

# 10 - DFS and BFS

[KOMS120403]

Desain dan Analisis Algoritma (2022/2023)

Dewi Sintari

Prodi S1 Ilmu Komputer  
Universitas Pendidikan Ganesha

Week 13 (May 2023)

# Daftar isi

- Algoritma traversal graf
- DFS
- BFS
- Graf dinamis

# Traversal graf

Algoritma **traversal graf** adalah algoritma yang mencari solusi permasalahan pada sebuah *struktur data graf*, dengan mengunjungi simpul-simpul pada graf secara sistematis (dengan asumsi bahwa graf tersebut *terhubung*).

- Depth first search (DFS)
- Breadth first search (BFS)

# Struktur data graf

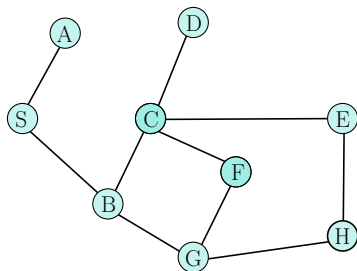
## Adjacency matrix (matriks ketetanggaan)

Sebuah **adjacency matrix** adalah matriks biner  $n \times n$  dimana nilai entri ke- $[i, j]$  adalah 1 jika dan hanya jika terdapat sisi yang memiliki simpul akhir  $i$  dan  $j$

## Adjacency list (daftar ketetanggaan)

**Adjacency list** adalah array dari daftar terpisah. Setiap elemen array adalah daftar simpul tetangga (atau terhubung langsung) yang sesuai. Dengan kata lain, daftar ke- $i$  dari adjacency list adalah daftar semua simpul yang terhubung langsung ke simpul ke- $i$ .

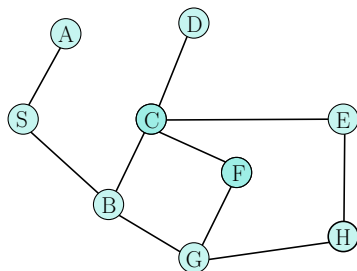
# Struktur data graf



	A	B	C	D	E	F	G	H	S
A	0	0	0	0	0	0	0	0	1
B	0	0	0	0	0	0	0	0	1
C	0	1	0	1	1	1	0	0	0
D	0	0	1	0	0	0	0	0	0
E	0	0	1	0	0	0	0	1	0
F	0	0	1	0	0	0	0	0	0
G	0	1	0	0	0	1	0	1	0
H	0	0	0	0	1	0	1	0	0
S	1	1	0	0	0	0	0	0	0

Figure: Graf dan matriks ketetanggaannya

# Struktur data graf



S: [A, B]  
A: [S]  
B: [S, C, G]  
C: [B, D, E, F]  
D: [C]  
E: [C, H]  
F: [C, G]  
G: [B, F, H]

**Adjacency list:** [[A,B], [S], [S,C,G], [B,D,E,F], [C], [C,H], [C,G], [B,F,H]]

**Figure:** Matriks dan list ketetanggaannya

# Representasi graf dalam proses pencarian

Dua pendekatan dalam proses pencarian solusi

- 1 **Graf statis:** graf dibangun *sebelum* proses pencarian. Graf direpresentasikan sebagai struktur data.
  - ▶ Contoh: BFS, DFS
- 2 **Graf dinamis:** graf dibangun bersama dengan proses pencarian.

# Bagian 1. Depth-First Search (DFS)



## DFS (1): Algoritma

DFS dimulai dari *simpul akar* dan memeriksa semua simpul tetangga.

- Kunjungi simpul  $v$ ;
- Kunjungi simpul  $w$  yang bersebelahan dengan  $v$ ;
- Ulangi DFS mulai dari simpul  $w$ ;
- Ketika simpul  $u$  tercapai sehingga semua tetangganya dikunjungi, pencarian “mundur” ke simpul yang terakhir dikunjungi yang masih memiliki tetangga yang belum dikunjungi.
- Lanjutkan langkah ini.
- Searching selesai bila tidak ada lagi simpul yang dapat dijangkau dari simpul yang dikunjungi.

# DFS (2): Pseudocode (rekursif)

---

## Algorithm 1 DFS in a graph

---

```
1: procedure DFS( $G$ )
2:   input: graf  $G = (V, E)$ 
3:   output: graf  $G$  dengan  $V(G)$  ditandai dengan bilangan bulat berurutan yang
      menunjukkan urutan DFS
4:    $\text{count} \leftarrow 0$ 
5:   initialize array  $\text{visited} = [ ]$ 
6:   for  $v \in V$  do
7:      $\text{visited}[v] = 0$ 
8:   end for
9:   for  $v \in V$  do
10:    if  $\text{visited}[v] = 0$  then
11:      DFS( $v$ )
12:    end if
13:  end for
14:  return  $\text{visited}$ 
15: end procedure
```

▷ 'count' is used to store the number of vertices that have been visited

▷ to store all vertices that have been visited

▷ vertex  $v$  has not been visited

▷ see next slide for the subroutine

---

## DFS (3): Pseudocode

---

### Algorithm 2 DFS a vertex

---

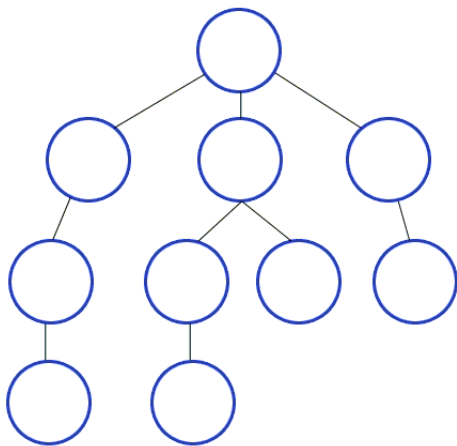
```
1: procedure DFS( $v$ )
2:   count  $\leftarrow$  count + 1
3:   visited[ $v$ ] = count
4:   for  $w \in N(v)$  do
5:     if visited[ $w$ ] = 0 then
6:       DFS( $w$ )
7:     end if
8:   end for
9: end procedure
```

$\triangleright$  vertex  $v$  is now marked as visited

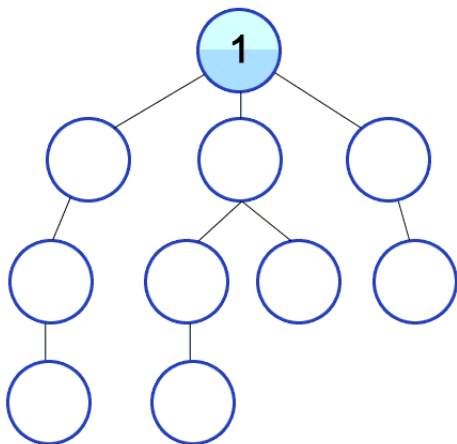
$\triangleright$  if  $w$  is marked as 'not visited', then visit  $v$  and DFS in one of its neighbors

