
# CSE 315: Artificial Intelligence

## Topic – 3: Solving Problems by Searching

Department of CSE

Daffodil International University

# Topic Contents

- Problem Solving Agents
- Problem Formulation and Search Spaces
- Example Problems
- Tree Search Algorithm
  - Breadth-First Search
  - Uniform-Cost Search
  - Depth-First Search
  - Depth-Limited Search
  - Iteratively Deepening Search
  - Bidirectional Search

# Introduction

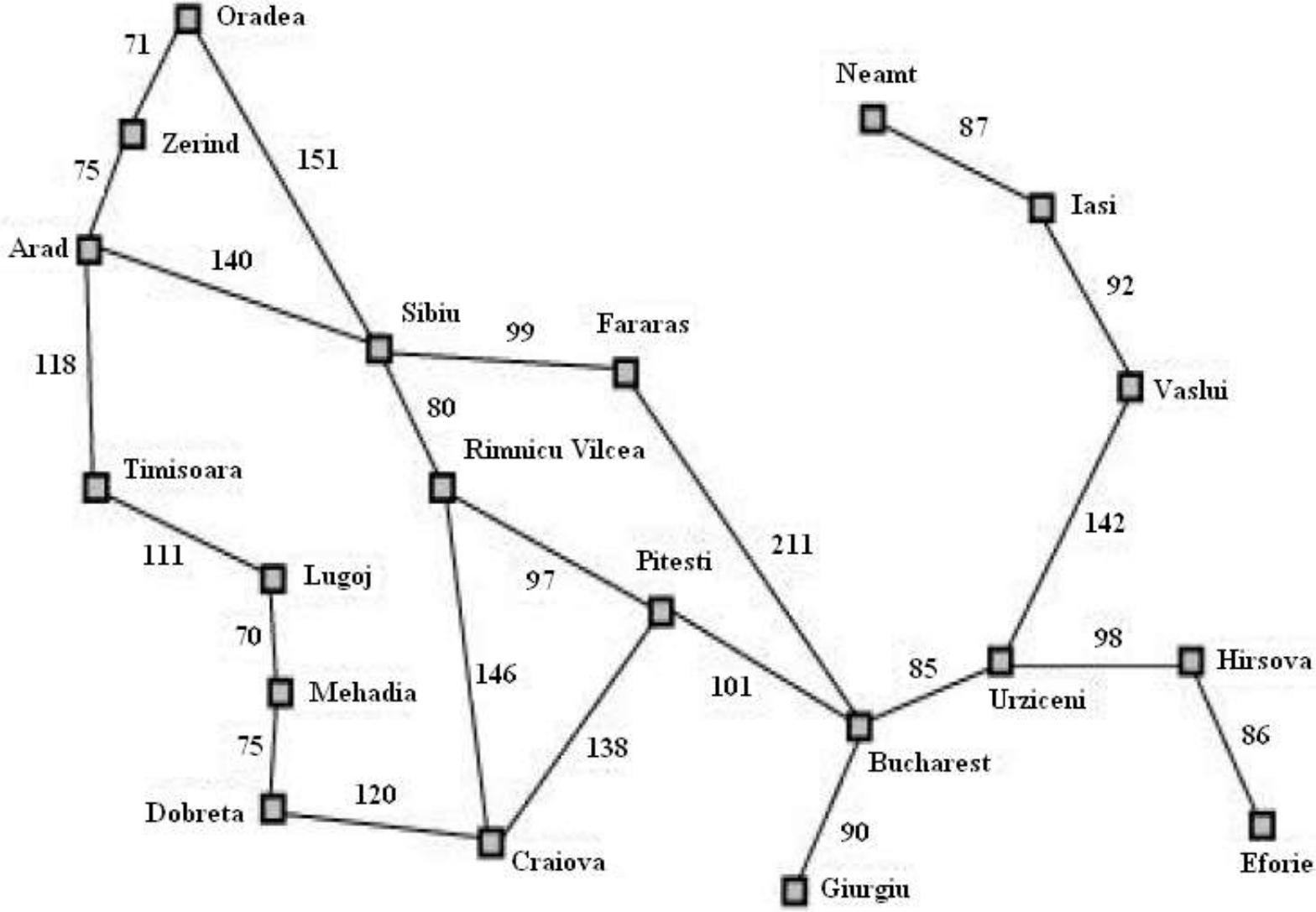
- In this topic, we see how an agent can find a sequence of actions that achieves its goals, when no single action will do.

# Problem-Solving Agent

Four general steps in problem solving:

- **Goal formulation**
  - What are the successful world states
- **Problem formulation**
  - What actions and states to consider given the goal
- **Search**
  - Examine different possible sequences of actions that lead to states of known value and then choose the best sequence
- **Execute**
  - Perform actions on the basis of the solution

# Example: Romania



# Example: Romania

On holiday the agent is in Romania visiting in Arad. Flight leaves tomorrow from Bucharest.

- Formulate goal
  - Be in Bucharest
- Formulate problem
  - States: various cities
  - Actions: drive between cities
- Find solution
  - Sequence of cities; e.g. Arad, Sibiu, Fagaras, Bucharest,…

# Problem Type

Given how we have defined the problem:

- Environment is **fully observable**
- Environment is **deterministic**
- Environment is **sequential**
- Environment is **static**
- Environment is **discrete**
- Environment is **single-agent**

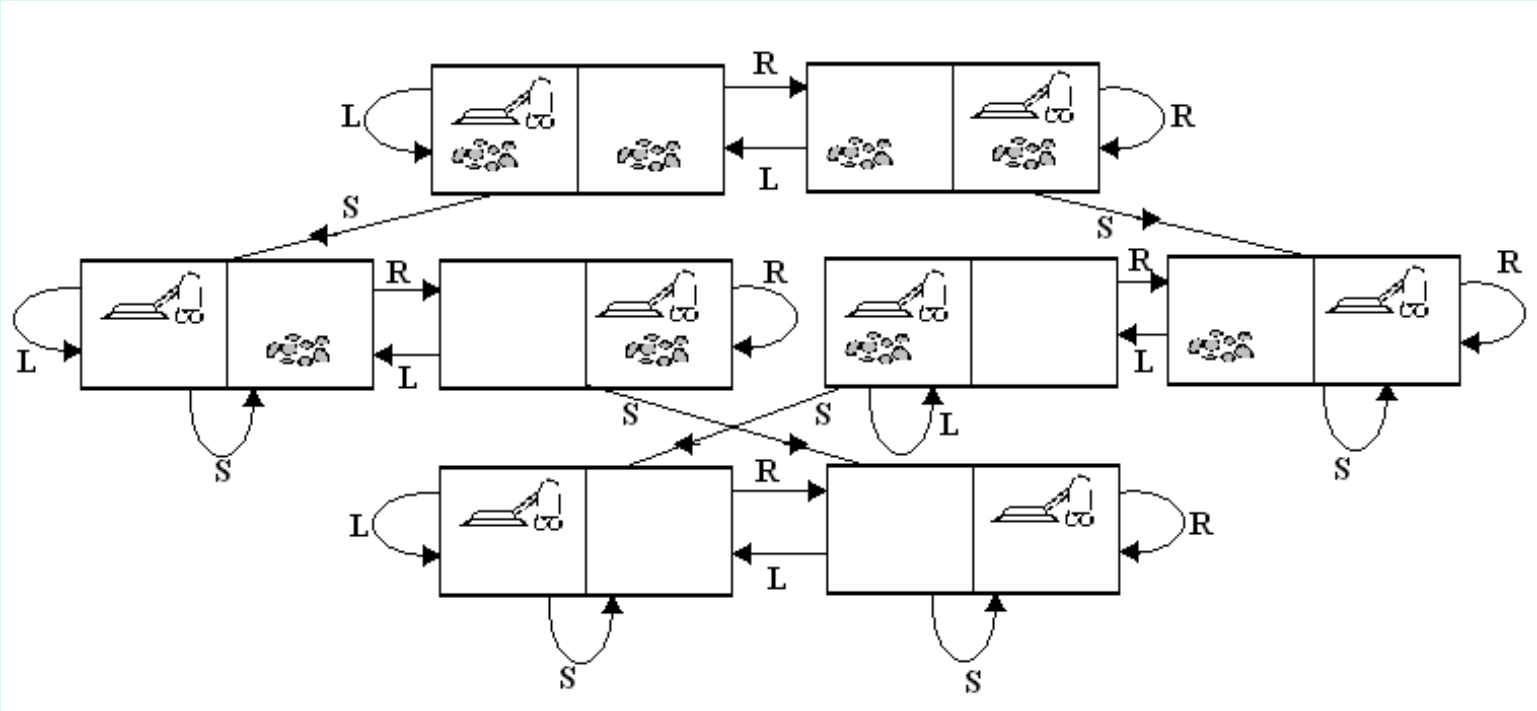
# Problem Formulation

- A problem is defined by:
  - An initial state, e.g. *In(Arad)*
  - A successor function *S*(*x*) = set of action-state pairs
    - *S(In(Arad)) = {(Go(Sibiu), In(Sibiu)), (Go(Zerind), In(Zerind)), (Go(Timisoara), In(Timisoara))}*
    - initial state + successor function = state space
  - Goal test
    - Explicit, e.g. *x='In(Bucharest)'*
  - Path cost (assume additive)
    - e.g. sum of distances, number of actions executed, …
- A solution is a sequence of actions from initial to goal state
  - the optimal solution has the lowest path cost

# State Space

- The state space of a problem is the set of all states reachable from the initial state.
- The state space forms a graph, called a state space graph, in which the nodes are states and the arcs (may be labeled with costs) between nodes are actions.
- The map of Romania shown in an earlier slide can be interpreted as a state space graph if we view each road as standing for two driving actions, one in each direction.

# State Space

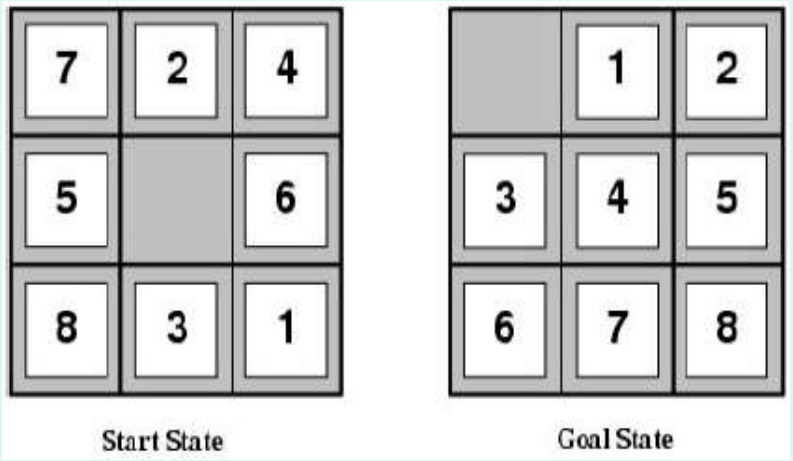



State space of the vacuum world

# Example Problems

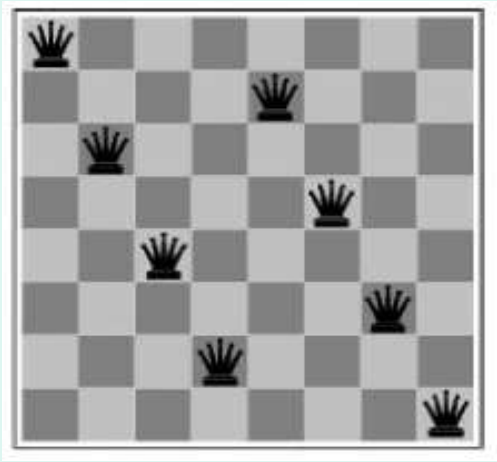
- The problem-solving approach has been applied to a vast array of task environments.
- Some of the best known are listed here, distinguishing between *toy and real-world* problems.
- A ***toy problem*** is intended to illustrate or exercise various problem-solving methods.
- It is usable by different researchers to compare the performance of algorithms.
- A ***real-world problem*** is one whose solutions people actually care about.
- We will mainly focus on toy problems in this topic.

# Toy Problems: 8-Puzzle




- **States:** location of each tile plus blank
- **Initial state:** Any state can be initial
- **Actions:** Move blank {*Left*, *Right*, *Up*, *Down*}
- **Goal test:** Check whether goal configuration is reached
- **Path cost:** Number of actions to reach goal

# Toy Problems: 8-Queens

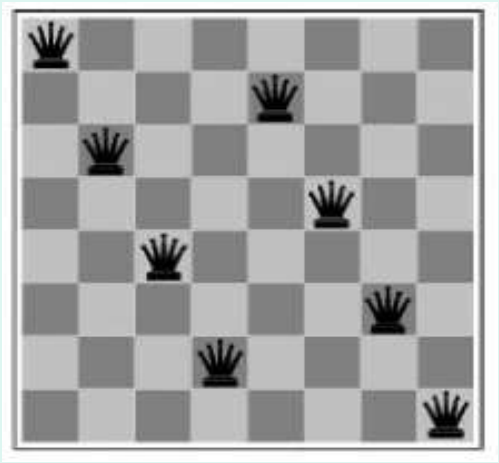


Place eight queens on a chessboard such that no queen attacks any other.

