09 - DFS and BFS

[KOMS119602] & [KOMS120403]

Design and Analysis of Algorithm (2021/2022)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 25-29 April 2022

Table of contents

Graph traversal

A graph traversal algorithm is an algorithm that looks for a problem solution in a *graph data structure*, by visiting the nodes in the graph systematically (assuming that the graph is *connected*).

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

Graph data structure

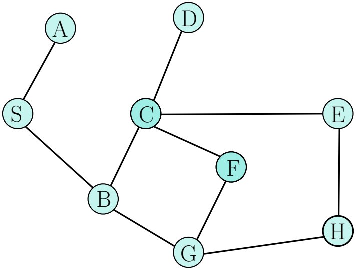
Adjacency matrix

An adjacency matrix is an $n \times n$ binary matrix in which value of $[i, j]$-th cell is 1 if there exists an edge having endpoints the i-th vertex and the j-th vertex, otherwise the value is 0.

Adjacency list

An adjacency list is an array of separate lists. Each element of array is a list of corresponding neighbour (or directly connected) vertices. In other words, the i-th list of an adjacency list is a list of all those vertices which is directly connected to i-th vertex.

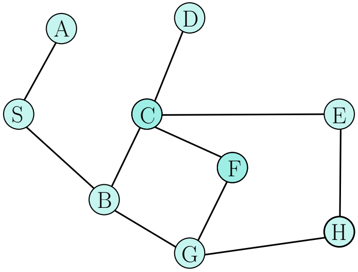
Graph data structure



$$\begin{array}{c|ccccccccc|}
 & A & B & C & D & E & F & G & H & S \\
A & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
B & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
C & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
D & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
E & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
F & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
G & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
H & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
S & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Figure: A graph and its adjacency matrix

Graph data structure



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: A graph and its adjacency list

Graph representation in searching process

Two approaches in the solution searching process

1. Static graph: the graph is constructed *before* the searching process. Graph is represented as a data structure.
    - Example: BFS, DFS
2. Dynamic graph: the graph is constructed along with the process of searching.

Depth-First Search (DFS)

DFS (1): Algorithm

DFS begins at a *root node* and inspects all the neighboring nodes.

- Visit the node v;
- Visit node w that is adjacent to v;
- Repeat DFS starting from node w;
- When vertex u is reached so that all its neighbor are visited, the searching is "backtracked" to the last visited node that still has an unvisited neighbor.
- Keep going on like this.
- Searching is finished when there is no more node that can be reached from the visited node.

DFS (2): Pseudocode (recursive)

Algorithm 1 DFS in a graph

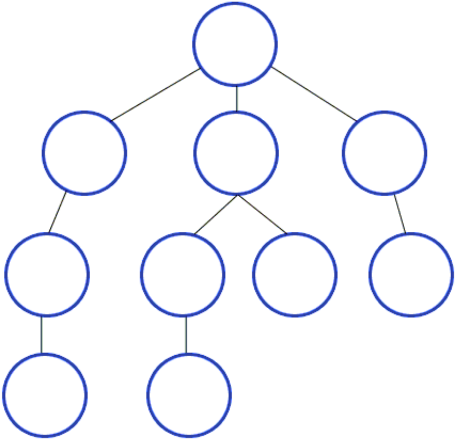
```
procedure DFS($G$)
    input: graph $G = (V, E)$
    output: graph $G$ with $V(G)$ marked with consecutive integers indicating the DFS-order
    count $\leftarrow 0$
    initialize array visited $= [\ ]$
    for $v \in V$ do
        visited$[v] = 0$
    end for
    for $v \in V$ do
        if visited$[v] = 0$ then
            DFS($v$)
        end if
    end for
    return visited
end procedure
```

DFS (3): Pseudocode

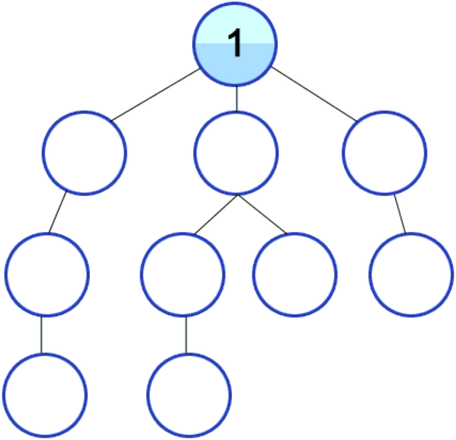
Algorithm 2 DFS a vertex

```
procedure DFS($v$)
    count ← count + 1
    visited[$v$] = count
    for $w \in N(v)$ do
        if visited[$w$] = 0 then
            DFS($w$)
        end if
    end for
end procedure
```