---

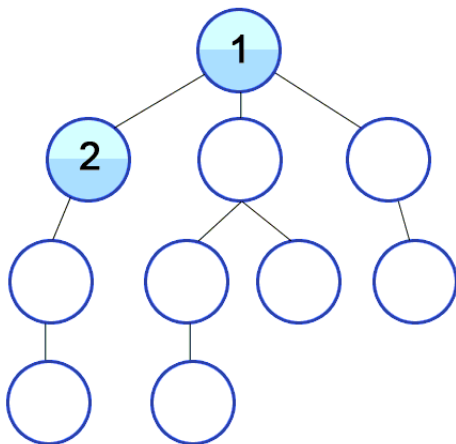
### DFS (4): Contoh pada tree



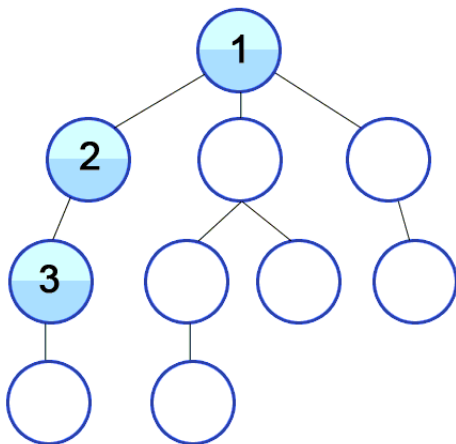
## DFS (4): Contoh pada tree



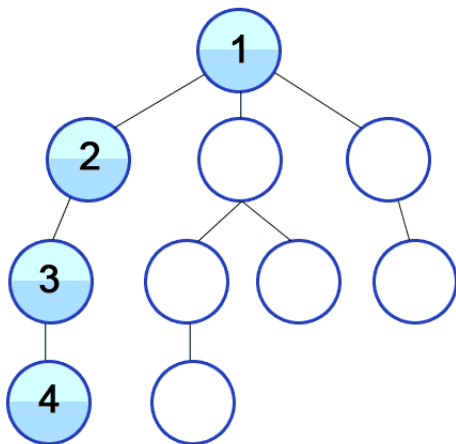
## DFS (4): Contoh pada tree



## DFS (4): Contoh pada tree

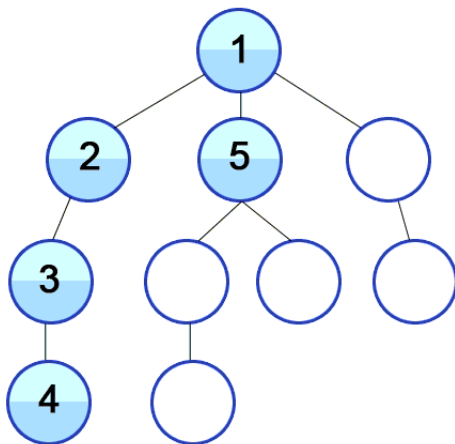


## DFS (4): Contoh pada tree

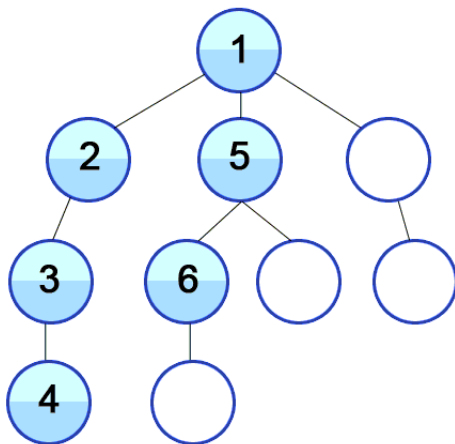




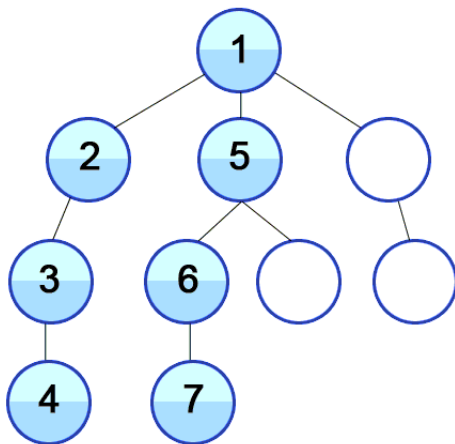
## DFS (4): Contoh pada tree



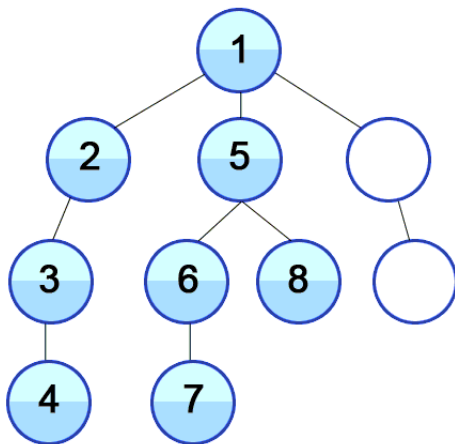
## DFS (4): Contoh pada tree



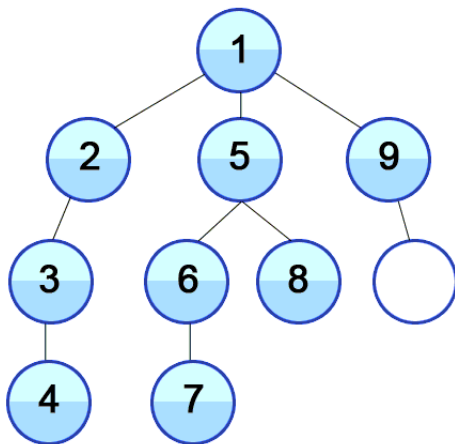
## DFS (4): Contoh pada tree



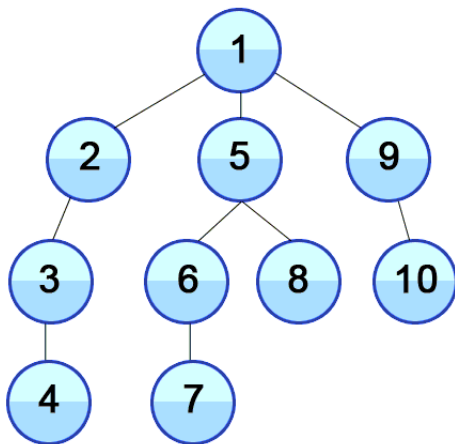
## DFS (4): Contoh pada tree



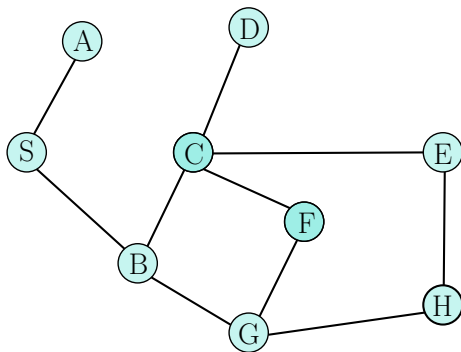
## DFS (4): Contoh pada tree



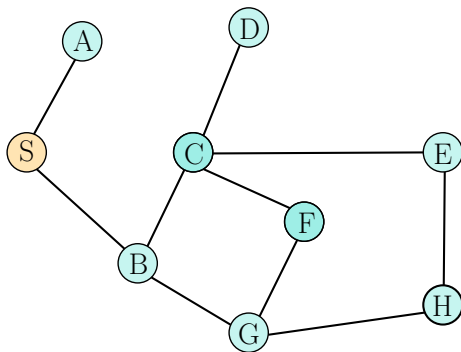
### DFS (4): Contoh pada tree



### DFS (5): Contoh pada sebuah graf

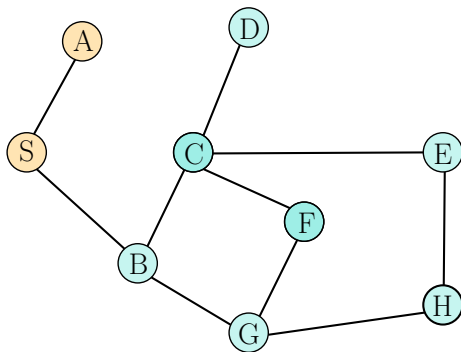


### DFS (5): Contoh pada sebuah graf

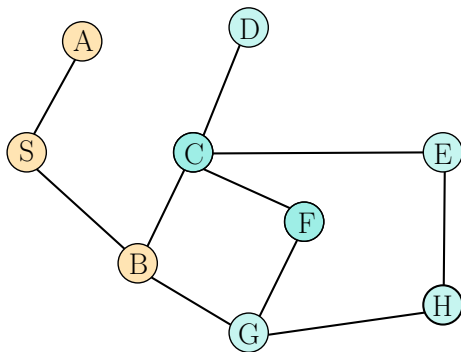




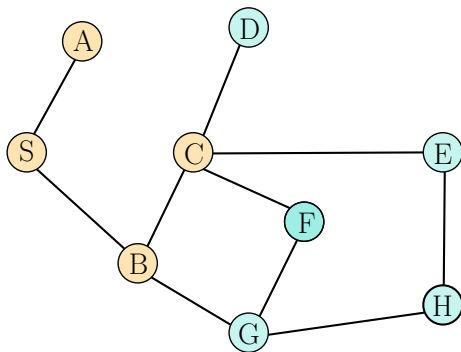
### DFS (5): Contoh pada sebuah graf



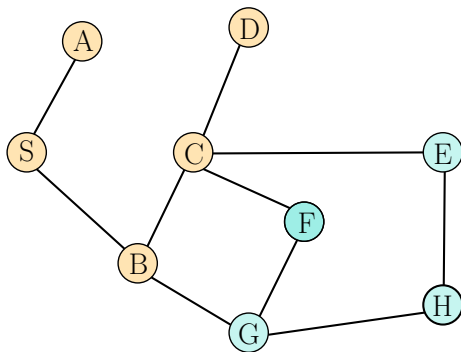