Two kinds of problem formulation:

- Complete-state formulation
- Incremental formulation

# Toy Problems: 8-Queens

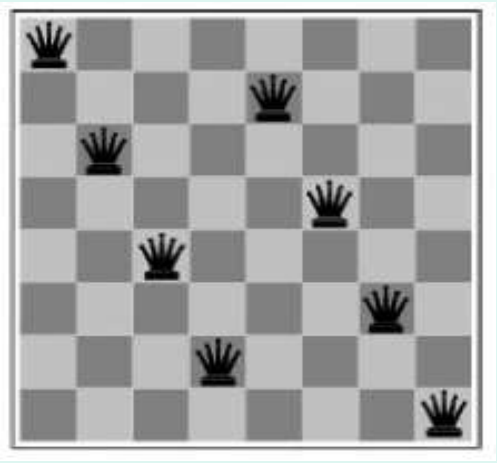


Complete-state formulation

- States: Any arrangement of 0 to 8 queens on the board
- Initial state: No queens
- Actions: Add queen to any empty square
- Goal test: 8 queens on board and none attacked
- Path cost: N/A

64.63.62…..57 ≈ 3 x $10^{14}$ possible sequences to investigate

# Toy Problems: 8-Queens



Incremental formulation

- States: *n* (0 to 8) queens on the board, one per column in the *n* leftmost columns with no queen attacking any other
- Actions: Add queen in leftmost empty column such that the queen is not attacking any other queen
- 2057 possible sequences to investigate; solutions are easy to find

But with 100 queens the problem is much harder

# Real-World Problems

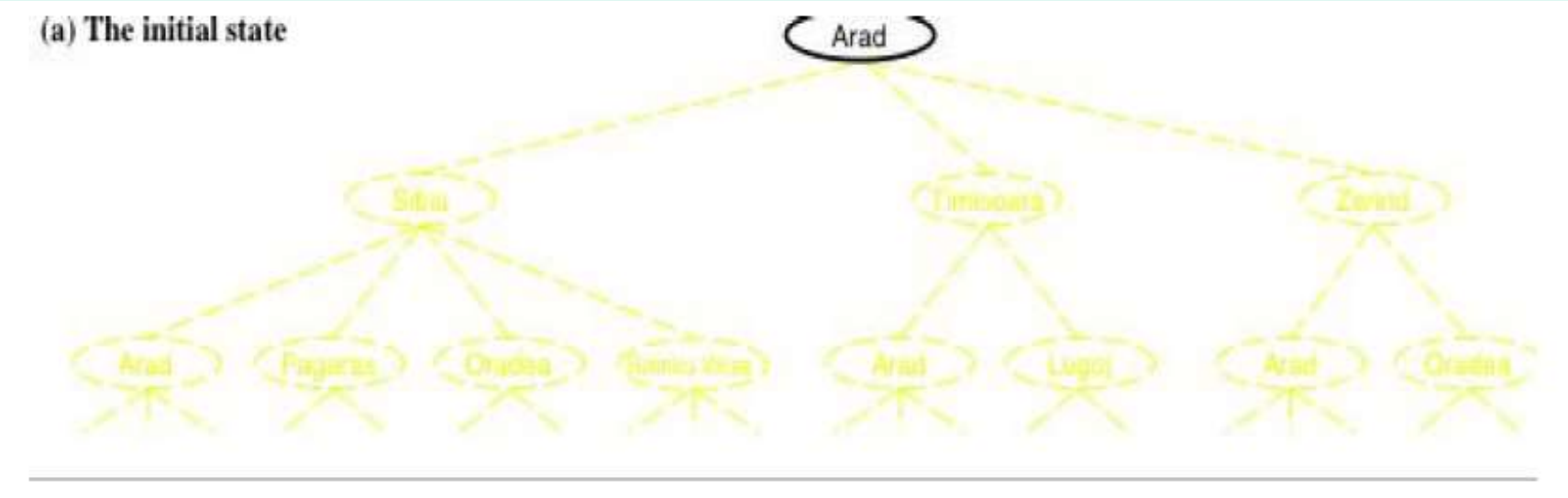
- Route-finding problems
- Touring problems
- Traveling Salesman problem
- VLSI layout problem
- Robot navigation
- Automatic assembly sequencing
- Internet searching

# Basic Search Algorithms

How do we find the solutions of previous problems?

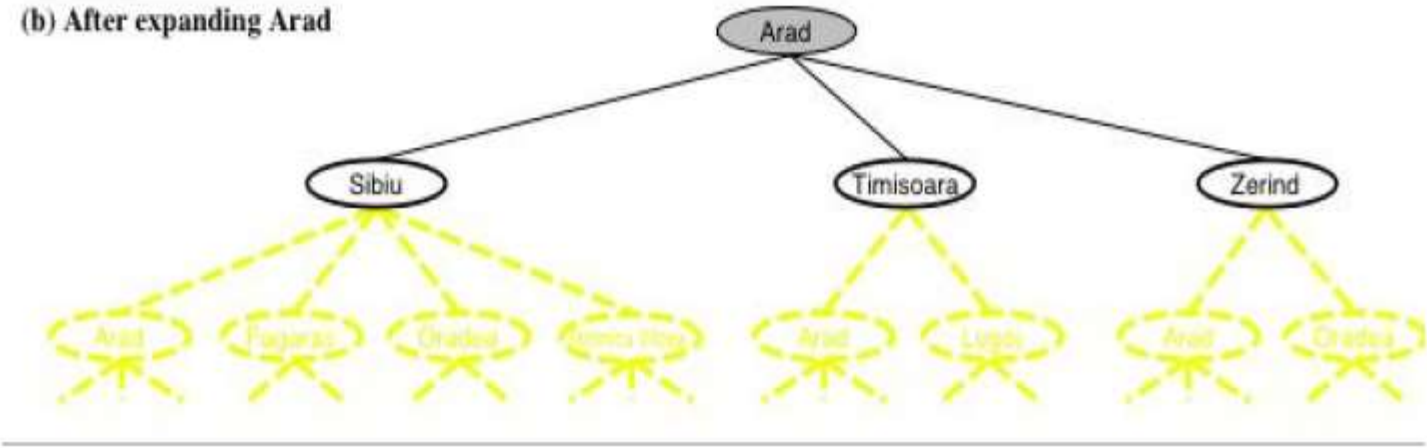
- Search the state space
  - State space can be represented by a **search tree**
  - Root of the tree is the initial state
  - Children generated through successor function
- In general we may have a search graph rather than tree (same state can be reached through multiple paths)

# Simple Tree Search Example




```
function TREE-SEARCH(problem, strategy) return a solution or failure
    Initialize search tree to the initial state of the problem
    do
        if no candidates for expansion then return failure
        choose leaf node for expansion according to strategy
        if node contains goal state then return solution
        else expand the node and add resulting nodes to the search tree
    enddo
```

# Simple Tree Search Example




**function** TREE-SEARCH(*problem, strategy*) **return** a solution or failure

Initialize search tree to the *initial state* of the *problem*

**do**

**if** no candidates for expansion **then return** *failure*

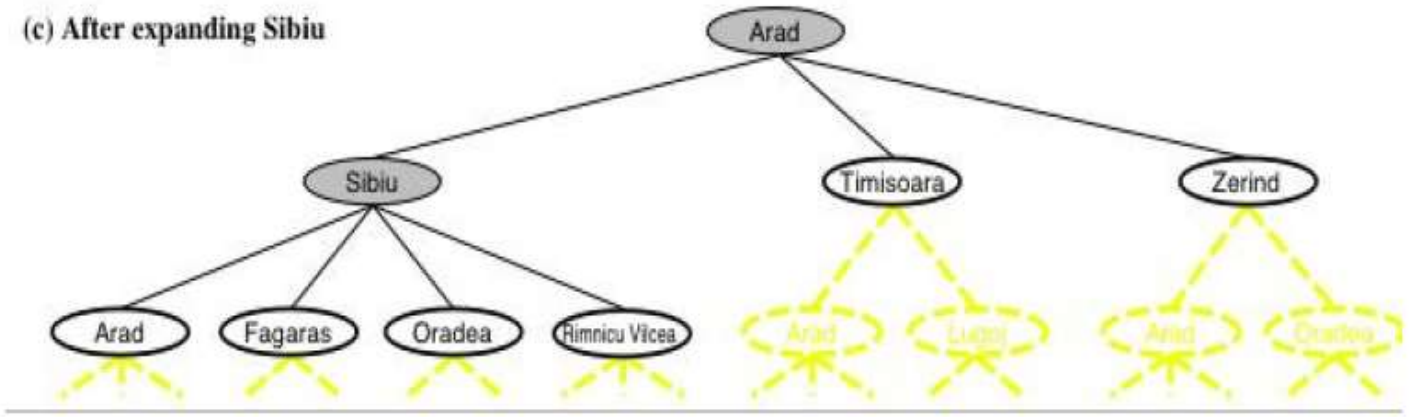
choose leaf node for expansion according to *strategy*

**if** node contains goal state **then return** *solution*

**else** expand the node and add resulting nodes to the search tree

**enddo**

# Simple Tree Search Example




**function** TREE-SEARCH(*problem, strategy*) **return** a solution or failure

Initialize search tree to the *initial state* of the *problem*

**do**

**if** no candidates for expansion **then return** *failure*

choose leaf node for expansion according to *strategy* ← **Determines search process**

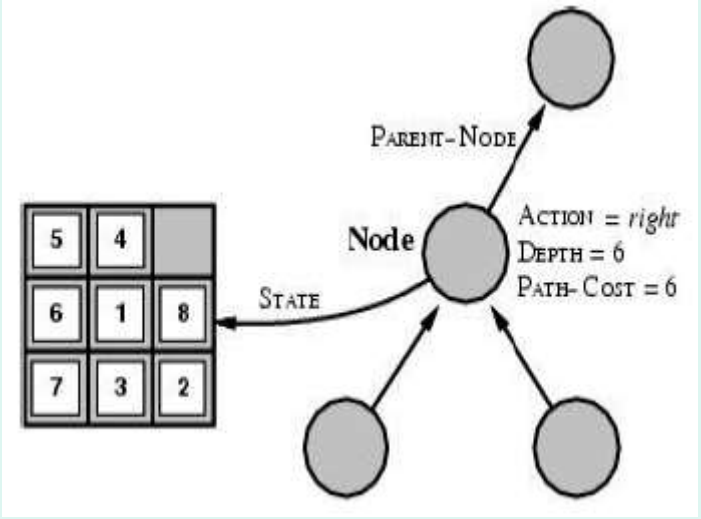
**if** node contains goal state **then return** *solution*

**else** expand the node and add resulting nodes to the search tree

**enddo**

# State Space vs. Search Tree

- A *state* corresponds to a configuration of the world
- A *node* is a data structure in a search tree
  - *e.g. node = <state, parent-node, action, path-cost, depth>*



# Search Strategies

- A search strategy is defined by picking the order of node expansion
- Problem-solving performance is measured in four ways:
  - **Completeness**: Is a solution found if one exists?
  - **Optimality**: Does the strategy find the optimal solution?
  - **Time Complexity**: How long does it take to find a solution*?*
  - **Space Complexity**: How much memory is needed to perform the search*?*
- Time and space complexity are measured in terms of problem difficulty defined by:
  - *b* - branching factor of the search tree
  - *d* - depth of the shallowest goal node
  - *m* - maximum length of any path in the state space

# Uninformed Search Strategies

- **Uninformed search** (or **blind search**)
  - Strategies have no additional information about states beyond that provided in the problem definition

- Uninformed strategies (defined by order in which nodes are expanded):
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
  - Bidirectional search

