Posts and Telecommunication Institute of Technology
Faculty of Information Technology 1

Introduction to Artificial Intelligence

# Solving problems by searching

Dao Thi Thuy Quynh

# Outline

- The search problem in state spaces
- Examples
- Basic search algorithms

# Outline

- The search problem in state spaces
  - Search and AI
  - Search problem formulation
  - Criteria for evaluating search algorithms
- Examples
- Basic search algorithms

# Search & AI

- Many problems can be formulated as a search problem
  - Game: find an optimal movement (take advantages)
  - Planning: find a solution that satisfies requirements (constraint satisfaction)
  - Route finding: find the optimal path (length, time, cost, …)

- Search is an important research direction of AI
  - Developing efficient search algorithms (especially in cases where the search space is large)
  - Foundations of many other research branches of AI
    - Machine learning, Natural language processing, inference

# Search problem formulation

A search problem can be formulated through 5 components

1. A finite set of possible states: $Q$
2. A set of initial states: $S \subseteq Q$
3. Operator or successor function $P(x)$, set of states reachable from $x$ by any single action
4. Goal test:
   - Explicitly: a set of possible goal states $G \subseteq Q$
   - Implicitly: specified by an abstract property
5. Path cost
   - Sum of the costs of the individual actions along the path
   - $c(x, a, y) \geq 0$, cost moving from state $x$ to state $y$ by taking action $a$

A solution is a path (a series of actions) from the initial state to a goal state

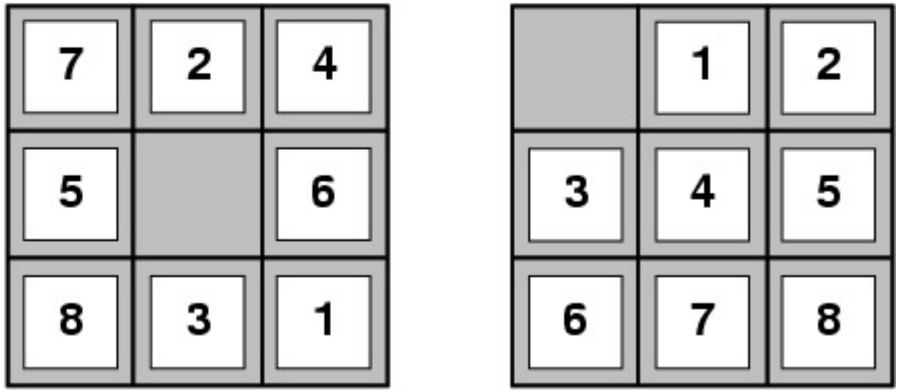
# Criteria for evaluating search algorithms

- Computational complexity (time complexity)
  - The amount of computation required to find the solution
  - Number of states to consider before finding a solution
- Space complexity
  - Number of states to store concurrently in memory when executing the algorithm
- Completeness
  - Is the algorithm guaranteed to find a solution when there is one?
- Optimality
  - Does the algorithm find the highest-quality solution when there are several different solutions?

# Outline

- The search problem in state spaces
- Examples
  - 8-puzzle game
  - 8-queen problem
- Basic search algorithms

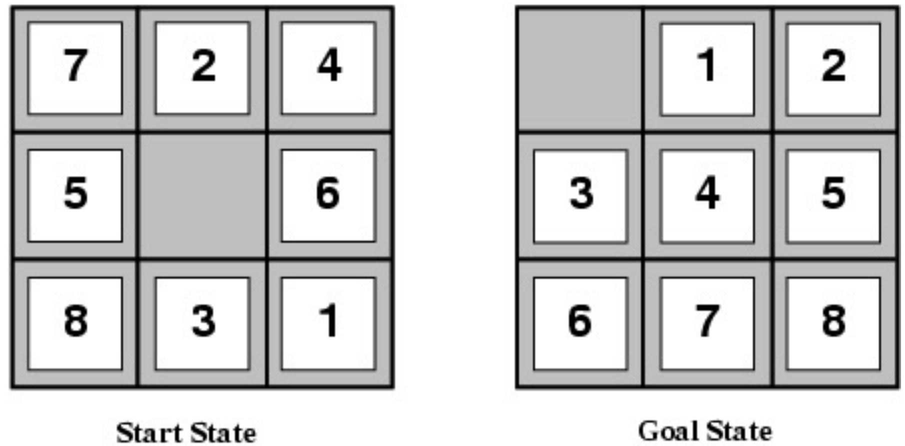
# 8-puzzle game (1/2)




Start State Goal State

(Russell & Norvig, 2010)

# 8-puzzle game (2/2)




(Russell & Norvig, 2010)

- States: combination of cell positions
- Initial state: arbitrary state
- Action: move the empty cell up, down, left, right
- Goal: a pre-defined goal state
- Cost: number of movements

# 8-queen problem (1/2)

Place 8 queens on an 8x8 chess board so that no queens threaten each other

# 8-queen problem (2/2)

Place 8 queens on an 8x8 chess board so that no queens threaten each other

- State: arrangement of 0 to 8 queens on the board
- Initial state: no queen on the board
- Action: place a queen on an empty cell
- Goal: 8 queens on an 8x8 chess board so that no queens threaten each other

# Outline

- The search problem in state spaces
- Examples
- Basic search algorithms
  - General search algorithm
  - Breadth-first search (BFS)
  - Uniform-cost search (UCS)
  - Depth-first search (DFS)
  - Iterative deepening search (IDS)

# General search algorithm (1/3)

- **General idea:** consider states, using successor functions to extend those states until the desired state is reached

- Expanding states creates a “search tree”
  - Each state is a node
  - Open nodes are nodes waiting for further expansion
  - Expanded nodes are called closed nodes

# General search algorithm (2/3)

$Search(Q, S, G, P)$

($Q$: state space, $S$: initial state, $G$: goals, $P$: successor function)

**Input**: search problem

**Output**: goal state (path to the goal state)

**Initialize**: $O \leftarrow S$ ($O$: the open node list)

**while**($O \neq \emptyset$) **do**

1. Select a node $n \in O$ and delete $n$ from $O$
2. **if** $n \in G$, **return** (path to $n$)
3. Add $P(n)$ to $O$

**return**: no solution

# General search algorithm (3/3)

$Search(Q, S, G, P)$

($Q$: state space, $S$: initial state, $G$: goals, $P$: successor function)