## DFS (5): Contoh pada sebuah graf



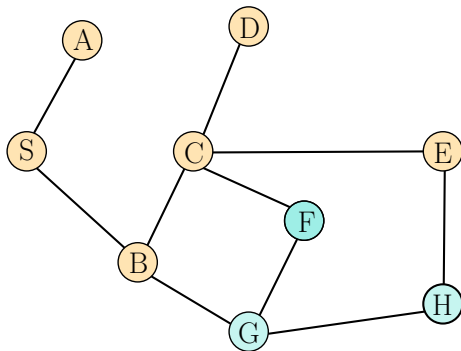
## DFS (5): Contoh pada sebuah graf



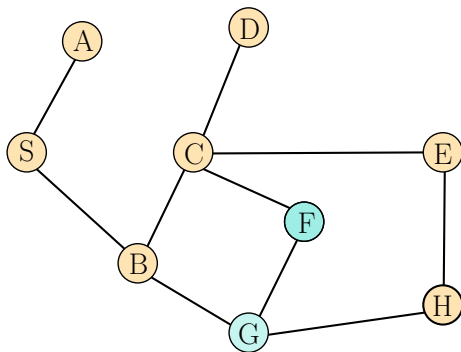
### DFS (5): Contoh pada sebuah graf



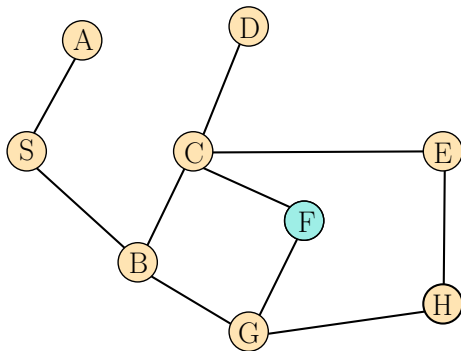
## DFS (5): Contoh pada sebuah graf



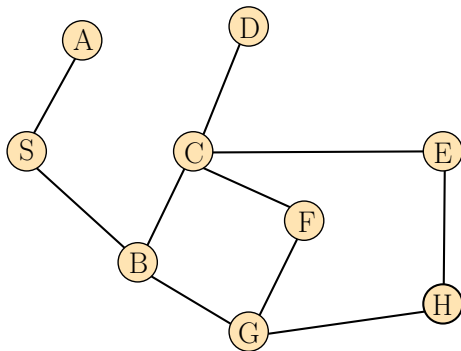
## DFS (5): Contoh pada sebuah graf



## DFS (5): Contoh pada sebuah graf



### DFS (5): Contoh pada sebuah graf





## DFS (6): DFS tree

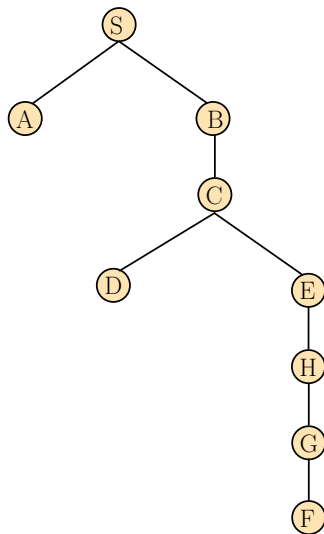


Figure: Tree setelah *running* BFS

# Bagian 2. Breadth-First Search (BFS)

## BFS (1): Algoritma

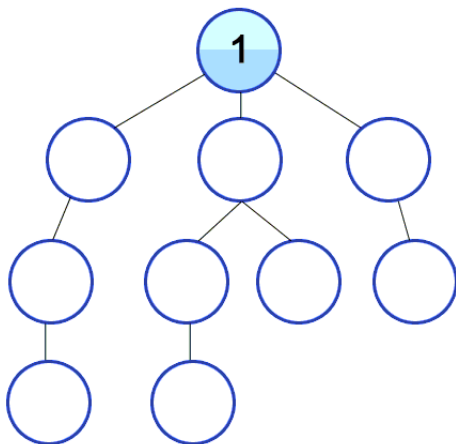
BFS dimulai pada *simpul akar* dan memeriksa semua simpul tetangga.

Kemudian untuk masing-masing simpul tetangga, secara bergiliran, ia memeriksa simpul tetangganya yang belum dikunjungi, dan seterusnya.

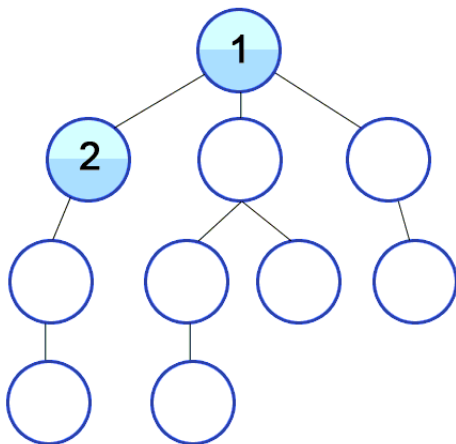
- Kunjungi simpul  $v$ ;
- Kunjungi semua simpul yang berdekatan dengan  $v$ ;
- Kunjungi semua simpul yang belum dikunjungi, dan berdekatan dengan simpul yang baru saja dikunjungi;
- Lanjutkan seperti ini...