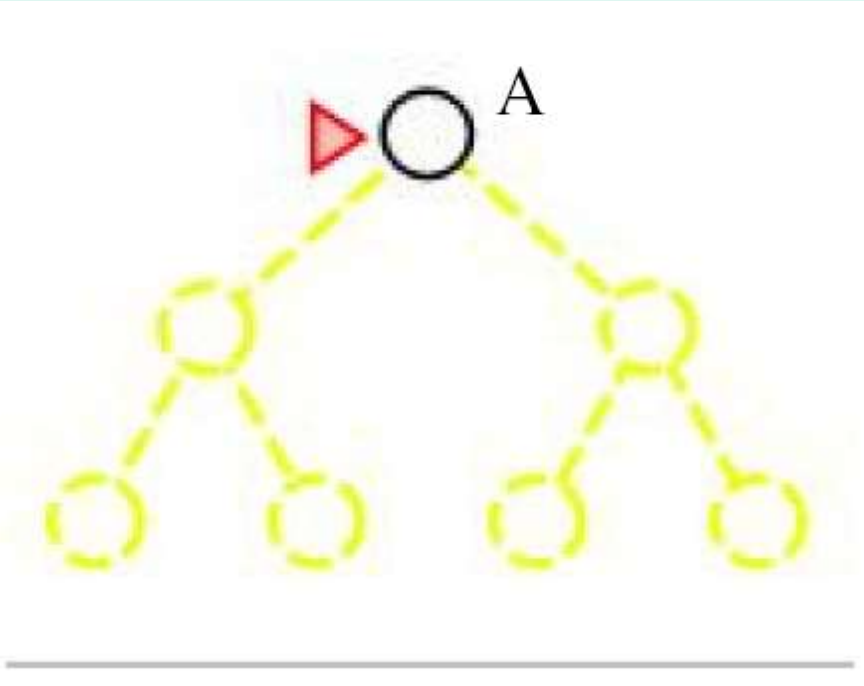
# Informed Search Strategies

- **Informed** (or **heuristic**) search
  - Search strategies know whether one state is more promising than another

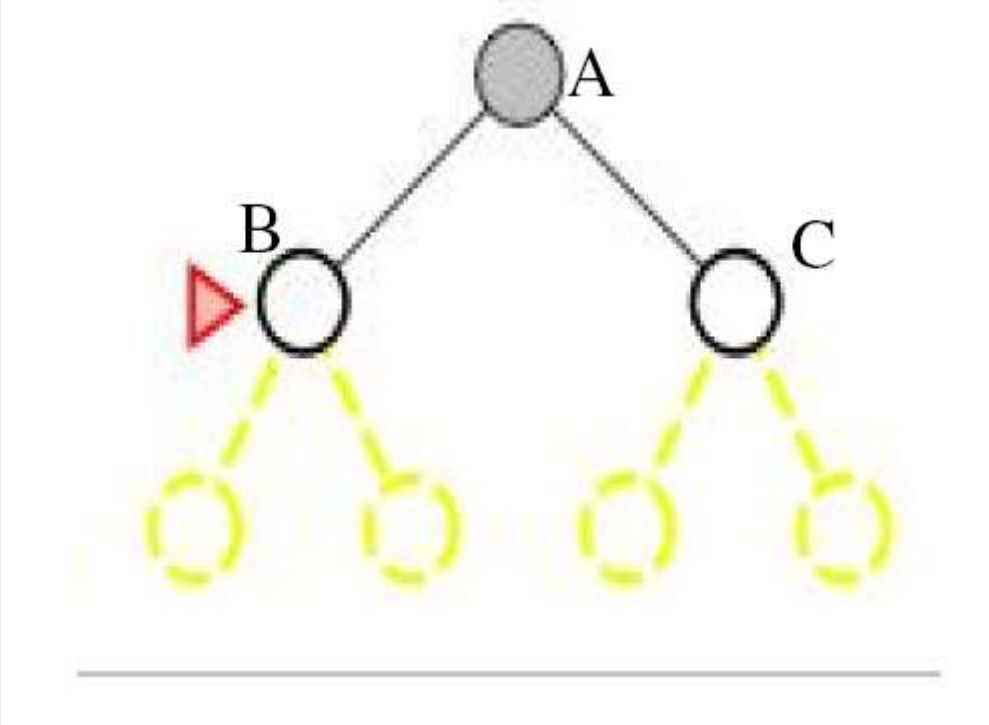
- Examples of informed search strategies:
  - Greedy best-first search
  - A* search
  - Iterative-deepening A* (IDA*) search
  - Recursive best-first search (RBFS)
  - Memory-bounded A* (MA*) search
  - Simplified memory-bounded A* (SMA*) search

# Breadth-First Search

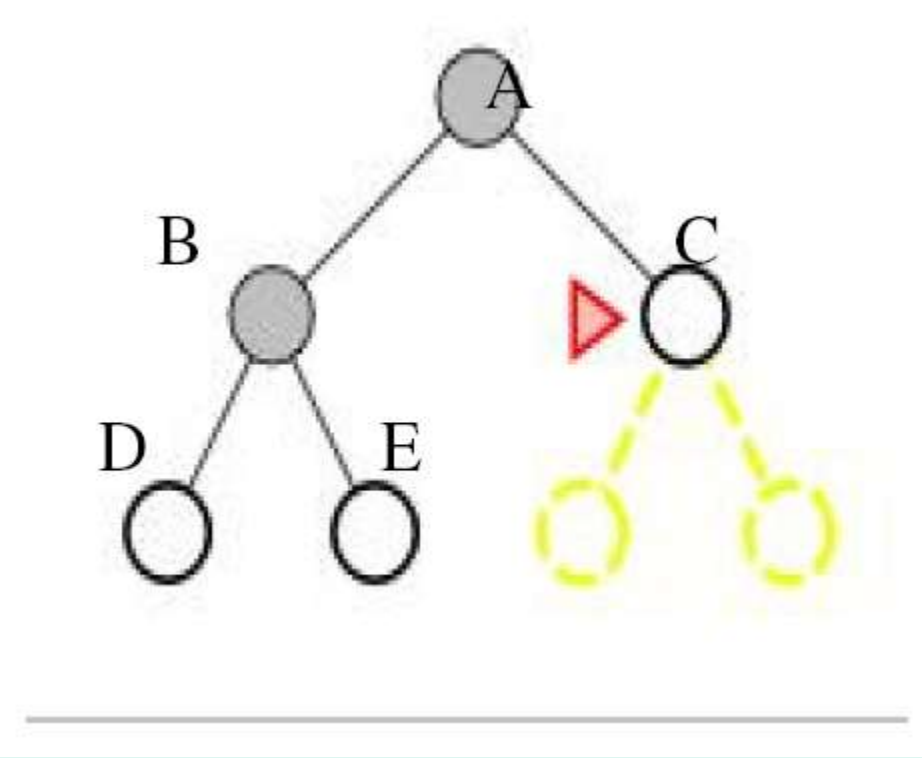
- Expand *shallowest* unexpanded node
- ***Fringe*** is the collection of nodes that have been generated but not yet expanded.
- The set of all leaf nodes available for expansion at any given point is called the ***frontier***.
- Implementation: *fringe/frontier* is a FIFO queue



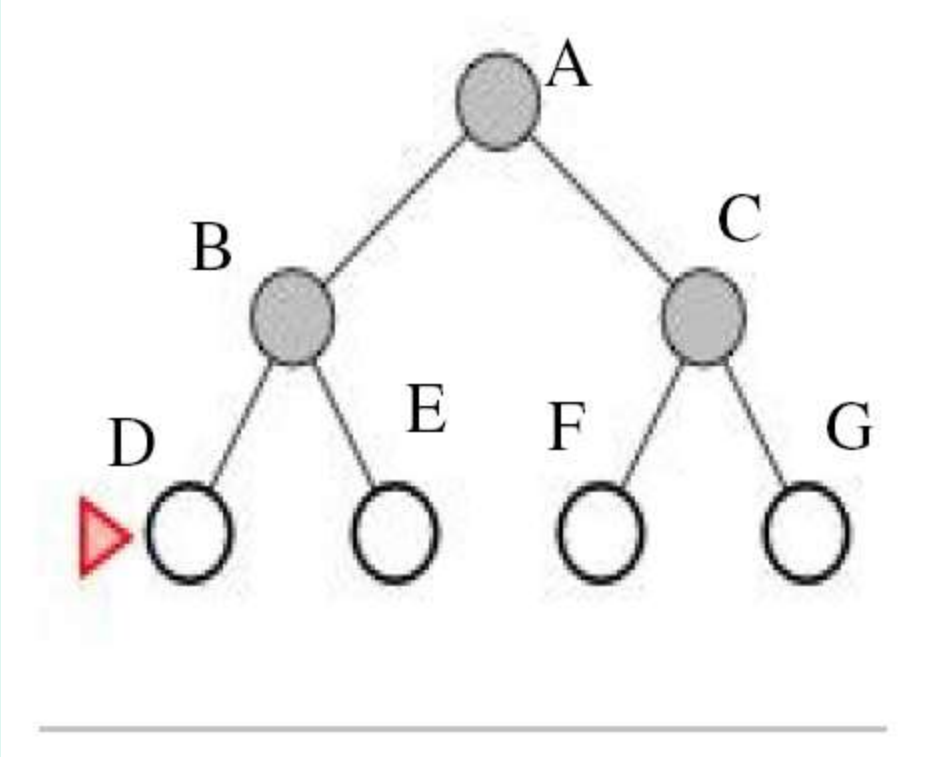
# Breadth-First Search



# Breadth-First Search



# Breadth-First Search



# Breadth-First Search

- Completeness: is a solution always found if one exists?
  - YES
    - If shallowest goal node is at some finite depth *d*
    - If branching factor *b* is finite
- BF search is optimal if the path cost is a non-decreasing function of the depth of the node

# Breadth-First Search

- Time complexity: Assume a state space where every state has *b* successors
  - Assume solution is at depth *d*
  - Worst case: expand all but the last node at depth *d*
  - Total number of nodes generated:
  - $b + b^2 + b^3 + ... + b^d + (b^{d+1} - b) = O(b^{d+1})$
- Space complexity: every node generated must remain in memory, so same as time complexity

# Breadth-First Search

- Memory requirements are a bigger problem than execution time.

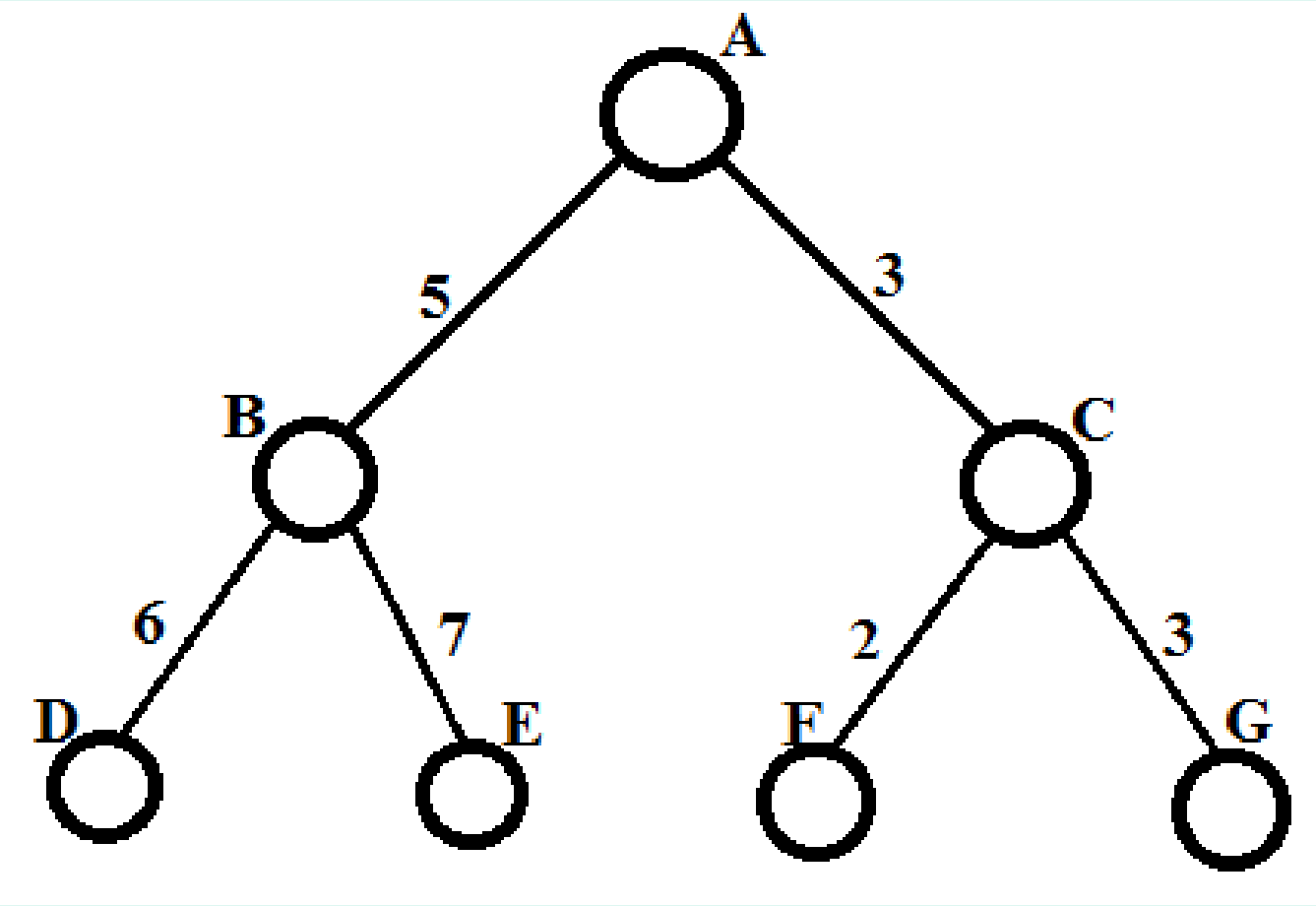
| DEPTH | NODES | TIME | MEMORY |
|---|---|---|---|
| 2 | 1100 | 0.11 seconds | 1 megabyte |
| 4 | 111100 | 11 seconds | 106 megabytes |
| 6 | $10^7$ | 19 minutes | 10 gigabytes |
| 8 | $10^9$ | 31 hours | 1 terabyte |
| 10 | $10^{11}$ | 129 days | 101 terabytes |
| 12 | $10^{13}$ | 35 years | 10 petabytes |
| 14 | $10^{15}$ | 3523 years | 1 exabyte |