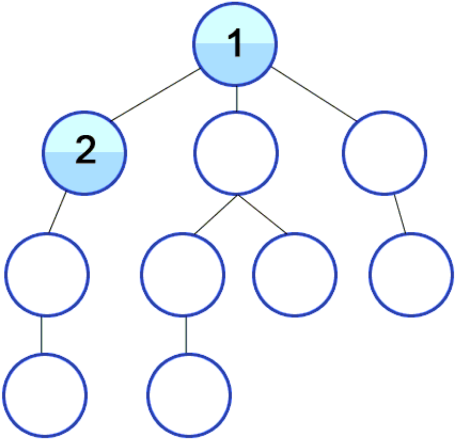
DFS (4): Example on a tree



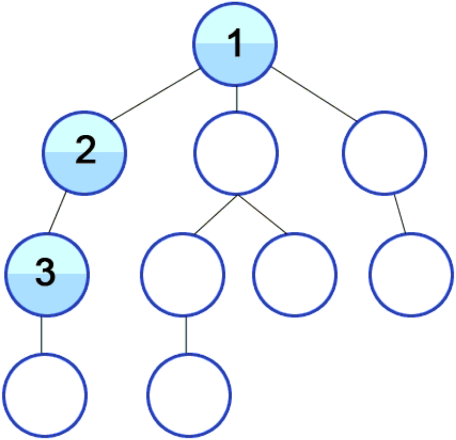
DFS (4): Example on a tree



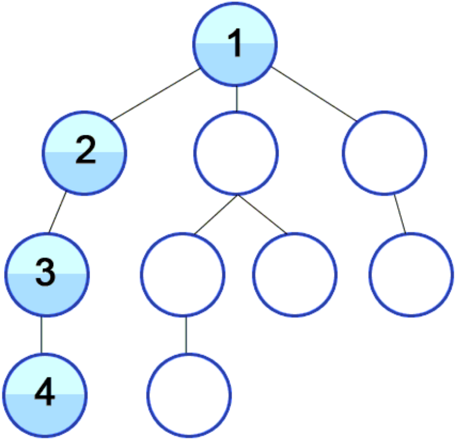
DFS (4): Example on a tree



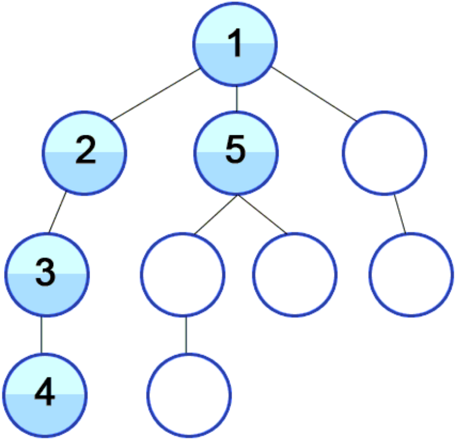
DFS (4): Example on a tree



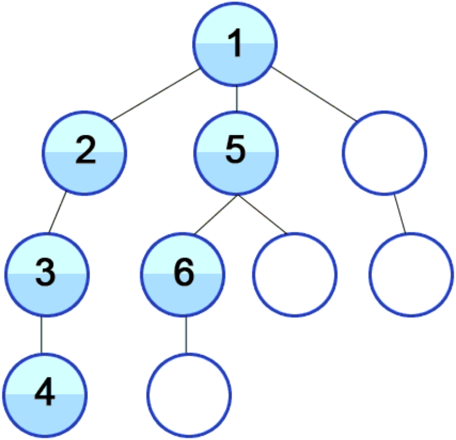
DFS (4): Example on a tree



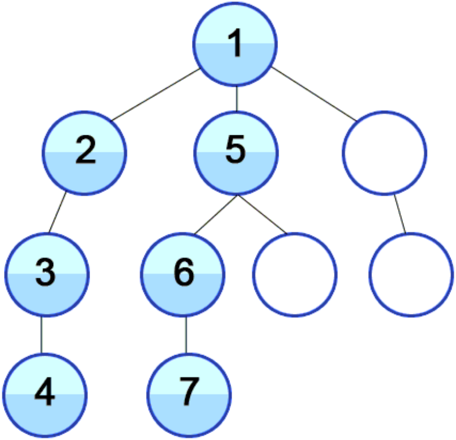
DFS (4): Example on a tree



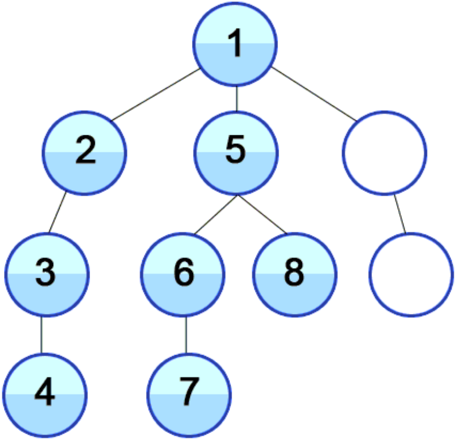
DFS (4): Example on a tree



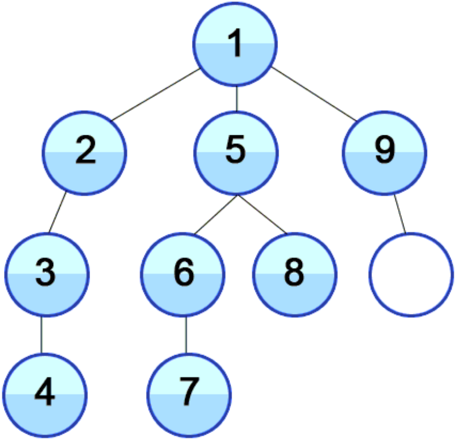
DFS (4): Example on a tree



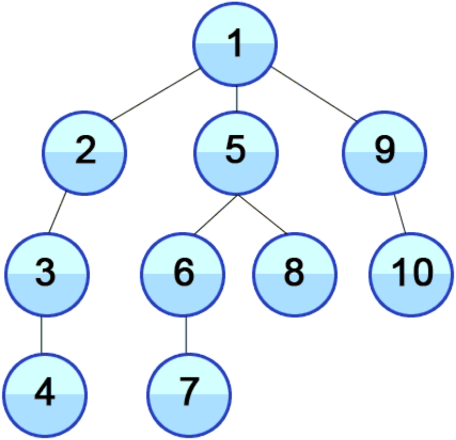
DFS (4): Example on a tree



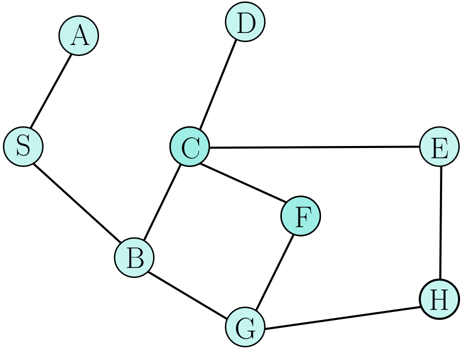
DFS (4): Example on a tree



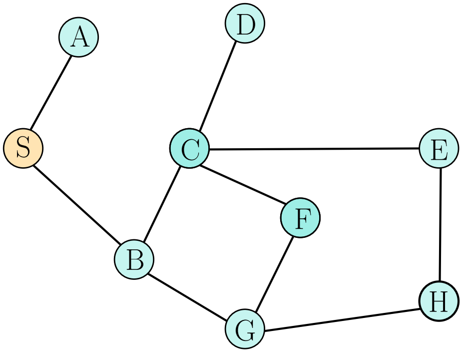
DFS (4): Example on a tree



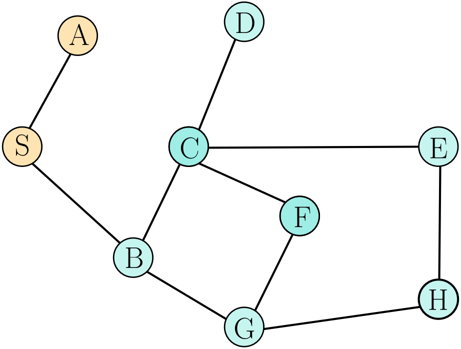