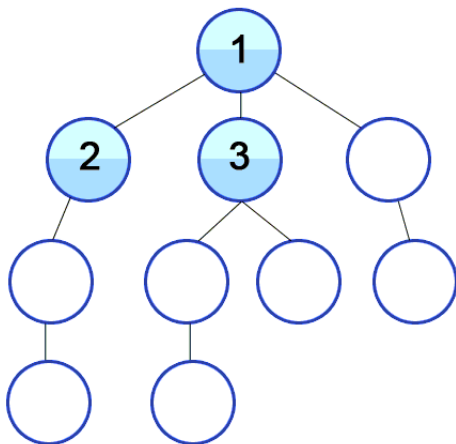
## BFS (2): Contoh pada tree



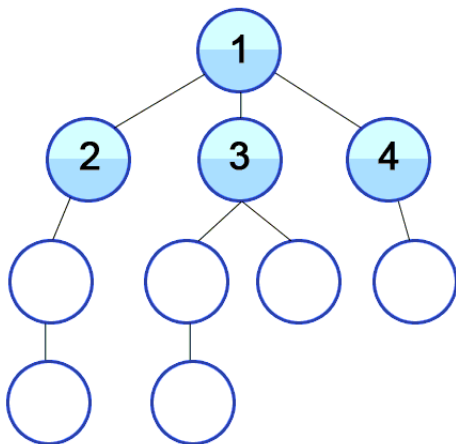
## BFS (2): Contoh pada tree



## BFS (2): Contoh pada tree

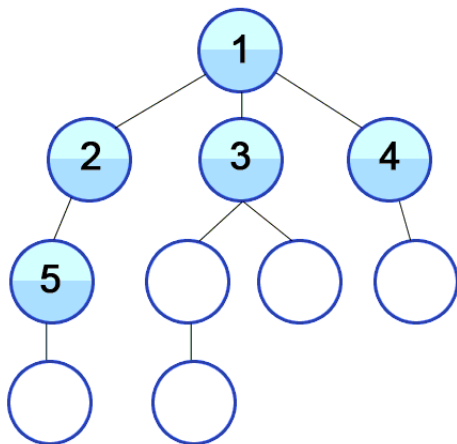


## BFS (2): Contoh pada tree

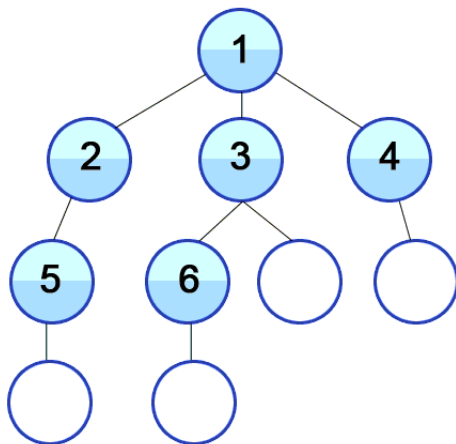




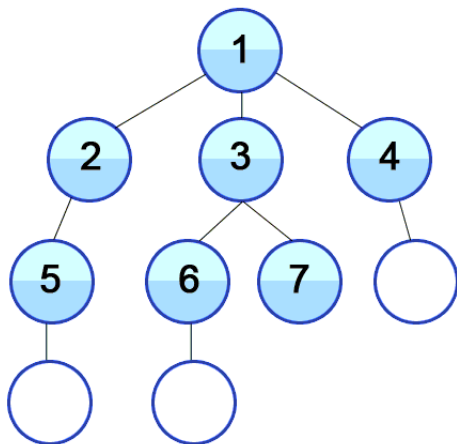
## BFS (2): Contoh pada tree



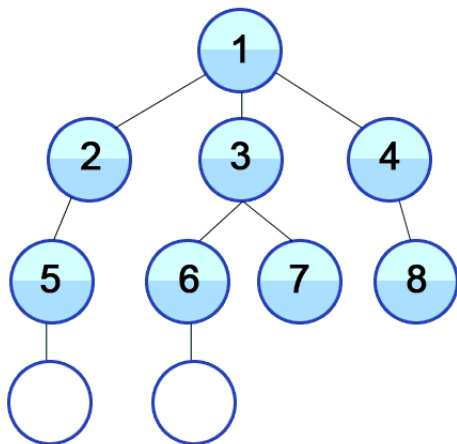
## BFS (2): Contoh pada tree



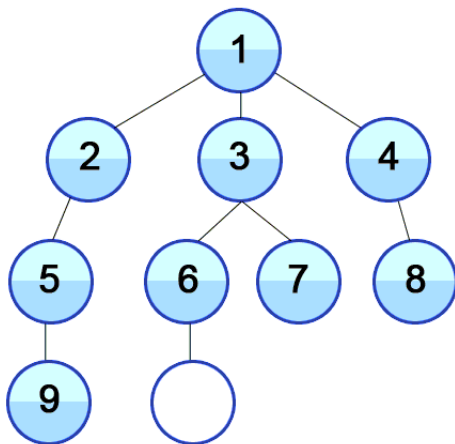
## BFS (2): Contoh pada tree



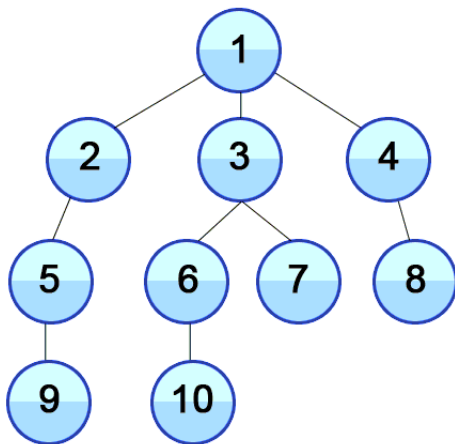
## BFS (2): Contoh pada tree



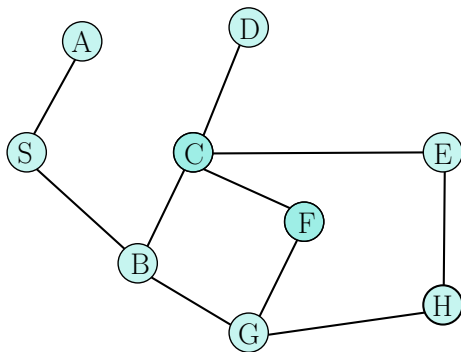
## BFS (2): Contoh pada tree



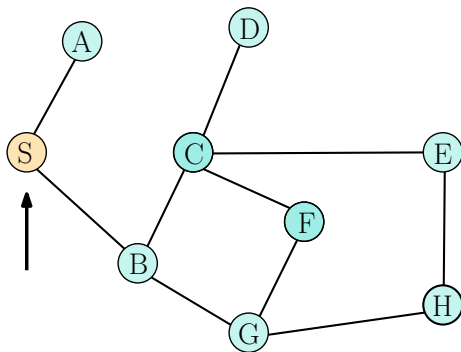
## BFS (2): Contoh pada tree



## BFS (3): Contoh pada graf

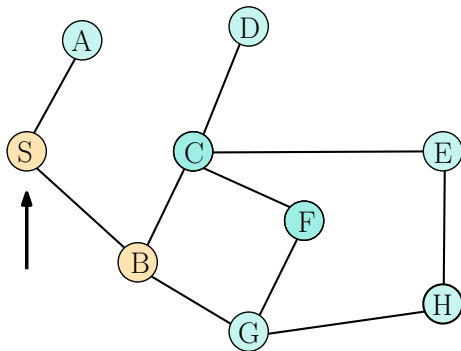


### BFS (3): Contoh pada graf

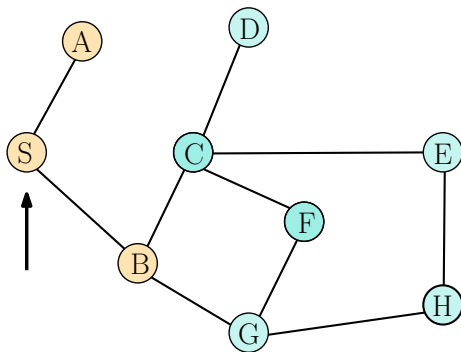




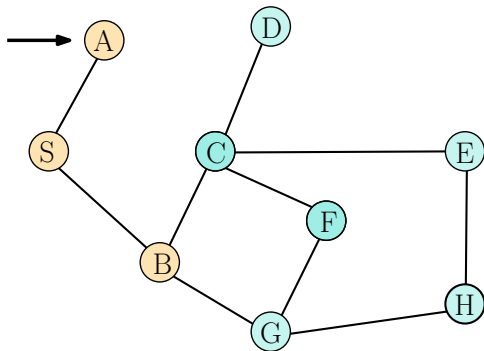
## BFS (3): Contoh pada graf



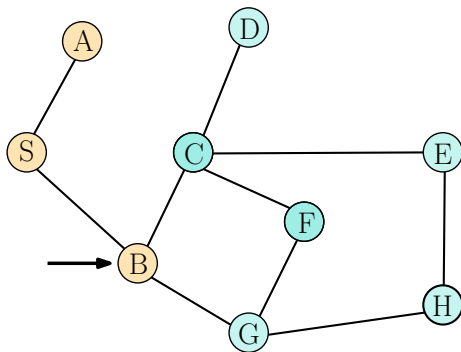
## BFS (3): Contoh pada graf



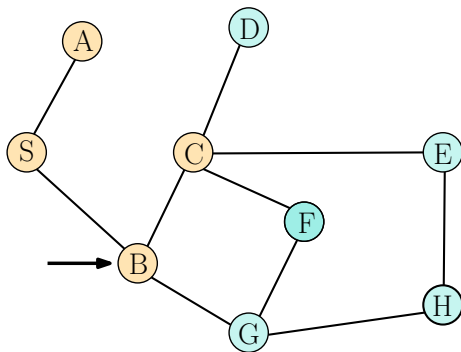
## BFS (3): Contoh pada graf



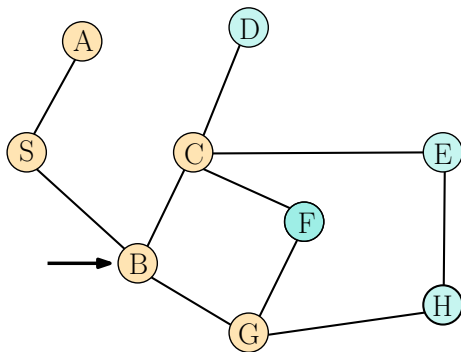
## BFS (3): Contoh pada graf



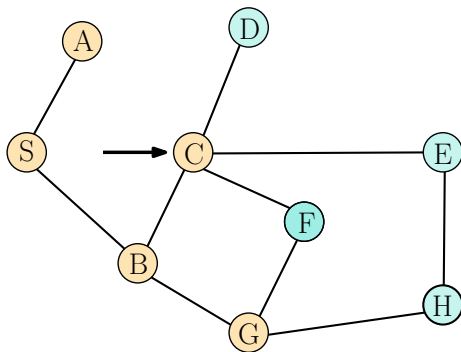
## BFS (3): Contoh pada graf



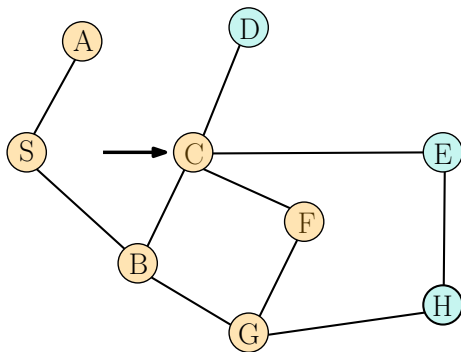
## BFS (3): Contoh pada graf



## BFS (3): Contoh pada graf

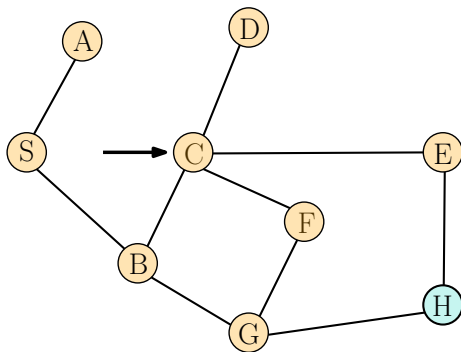


