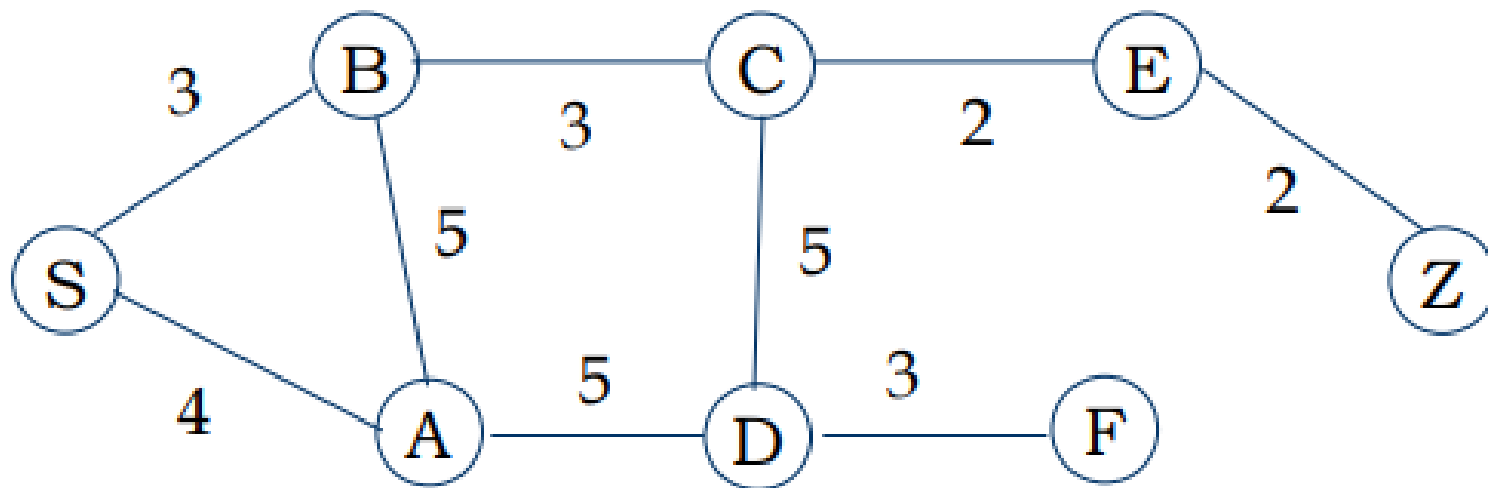


Kecerdasan Buatan/ Artificial Intelligence

Penyelesaian Problem Dengan Pencarian (Blind / Un-Informed Searching)

Satrio Hadi Wijoyo

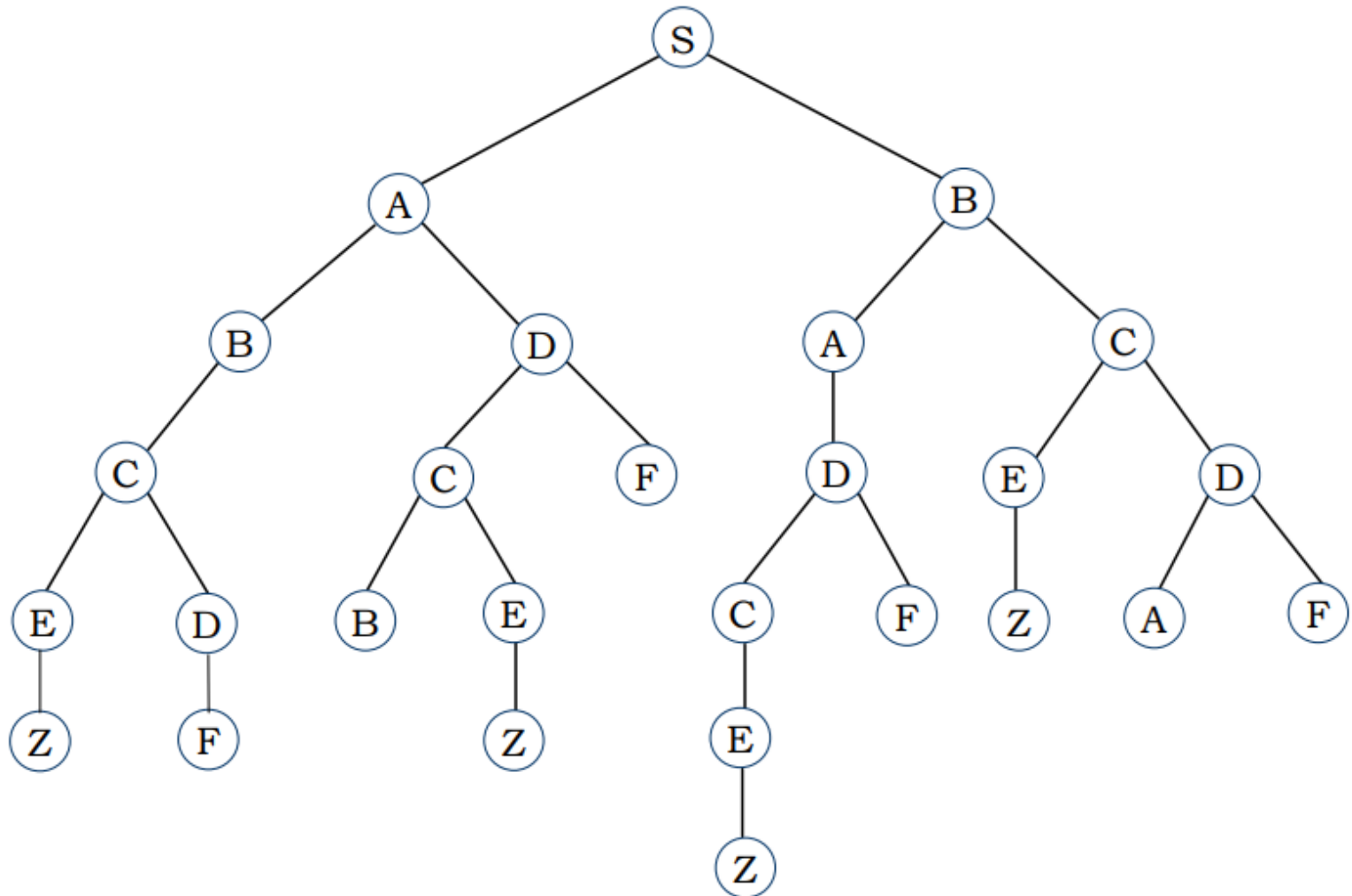




Gambar 4.3 Contoh Graph yang Berisi Path Antar Kota

Informed Search

Un-Informed Searching



Gambar 4.4 Struktur Tree dari Graph Gambar 4.3

Agen Penyelesaian Problem

- ❑ Agen pemecahan masalah adalah jenis agen berbasis tujuan.
- ❑ Memutuskan apa yang harus dilakukan dengan mencari urutan tindakan yang mengarah pada keadaan (*states*) yang diinginkan (Goal) .
- ❑ Merumuskan Tujuan lalu Perumusan masalah.
- ❑ Search → mengambil masalah sebagai masukan dan solusi pengembalian dalam bentuk urutan tindakan.

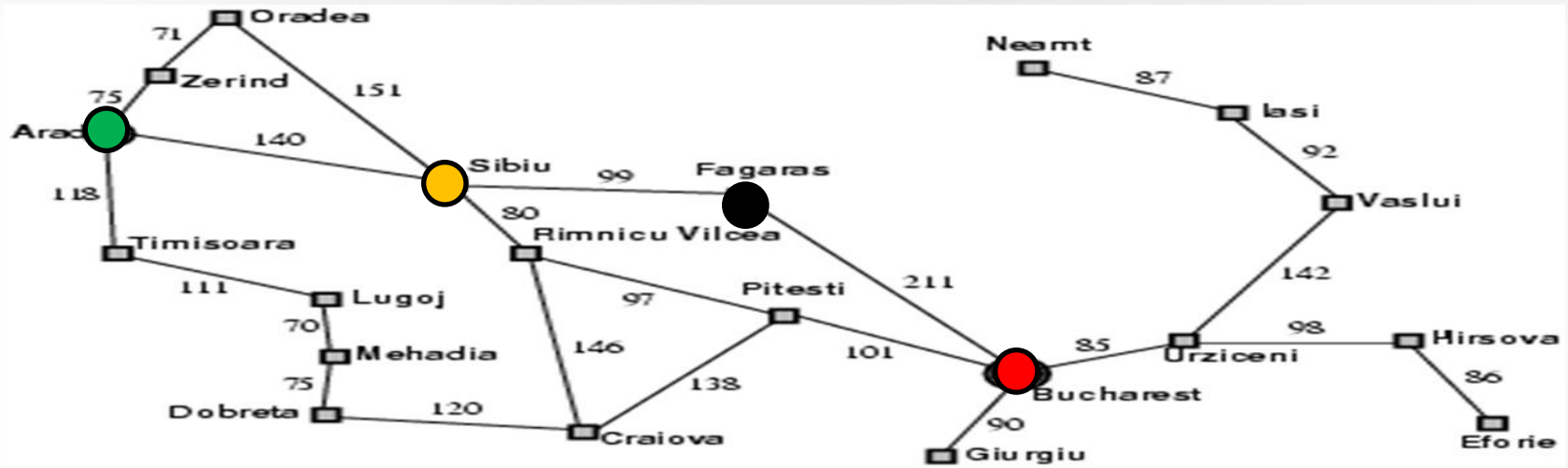
Agen Penyelesaian Problem

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

- ❑ Pertama-tama, agen akan **merumuskan tujuannya**, kemudian akan **merumuskan masalah yang solusinya adalah jalur** (urutan tindakan) ke tujuan, dan kemudian akan **menyelesaikan masalah tersebut dengan menggunakan pencarian**

Agen Penyelesaian Problem



□ Contoh : Romania

- Berlibur ke Rumania, saat ini berada di **Arad**.
- Penerbangan (keberangkatan) dilakukan besok dari **Bucharest**
- Merumuskan tujuan (**Formulate goal**) :
 - Berada di Bucharest
- Merumuskan masalah (**Formulate problem**) :
 - States : berbagai kota sebagai alternatif tempat yang akan dilalui
 - Actions : drive antara kota
- Cari solusi (**Find solution**) :
 - Urutan kota yang dilalui untuk mencapai tujuan. Misalnya ; **Arad**, **Sibiu**, **Fagaras**, **Bucharest**

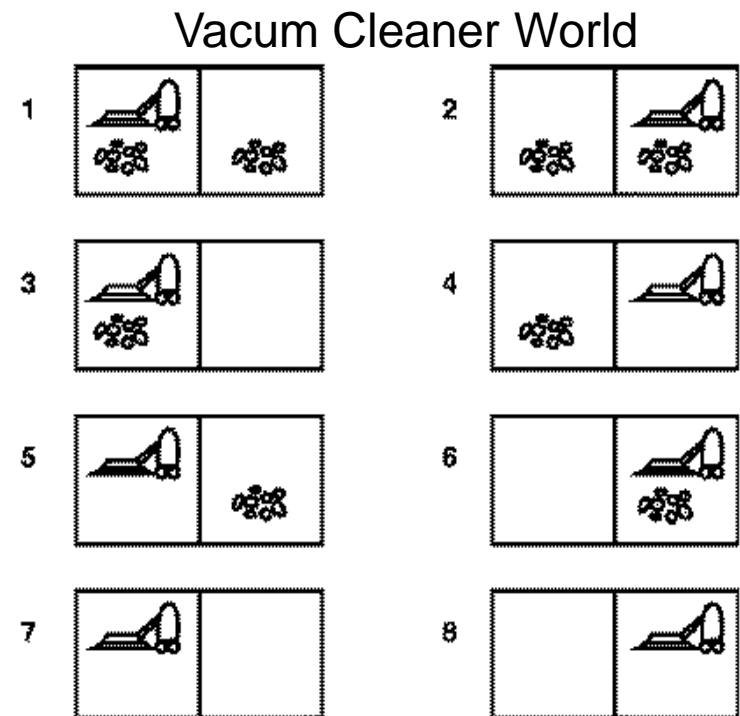
Jenis Problem/Masalah

- ❑ Single State (Deterministic, fully observable) → Single-state problem
 - Agen tahu persis keadaan sesuatu yang akan diamati.
- ❑ Multistate (Non-observable) → Sensorless / multistate problem (conformant problem)
 - Agen mungkin tidak mengetahui dimana keberadaan sesuatu yang dicari.
- ❑ Nondeterministic and/or partially observable → Contingency problem (keadaan yang tidak pasti)
 - Persepsi yang dapat memberikan informasi baru tentang keadaan saat ini.
- ❑ Unknown state space → Exploration problem (Masalah eksplorasi)