DFS (5): Example on a graph



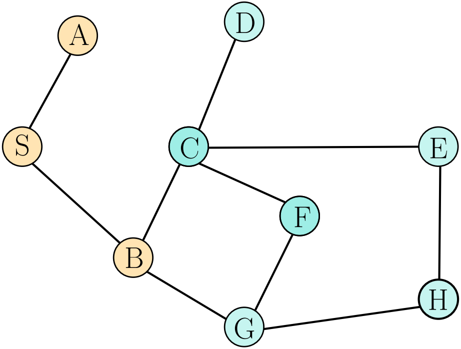
DFS (5): Example on a graph



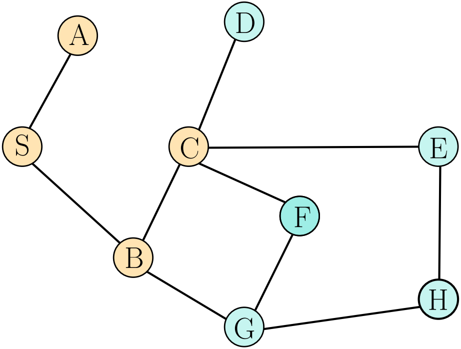
DFS (5): Example on a graph



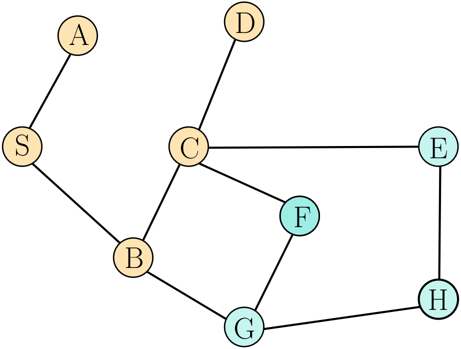
DFS (5): Example on a graph



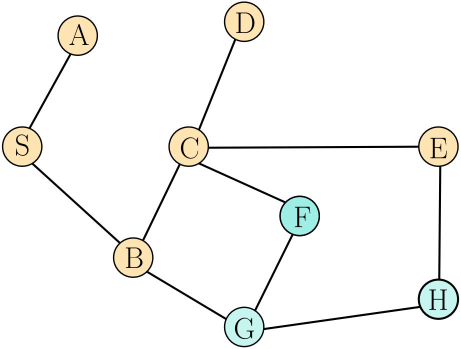
DFS (5): Example on a graph



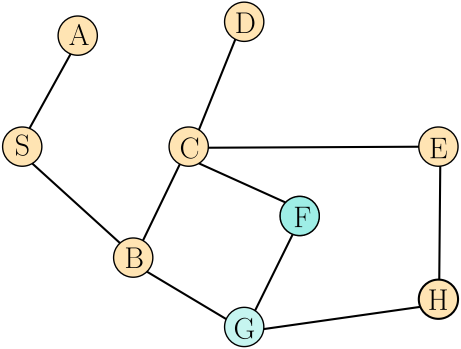
DFS (5): Example on a graph



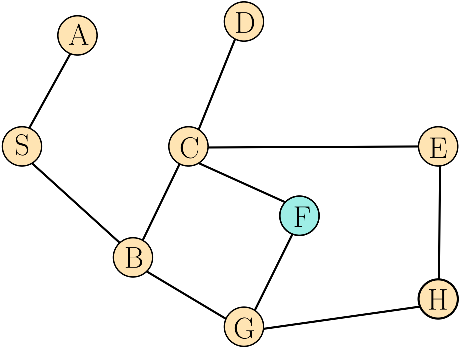
DFS (5): Example on a graph



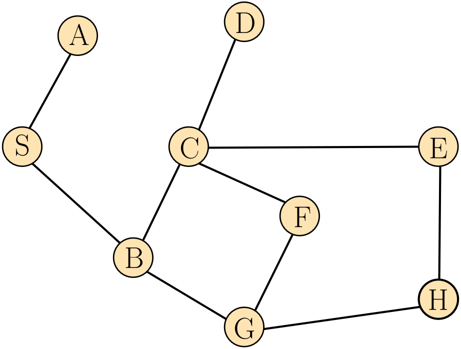
DFS (5): Example on a graph



DFS (5): Example on a graph



DFS (5): Example on a graph



DFS (6): DFS tree

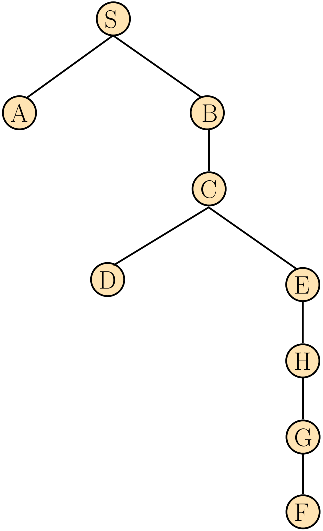


Figure: Tree after DFS run

Breadth-First Search (BFS)

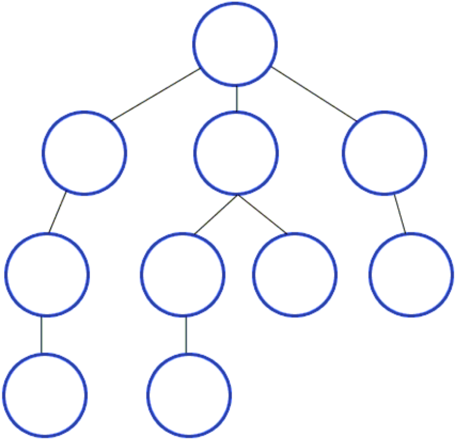
BFS (1): Algorithm

BFS begins at a *root node* and inspects all the neighboring nodes.