## BFS (3): Contoh pada graf

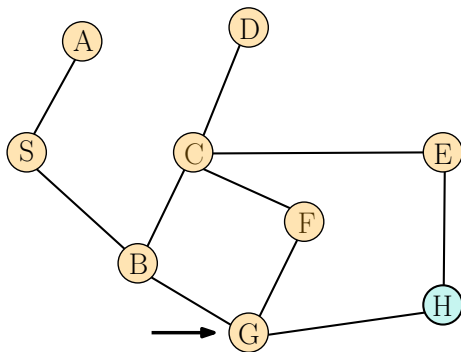




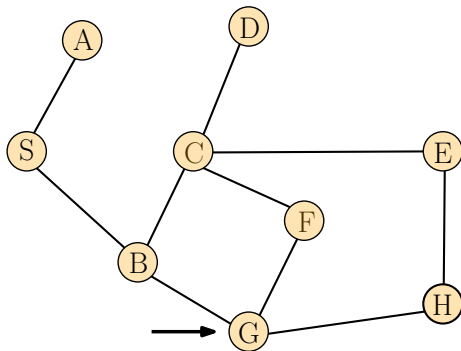
## BFS (3): Contoh pada graf



## BFS (3): Contoh pada graf



## BFS (3): Contoh pada graf



## BFS (4): BFS tree

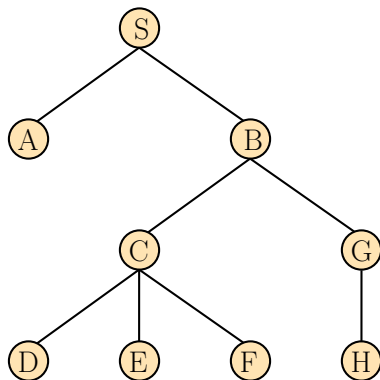


Figure: Struktur graf pohon setelah *running* BFS

## BFS (5): Struktur data

- ➊ The adjacency matrix  $A = [a_{ij}]$  of size  $n \times n$ ,
  - ▶  $a_{ij} = 1$ , jika simpul  $i$  dan simpul  $j$  berdekatan
  - ▶  $a_{ij} = 0$ , jika simpul  $i$  dan simpul  $j$  tidak berdekatan
- ➋ Antrian  $Q$  untuk menyimpan simpul yang dikunjungi.
- ➌ Boolean array, bernama “Visited”, dengan ukuran  $1 \times n$ 
  - ▶  $\text{visited}[i]$ : *True* jika simpul  $i$  telah dikunjungi
  - ▶  $\text{visited}[i]$ : *False* jika simpul  $i$  belum dikunjungi
- ➍ “Visited” juga dapat diatur sebagai array integer, yang menunjukkan urutan simpul yang dikunjungi setelah prosedur BFS diimplementasikan.

## BFS (6): Pseudocode (recursive)

---

### Algorithm 3 BFS in a graph

---

```
1: procedure BFS( $G$ )
2:   input: graf  $G = (V, E)$ 
3:   output: graf  $G$  dengan  $V(G)$  ditandai dengan bilangan bulat berurutan yang
      menunjukkan urutan BFS
4:   count  $\leftarrow 0$ 
5:   initialize array visited = [ ]
6:   for  $v \in V$  do
7:     visited[ $v$ ] = 0
8:   end for
9:   for  $v \in V$  do
10:    if visited[ $v$ ] = 0 then