Contoh Problem

□ Contoh : vacuum world, Goal : ruangan bersih

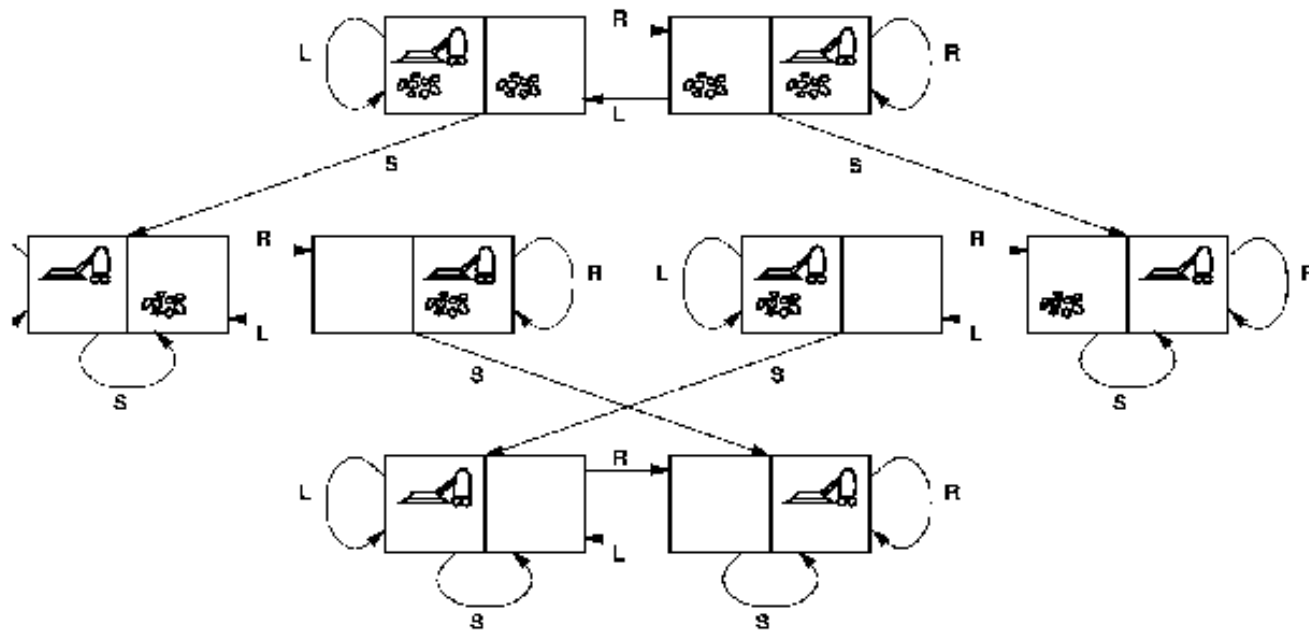
- Single-state,
- Vacuum Cleaner tahu dia ada di ruangan mana, kondisi ruangan pada saat itu bersih atau tidak.
- the goal is { state 7, state 8 }.
- Contoh state awal di 5
- Solusi : *[Right, Suck]*



Contoh Problem

- Contoh : vacuum world
 - Single-state, deterministic

Let's draw the state space:



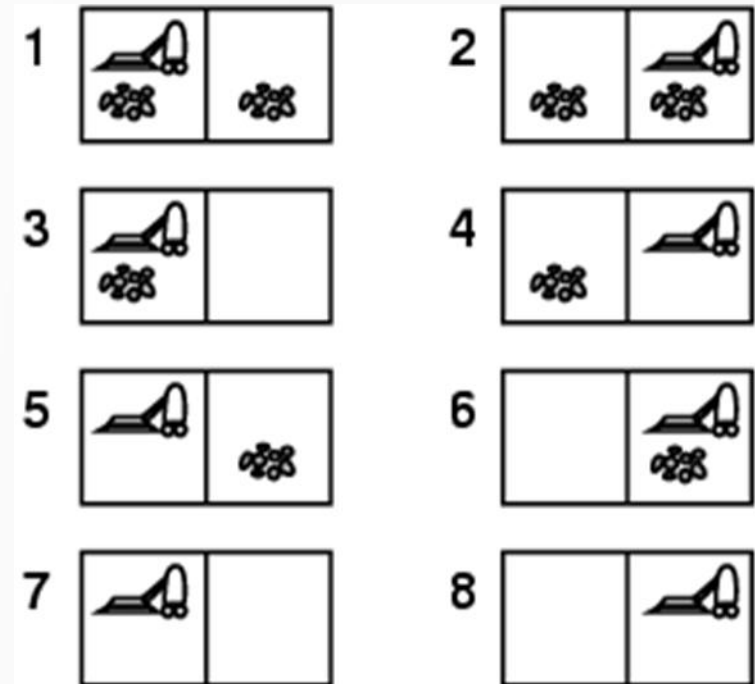
Contoh Problem

□ Contoh : vacuum world

- Multistate, Sensorless,
- Agent tdk tahu dia pada ruangan mana, dengan kondisi seperti apa
- start in {1,2,3,4,5,6,7,8}

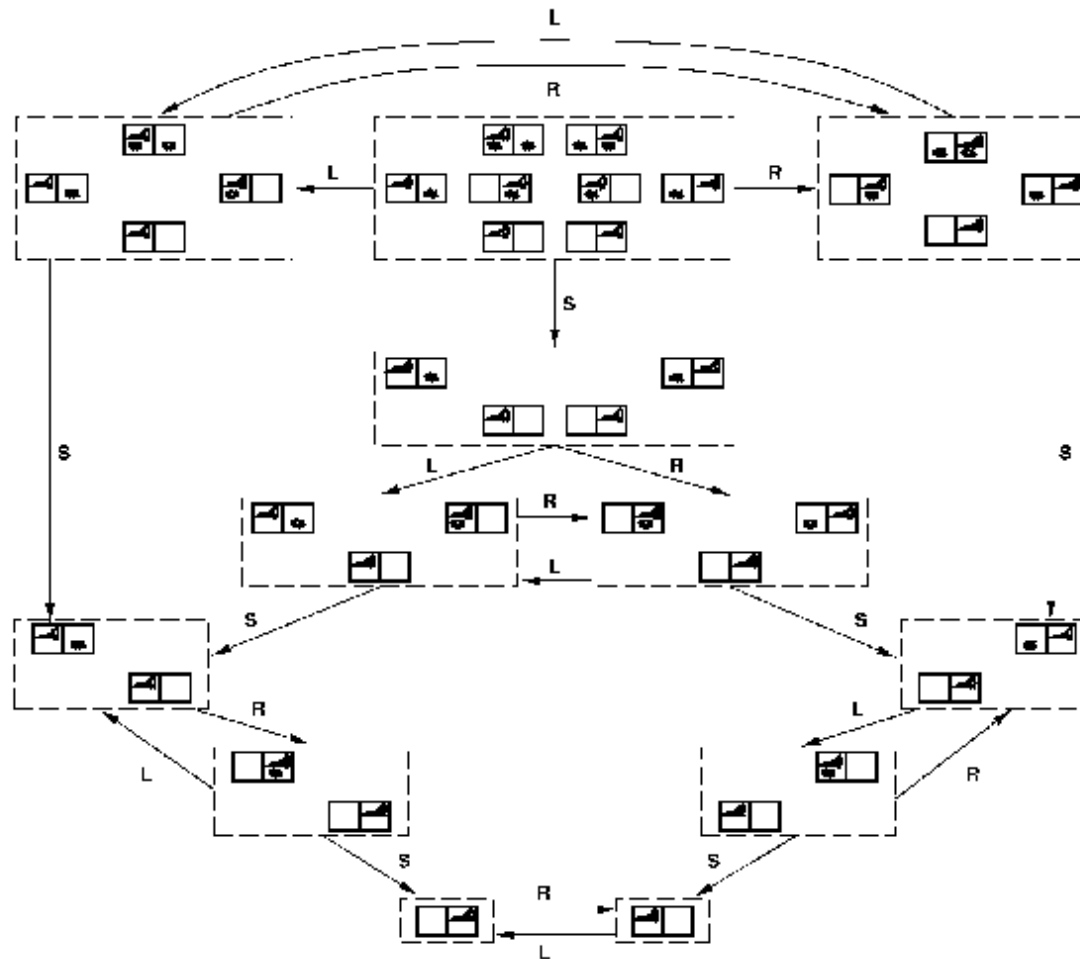
Solution?

- *[Right, Suck, Left, Suck]*



Multiset Sensorless

Then it must consider sets of possible states.



Contoh Problem

- Contingency / kemungkinan - Kemungkinan

- Nondeterministic: *Suck* may dirty a clean carpet
- Now [right,suck,left,suck] is NOT a correct plan.
- There doesn't exist any FIXED plan that always works.
- Agen untuk lingkungan ini HARUS memiliki sensor dan harus menggabungkan pengambilan keputusan, penginderaan, dan eksekusi.

- Exploration Problem

- Contoh : Agent yang masih belum memiliki pengetahuan apapun [bodoh] untuk mencapai tujuan robot tersebut harus melakukan aksi sambil mempelajari efek dari setiap aksinya
- agen harus belajar tentang lingkungan. ia harus mengambil tindakan untuk tujuan memperoleh pengetahuan tentang efeknya, BUKAN hanya untuk kontribusinya dalam mencapai suatu tujuan

Formulasi Problem

❑ Single-state problem formulation

- Suatu **problem** didefinisikan dalam 4 item :

1. **initial state** e.g., "at Arad"

2. **actions** or **successor function** $S(x)$ = set of action–state pairs

Successor function $S(x)$ yields a set of states that can be reached from x via a (any) single action.

- $\langle \text{action}, \text{successor}(\text{state}) \rangle$
- e.g., $S(\text{Arad}) = \{ \langle \text{Berangkat}(\text{Arad} \rightarrow \text{Zerind}), \text{BeradaDi}(\text{Zerind}) \rangle, \dots \}$

• $S(x) = \{ \langle a, y \rangle, \langle b, z \rangle \}$

y via action a , z via action b .

• $S(x) = \{ \langle a, y \rangle, \langle b, y \rangle \}$

y via action a , also y via alternative action b

* Penulisan bisa dibalik

3. **goal test**, can be

- **explicit**, e.g., $x = \text{"at Bucharest"}$
- **implicit**, e.g., $\text{beradaDi}(x)$

4. **path cost** (additive)

- Menetapkan besarnya biaya untuk setiap jalur yang ada.
- Mis., jumlah jarak tempuh, jumlah tindakan lain yang dilakukan, dll.
- $c(x, a, y)$ adalah cost action a dari state x ke state y , diasumsikan ≥ 0 .

• Solusi adalah suatu urutan tindakan yang mengarah dari keadaan awal (initial state) ke keadaan tujuan (goal state).

• Kualitas suatu solusi dapat diukur dari nilai fungsi biaya (cost function) yang paling minimal dari jalur (path) yang dilalui.

Formulasi Problem

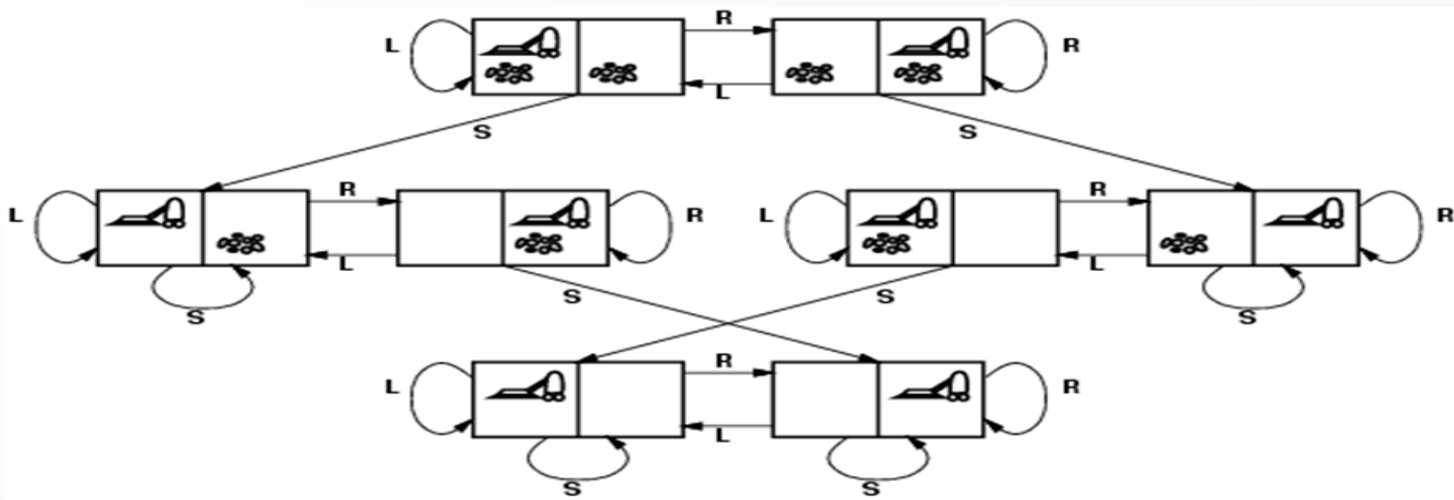
❑ Selecting a **state space**

- Dunia nyata luar biasa kompleks dan rumit! State space harus merupakan **abstraksi masalah** supaya bisa dipecahkan.
 - **State** = himpunan “keadaan nyata”. Mis : BeradaDi (Arad) – dengan siapa? kondisi cuaca?
 - **Action** = kombinasi berbagai “tindakan nyata”. Mis : Berangkat(Arad , Sibiu) – jalan tikus, isi bensin, istirahat, dll.
 - **Solution** = representasi berbagai “path nyata” yang mencapai tujuan
- Abstraksi ini membuat **masalah yang nyata lebih mudah dipecahkan**.