**Input**: search problem

**Output**: goal state (path to the goal state)

**Initialize**: $O \leftarrow S$ ($O$: the open node list)

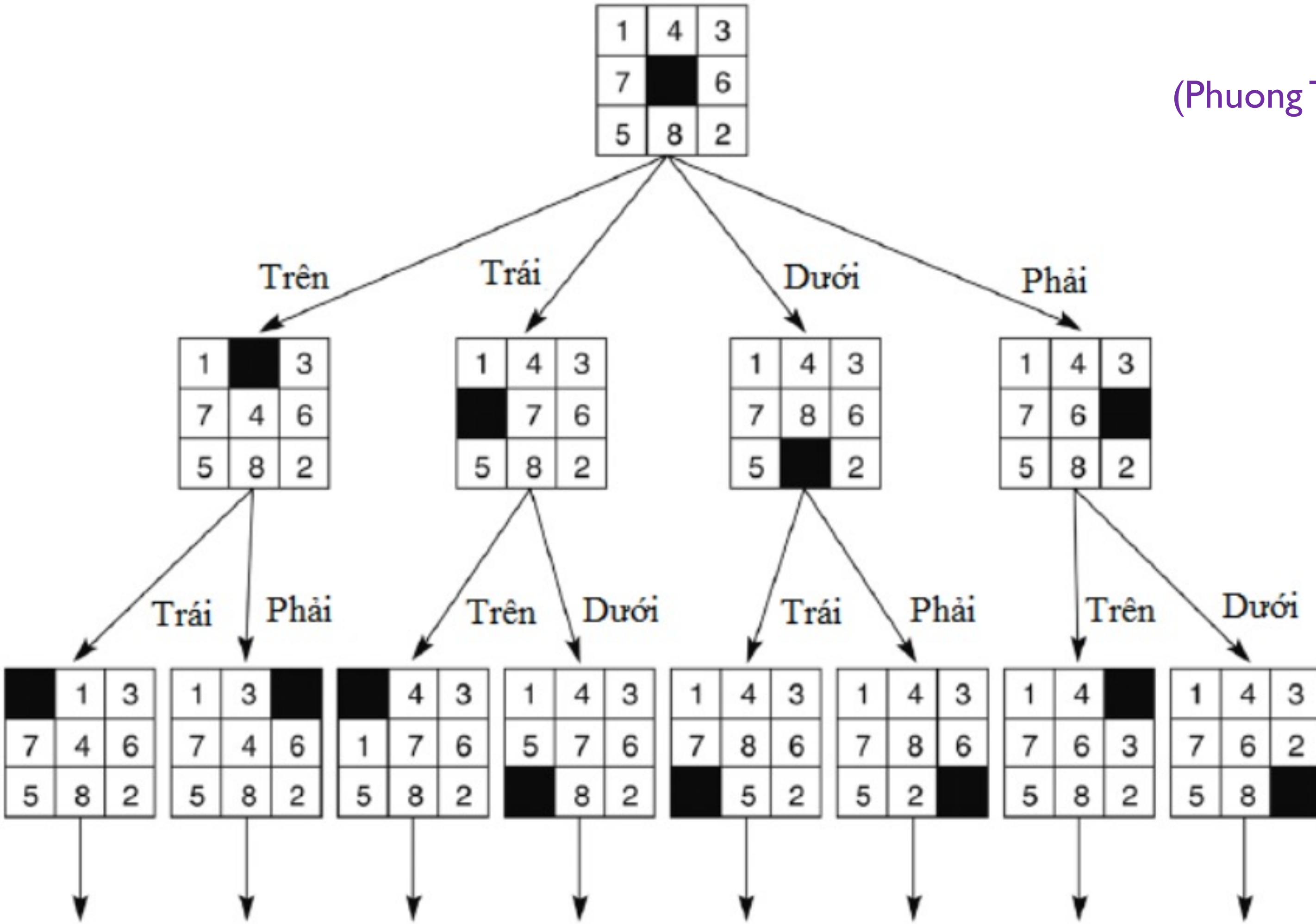
**while**($O \neq \emptyset$) **do**

1. Select a node $n \in O$ and delete $n$ from $O$
2. **if** $n \in G$, **return** (path to $n$)
3. Add $P(n)$ to $O$

**return**: no solution

How to select node n?

# Example of a search tree




(Phuong TM, 2016)

# Search strategies

- A search strategy is determined by the order in which the nodes in the search tree are expanded

- Criteria for evaluating search strategies:
  - Completeness: is guaranteed to find a solution (when there is)?
  - Computational complexity: number of generated nodes
  - Space complexity: number of nodes stored concurrently in memory
  - Optimality: does the algorithm find the best solution?
- The complexity is calculated based on the following parameters
  - $b$: branching factor (of the search tree)
  - $d$: depth of the solution
  - $m$: maximum depth of the state space (may be $\infty$)

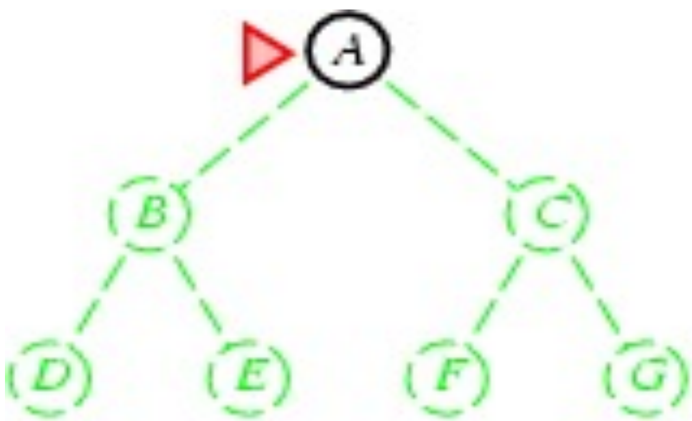
# Blind search (Uninformed search)

- Blind search only uses information according to the problem statement during the search process

- Blind search algorithms
  - Breadth-first search (BFS)
  - Uniform-cost search (UCS)
  - Depth-first search (DFS)
  - Iterative deepening search (IDS)

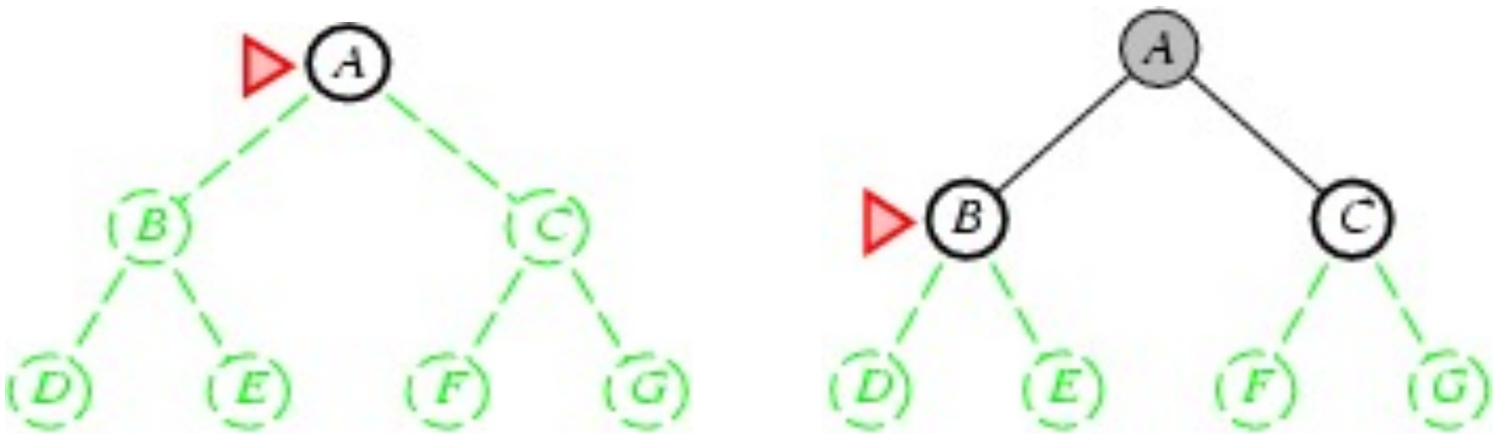
# Breadth-first search – BFS (1/4)

- **Principle**: among open nodes, choose the shallowest node (closest to the root) to expand