11:      BFS( $v$ )
12:    end if
13:  end for
14:  return visited
15: end procedure
```

---

# BFS (7): Pseudocode

---

## Algorithm 4 BFS a vertex

---

```
1: procedure BFS( $v$ )
2:   count  $\leftarrow$  count + 1
3:   visited[ $v$ ] = count
4:   initialize queue  $Q = [v]$ 
5:   while  $Q \neq []$  do
6:     for  $w \in N(Q[0])$  do
7:       if visited[ $w$ ] = 0 then
8:         count  $\leftarrow$  count + 1
9:         visited[ $w$ ] = count
10:        add  $w$  to  $Q$ 
11:       end if
12:     end for
13:     remove  $Q[0]$  from  $Q$ 
14:   end while
15: end procedure
```

▷  $Q$  is the list of vertices whose neighbors may need to be visited

▷  $Q[0]$  is the first element in the queue  $Q$

▷  $w$  is put as the last element of the array  $Q$

# Penerapan DFS dan BFS



# Bagian 3. Graf dinamis (*dynamic graph*)

# Graf dinamis

**Graf:**  $G(V, E)$ , dimana  $V$  adalah himpunan simpul, dan  $E$  adalah himpunan sisi.

**Graf dinamis:**  $G = (G_1, G_2, \dots, G_t)$  dimana  $G_t = (V_t, E_t)$  dan adalah ukuran graf pada waktu  $t$ .

- Pada graf dinamis, simpul baru dapat dibentuk dan membuat tautan dengan simpul yang sudah ada; atau simpul bisa hilang, sehingga mengakhiri tautan yang ada.

## Mengapa graf dinamis dibutuhkan?

- Situasi di dunia nyata yang dimodelkan dengan graf bisa sangat kompleks. Grafnya **tidak statis** dan dapat **berkembang sepanjang waktu**.