# Uniform-Cost Search

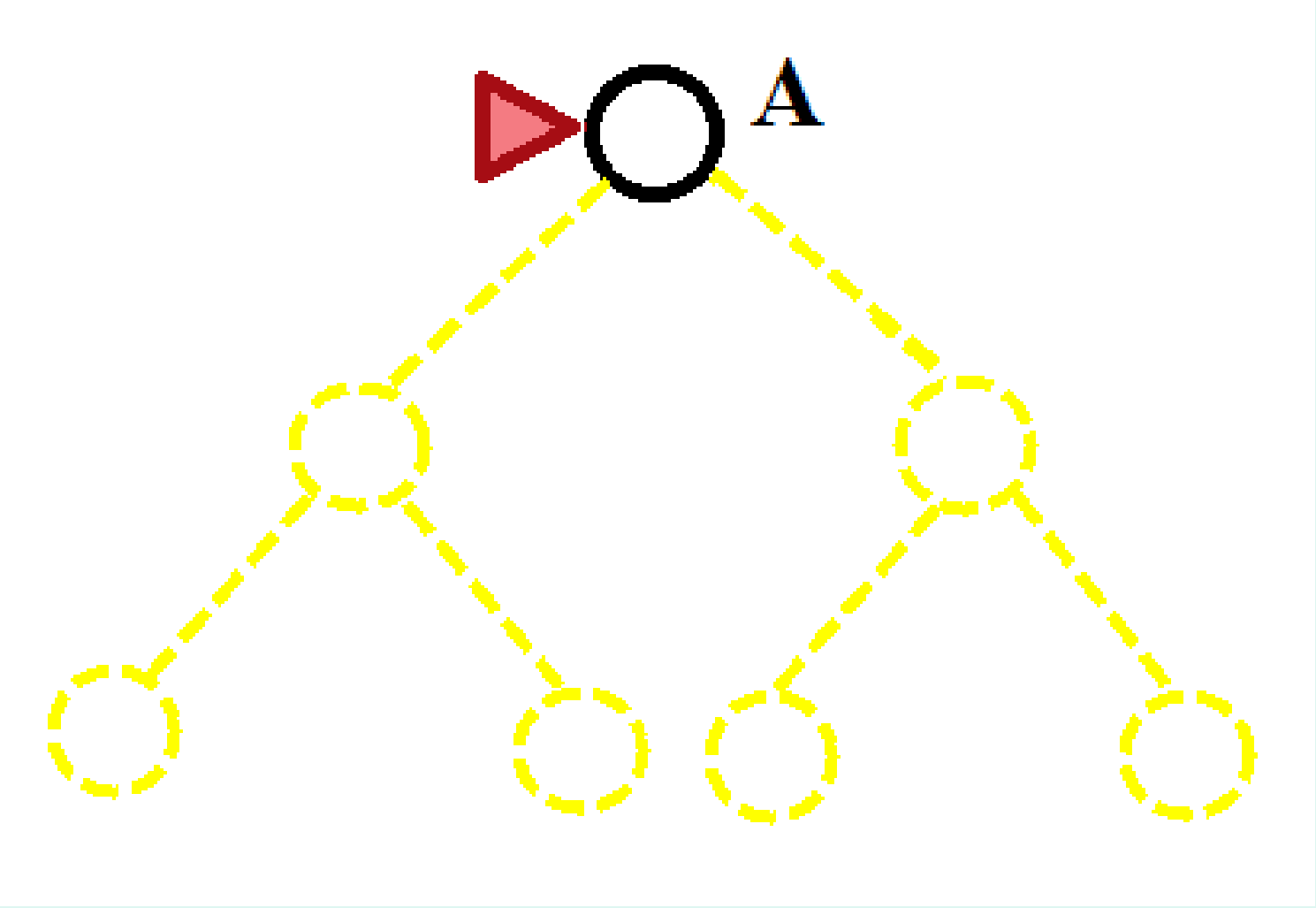
- Extension of BF-search:
  - Expand node with *lowest path cost*
- Implementation: *fringe* = queue ordered by path cost
- UC-search is the same as BF-search when all step costs are equal

# Uniform-Cost Search

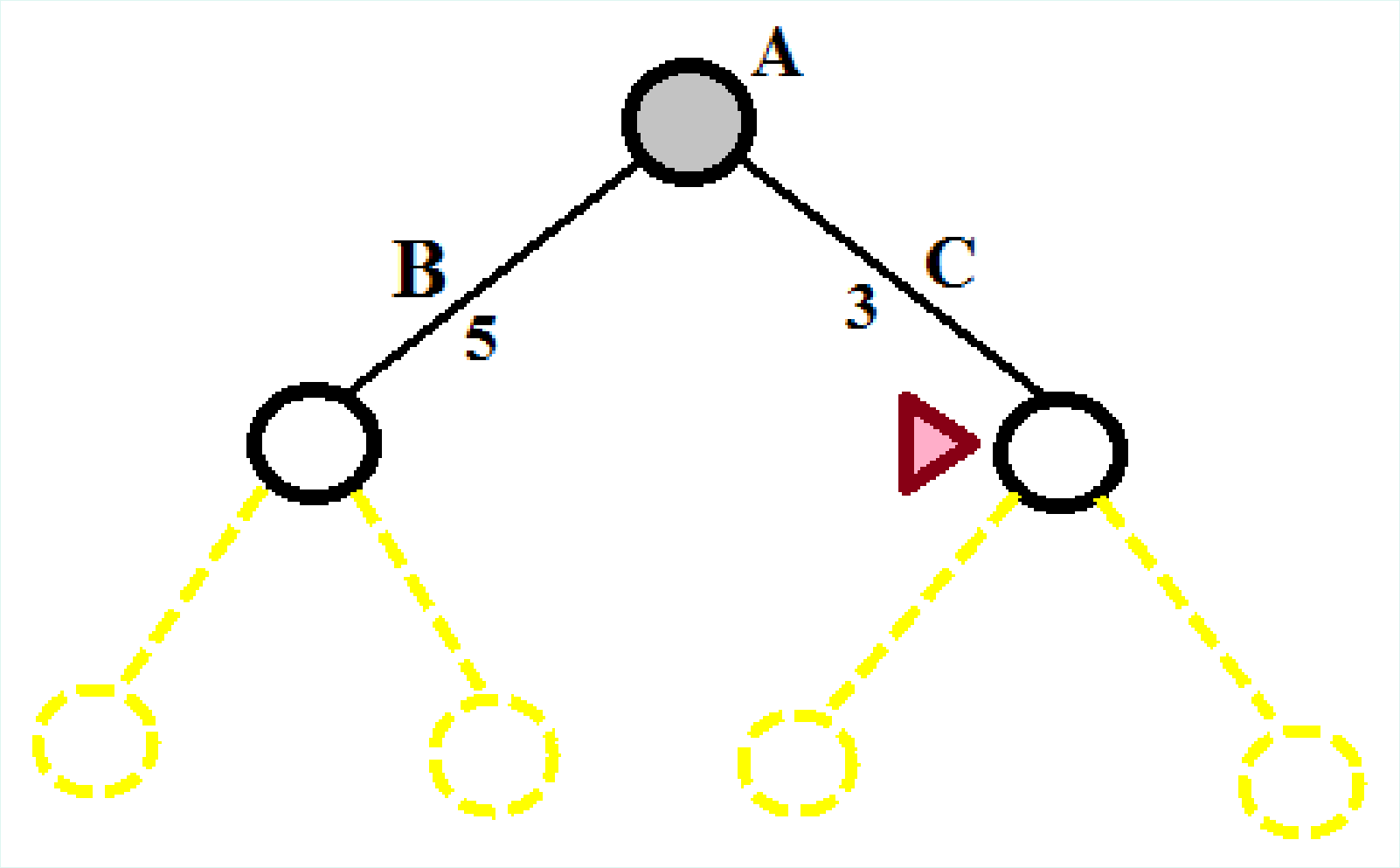
- Let us consider the following search tree, where ***A*** and ***G*** are the initial and goal node, respectively.