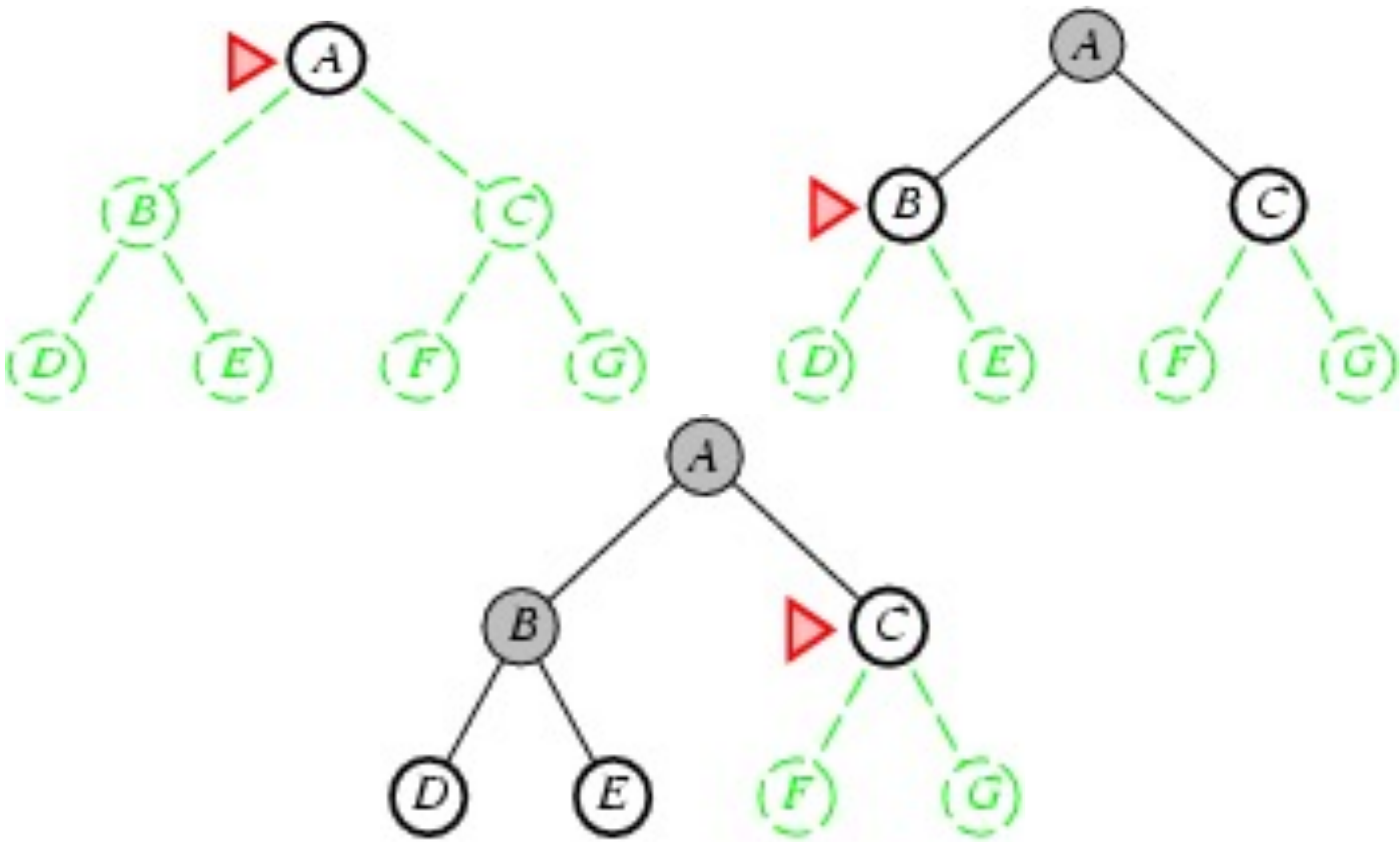
# Breadth-first search – BFS (2/4)

- **Principle**: among open nodes, choose the shallowest node (closest to the root) to expand



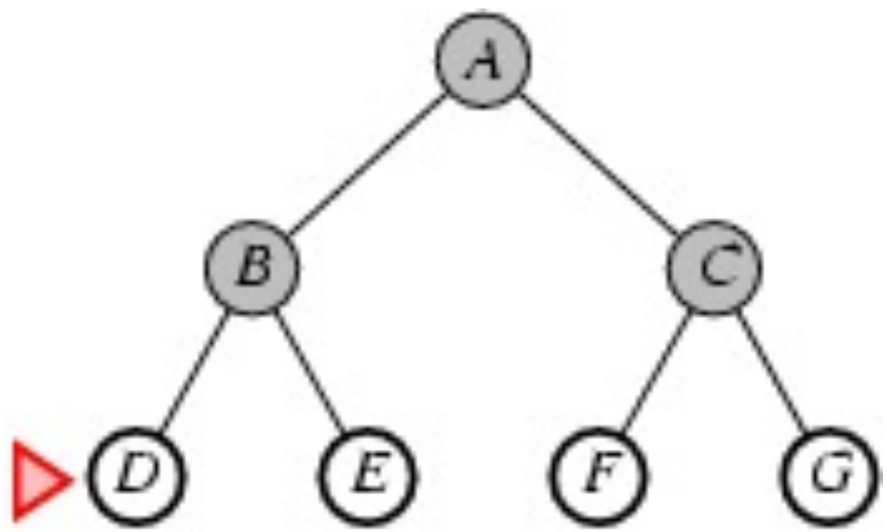
# Breadth-first search – BFS (3/4)

- **Principle**: among open nodes, choose the shallowest node (closest to the root) to expand



# Breadth-first search – BFS (4/4)

- Remember the path
  - When switching to a node, remember the parent node of that node by using a back pointer
  - After reaching the goal, the back pointer is used to find the path back to the initial node



# BFS algorithm (1/2)

$Search(Q, S, G, P)$

($Q$: state space, $S$: initial state, $G$: goals, $P$: successor function)

**Input**: search problem

**Output**: goal state (path to the goal state)

**Initialize**: $O \leftarrow S$ ($O$: the open node list)

**while**($O \neq \emptyset$) **do**

1. take the first node $n$ from $O$
2. **if** $n \in G$, **return** (path to $n$)
3. add $P(n)$ to the end of $O$

**return** no solution

# BFS algorithm (2/2)

$Search(Q, S, G, P)$

($Q$: state space, $S$: initial state, $G$: goals, $P$: successor function)

**Input**: search problem

**Output**: goal state (path to the goal state)

**Initialize**: $O \leftarrow S$ ($O$: the open node list)

**while**($O \neq \emptyset$) **do**

1. take the first node $n$ from $O$
2. **if** $n \in G$, **return** (path to $n$)
3. add $P(n)$ to the end of $O$