# Contoh graf dinamis

## Evolusi social network

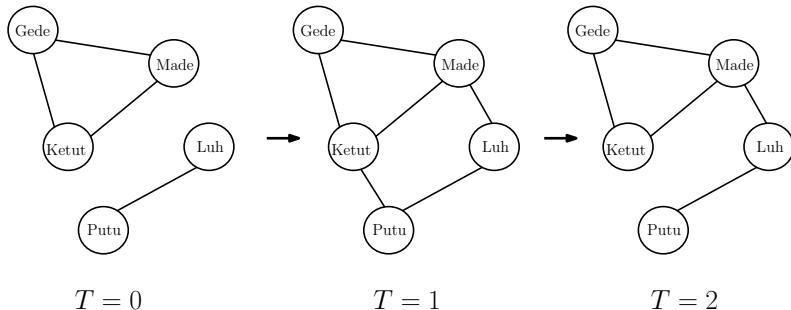


Figure: Evolusi *social network* (source: [towardsdatascience.com](http://towardsdatascience.com))

- Evolusi menunjukkan 3 snapshot pada 3 titik waktu
- Beberapa pertemanan baru dibuat dan beberapa pertemanan putus
- Terdapat simpul masuk baru (orang yang bergabung dengan jaringan) dan beberapa simpul keluar (orang yang keluar dari jaringan)

# Solusi pencarian dengan DFS/BFS

Pencarian solusi → membuat **dynamic tree**

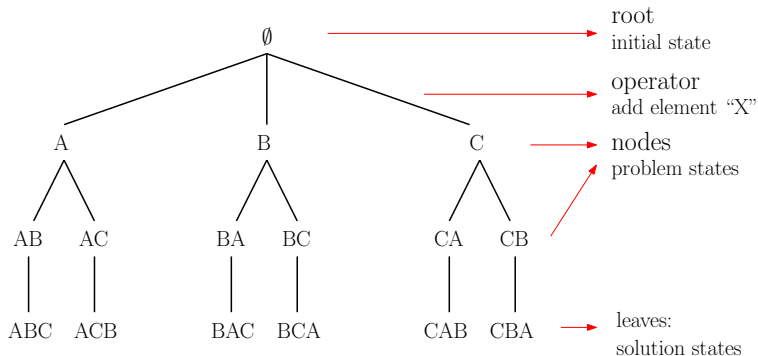
- Setiap simpul diperiksa, untuk melihat apakah solusi (tujuan) diperoleh.
- Jika simpul adalah solusi, pencarian selesai (untuk satu solusi); atau dilanjutkan untuk mencari solusi lain.

## Representasi pohon dinamis

- **State-space tree**: pohon ruang status
- Setiap simpul mewakili status masalah
  - ▶ **Root**: initial state
  - ▶ **Leaves**: solution/goal state
- **Branch**: operator/operasi
- **State space**: himpunan semua simpul
- **Solution space**: himpunan status solusi

Solusi masalah dalam pohon dinamis ditampilkan menggunakan **lintasan dari simpul akar ke simpul status solusi**.

## Contoh *state-space tree*: Permutasi



**Solution space:** set of all solution states

**State space:** all nodes in dynamic tree

**Figure:** State space tree dari "Permutasi A, B, C"

# BFS untuk membangun state-space tree

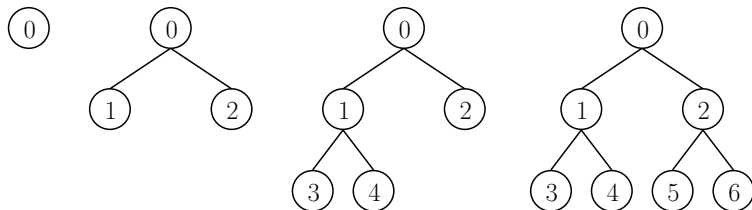
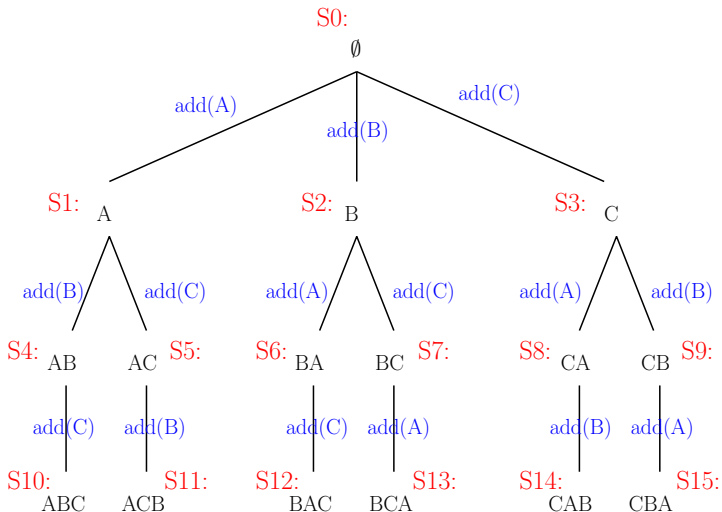


Figure: State space tree dari "Permutasi A, B, C"

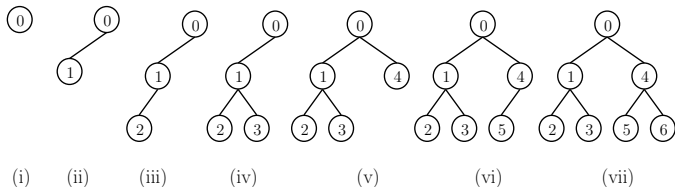
- Inisialisasi keadaan awal sebagai akar (*root*), tambahkan simpul anak.
- Semua simpul pada level  $d$  dibangun sebelum membangun simpul pada level  $d + 1$ .

# DFS untuk membangun state-space tree



# DFS untuk membangun state-space tree

## DFS



## BFS

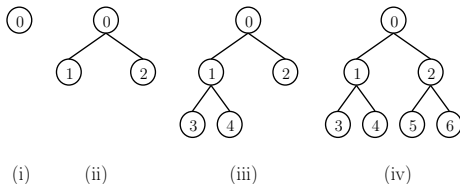
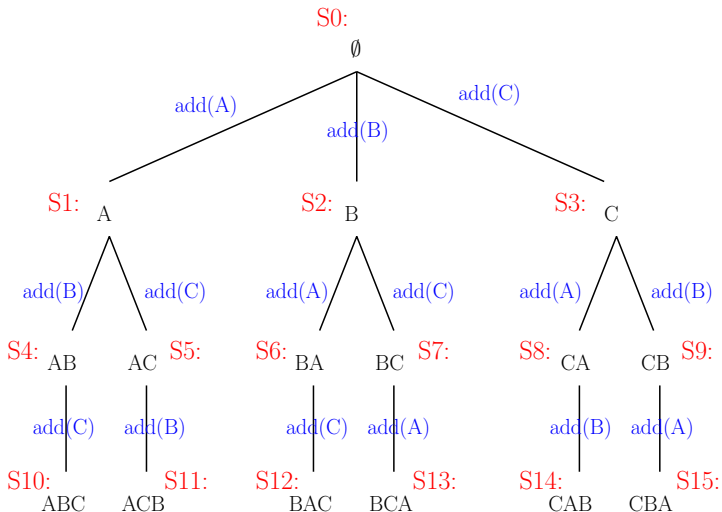