Contoh Problem

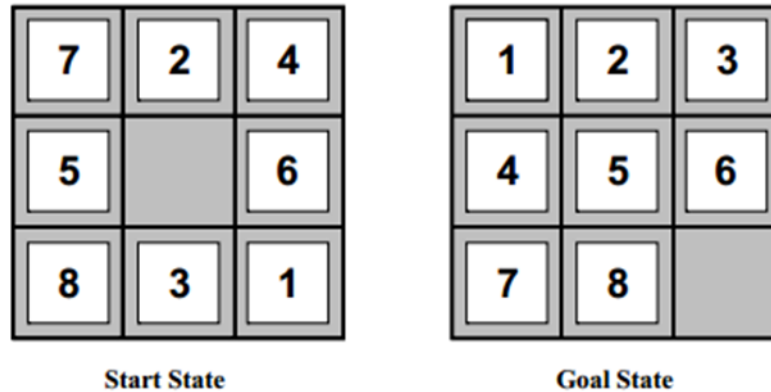
□ Contoh : Vacuum Cleaner World

- **State**: lokasi agent, status debu.
- **Possible action**: DoKeKiri(L), DoKeKanan(R), DoSedot(S).
- **Goal test**: semua ruangan sudah bebas debu.
- **Path cost**: asumsi step cost sama untuk semua action, mis: Path cost = 1 per action.
- Successor function mendefinisikan state space sbb:



Contoh Problem

❑ Contoh: 8-Puzzle



- **State**: lokasi 8 buah angka dalam matriks 3x3
- **Possible action** (move, blank) : left, right, up, down
- **Goal test**: apakah konfigurasi angka sudah seperti goal state di atas.
- **Path cost**: asumsi, 1 step cost = 1 per move.
Path cost = jumlah langkah dalam path.

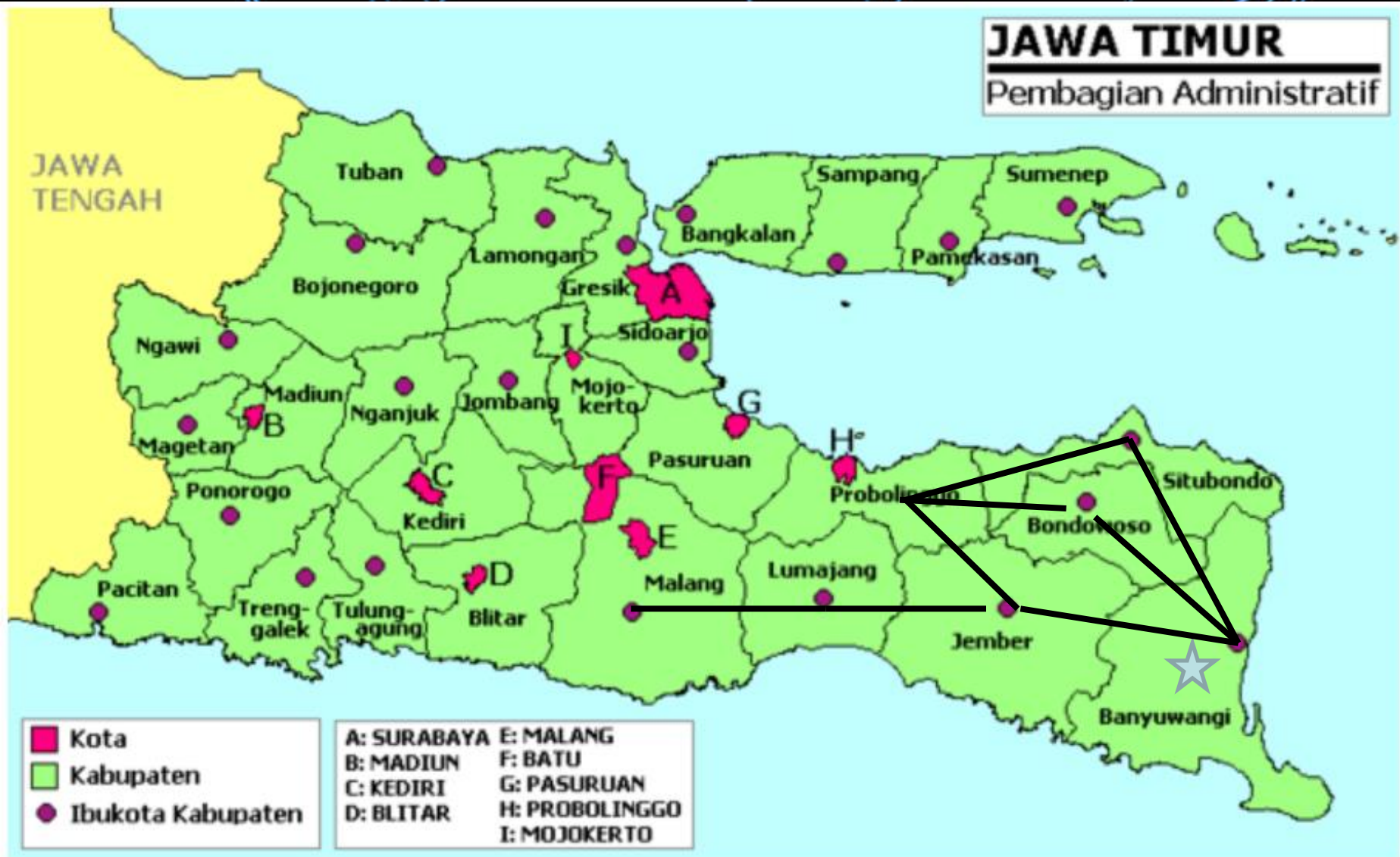
Algoritma Pencarian Dasar

□ Tree search algorithms :

- Setelah merumuskan masalah → cari solusinya menggunakan sebuah **search algorithm**
- **Search tree** merepresentasikan **state space**.
- Search tree terdiri dari kumpulan **node**: struktur data yang **merepresentasikan suatu state** pada suatu path, dan memiliki **parent**, **children**, **depth**, dan **path cost**.
- **Root node** merepresentasikan initial state.
- Penerapan successor function terhadap (state yang diwakili) node menghasilkan children baru → ini disebut **node expansion**.
- Kumpulan semua node yang belum di-expand disebut **fringe** (pinggir) sebuah search tree.

JAWA TIMUR

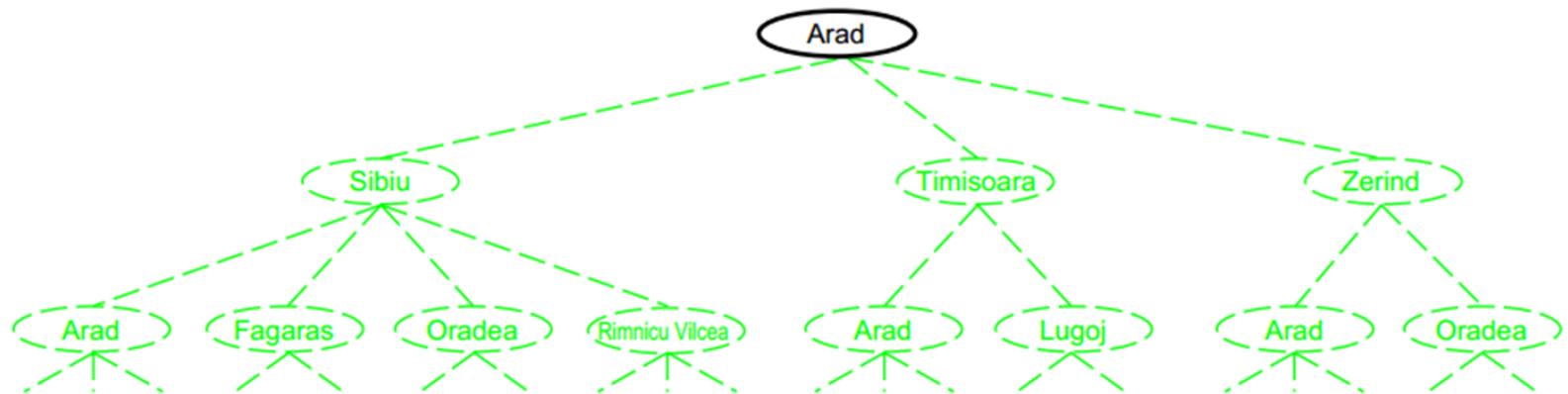
Pembagian Administratif



Algoritma Pencarian Dasar

□ Tree search algorithms (Basic idea) :

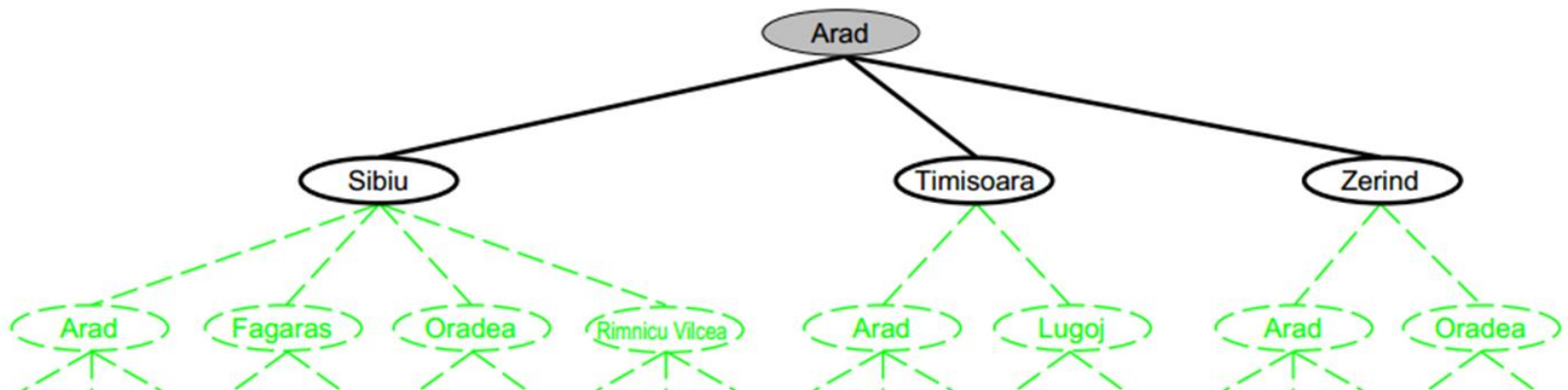
- Mulai dari root node (Arad) sebagai **current node**.
- Lakukan node expansion terhadapnya.
- Pilih salah satu node yang di-expand sebagai current node yang baru. Ulangi langkah sebelumnya.



Algoritma Pencarian Dasar

□ Tree search algorithms (Basic idea) :

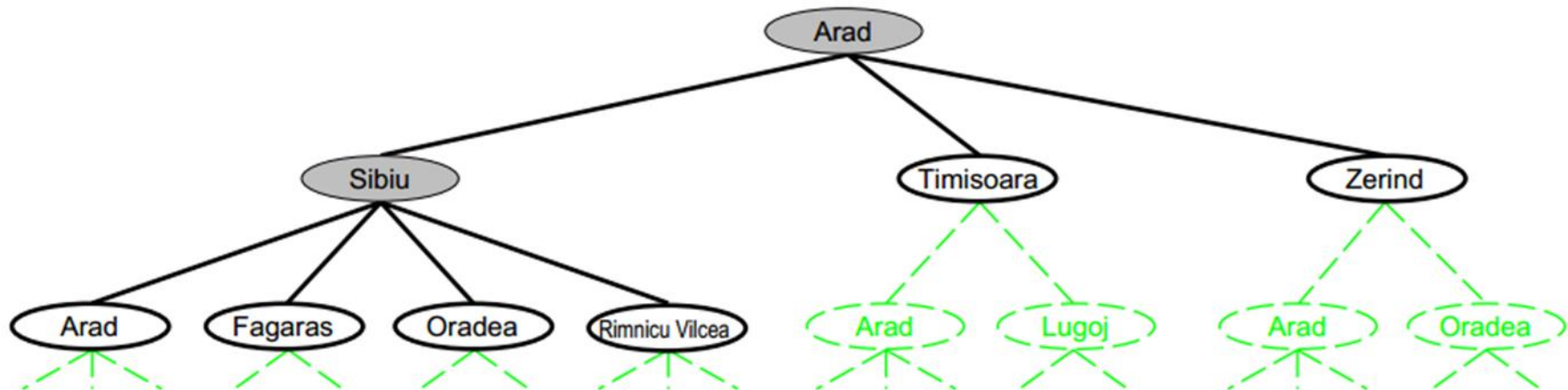
- Mulai dari root node (Arad) sebagai **current node**.
- Lakukan **node expansion** terhadapnya.
- Pilih salah satu node yang di-expand sebagai current node yang baru. Ulangi langkah sebelumnya.



Algoritma Pencarian Dasar

□ Tree search algorithms (Basic idea) :

- Mulai dari root node (Arad) sebagai **current node**.
- Lakukan **node expansion** terhadapnya.
- Pilih salah satu node yang di-expand sebagai **current node yang baru**. Ulangi langkah sebelumnya.



Algoritma Pencarian Dasar

□ Tree search algorithms (Basic idea) :

1. Pada awalnya, fringe = himpunan node yang mewakili initial state.
2. Pilih satu node dari fringe sebagai **current node** (Kalau fringe kosong, selesai dengan gagal).
3. Jika node tsb. lolos goal test, selesai dengan sukses!
4. Jika tidak, lakukan **node expansion** terhadap current node tsb. Tambahkan semua node yang dihasilkan ke fringe.
5. Ulangi langkah 2.

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Algoritma Pencarian Dasar

□ Tree search algorithms (Basic idea) :

1. Pada awalnya, fringe = himpunan node yang mewakili initial state.
2. Pilih satu node dari fringe sebagai **current node** (Kalau fringe kosong, selesai dengan gagal).
3. Jika node tsb. lolos goal test, selesai dengan sukses!
4. Jika tidak, lakukan **node expansion** terhadap current node tsb. Tambahkan semua node yang dihasilkan ke fringe.
5. Ulangi langkah 2.