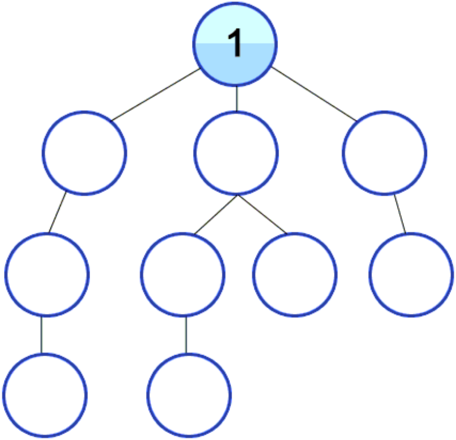
Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on.

- Visit the node v;
- Visit all nodes that are adjacent to v;
- Visit all nodes not yet visited, and are adjacent to the nodes that just visited;
- Keep going on like this...

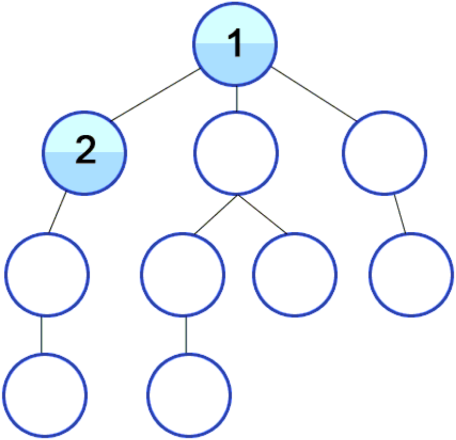
BFS (2): Example on a tree



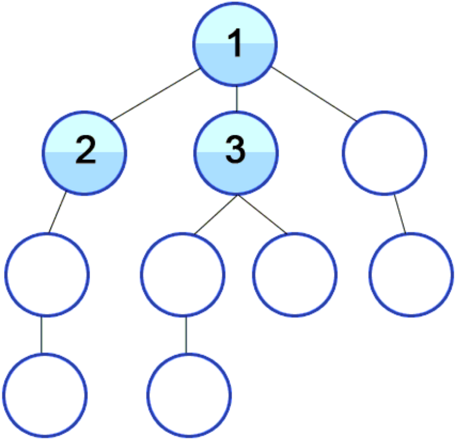
BFS (2): Example on a tree



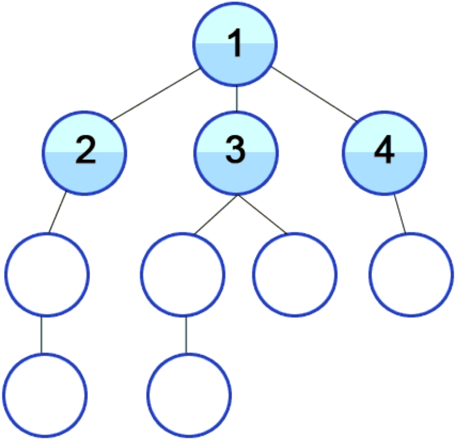
BFS (2): Example on a tree



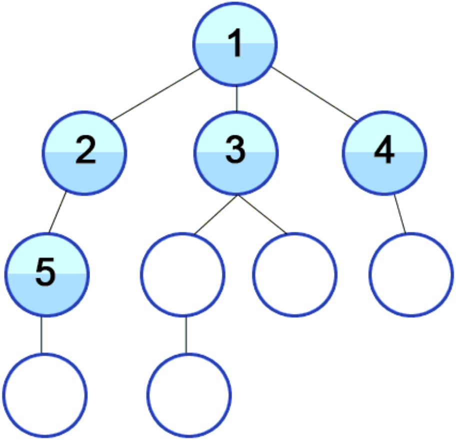
BFS (2): Example on a tree



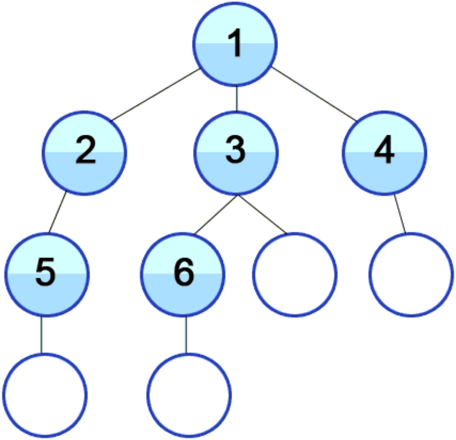
BFS (2): Example on a tree



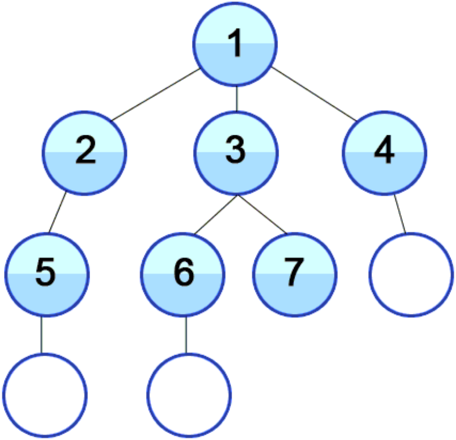
BFS (2): Example on a tree



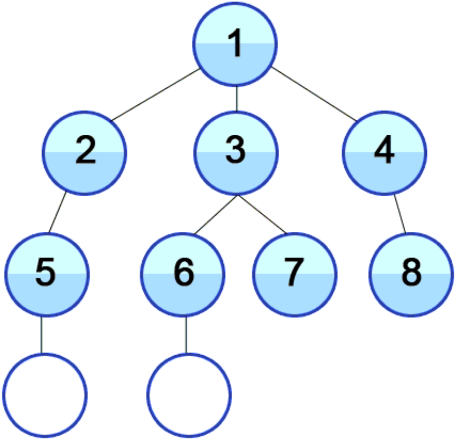
BFS (2): Example on a tree



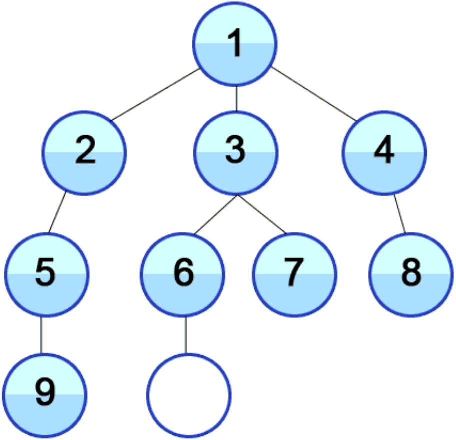
BFS (2): Example on a tree



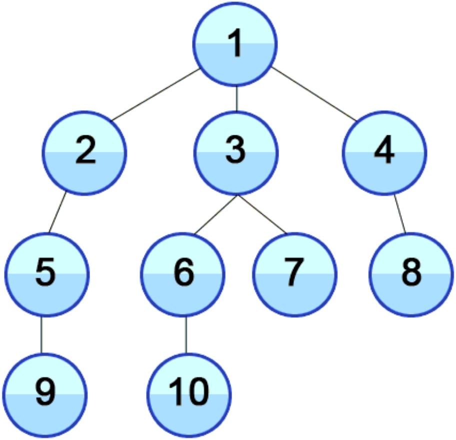
BFS (2): Example on a tree



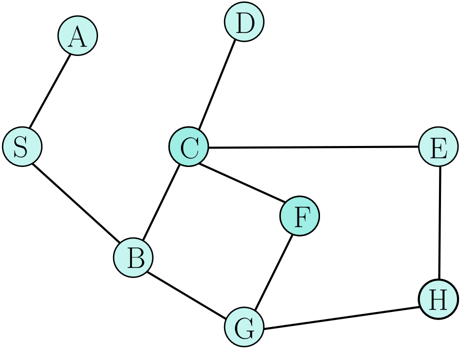
BFS (2): Example on a tree



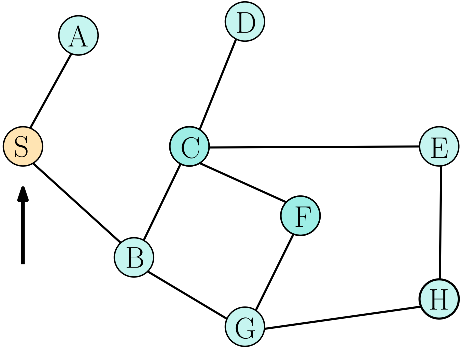
BFS (2): Example on a tree



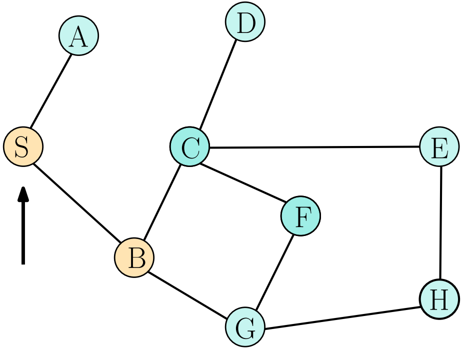
BFS (3): Example on a graph



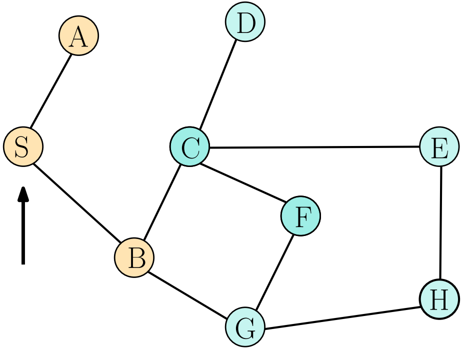
BFS (3): Example on a graph



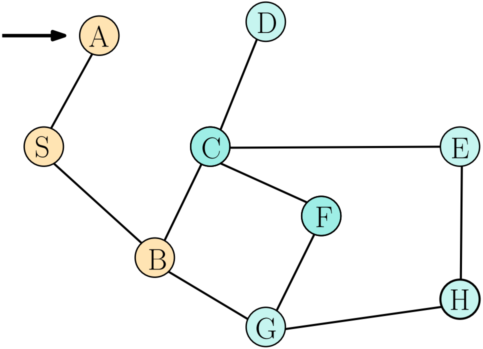
BFS (3): Example on a graph



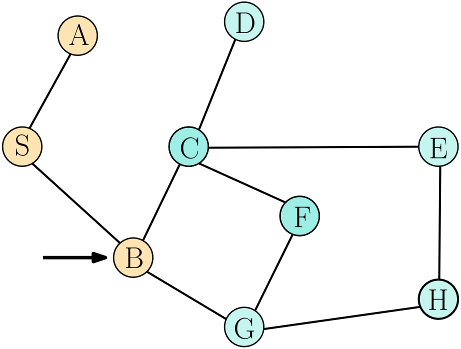
BFS (3): Example on a graph



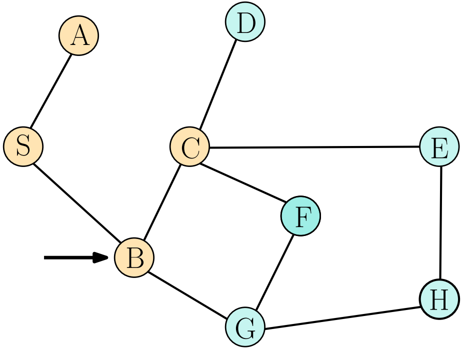
BFS (3): Example on a graph



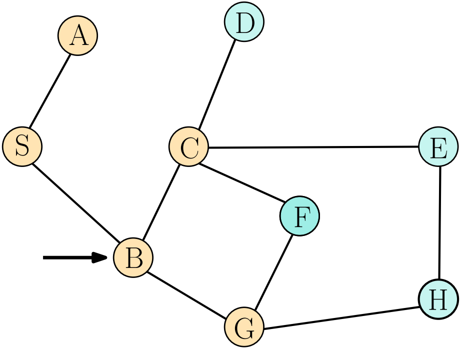