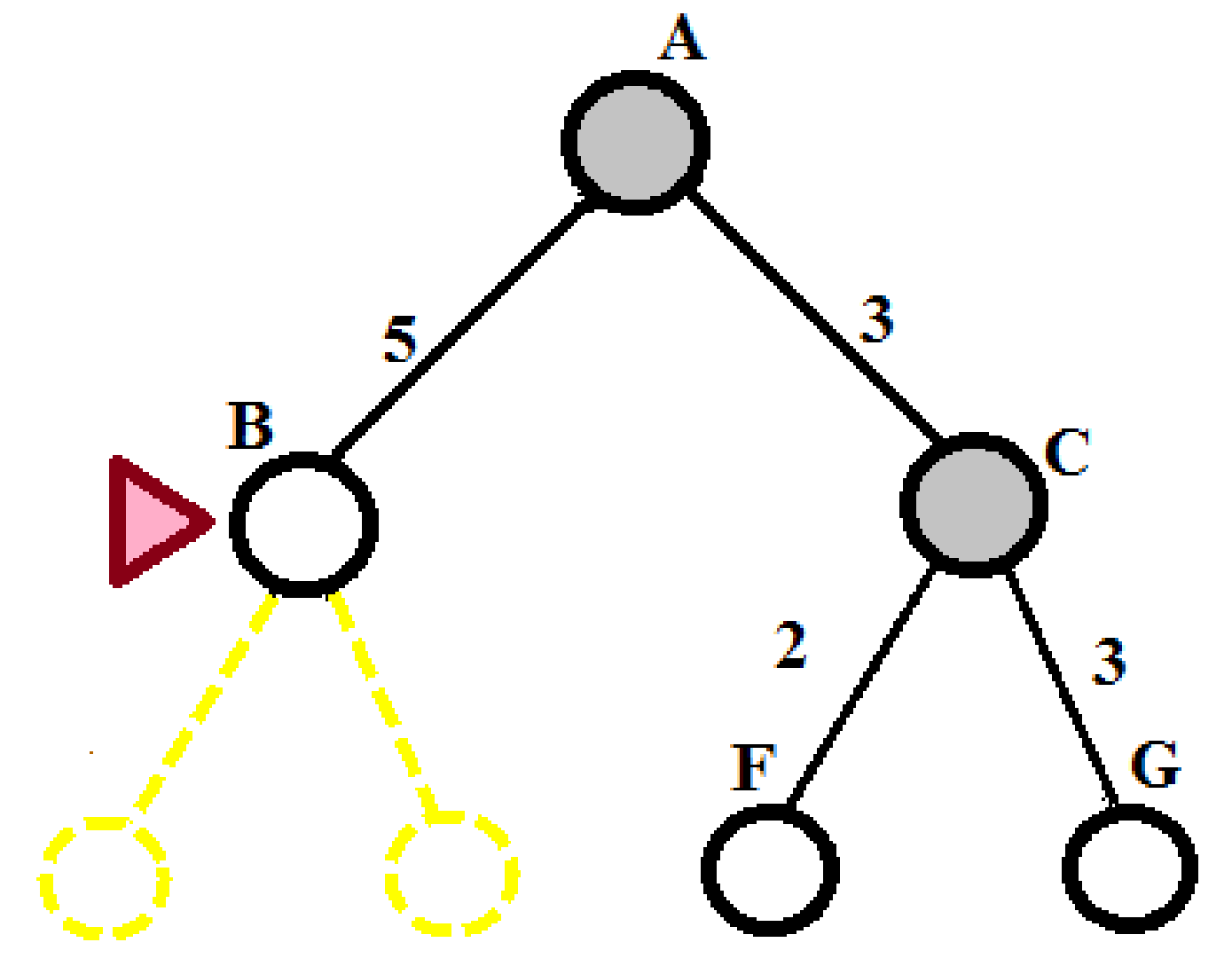
# Uniform-Cost Search



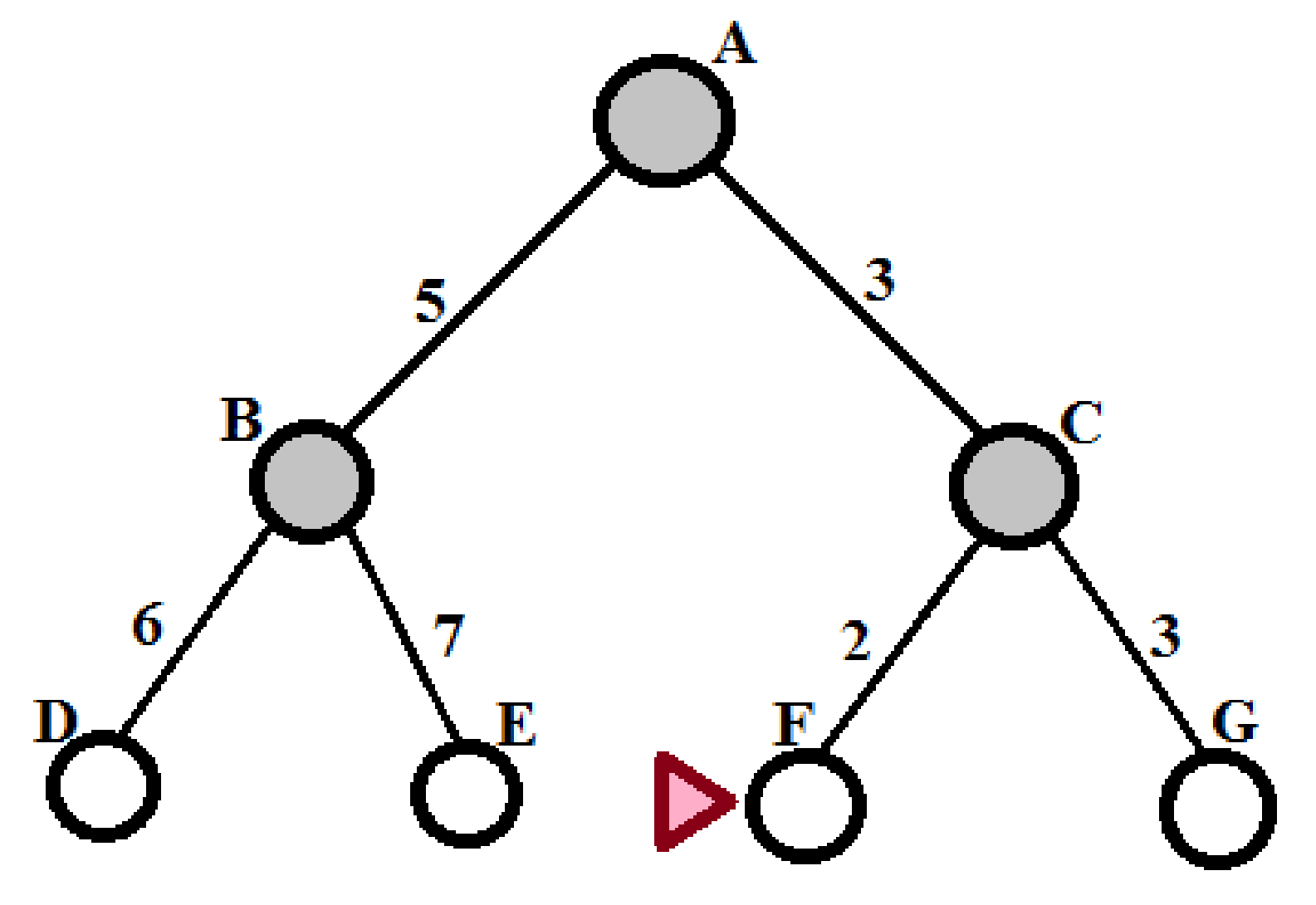
# Uniform-Cost Search



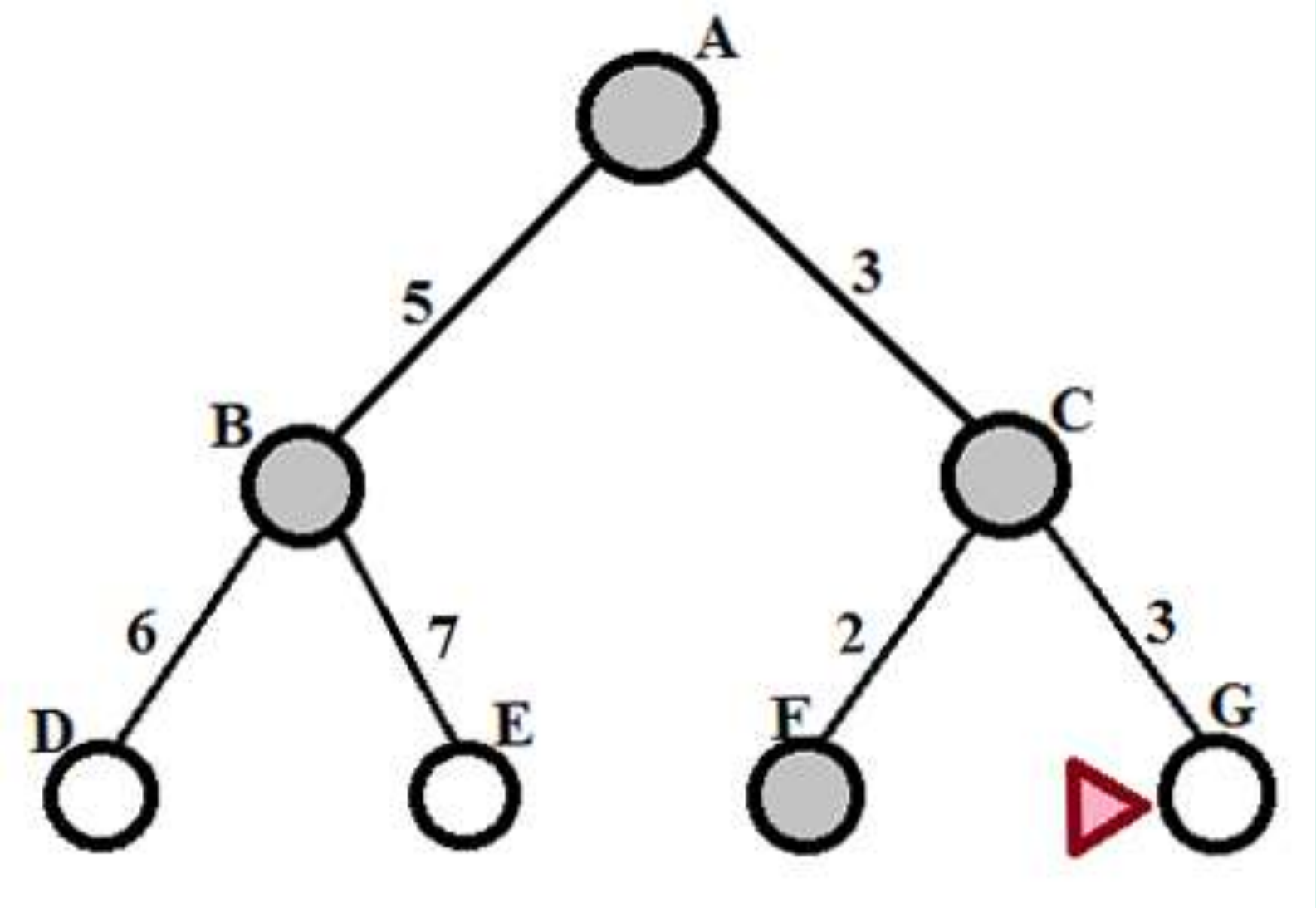
# Uniform-Cost Search



# Uniform-Cost Search

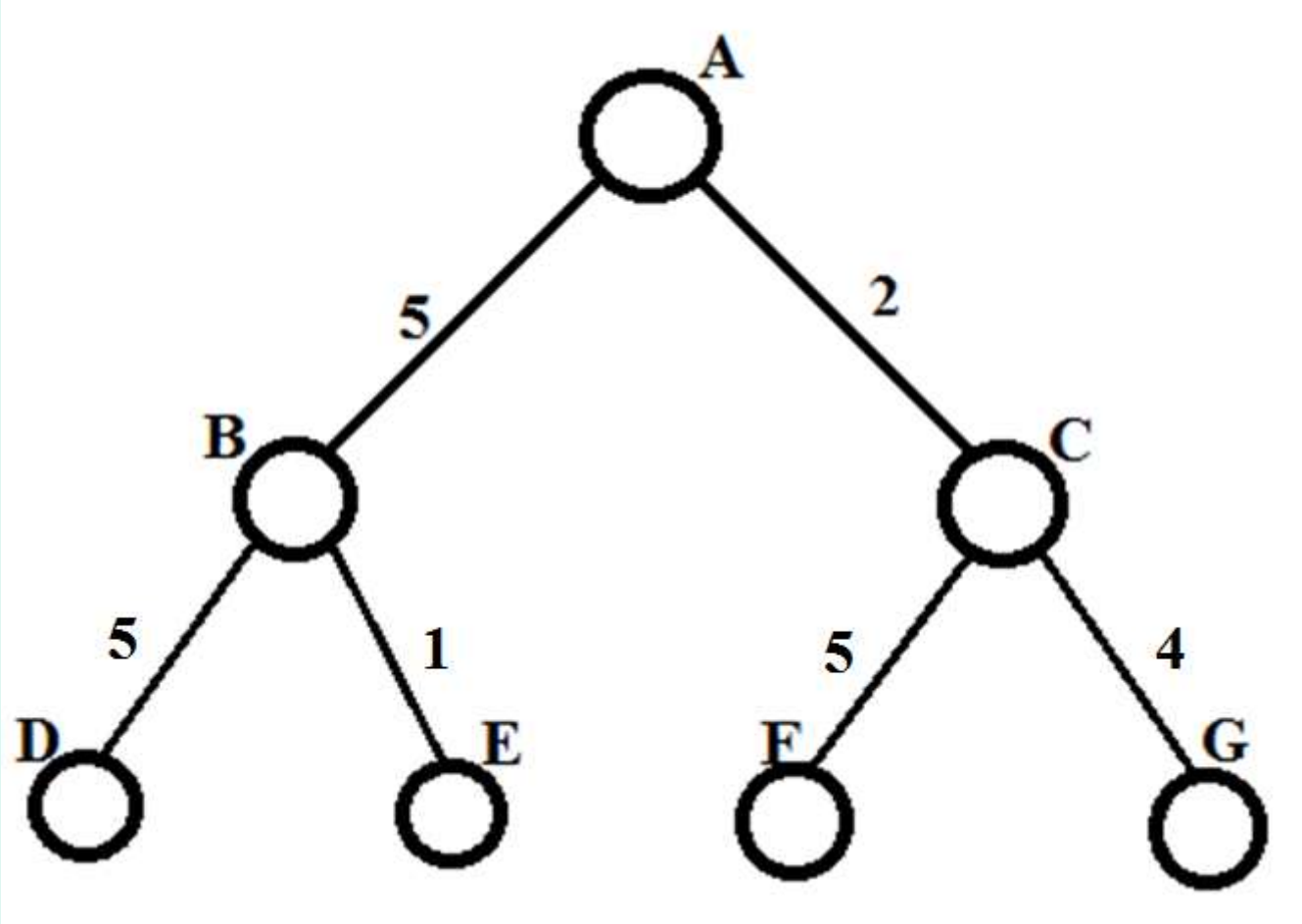