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Algoritma Pencarian Dasar

□ Tree search algorithms (Basic idea) :

1. Pada awalnya, fringe = himpunan node yang mewakili initial state.
2. Pilih satu node dari fringe sebagai **current node** (Kalau fringe kosong, selesai dengan gagal).
3. Jika node tsb. lolos goal test, selesai dengan sukses!
4. Jika tidak, lakukan **node expansion** terhadap current node tsb. Tambahkan semua node yang dihasilkan ke fringe.
5. Ulangi langkah 2.

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    fringe ← INSERTALL(EXPAND(node, problem), fringe)
```


Algoritma Pencarian Dasar

□ Tree search algorithms (Basic idea) :

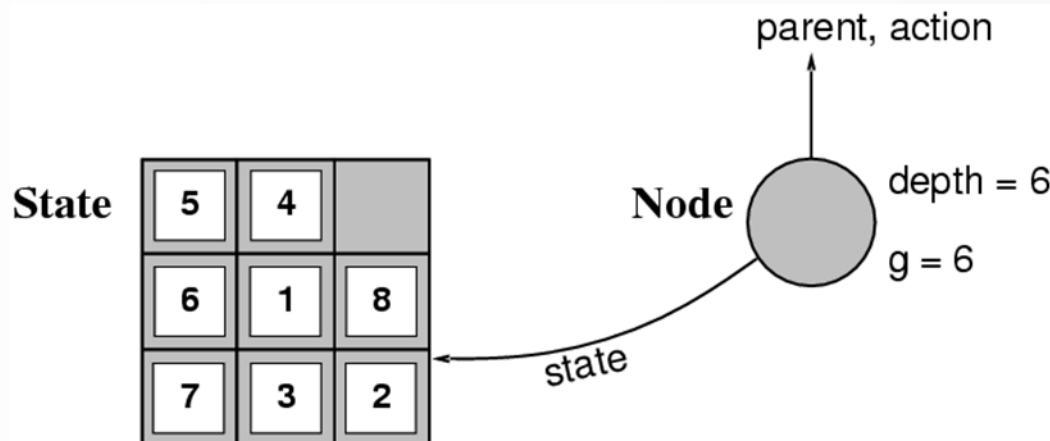
1. Pada awalnya, fringe = himpunan node yang mewakili initial state.
2. Pilih satu node dari fringe sebagai **current node** (Kalau fringe kosong, selesai dengan gagal).
3. Jika node tsb. lolos goal test, selesai dengan sukses!
4. Jika tidak, lakukan **node expansion** terhadap current node tsb. Tambahkan semua node yang dihasilkan ke fringe.
5. Ulangi langkah 2.

```
function EXPAND(node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node; ACTION[s] ← action; STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Algoritma Pencarian Dasar

❑ Implementation: states vs. nodes

- Sebuah **state** merepresentasikan **abstraksi keadaan nyata** dari masalah.
- Sebuah **node** adalah struktur data yang menjadi **bagian dari search tree**.
- State tidak memiliki parent, children, depth, path cost!
- Node = state pada path tertentu. Dua node berbeda bisa mewakili state yang sama!



Strategi Pencarian Uninformed

❑ Strategi pencarian :

- Terdapat berbagai jenis strategi untuk melakukan search.
- Semua strategi ini berbeda dalam satu hal: urutan dari node expansion.
- Search strategy di-evaluasi berdasarkan:
 - completeness: apakah solusi (jika ada) pasti ditemukan?
 - time complexity: jumlah node yang di-expand.
 - space complexity: jumlah maksimum node di dalam memory.
 - optimality: apakah solusi dengan minimum cost pasti ditemukan?
- Time & space complexity diukur berdasarkan
 - b - branching factor dari search tree
 - d - depth (kedalaman) dari solusi optimal
 - m - kedalaman maksimum dari search tree (bisa infinite!)

Strategi Pencarian Uninformed

❑ Strategi Pencarian Uninformed :

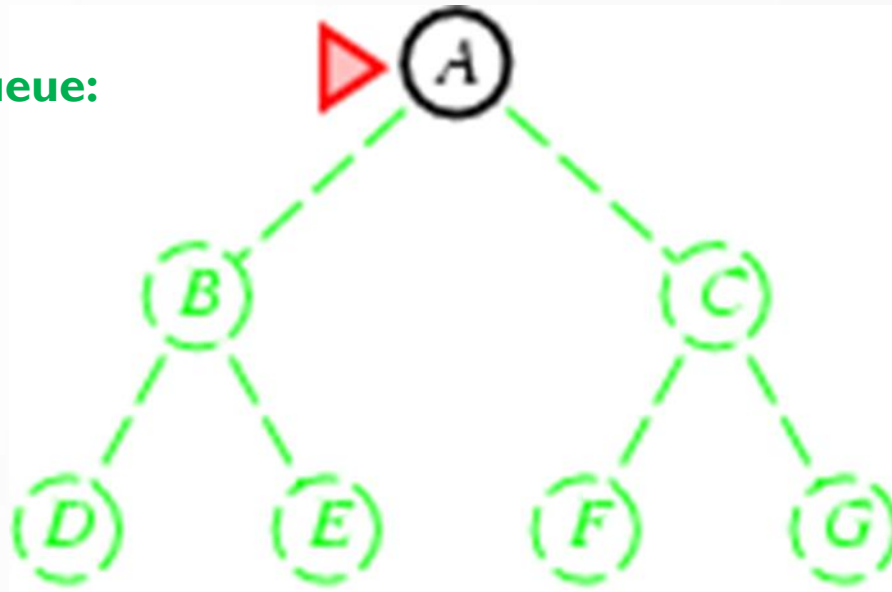
- Uninformed strategy hanya menggunakan informasi dari definisi masalah.
- Bisa diterapkan secara generik terhadap semua jenis masalah yang bisa direpresentasikan dalam sebuah state space.
- Ada beberapa jenis :
 - **Breadth-first search**
 - Uniform-cost search
 - **Depth-first search**
 - Depth-limited search
 - Iterative-deepening search

Breadth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling dekat ke root
 - Implementasi: fringe adalah sebuah queue, data struktur FIFO (First In First Out)
 - Hasil node expansion (successor function) ditaruh di belakang

BFS traversal queue:

a
bc
cde
defg
efg
fg
g

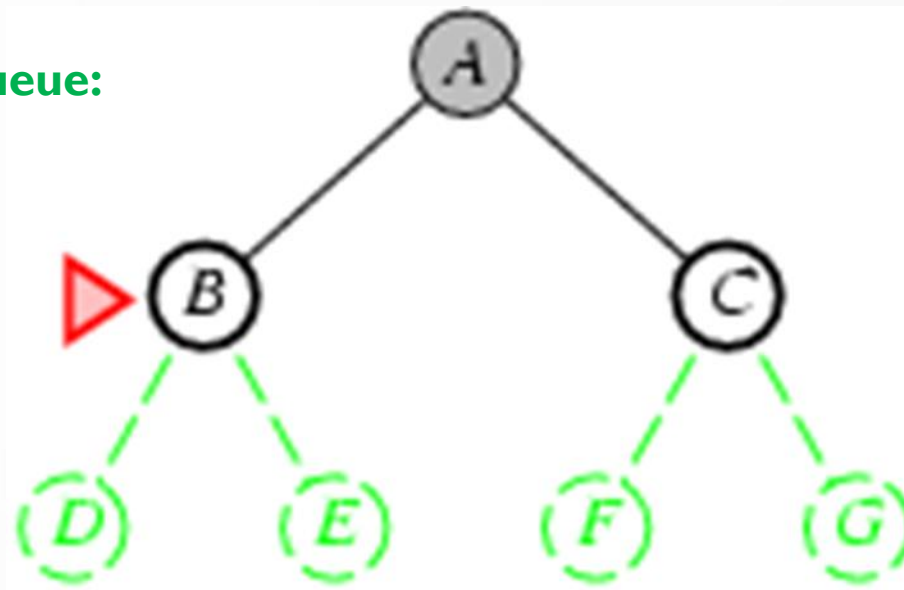


Breadth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling dekat ke root
 - Implementasi: fringe adalah sebuah queue, data struktur FIFO (First In First Out)
 - Hasil node expansion (successor function) ditaruh di belakang

BFS traversal queue:

a
bc
cde
defg
efg
fg
g

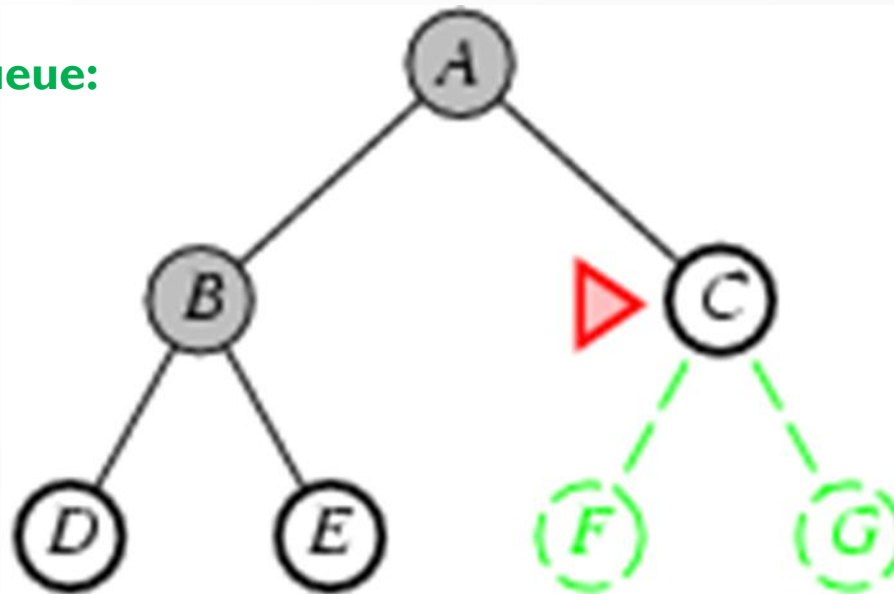


Breadth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling dekat ke root
 - Implementasi: fringe adalah sebuah queue, data struktur FIFO (First In First Out)
 - Hasil node expansion (successor function) ditaruh di belakang

BFS traversal queue:

a
bc
cde
defg
efg
fg
g

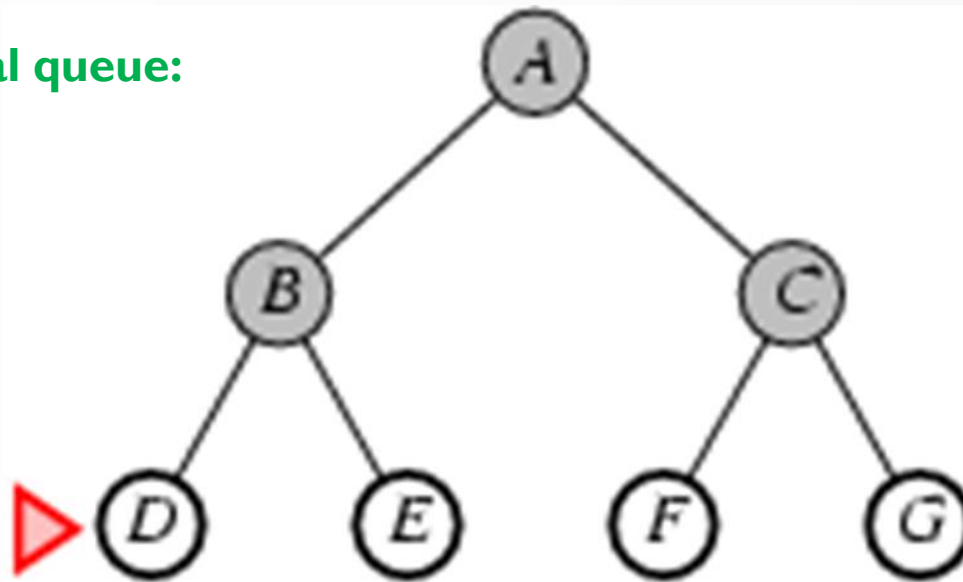


Breadth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling dekat ke root
 - Implementasi: fringe adalah sebuah queue, data struktur FIFO (First In First Out)
 - Hasil node expansion (successor function) ditaruh di belakang