**return** no solution

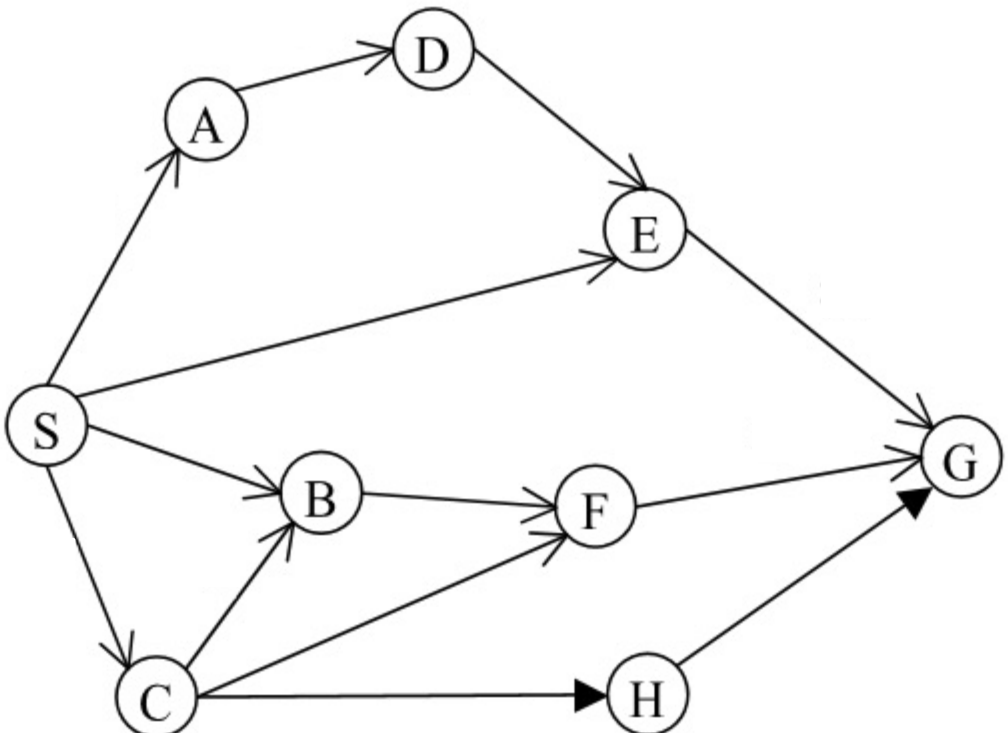
FIFO data structure (Queue)

# Avoiding repeated nodes

- There can be multiple paths reaching to a node
  - The algorithm can expand a node many times
  - Can lead to an infinite loop

- Solution
  - Do not add a node to the queue if the node is already expanded or is in the queue (waiting to be expanded)
    - Remember less nodes; check goal faster
    - Avoid the infinite loop

# BFS example (1/2)




(Phuong TM, 2016)

# BFS example (2/2)

| # | Expanded node | Open node list O (Queue) |
|---|---|---|
| 0 | | $S$ |
| 1 | $S$ | $A_S, B_S, C_S, E_S$ |
| 2 | $A_S$ | $B_S, C_S, E_S, D_A$ |
| 3 | $B_S$ | $C_S, E_S, D_A, F_B$ |
| 4 | $C_S$ | $E_S, D_A, F_B, H_C$ |
| 5 | $E_S$ | $D_A, F_B, H_C, G_E$ |
| 6 | $D_A$ | $F_B, H_C, G_E$ |
| 7 | $F_B$ | $H_C, G_E$ |
| 8 | $H_C$ | $G_E$ |
| 9 | $G_E$ | Goal |

Path: $G \leftarrow E \leftarrow S$

# Properties of BFS

- Completeness?
  - Yes (if $b$ is finite)
- Time?
  - $1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$
- Space?
  - $O(b^d)$ (store all nodes)
- Optimality?
  - Yes (if cost = 1 for every action)
  - Always search all the nodes at the higher level before searching nodes at the lower level

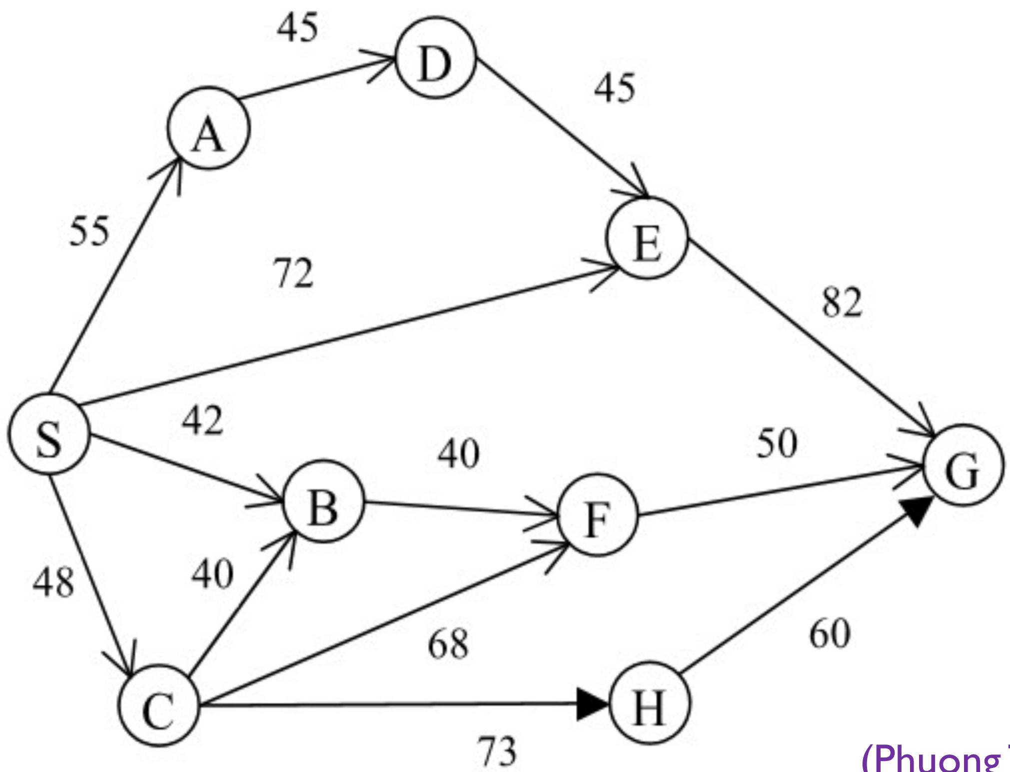
- Memory is more important than time

# Uniform-cost search (UCS)

- When the costs of actions (moving between two nodes) are different
  - BFS does not give an optimal solution
  - Need to use uniform-cost search (a variant of BFS)

- **Principle**: choose the node with the smallest cost to expand first instead of choosing the shallowest node like in BFS

# UCS example (1/2)




(Phuong TM, 2016)