# Uniform-Cost Search

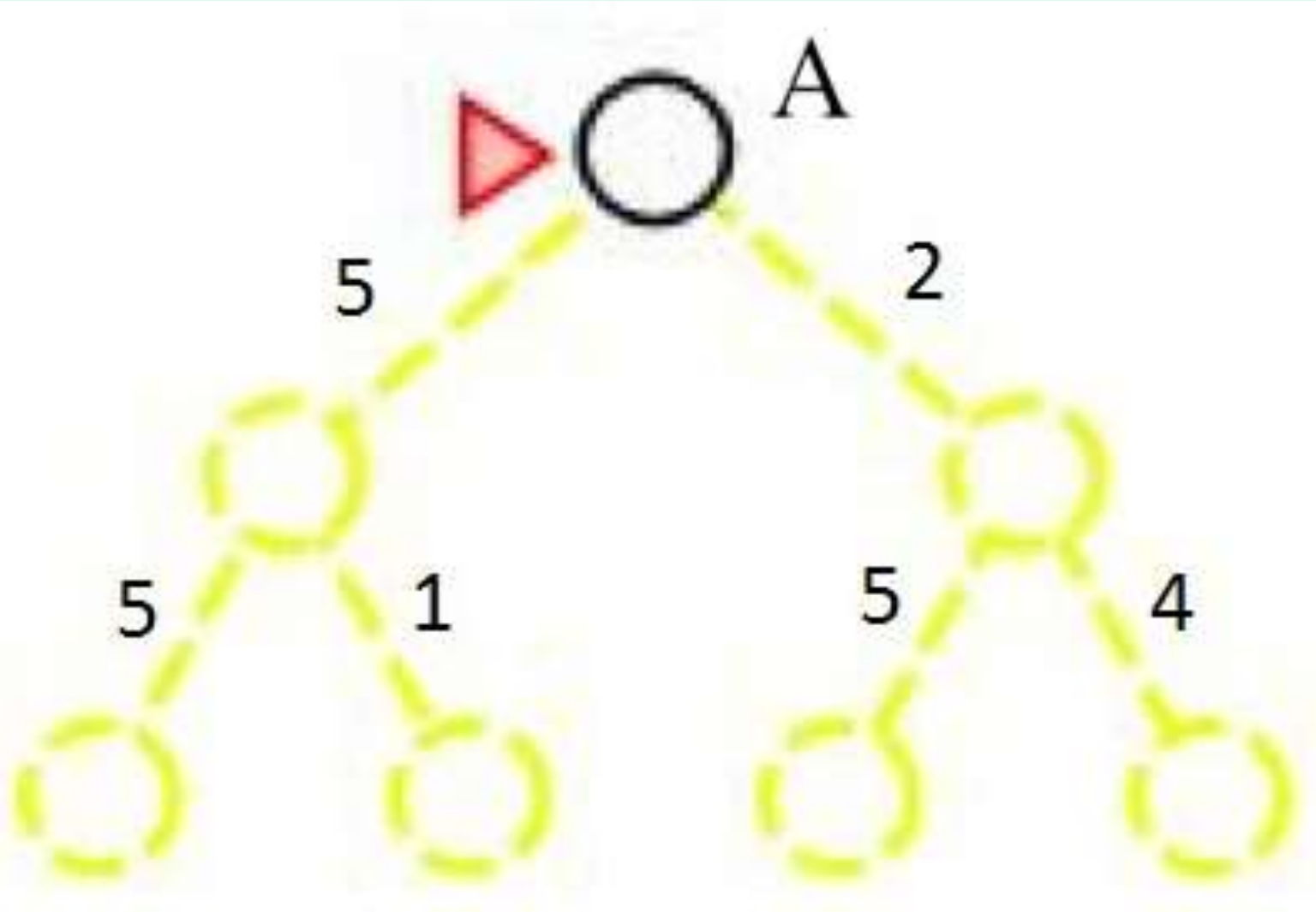


# Uniform-Cost Search

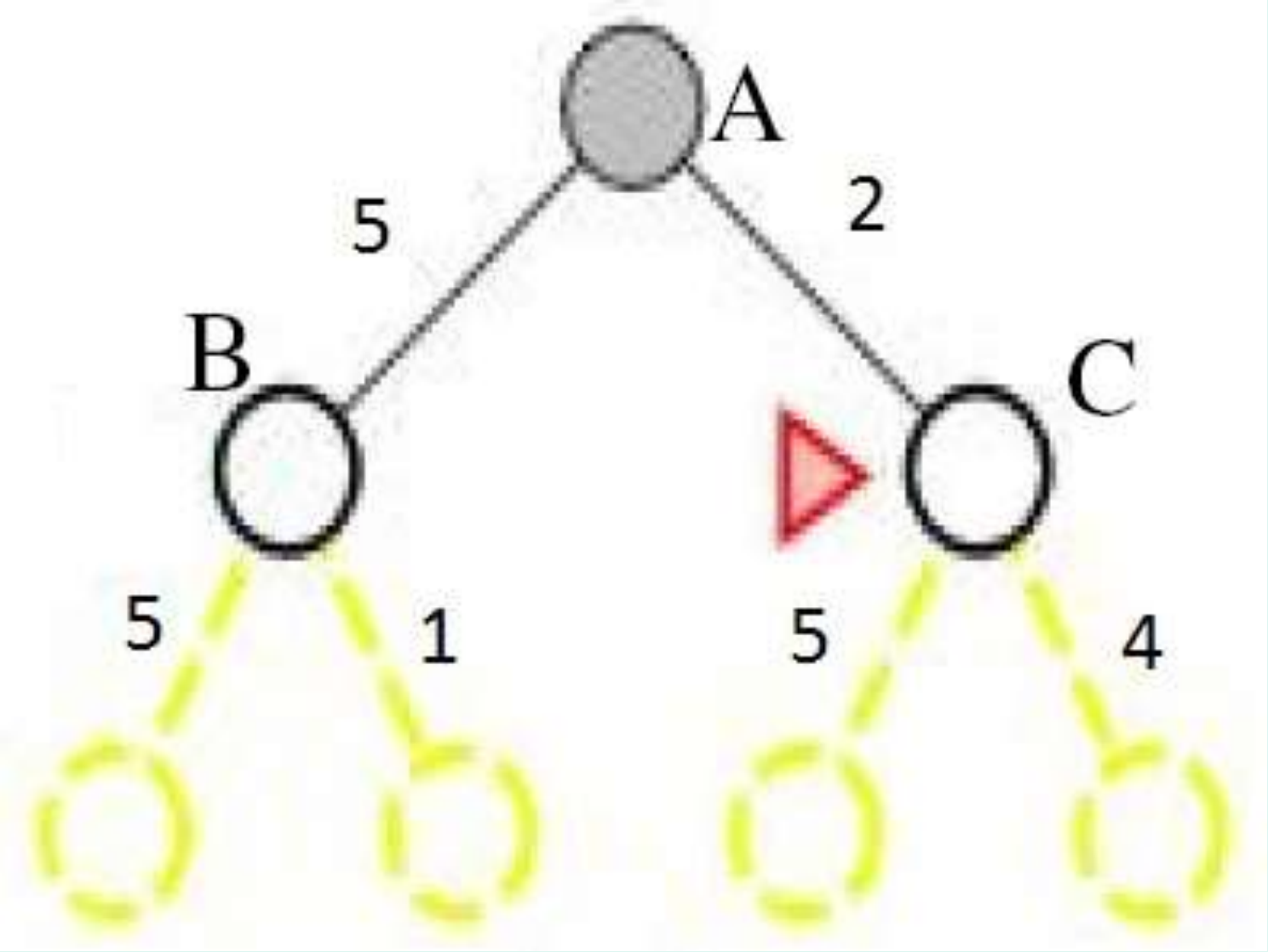
- Let us consider the another search tree, where ***A*** and ***G*** are the initial and goal node, respectively.