BFS (3): Example on a graph



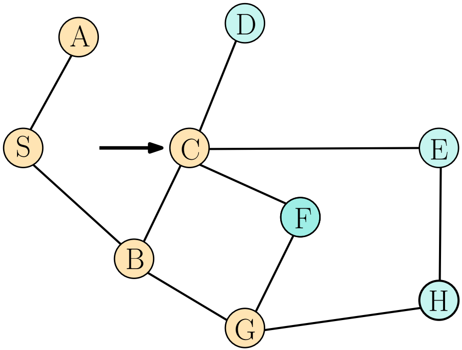
BFS (3): Example on a graph



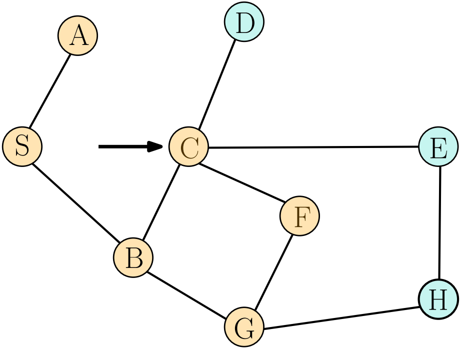
BFS (3): Example on a graph



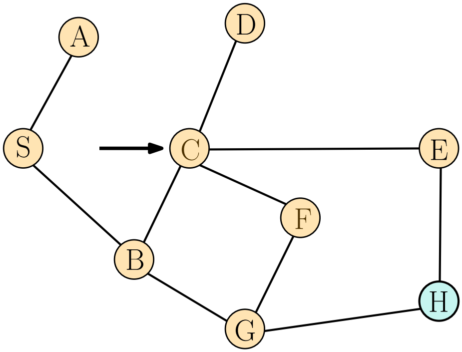
BFS (3): Example on a graph



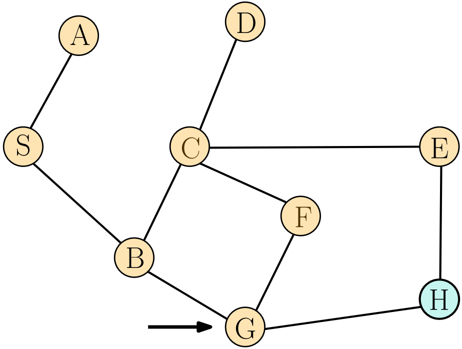
BFS (3): Example on a graph



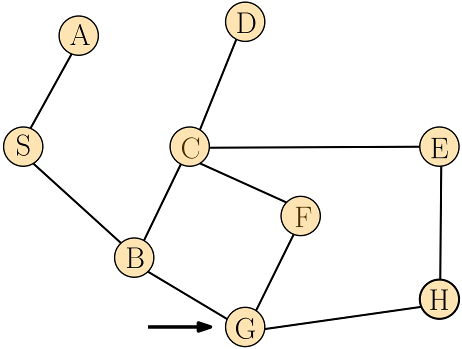
BFS (3): Example on a graph



BFS (3): Example on a graph



BFS (3): Example on a graph



BFS (4): BFS tree

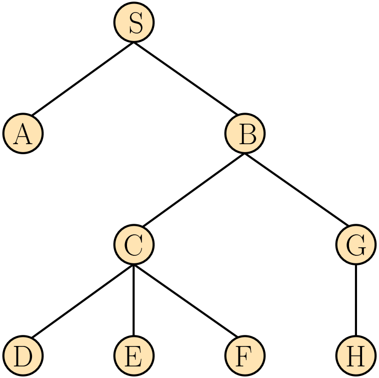


Figure: Tree after BFS run

BFS (5): Data structure

1. The adjacency matrix $A = [a_{ij}]$ of size $n \times n$,
    - $a_{ij} = 1$, if node i and node j are adjacent
    - $a_{ij} = 0$, if node i and node j are non-adjacent
2. Queue Q to store the visited nodes.
3. Boolean array, named "Visited", of size $1 \times n$
    - visited$[i]$: *True* if node i has been visited
    - visited$[i]$: *False* if node i has not been visited
4. "Visited" can be also set as an integer array, indicating the order of the visited nodes after BFS procedure is implemented.

BFS (6): Pseudocode (recursive)

Algorithm 3 BFS in a graph

```
procedure BFS($G$)
    input: graph $G = (V, E)$
    output: graph $G$ with $V(G)$ marked with consecutive integers indicating the BFS-order
    count $\leftarrow 0$
    initialize array visited = [ ]
    for $v \in V$ do
        visited[$v$] = 0
    end for
    for $v \in V$ do
        if visited[$v$] = 0 then
            BFS($v$)