# UCS example (2/2)

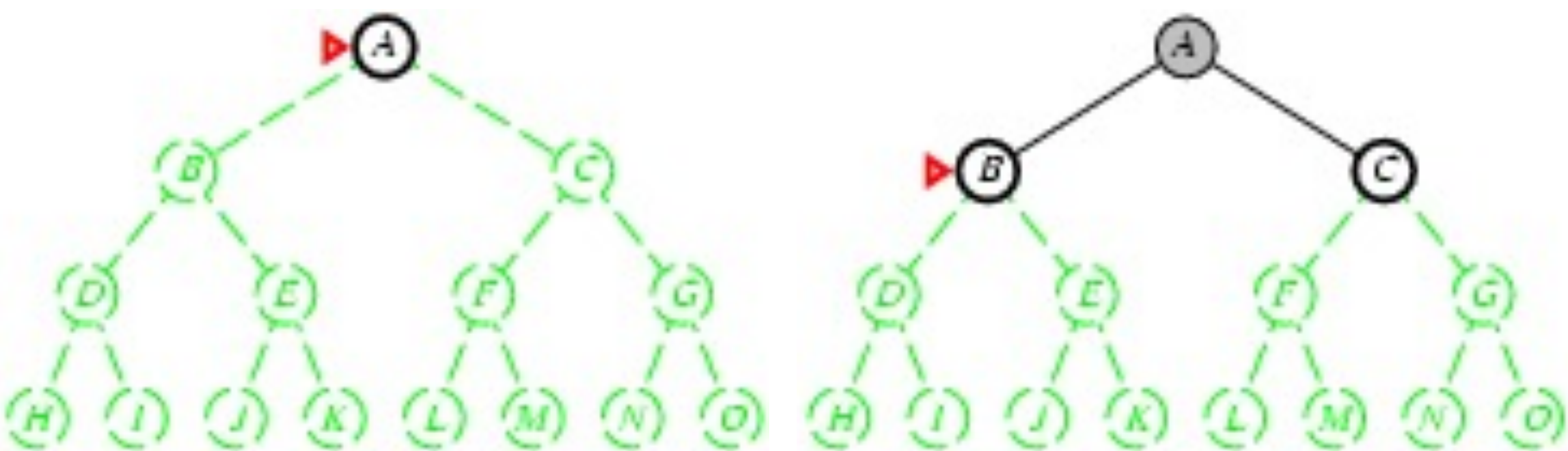
| # | Expanded node | Open node list O |
|---|---|---|
| 0 | | $S(0)$ |
| 1 | $S$ | $A_S(55), B_S(42), C_S(48), E_S(72)$ |
| 2 | $B_S$ | $A_S(55), C_S(48), E_S(72), F_B(82)$ |
| 3 | $C_S$ | $A_S(55), E_S(72), F_B(82),\ H_C(121)$ |
| 4 | $A_S$ | $E_S(72), F_B(82), H_C(121),\ D_A(100)$ |
| 5 | $E_S$ | $F_B(82), H_C(121), D_A(100),\ G_E(154)$ |
| 6 | $F_B$ | $H_C(121), D_A(100),\ G_F(132)$ |
| 7 | $D_A$ | $H_C(121), G_F(132)$ |
| 8 | $H_C$ | $G_F(132)$ |
| 9 | $G_F$ | Goal |

Update path to $G$

Path: $G \leftarrow F \leftarrow B \leftarrow S$

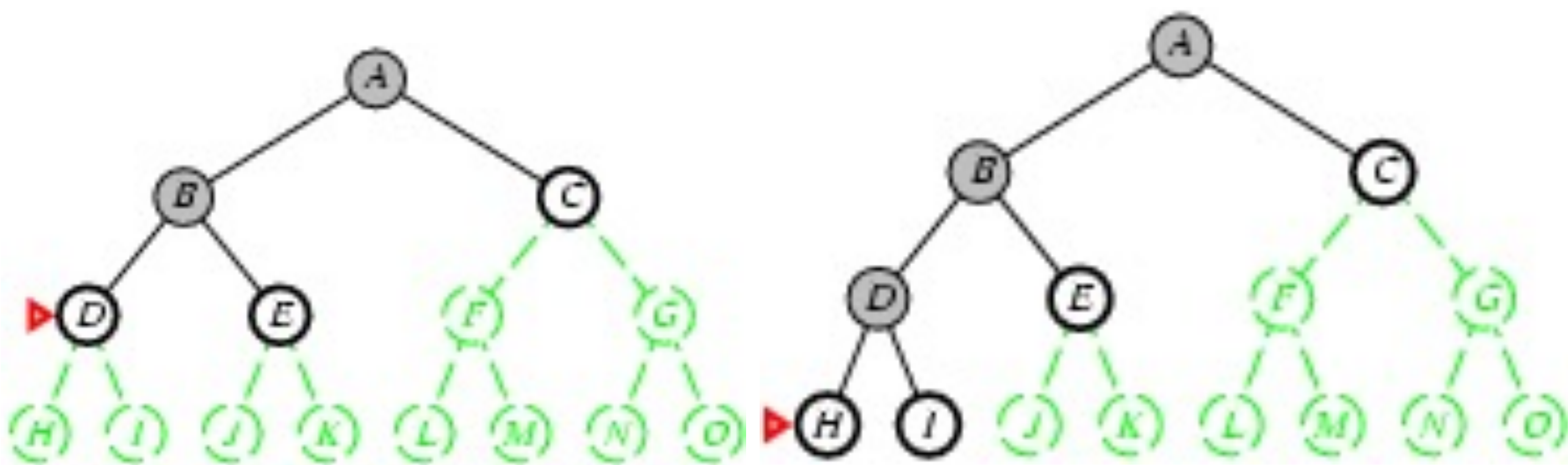
# Depth-first search - DFS (1/4)

- **Principle**: among open nodes, choose the deepest node (farthest to the root) to expand



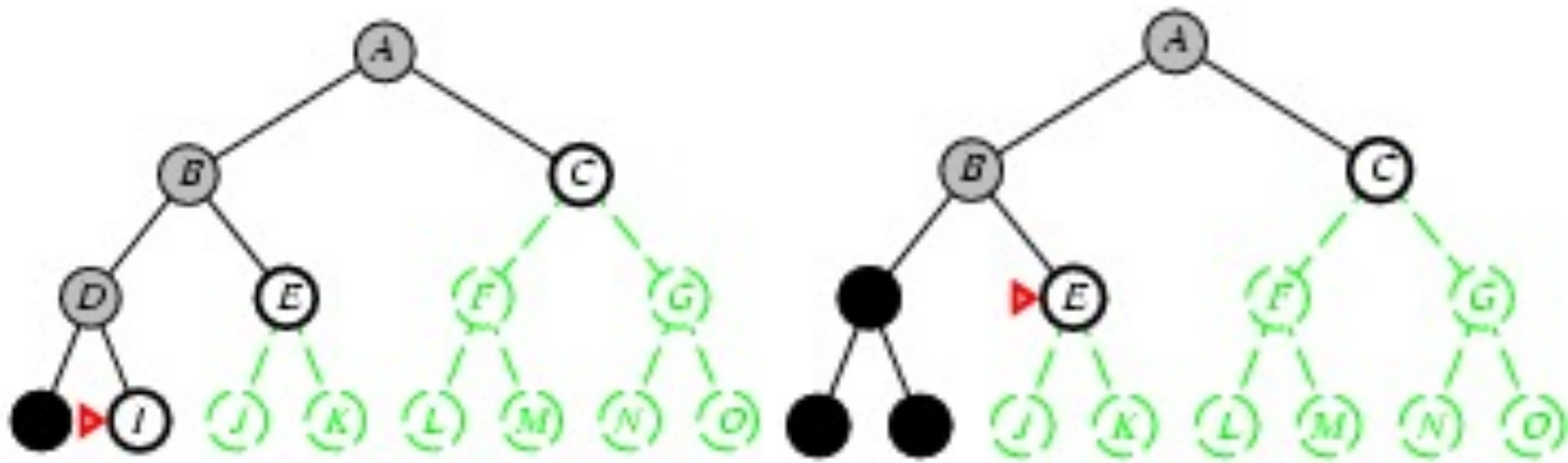
# Depth-first search - DFS (2/4)

- **Principle**: among open nodes, choose the deepest node (farthest to the root) to expand



