Artificial Intelligence. Springer India.