        end if
    end for
    return visited
end procedure
```

BFS (7): Pseudocode

Algorithm 4 BFS a vertex

```
procedure BFS($v$)
    count ← count + 1
    visited[$v$] = count
    initialize queue $Q = [v]$
    while $Q \neq [\,]$ do
        for $w \in N(Q[0])$ do                    ▷ Q[0] is the first element in the queue Q
            if visited[$w$] = 0 then
                count ← count + 1
                visited[$w$] = count
                add $w$ to $Q$
            end if
        end for
        remove $v$ from $Q$
    end while
end procedure
```

Applications of DFS and BFS

Tugas: Buat rangkuman tentang satu aplikasi algoritma DFS atau BFS. Jelaskan apa permasalahannya, dan bagaimana algoritma DFS/BFS digunakan untuk menyelesaikan permasalahan tersebut!

Setiap mahasiswa diwajibkan memberikan contoh yang berbeda dengan mahasiswa lain!

Tugas diketik dalam Bahasa Indonesia ±1 halaman.

Tulis topik pada list berikut . . .

Dynamic graph

Dynamic graph

Graph: $G(V,E)$, where V: set of vertices and E: set of edges.

Dynamic graph: $G = (G_1, G_2, \ldots, G_t)$ where $G_t = (V_t, E_t)$ and is the current number of *snapshots*.

- In dynamic graph, new nodes can be formed and create links with the existing nodes; or nodes can disappear, thus terminating the existing links.

Why need dynamic graphs?

- Real-life situations that are modeled with graphs can be very complex. The graph is **not static** and can **evolve through time**.