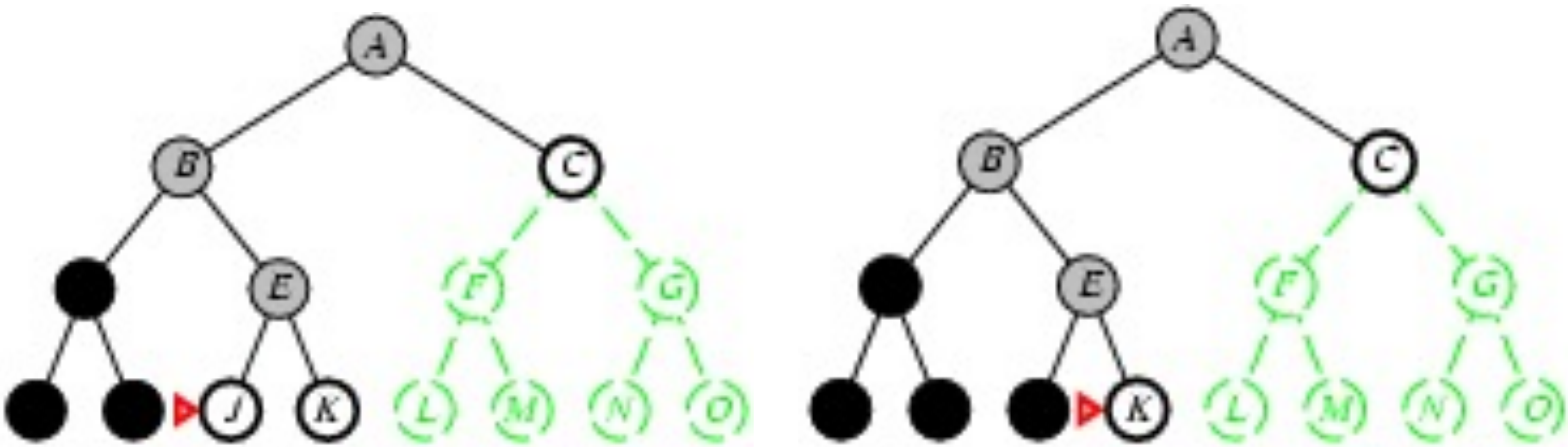
# Depth-first search - DFS (3/4)

- **Principle**: among open nodes, choose the deepest node (farthest to the root) to expand



# Depth-first search - DFS (4/4)

- **Principle**: among open nodes, choose the deepest node (farthest to the root) to expand



# DFS algorithm (1/2)

$Search(Q, S, G, P)$

($Q$: state space, $S$: initial state, $G$: goals, $P$: successor function)

**Input**: search problem

**Output**: goal state (path to the goal state)

**Initialize**: $O \leftarrow S$ ($O$: the open node list)

**while**($O \neq \emptyset$) **do**

1. take the first node $n$ from $O$
2. **if** $n \in G$, **return** (path to $n$)
3. add $P(n)$ to the head of $O$

**return** no solution

# DFS algorithm (2/2)

$Search(Q, S, G, P)$

($Q$: state space, $S$: initial state, $G$: goals, $P$: successor function)

**Input**: search problem

**Output**: goal state (path to the goal state)

**Initialize**: $O \leftarrow S$ ($O$: the open node list)

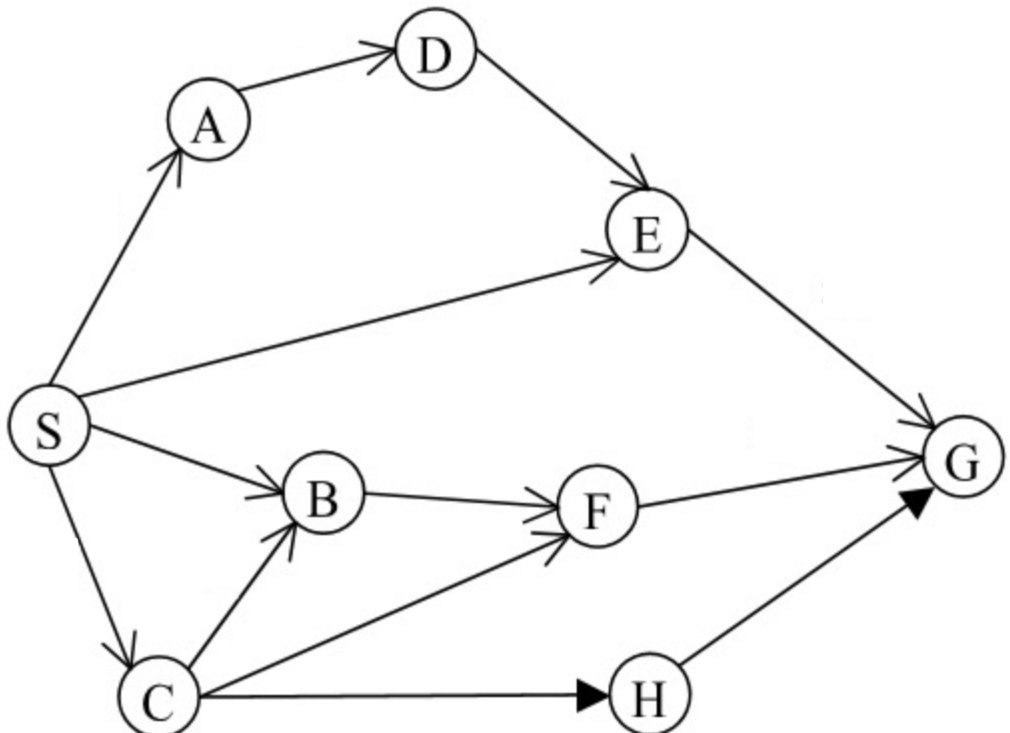
**while**($O \neq \emptyset$) **do**

1. take the first node $n$ from $O$
2. **if** $n \in G$, **return** (path to $n$)
3. add $P(n)$ to the head of $O$

**return** no solution

LIFO data structure (Stack)

# DFS example (1/2)




(Phuong TM, 2016)

# DFS example (2/2)

| # | Expanded node | Open node list O (Stack) |
| --- | --- | --- |
| 0 | | $S$ |
| 1 | $S$ | $A_S, B_S, C_S, E_S$ |
| 2 | $A_S$ | $D_A, B_S, C_S, E_S$ |
| 3 | $D_A$ | $E_D, B_S, C_S, E_S$ |
| 4 | $E_D$ | $G_E, B_S , C_S, E_S$ |
| 5 | $G_E$ | Goal |