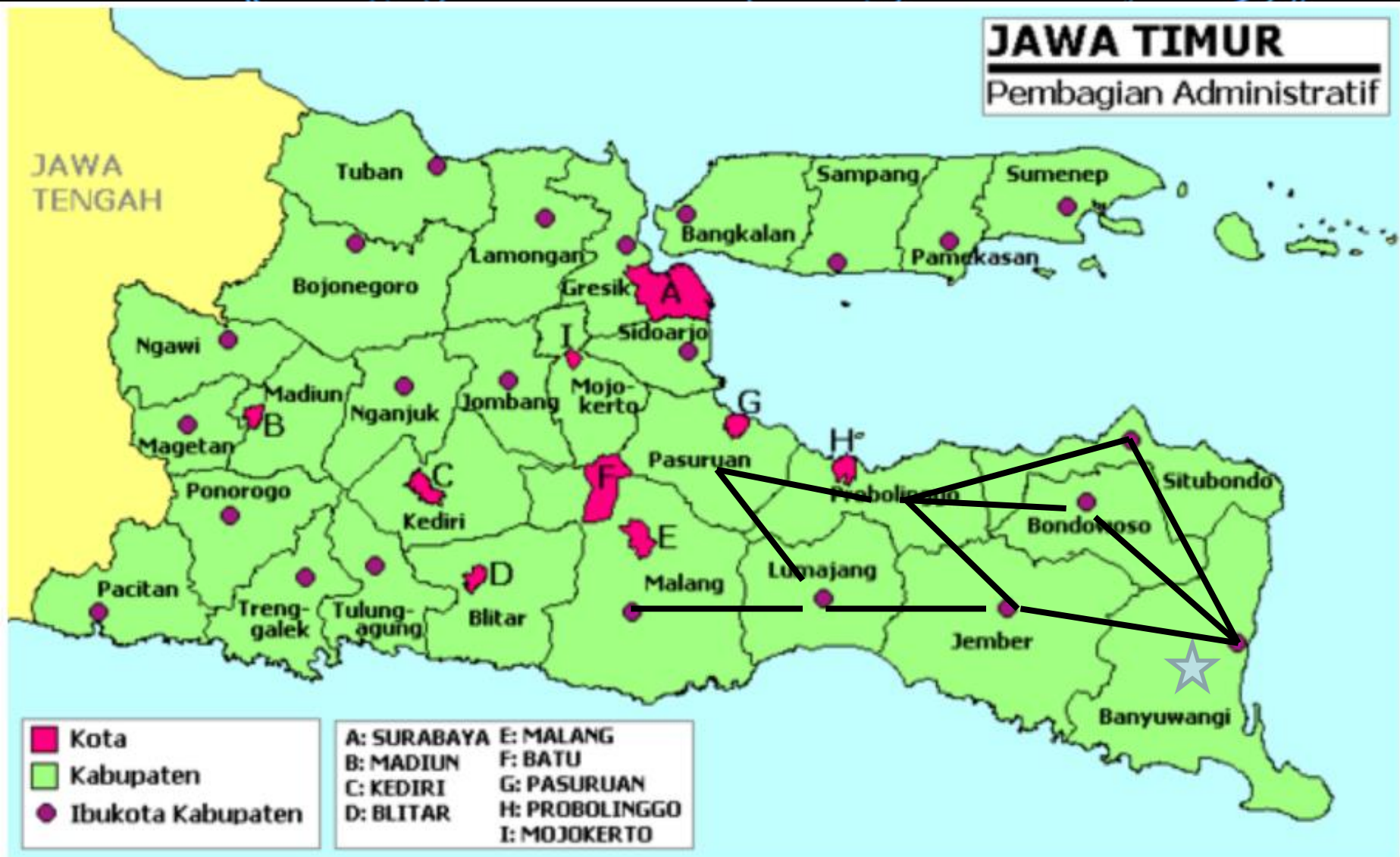
BFS traversal queue:

a
bc
cde
defg
efg
fg
g



JAWA TIMUR

Pembagian Administratif



Breadth-First Search dengan Python

```
graph = {'A':['B','C'],  
        'B':['D','E'],'C':['F','G'],  
        'D':['H','I'],'E':['J','K'],'F':  
        :['L','M'],'G':['N','O'],'H':  
        [],'I':[],'J':[],'K':[],'L':[],'M':  
        :[],'N':[],'O':[]}
```

```
visited = []
```

```
queue = []
```

```
def bfs(visited, graph, node):
```

```
    visited.append(node)
```

```
    queue.append(node)
```

```
    while queue:
```

```
        s = queue.pop(0)
```

```
        print(s,end="")
```

```
        for neighbour in graph[s]:
```

```
            if neighbour not in visited:
```

```
                visited.append(neighbour)
```

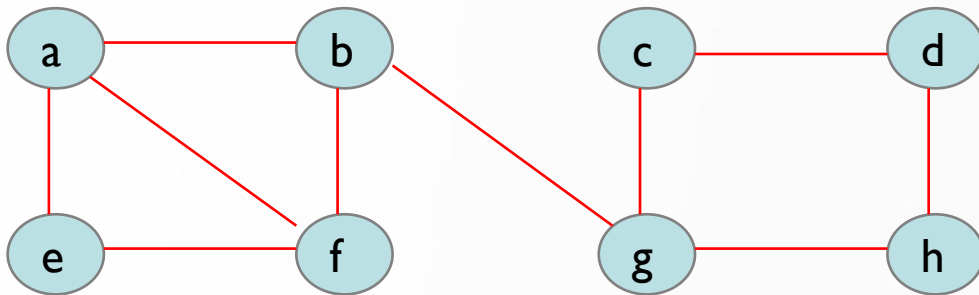
```
                queue.append(neighbour)
```

```
    bfs(visited, graph, 'A')
```

```
In [15]: runfile('C:/Users/Satrio Hadi Wijoyo/Documents/Ganjil 2020-2021/MK  
Pengantar Sains Data/cobaalgobfs.py', wdir='C:/Users/Satrio Hadi Wijoyo/  
Documents/Ganjil 2020-2021/MK Pengantar Sains Data')  
ABCDEFGHIJKLMNO
```


Breadth-First Search

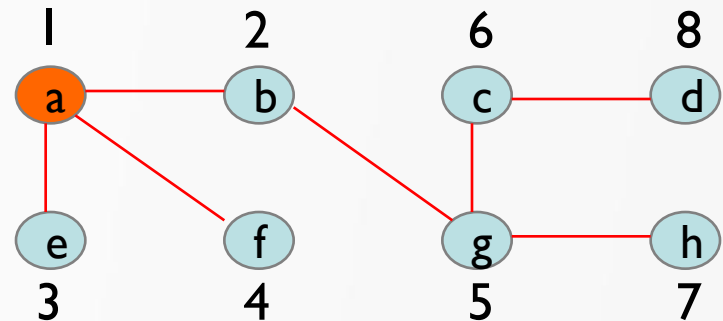
- ❑ BFS menggunakan prinsip queue (Contoh lain)



BFS traversal queue:

a
bef
efg
fg
g
ch
hd
d

BFS tree:



Breadth-First Search

□ Properties of breadth-first search :

- Complete? Ya, jika b terbatas
- Time complexity? $b + b^2 + b^3 + \dots + b^d = O(b^{d+1} - 1)$ → eksponensial dalam d.
- Space complexity? $O(b^{d+1} - 1)$ karena semua node yang di-generate harus disimpan.
- Optimal? Ya, jika semua step cost sama, tapi pada umumnya tidak optimal.
- Masalah utama breadth-first search adalah **space** :
 - Mis: 1 node memakan 1000 byte, dan $b = 10$
 - Jika $d = 7$, ada 10^7 node ≈ 10 gigabyte.
 - Jika $d = 13$, ada 10^{13} node ≈ 10 petabyte!

Depth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling jauh dari root.
 - Implementasi: fringe adalah sebuah stack, data struktur LIFO (Last In First Out)
 - Hasil node expansion ditaruh di depan
 - Depth-first search sangat cocok diimplementasikan secara rekursif.

DFS traversal stack:

a	abek	acg
ab	abe	acgn
abd	ab	acg
abdh	a	acgo
abd	ac	acg
abdi	acf	ac
abd	acfl	a
ab	acf	null
abe	acfm	
abej	acf	
abe	ac	
...	...	

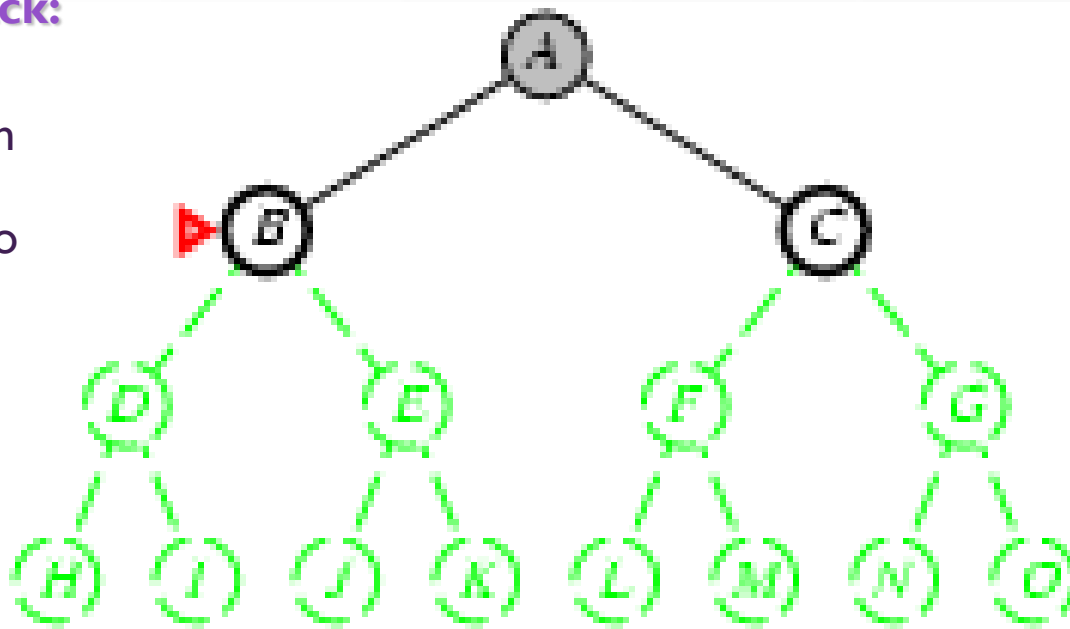


Depth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling jauh dari root.
 - Implementasi: fringe adalah sebuah stack, data struktur LIFO (Last In First Out)
 - Hasil node expansion ditaruh di depan
 - Depth-first search sangat cocok diimplementasikan secara rekursif.

DFS traversal stack:

a	abek	acg
ab	abe	acgn
abd	ab	acg
abdh	a	acgo
abd	ac	acg
abdi	acf	ac
abd	acfl	a
ab	acf	null
abe	acfm	
abej	acf	
abe	ac	
...	...	



Depth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling jauh dari root.
 - Implementasi: fringe adalah sebuah stack, data struktur LIFO (Last In First Out)
 - Hasil node expansion ditaruh di depan
 - Depth-first search sangat cocok diimplementasikan secara rekursif.

DFS traversal stack:

a	abek	acg
ab	abe	acgn
abd	ab	acg
abdh	a	acgo
abd	ac	acg
abdi	acf	ac
abd	acfl	a
ab	acf	null
abe	acfm	
abej	acf	
abe	ac	
...	...	



Depth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling jauh dari root.
 - Implementasi: fringe adalah sebuah stack, data struktur LIFO (Last In First Out)
 - Hasil node expansion ditaruh di depan
 - Depth-first search sangat cocok diimplementasikan secara rekursif.

DFS traversal stack:

a	abek	acg
ab	abe	acgn
abd	ab	acg
abdh	a	acgo
abd	ac	acg
abdi	acf	ac
abd	acfl	a
ab	acf	null
abe	acfm	
abej	acf	
abe	ac	
...	...	



Depth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling jauh dari root.
 - Implementasi: fringe adalah sebuah stack, data struktur LIFO (Last In First Out)
 - Hasil node expansion ditaruh di depan
 - Depth-first search sangat cocok diimplementasikan secara rekursif.

DFS traversal stack:

a	abek	acg
ab	abe	acgn
abd	ab	acg
abdh	a	acgo
abd	ac	acg
abdi	acf	ac
abd	acfl	a
ab	acf	null
abe	acfm	
abej	acf	
abe	ac	
...	...	



Depth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling jauh dari root.
 - Implementasi: fringe adalah sebuah stack, data struktur LIFO (Last In First Out)
 - Hasil node expansion ditaruh di depan
 - Depth-first search sangat cocok diimplementasikan secara rekursif.

DFS traversal stack:

a	abek	acg
ab	abe	acgn
abd	ab	acg
abdh	a	acgo
abd	ac	acg
abdi	acf	ac
abd	acfl	a
ab	acf	null
abe	acfm	
abej	acf	
abe	ac	
...	...	

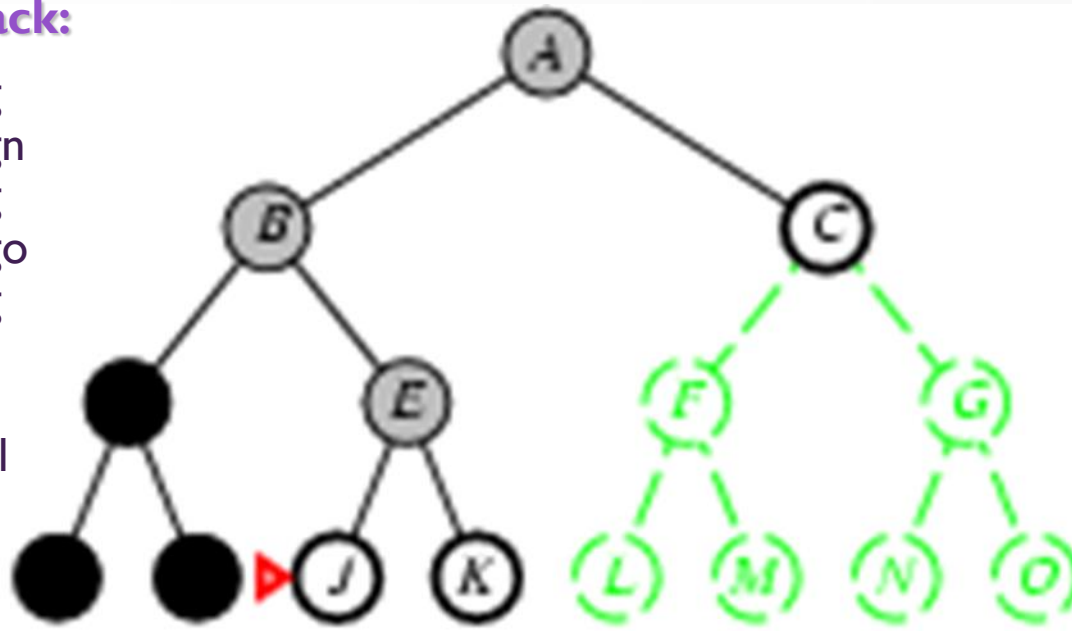


Depth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling jauh dari root.
 - Implementasi: fringe adalah sebuah stack, data struktur LIFO (Last In First Out)
 - Hasil node expansion ditaruh di depan
 - Depth-first search sangat cocok diimplementasikan secara rekursif.