Dynamic graph: example

Evolution of a social network

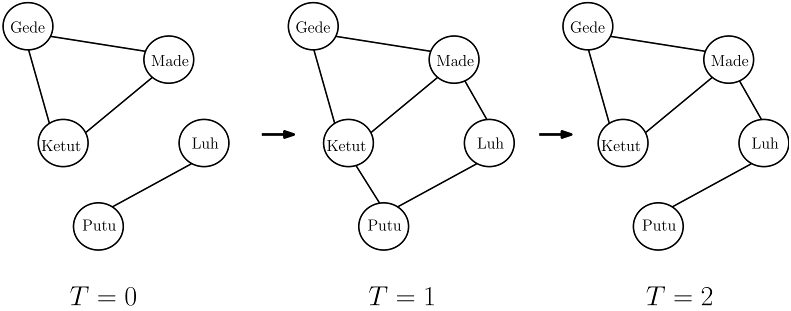


Figure: Evolution of a social network (source: `towardsatascience.com`)

- The evolution shows 3 snapshots at 3 time-points
- Some new friendships being made and also some get broken
- There are new incoming nodes (people joining the network) and some outgoing nodes (people leaving the network)

Solution searching via DFS/BFS

Solution searching → creating dynamic tree

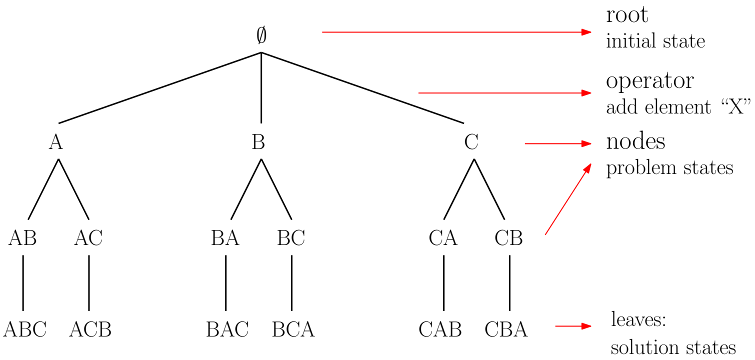
- Each node is checked, to see if the solution (goal) is obtained.
- If a node is a solution, the searching is finished (for one solution); or is continued to look for other solutions.

Representation of dynamic tree

- **State-space tree**: tree of problem's states
- Each node represents a problem state
  - **Root**: initial state
  - **Leaves**: solution/goal state
- **Branch**: operator/operation
- **State space**: set of all nodes
- **Solution space**: set of solution state

A problem solution in a dynamic tree is showed using a path from the root to a solution state.

State-space tree example: Permutation



root
initial state

operator
add element “X”

nodes
problem states

leaves:
solution states

Solution space: set of all solution states

State space: all nodes in dynamic tree

Figure: State space tree of “Permutation of A, B, C”

BFS for constructing state-space tree

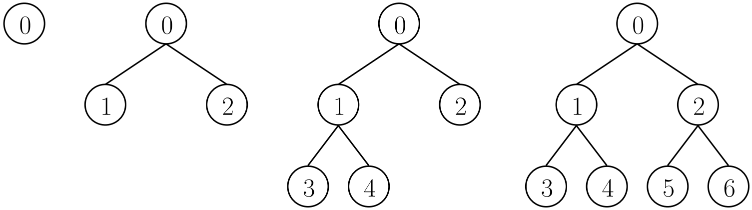
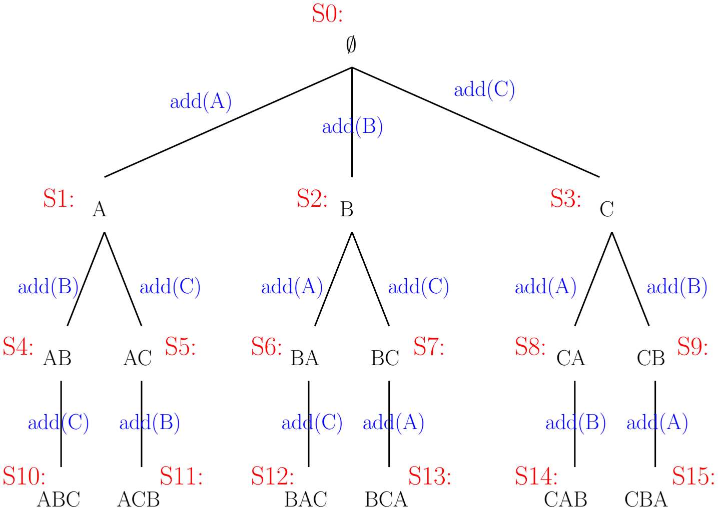


Figure: State space tree of “Permutation of A, B, C”

- Initialize the initial state as the root, add children nodes.
- All nodes at level d are constructed before constructing the nodes at level $d + 1$.

DFS for constructing state-space tree



DFS for constructing state-space tree

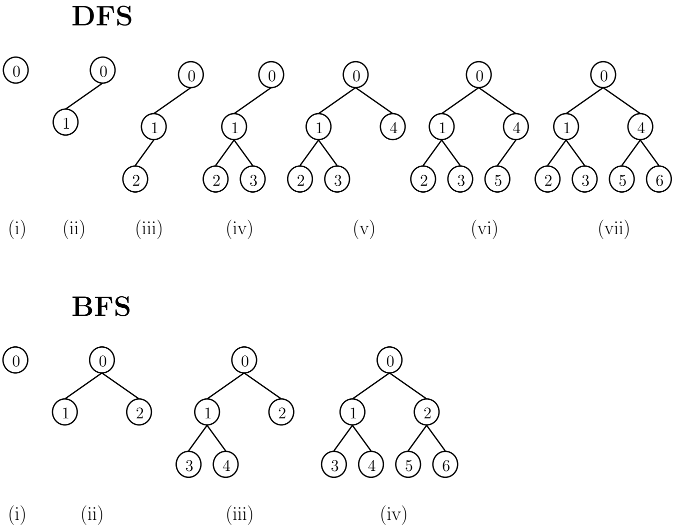
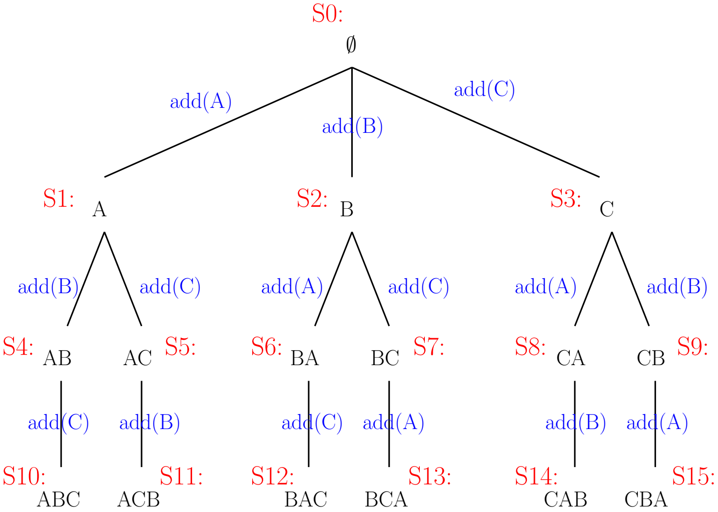
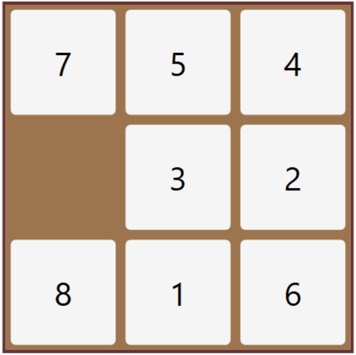