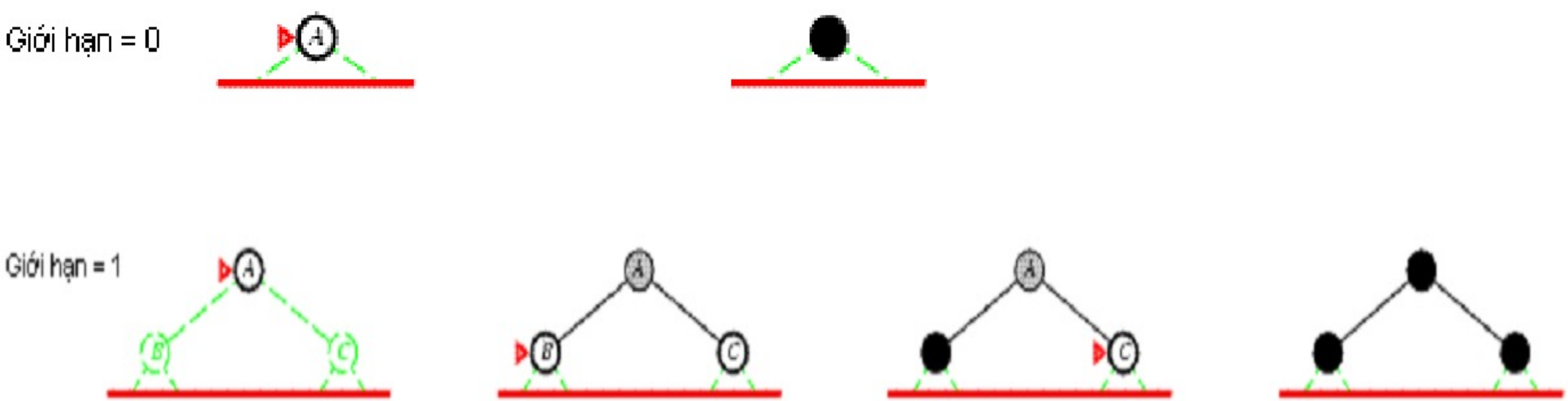
Path: $G \leftarrow E \leftarrow D \leftarrow A \leftarrow S$
Depth: 4

# Properties of DFS

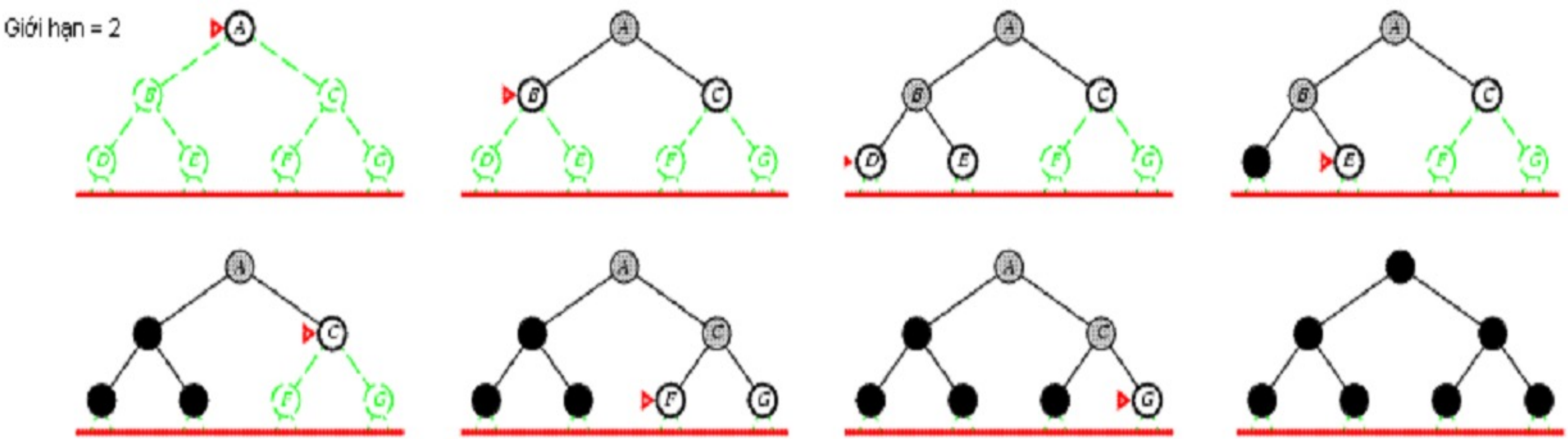
- Completeness?
  - No: when the depth of state space is infinite
- Optimality?
  - No
- Time?
  - $O(b^m)$: very large if $m$ is greater than $d$
  - If there are many solutions, DFS can be much faster than BFS
- Space?
  - $O(bm)$: much better than BFS

# Iterative deepening search–IDS(1/3)

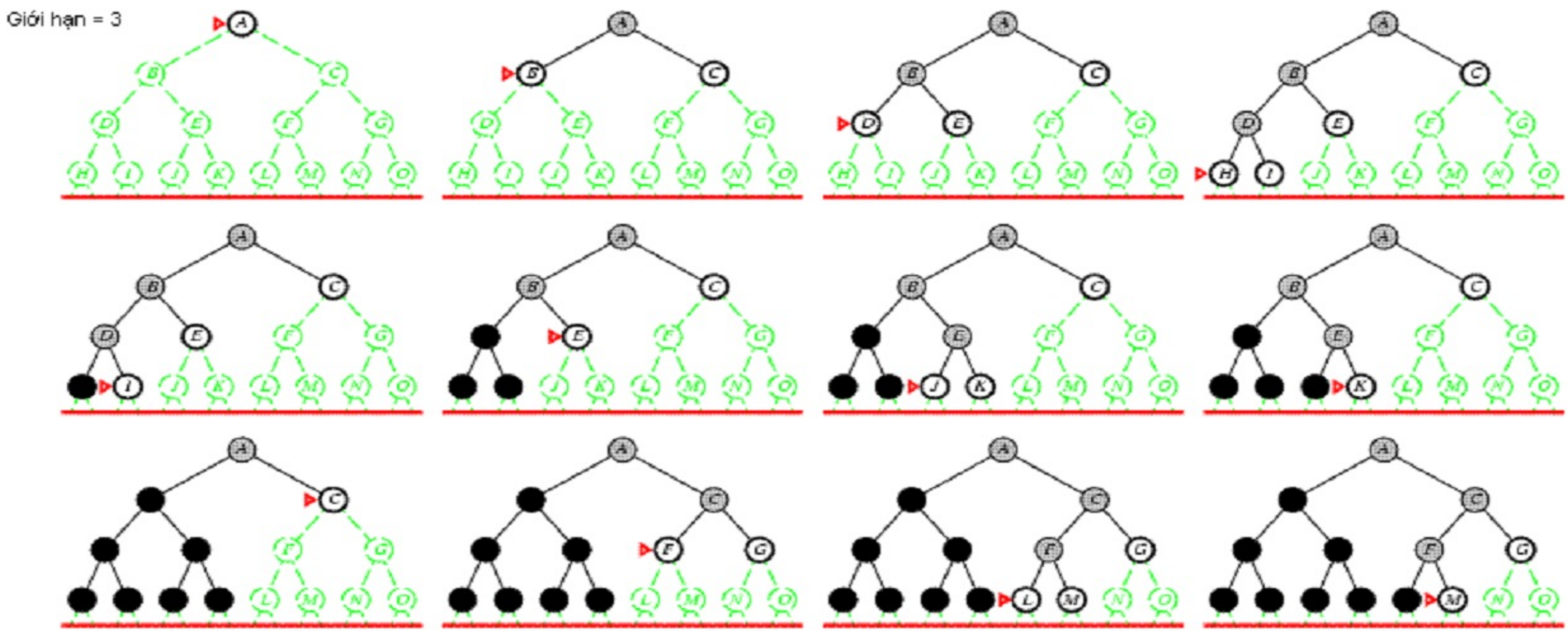
- **Principle**: use DFS but never extend nodes with depth beyond a certain limit. The depth limit will be gradually increased until a solution is found.