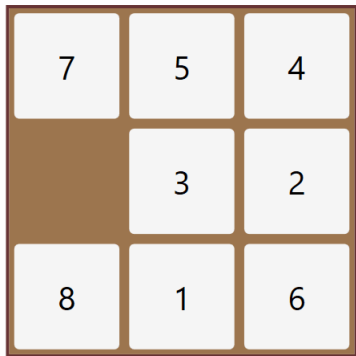
Figure: Konstruksi state space tree - DFS vs BFS



# BFS untuk membangun state-space tree



## Bagian 4. 8-puzzle game



## Merancang DFS/BFS untuk 8-puzzle

2	8	3
1	6	4
7		5

initial state

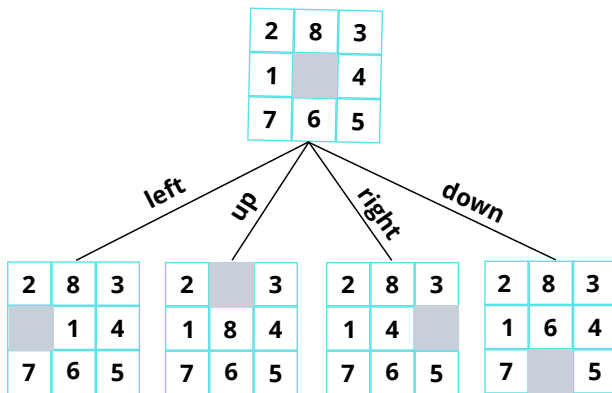
1	2	3
8		4
7	6	5

goal state

- **State**: status ditentukan berdasarkan posisi *kotak kosong*

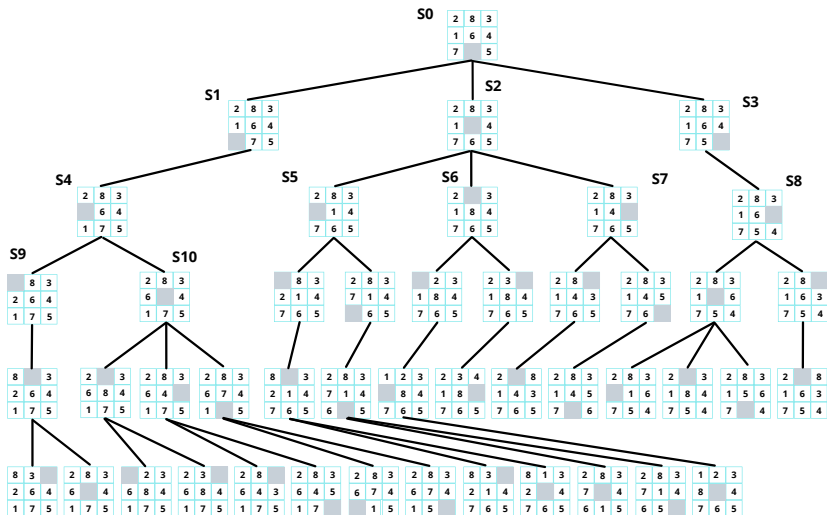
## Merancang DFS/BFS untuk 8-puzzle

- **Operator:** up, down, left, right

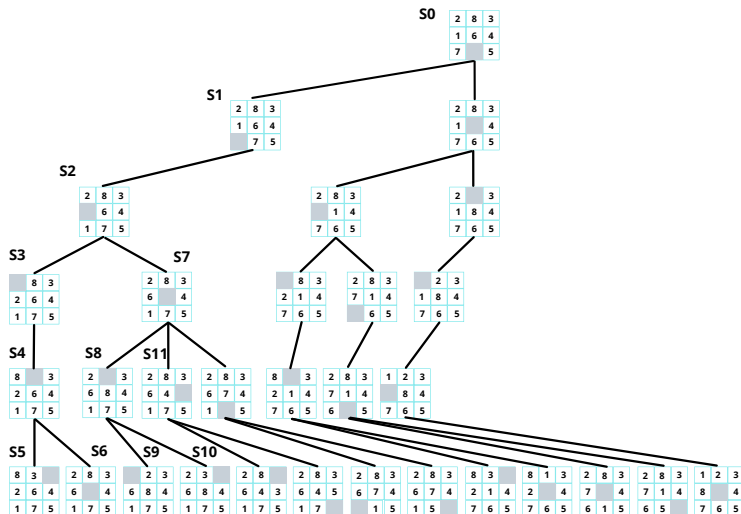


*Catatan:* saat membuat pohon ruang keadaan, urutan operator harus konsisten

# BFS state-space tree untuk 8-puzzle game



# DFS state-space tree untuk 8-puzzle game



# Bagian 5. Efisiensi BFS & DFS

# Efisiensi of DFS and BFS

- **Completeness:** jika solusinya ada, apakah algoritma menjamin bahwa solusi optimal ditemukan?
- **Optimality:** apakah algoritma menjamin solusi yang diperoleh optimal (misal: *lintasan solusi memiliki biaya terendah*)
- Kompleksitas **waktu & ruang**

Kompleksitas waktu dan ruang diukur berdasarkan faktor-faktor berikut:

- $b$  (*branching factor*): jumlah maksimum kemungkinan cabang dari sebuah simpul
- $d$  (*depth*): kedalaman solusi terbaik (lintasan dengan bobot/*cost* terendah)
- $m$ : kedalaman maksimum ruang status (mungkin bernilai  $\infty$ )



# Efisiensi BFS

- **Completeness:** YA selama  $b$  dibatasi (berhingga/*finite*)
- **Optimality:** YA jika biaya ditentukan oleh *jumlah langkah*
- **Time complexity:**  $1 + b + b^2 + b^3 + \dots + b^d = \mathcal{O}(b^d)$
- **Space complexity:**  $\mathcal{O}(b^d)$ , karena kita harus menyimpan semua status di setiap kedalaman.

# Efisiensi DFS

- **Completeness:** YA selama  $b$  dibatasi (berhingga/*finite*), dan “lintasan berlebih (*redundant*)” dan “lintasan berulang” ditangani.
- **Optimality:** TIDAK SELALU, karena kita mungkin akan melintasi banyak negara bagian sebelum mencapai solusi.
- **Time complexity:**  $\mathcal{O}(b^m)$ , karena kita harus membuat status berdasarkan kedalaman.
- **Space complexity:**  $\mathcal{O}(bm)$ , karena kita hanya menyimpan status yang mengarah ke solusi.

*end of slide...*