DFS traversal stack:

a	abek	acg
ab	abe	acgn
abd	ab	acg
abdh	a	acgo
abd	ac	acg
abdi	acf	ac
abd	acfl	a
ab	acf	null
abe	acfm	
abej	acf	
abe	ac	
...	...	

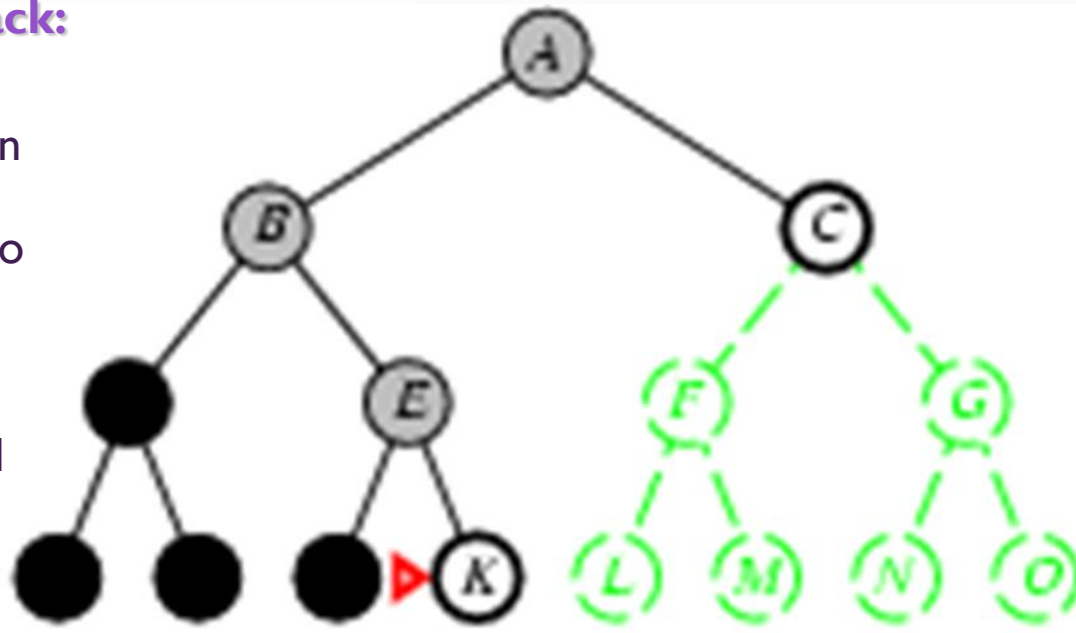


Depth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling jauh dari root.
 - Implementasi: fringe adalah sebuah stack, data struktur LIFO (Last In First Out)
 - Hasil node expansion ditaruh di depan
 - Depth-first search sangat cocok diimplementasikan secara rekursif.

DFS traversal stack:

a	abek	acg
ab	abe	acgn
abd	ab	acg
abdh	a	acgo
abd	ac	acg
abdi	acf	ac
abd	acfl	a
ab	acf	null
abe	acfm	
abej	acf	
abe	ac	
...	...	

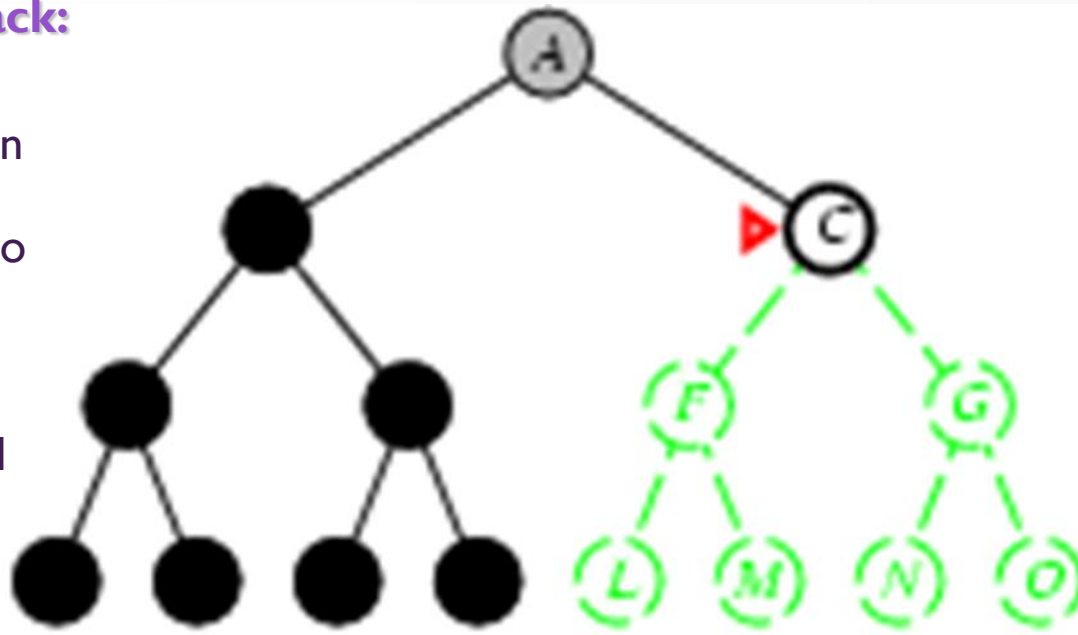


Depth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling jauh dari root.
 - Implementasi: fringe adalah sebuah stack, data struktur LIFO (Last In First Out)
 - Hasil node expansion ditaruh di depan
 - Depth-first search sangat cocok diimplementasikan secara rekursif.

DFS traversal stack:

a	abek	acg
ab	abe	acgn
abd	ab	acg
abdh	a	acgo
abd	ac	acg
abdi	acf	ac
abd	acfl	a
ab	acf	null
abe	acfm	
abej	acf	
abe	ac	
...	...	

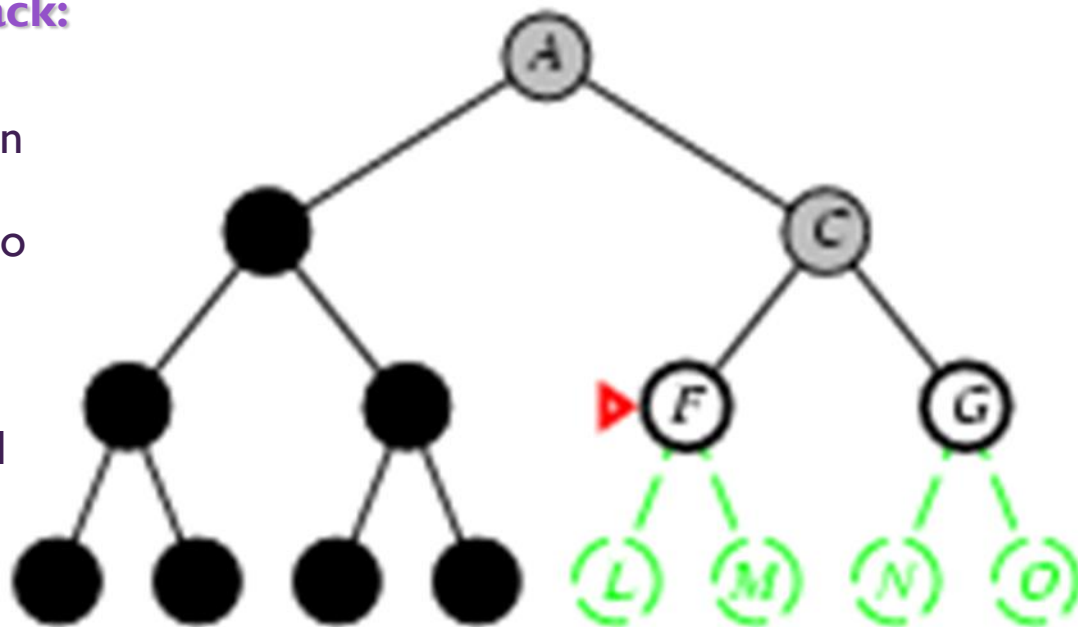


Depth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling jauh dari root.
 - Implementasi: fringe adalah sebuah stack, data struktur LIFO (Last In First Out)
 - Hasil node expansion ditaruh di depan
 - Depth-first search sangat cocok diimplementasikan secara rekursif.

DFS traversal stack:

a	abek	acg
ab	abe	acgn
abd	ab	acg
abdh	a	acgo
abd	ac	acg
abdi	acf	ac
abd	acfl	a
ab	acf	null
abe	acfm	
abej	acf	
abe	ac	
...	...	

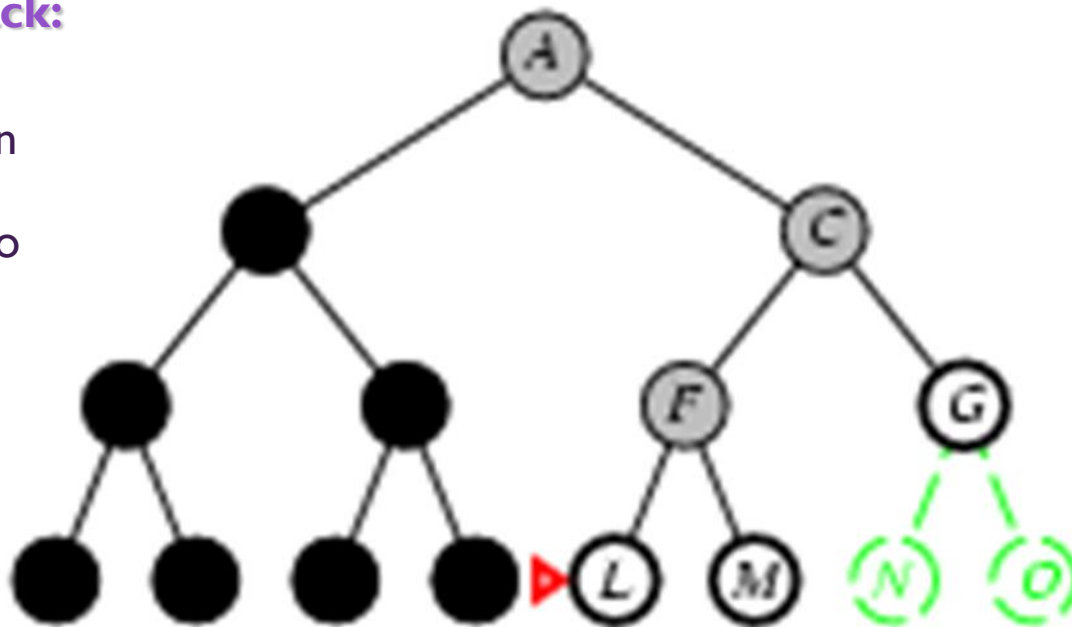


Depth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling jauh dari root.
 - Implementasi: fringe adalah sebuah stack, data struktur LIFO (Last In First Out)
 - Hasil node expansion ditaruh di depan
 - Depth-first search sangat cocok diimplementasikan secara rekursif.

DFS traversal stack:

a	abek	acg
ab	abe	acgn
abd	ab	acg
abdh	a	acgo
abd	ac	acg
abdi	acf	ac
abd	acfl	a
ab	acf	null
abe	acfm	
abej	acf	
abe	ac	
...	...	