# Iterative deepening search–IDS(2/3)



# Iterative deepening search–IDS(3/3)



# IDS algorithm

$Search(Q, S, G, P)$

($Q$: state space, $S$: initial state, $G$: goals, $P$: successor function)

**Input**: search problem

**Output**: goal state (path to the goal state)

**Initialize**: $O \leftarrow S$ ($O$: the open node list)

$c = 0$ (current depth)

**while** (1) **do**

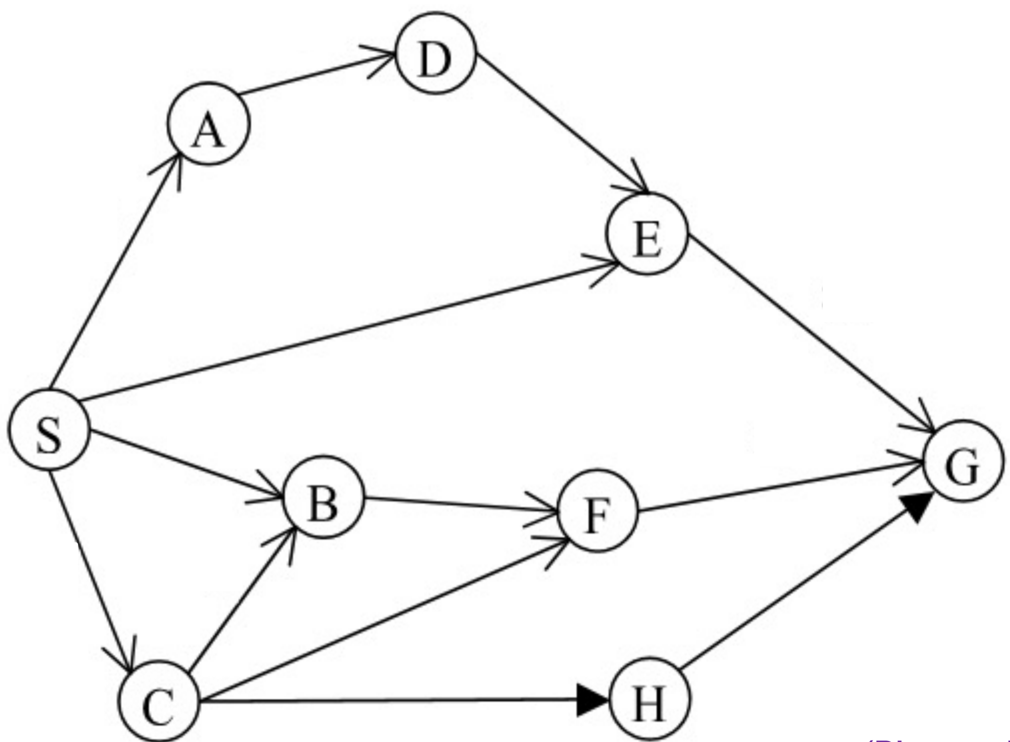
1. **while** ($O \neq \emptyset$) **do**
   a. take the first node $n$ from $O$
   b. **if** $n \in G$, **return** (path to n)
   c. **if** $depth(n) < c$ **then**
      add $P(n)$ to the head of $O$
2. $c++$; $O = S$

# Properties of IDS

- Completeness?
  - Yes
- Optimality?
  - Yes: iff there are multiple solutions, IDS can find the solution closest to the root
- Space?
  - $O(bd)$: small
- Time?
  - $(d+1)1 \ + \ db \ + \ (d-1)b^2 + \ldots + 2b^{d-1} + 1b^d = O(b^d)$

Has advantages of both BFS and DFS

# IDS example




(Phuong TM, 2016)

# Summary

| | BFS | UCS | DFS | IDS |
|---|---|---|---|---|
| Complete? | Yes | Yes | No | Yes |
| Optimal? | Yes | Yes | No | Yes |
| Time | $O(b^d)$ | $O(b^{\lceil C^*/\epsilon \rceil})$ | $O(b^m)$ | $O(b^d)$ |
| Space | $O(b^d)$ | $O(b^{\lceil C^*/\epsilon \rceil})$ | $O(bm)$ | $O(bd)$ |

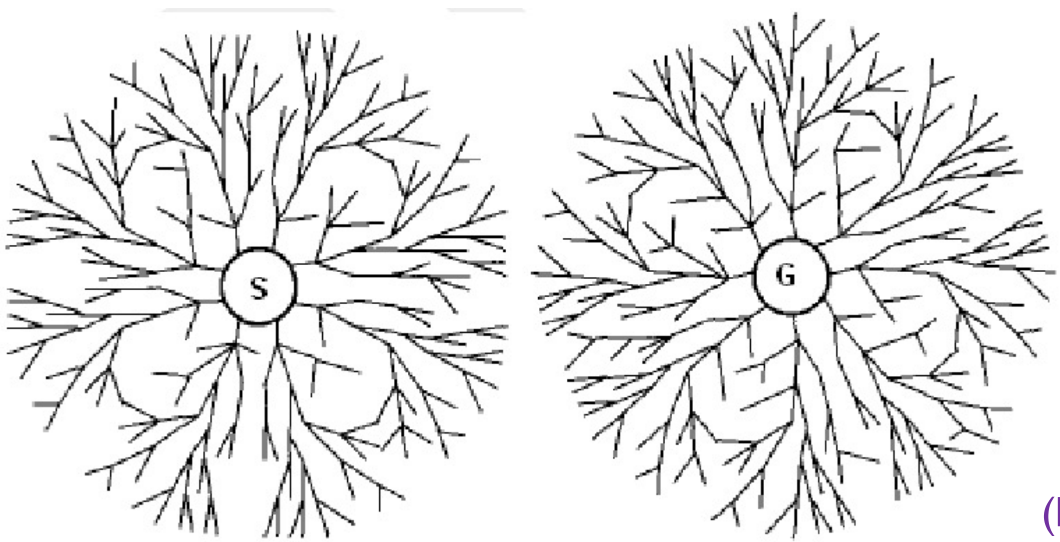
- Choose BFS if the branching factor is small
- Choose DFS if the maximum depth is known in advance and there are multiple goal states
- Choose IDF if the search tree has a large depth ($m$)

# When to add repeated nodes to the open node list?

- BFS
  - **No**: adding repeated nodes does not change the order of expanding nodes in the queue; does not change the solution of the problem; may lead to loops.
- UCS
  - In cases the repeated node has better cost, it will be added to the list (if it is already expanded) or updated to replace the old node (if it is on the list).
- DFS
  - **Yes**: adding a repeated node to the stack changes the order of expanding nodes in the stack; changes the solution.
  - Do not add expanded nodes to the stack
- IDS:
  - **Yes**: to ensure the optimality

# Bidirectional search (1/2)

- **Principle**: simultaneous search from the initial and goal nodes
  - There exist two search trees, one rooted at the initial node and the other rooted at the goal node
  - Search ends when a leaf of one tree matches a leaf of the other
- Illustration of search trees




(Phuong TM, 2016)

# Bidirectional search (2/2)

- Note
  - Need to use BFS
    - DFS may not yield a solution if two search trees grow along two branches that do not match each other
  - Functions
    - $P(x)$: set of child nodes of $x$
    - $D(x)$: set of father nodes of $x$
- Properties
  - Node matching is time consuming ($b^d$ nodes for each tree)
  - Computational complexity $O(b^{d/2})$
    - Number of expanded nodes is significantly reduced