Figure: State space tree construction - DFS vs BFS

BFS for constructing state-space tree



8-puzzle game



Designing DFS/BFS for 8-puzzle

2	8	3
1	6	4
7		5

initial state

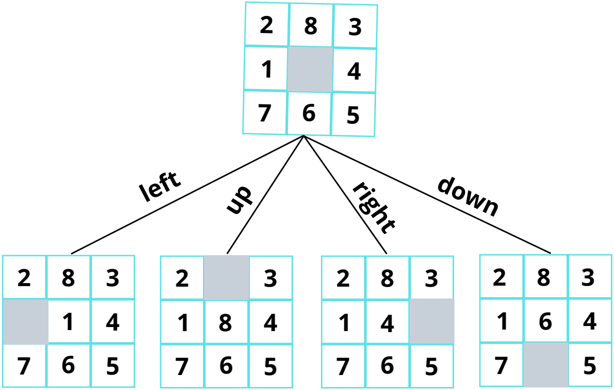
1	2	3
8		4
7	6	5

goal state

- **State**: the states are defined based on the *empty box*

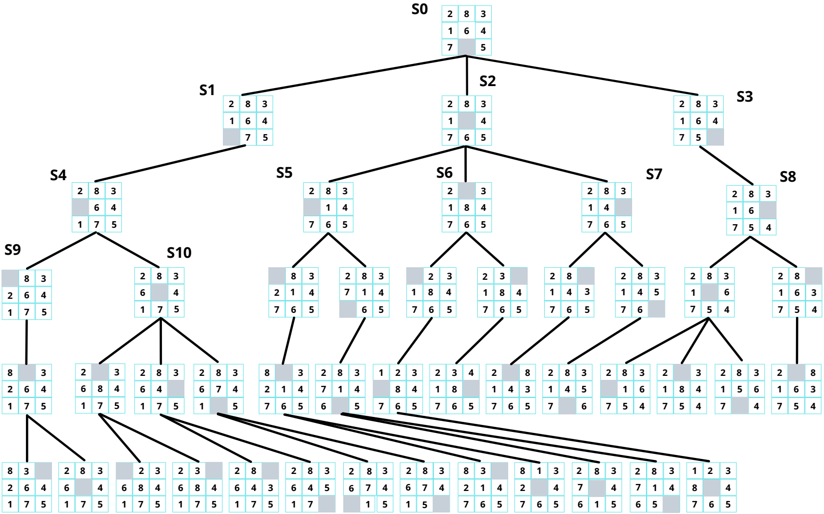
Designing DFS/BFS for 8-puzzle

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

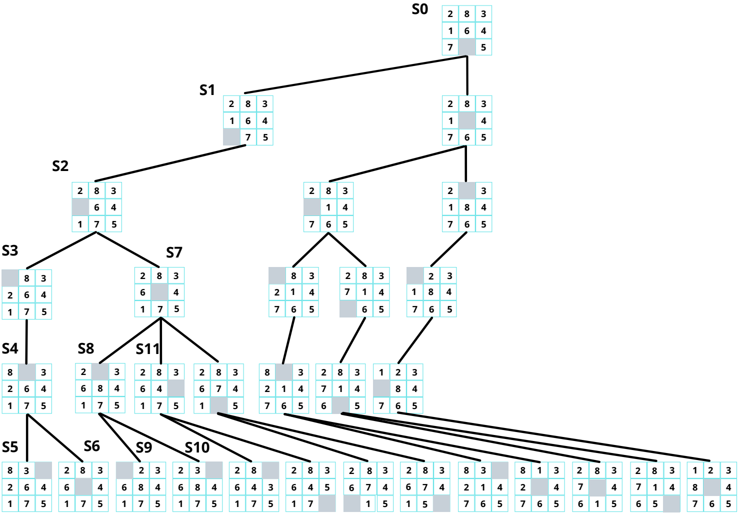


Remark: when creating the state-space tree, the order of the operator must be consistent

BFS state-space tree for 8-puzzle game



DFS state-space tree for 8-puzzle game



Efficiency of DFS and BFS

- **Completeness**: if the solution exists, does the algorithm guarantees that an optimal solution is found?
- **Optimality**: does the algorithm guarantees that the solution obtained is optimal (eg: *the solution path has the lowest cost*)
- **Time & space complexities**

The time and space complexities are measured based on the following factors:

- b (*branching factor*): the maximum number of possible branches from a node
- d (*depth*): the depth of the best solution (the lowest-cost path)
- m: the maximum depth of the state space (can be ∞)

Efficiency of BFS

- **Completeness**: yes as long as b is bounded (finite)
- **Optimality**: yes if the cost is determined by *the number of steps*
- **Time complexity**: $1 + b + b^2 + b^3 + \cdots + b^d = \mathcal{O}(b^d)$
- **Space complexity**: $\mathcal{O}(b^d)$, because we have to store all states at each depth.

Efficiency of DFS

- **Completeness**: yes as long as b is bounded (finite), and the "redundant paths" and "repeated paths" are handled.
- **Optimality**: not always, because we might end up traversing many states before reaching the solution
- **Time complexity**: $\mathcal{O}(b^m)$, because we have to generate the states based on the depth
- **Space complexity**: $\mathcal{O}(bm)$, because we only store the states that lead to a solution