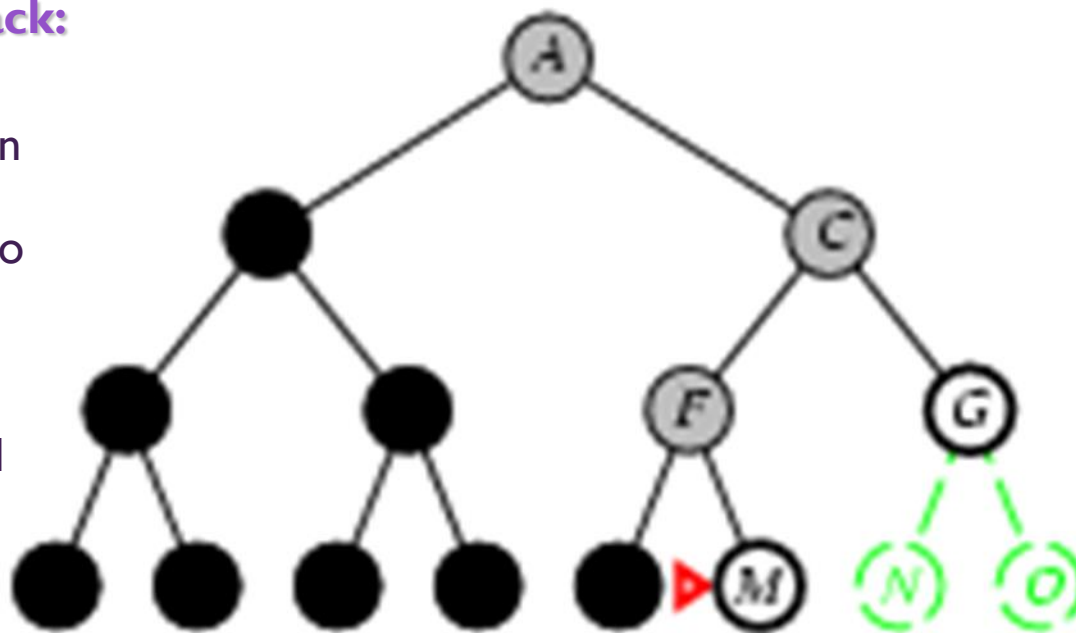


Depth-First Search

- ❑ Lakukan node expansion terhadap node di fringe yang paling jauh dari root.
 - Implementasi: fringe adalah sebuah stack, data struktur LIFO (Last In First Out)
 - Hasil node expansion ditaruh di depan
 - Depth-first search sangat cocok diimplementasikan secara rekursif.

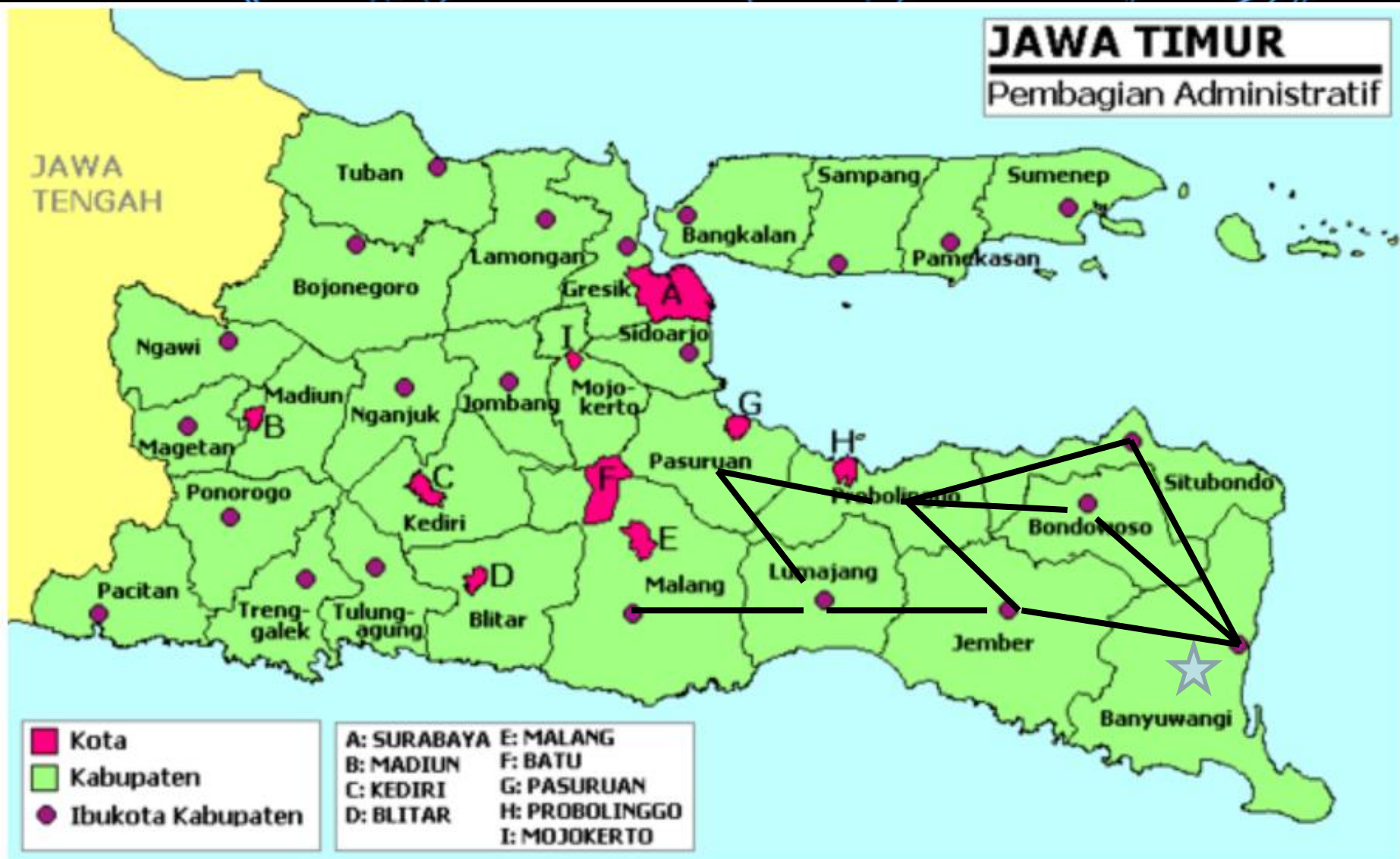
DFS traversal stack:

a	abek	acg
ab	abe	acgn
abd	ab	acg
abdh	a	acgo
abd	ac	acg
abdi	acf	ac
abd	acfl	a
ab	acf	null
abe	acfm	
abej	acf	
abe	ac	
...	...	



JAWA TIMUR

Pembagian Administratif



Depth-First Search dengan Python

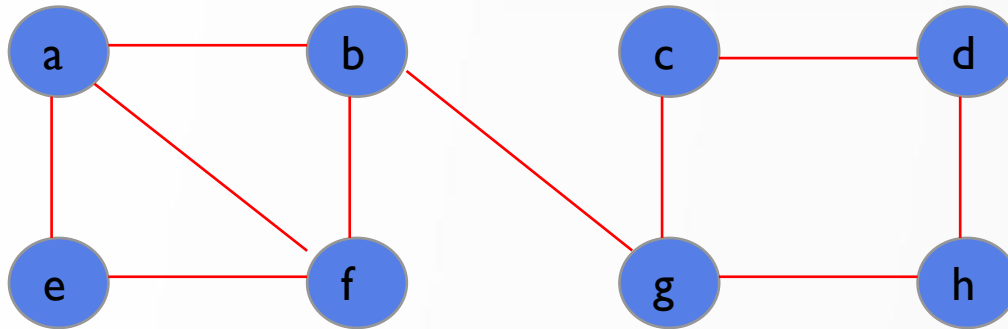
```
graph = {'A':['B','C'],  
        'B':['D','E'],'C':['F','G'],  
        'D':['H','I'],'E':['J','K'],'F':  
        :['L','M'],'G':['N','O'],'H':  
        [], 'I':[], 'J':[], 'K':[], 'L':[], 'M':  
        :[], 'N':[], 'O':[]}
```

```
visited = set()  
  
def dfs(visited, graph, node):  
    if node not in visited:  
        print(node, end="")  
        visited.add(node)  
        for neighbour in graph[node]:  
            dfs(visited, graph, neighbour)  
  
dfs(visited, graph, 'A')
```

```
In [16]: runfile('C:/Users/Satrio Hadi Wijoyo/Documents/Ganjil 2020-2021/MK  
Pengantar Sains Data/cobaalgodfs.py', wdir='C:/Users/Satrio Hadi Wijoyo/  
Documents/Ganjil 2020-2021/MK Pengantar Sains Data')  
ABDHIEJKCFLMGNO
```

Depth-First Search

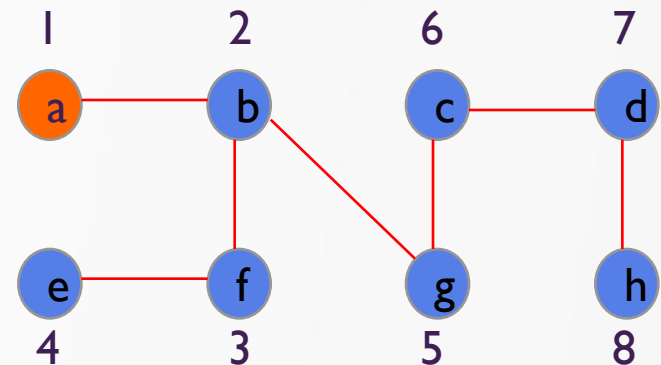
- DFS menggunakan prinsip stack (Contoh lain)



a **DFS traversal stack:**

ab
abf
abfe
abf
ab
abg
abgc
abgcd
abgcdh
abgcd
...

DFS tree:



Depth-First Search

□ Properties of depth-first search

- Complete? Tidak, bisa gagal jika m tak terbatas, atau state space dengan loop . Ya jika M terbatas
- Time complexity? $O(b^m)$ → jika $m \gg d$, sangat lama
- Space complexity? $O(bm)$ → linear space!
- Optimal? Tidak.
- Depth-first search mengatasi masalah **space** :
 - Mis: 1 node memakan 1000 byte, dan $b = 10$
 - Jika $m = 12$ atau $d = 12$, space yang dibutuhkan hanya 118 kilobyte
 - bandingkan dengan 10 petabyte!

Graph Search

- ❑ Solusinya adalah untuk mencatat state mana yang sudah pernah dicoba. Catatan ini disebut **closed list** (fringe = **open list**).
- ❑ Modifikasi algoritma **TreeSearch** dengan closed list menjadi **GraphSearch**.

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Latihan Individu

- Ada dua gelas air kapasitas 4 liter dan 3 Liter kosong. Ingin diisi tepat 2 liter masing masing gelas. Tentukan Langkah penyelesaian dengan BFS dan DFS. X = gelas 4 liter , Y = Gelas 3 liter

- State Space

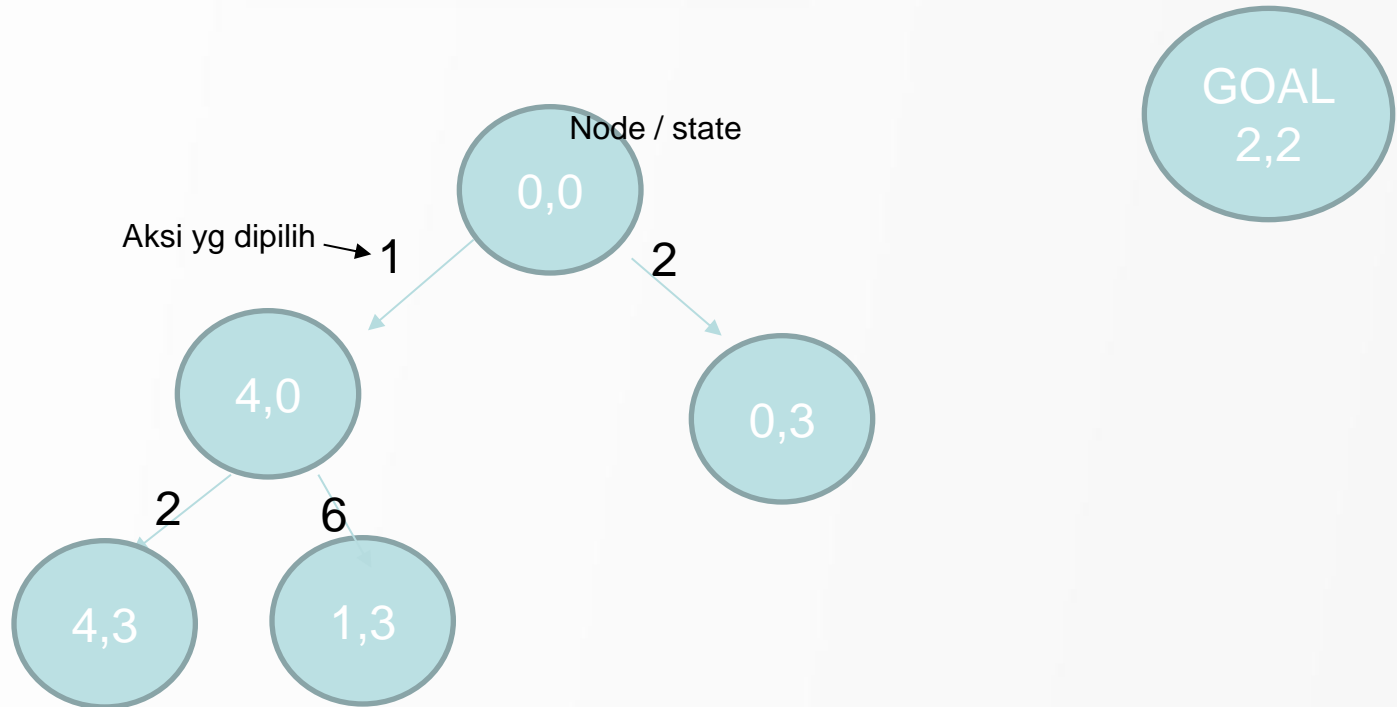
X Y	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)
3	(3,0)	(3,1)	(3,2)	(3,3)
4	(4,0)	(4,1)	(4,2)	(4,3)

- Aksi :

1. Isi Penuh gelas 4 Liter
2. isi Penuh gelas 3 Liter
3. Kosongkan gelas 4 Liter
4. Kosongkan gelas 3 Liter
5. Tuangkan sebagian isi gelas 3 Liter ke gelas 4 liter hingga gelas 4 liter penuh
6. Tuangkan sebagian isi gelas 4 Liter ke gelas 3 liter hingga gelas 3 liter penuh
7. Tuangkan seluruh isi gelas 3 Liter ke gelas 4 liter
8. Tuangkan seluruh isi gelas 4 Liter ke gelas 3 liter