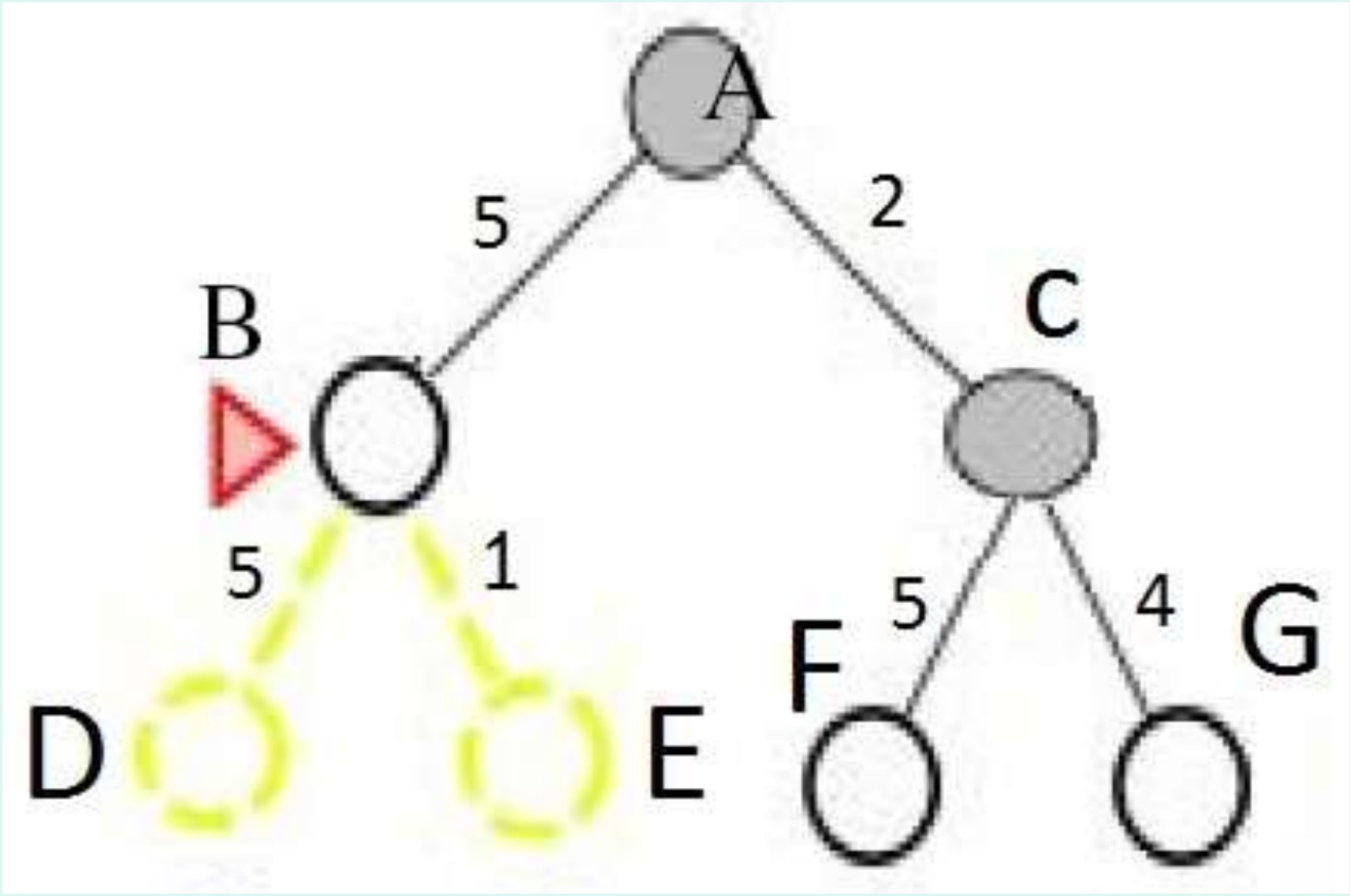
# Uniform-Cost Search



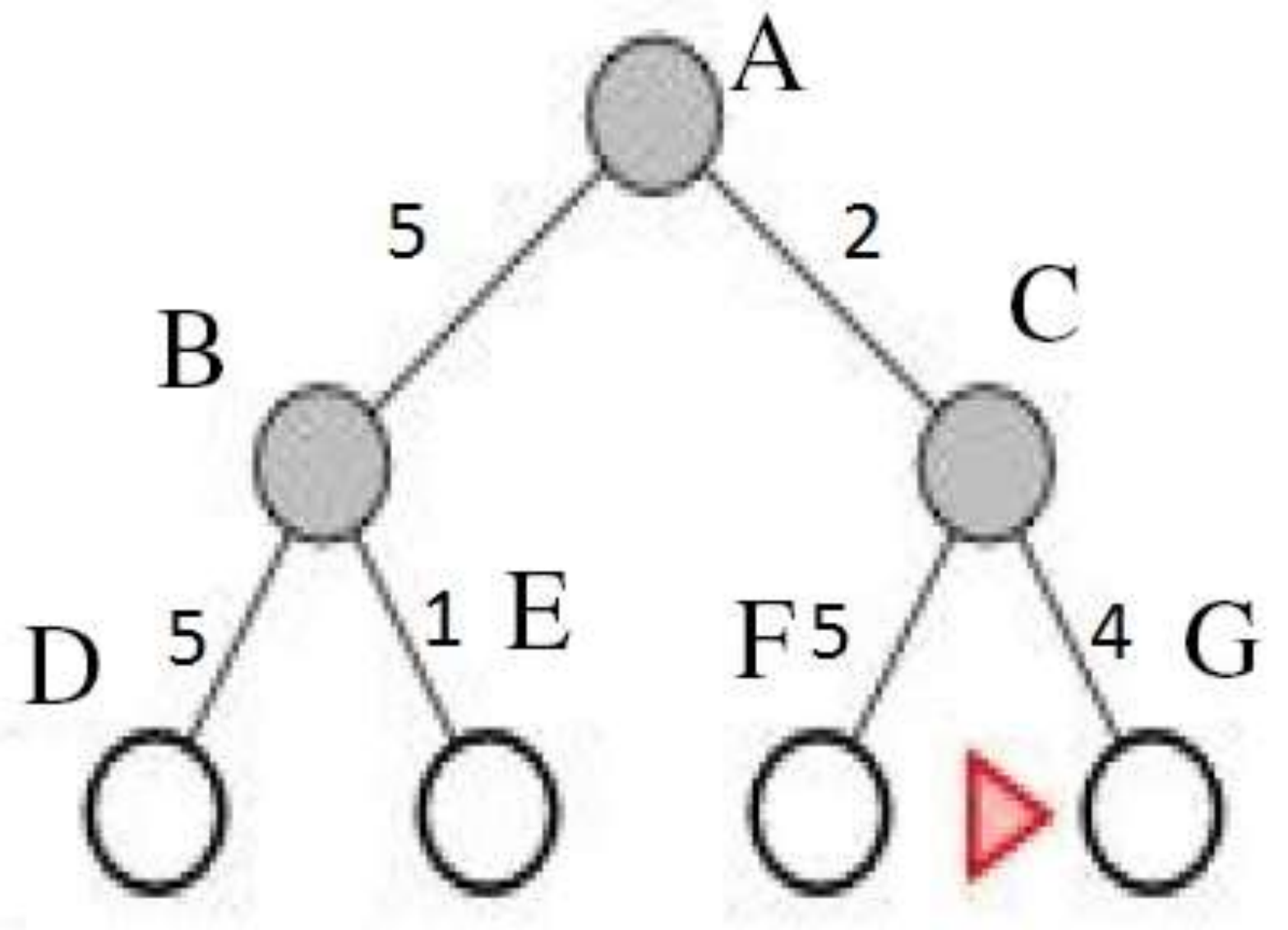
# Uniform-Cost Search



# Uniform-Cost Search



# Uniform-Cost Search

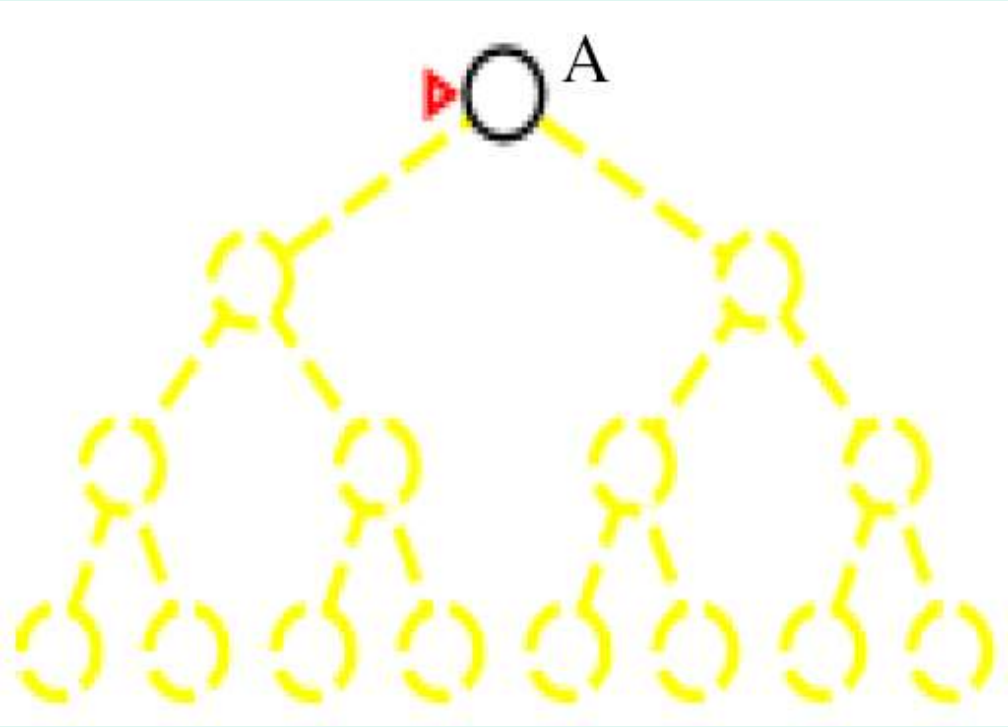


# Uniform-Cost Search

- Completeness:
  - YES, if step-cost > some small positive constant ε
- Optimality:
  - nodes expanded in order of increasing path cost
  - YES, if complete
- Time and space complexity:
  - Uniform-cost search is guided by path costs rather than depths, so its complexity cannot easily be characterized in terms of *b and d.*
  - Instead, assume $C^*$ is the cost of the optimal solution
  - Assume that every action costs at least ε
  - Worst-case: $O(b^{C^*/\varepsilon})$

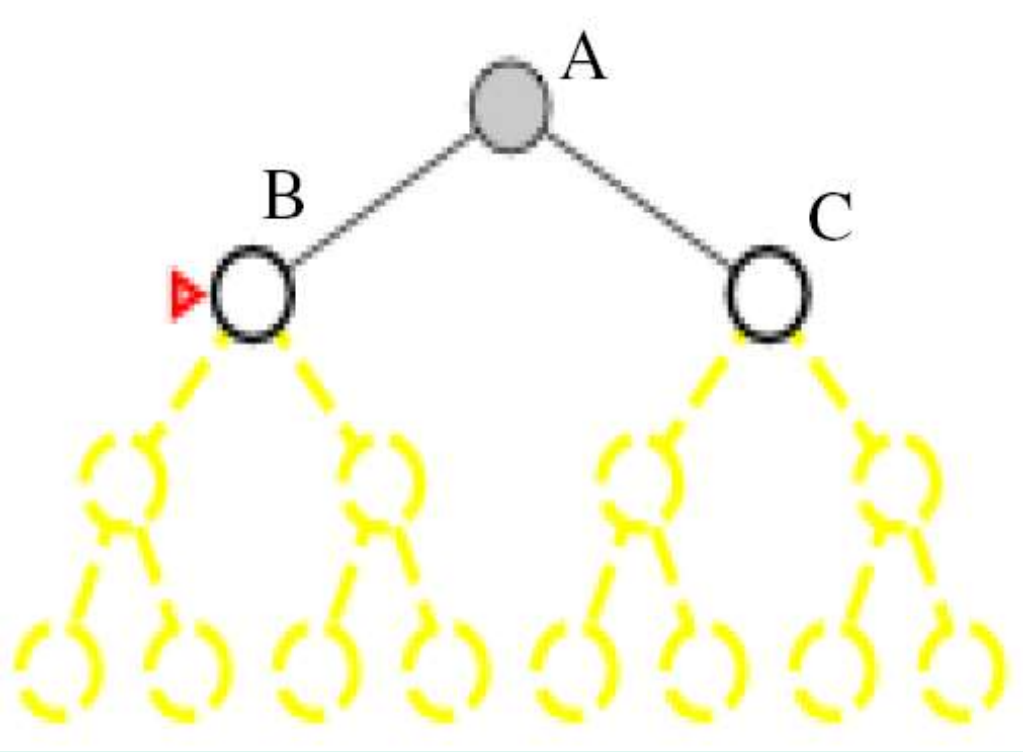
# Depth-First Search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (= stack)



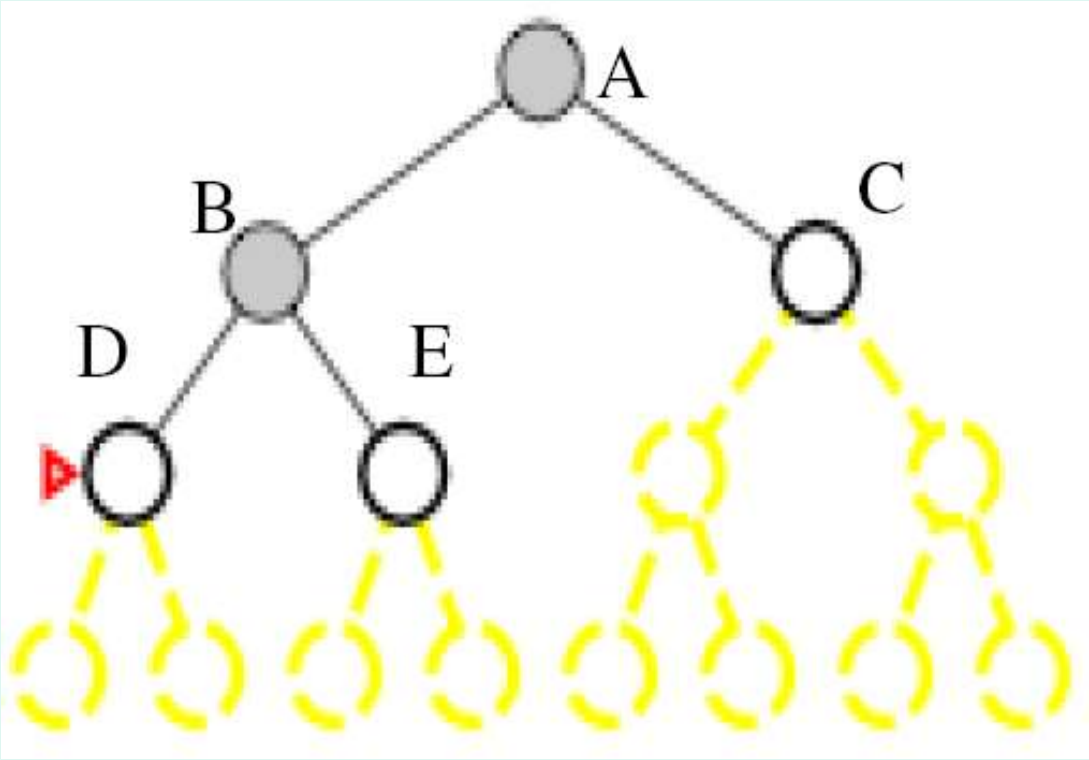
# Depth-First Search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (= stack)



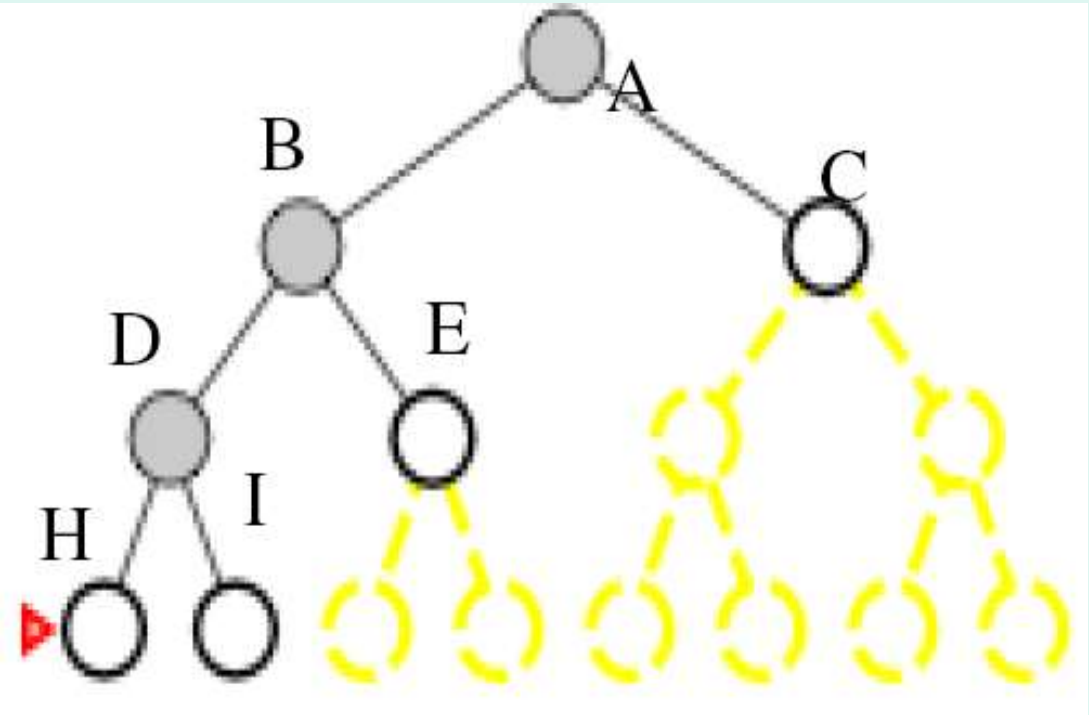
# Depth-First Search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (= stack)