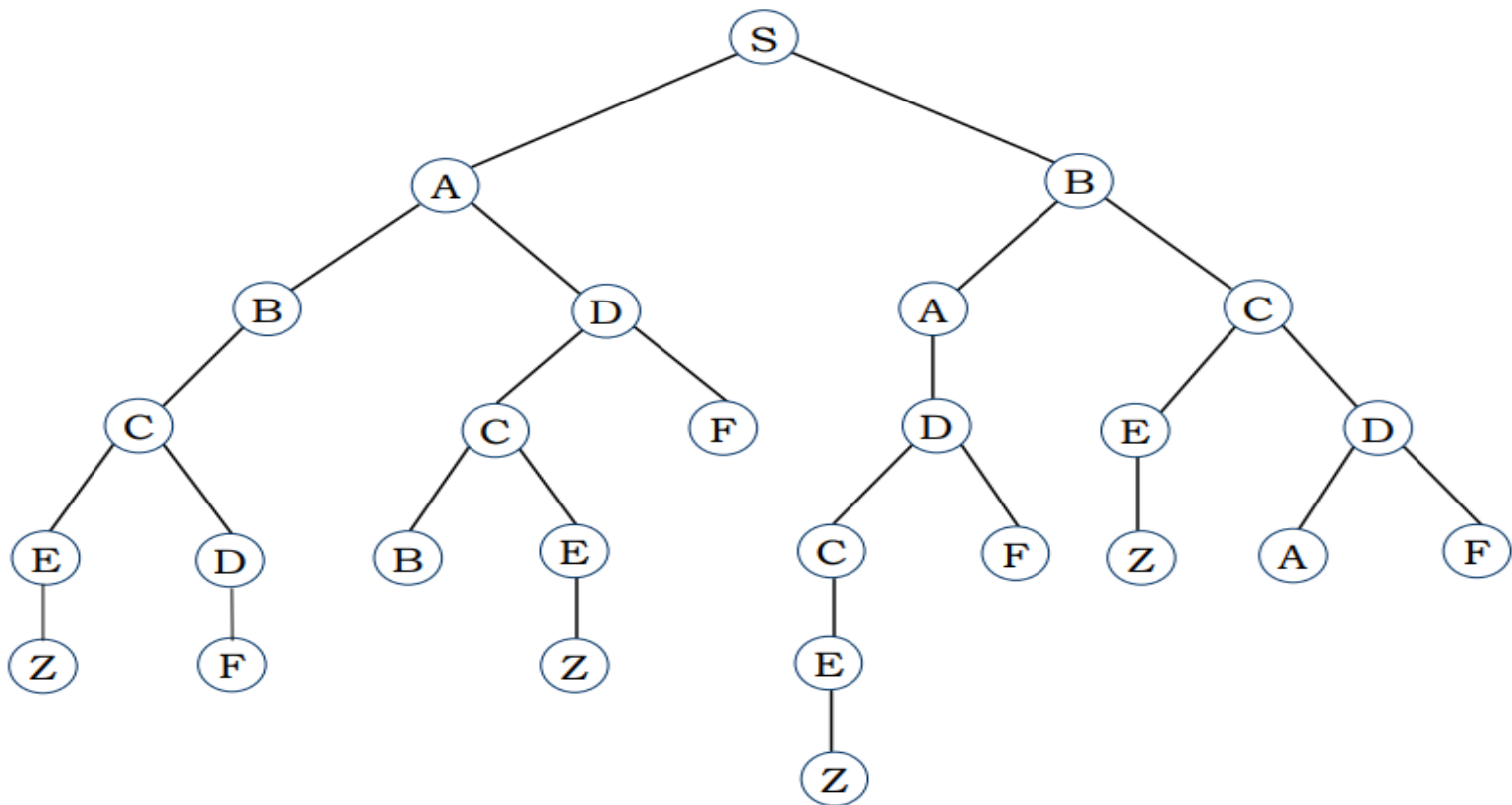
Contoh Penyelesaian

Lanjutkan hingga menemui solusi



Soal Latihan

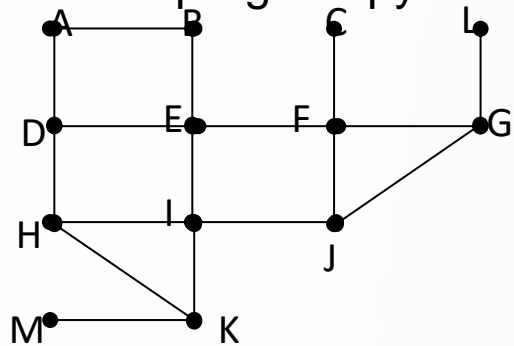
1. Selesaikan graf di bawah ini menggunakan algoritma DFS dan BFS berikut dengan hasil setiap langkah (Traversal dan Tree) dimulai dari node S ke tujuan node Z ! Lebih optimal mana DFS atau BFS ?



Gambar 4.4 Struktur Tree dari Graph Gambar 4.3

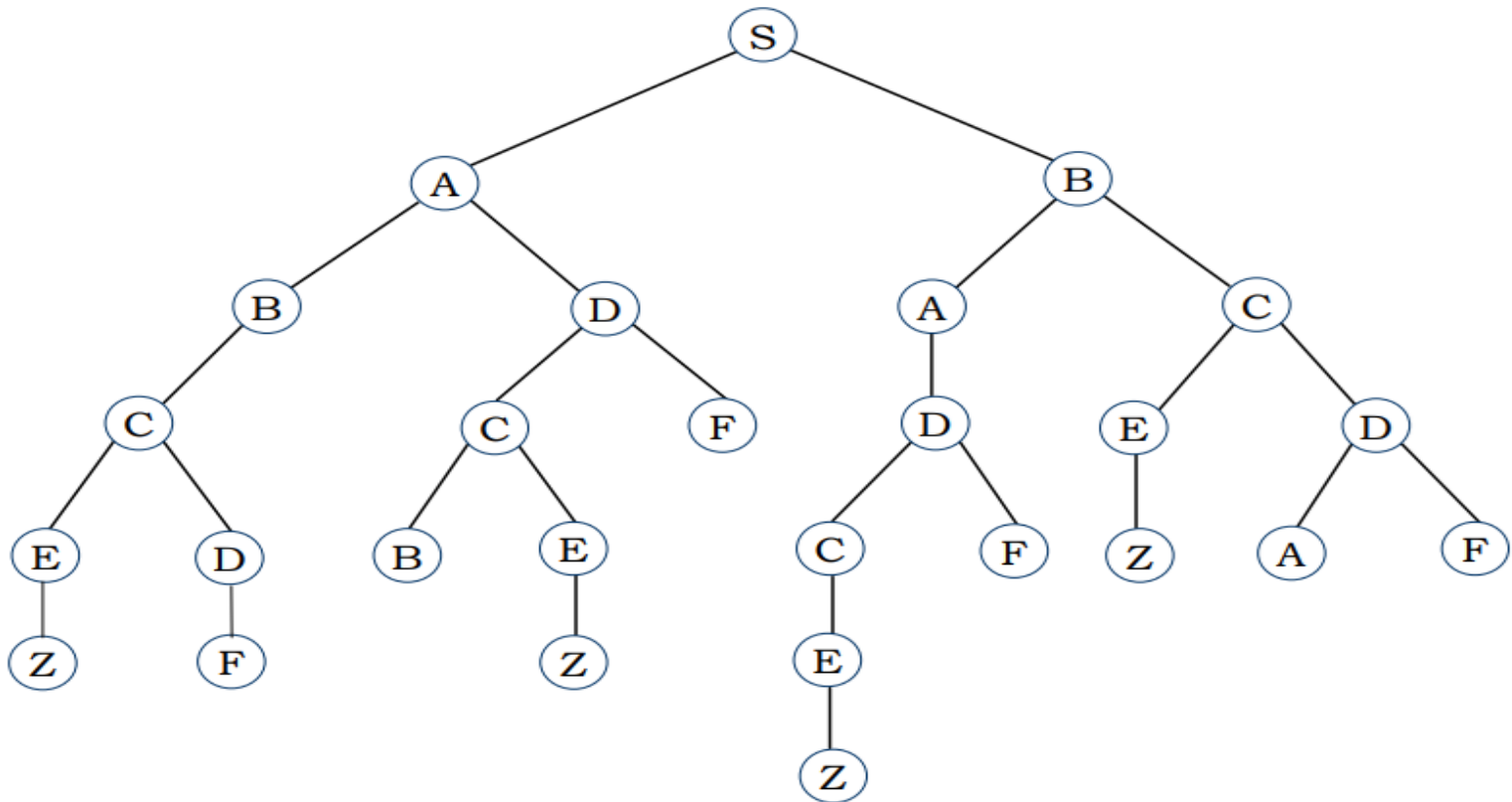
Tugas Individu 01

1. Selesaikan graf di bawah ini menggunakan algoritma DFS dan BFS berikut dengan hasil setiap langkah (Traversal dan Tree) dimulai dari node K! buat program python.

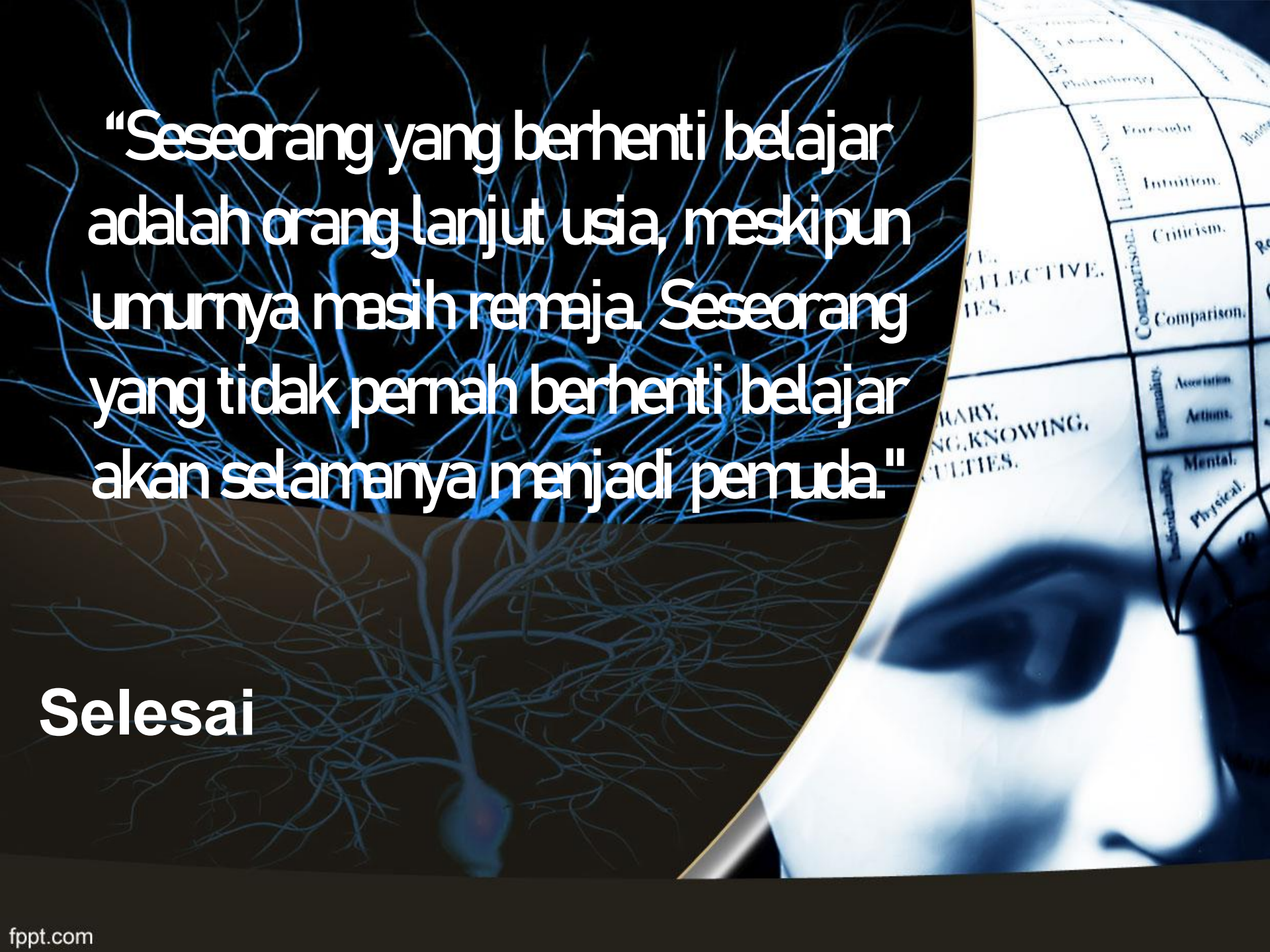


Tugas Individu 2

2. Selesaikan graf di bawah ini menggunakan algoritma DFS dan BFS berikut dengan hasil setiap langkah (Traversal dan Tree) dimulai dari node S ke tujuan node Z dalam bentuk kode program python



Gambar 4.4 Struktur Tree dari Graph Gambar 4.3



"Seseorang yang berhenti belajar
adalah orang lanjut usia, meskipun
umurnya masih remaja. Seseorang
yang tidak pernah berhenti belajar
akan selamanya menjadi pemuda."

Selesai