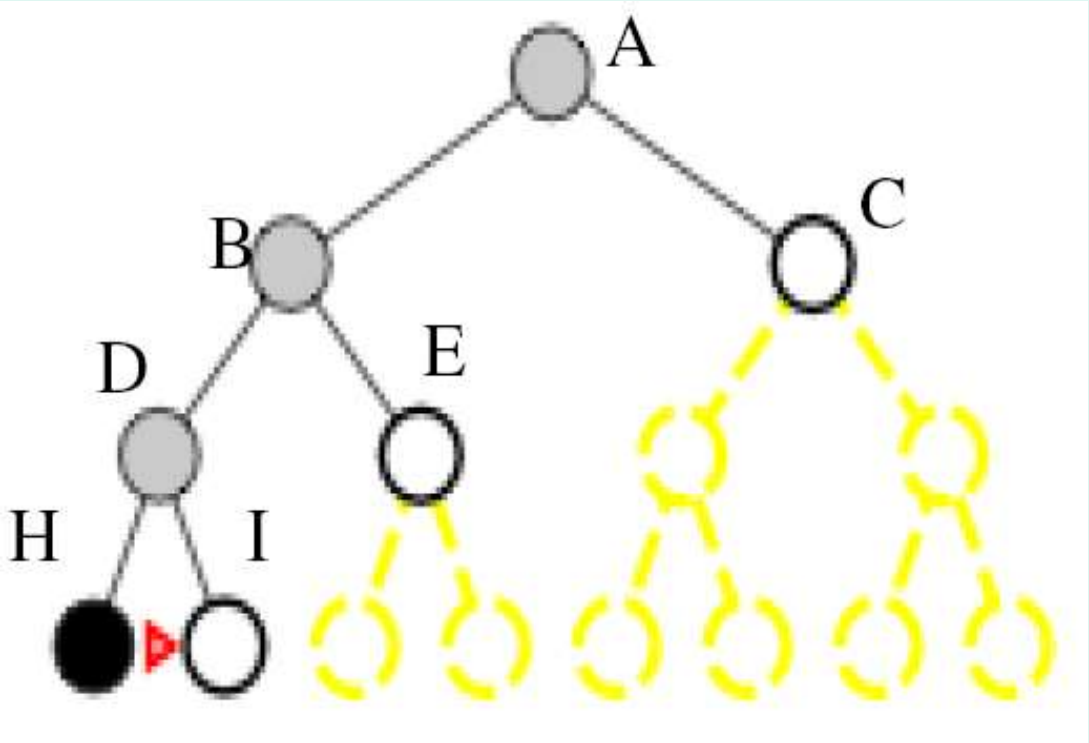
# Depth-First Search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (= stack)



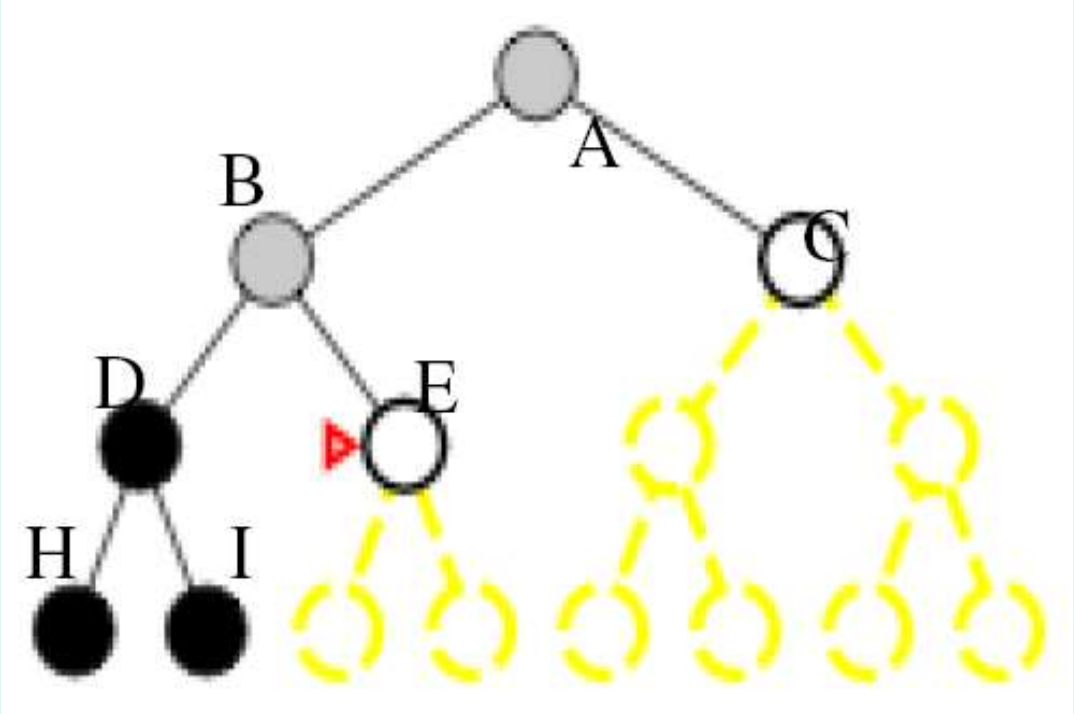
# Depth-First Search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (= stack)



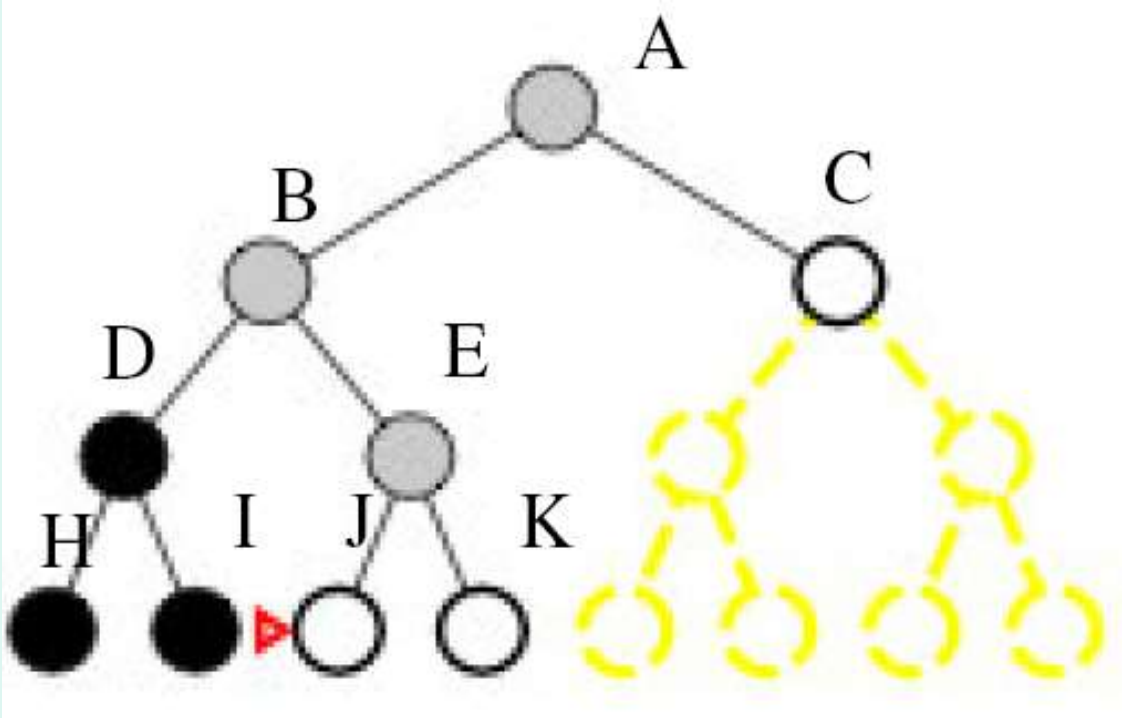
# Depth-First Search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (= stack)



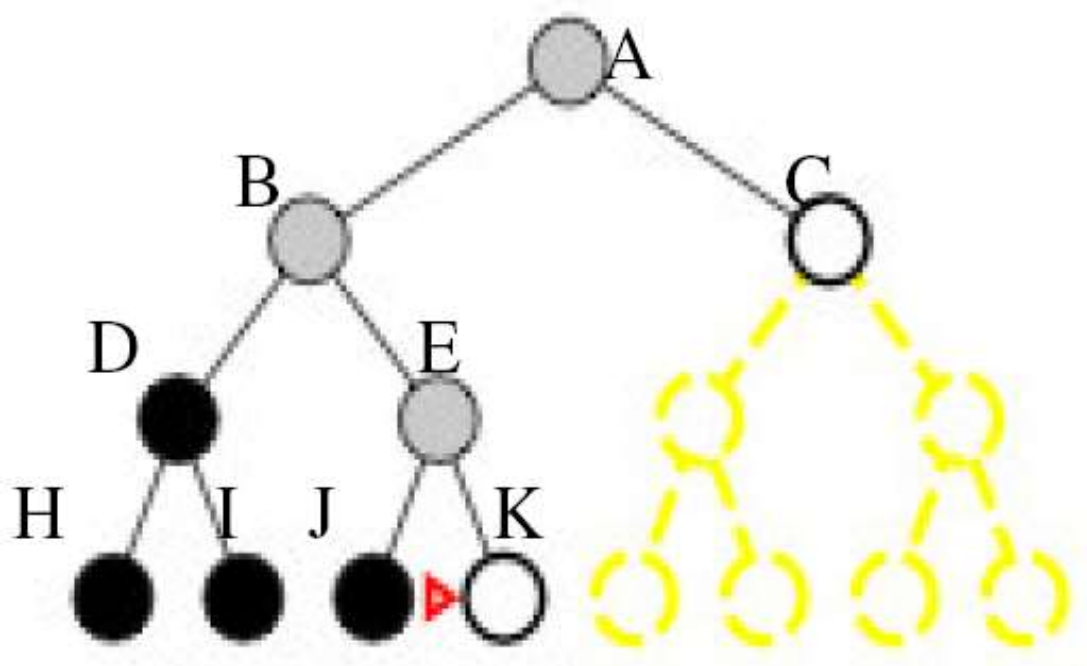
# Depth-First Search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (= stack)



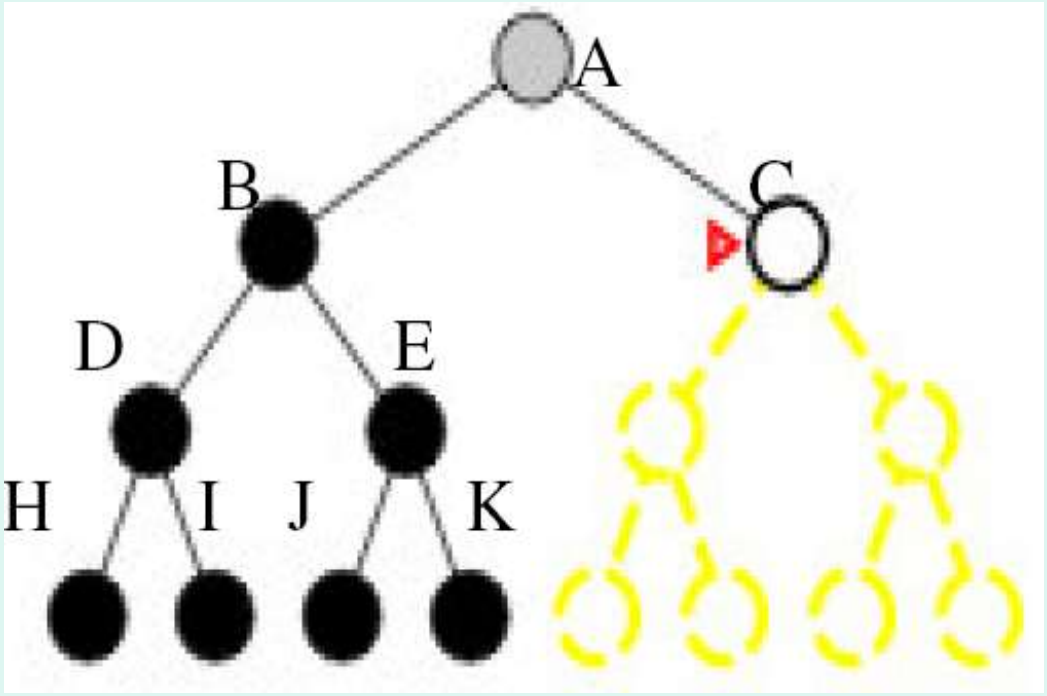
# Depth-First Search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (= stack)



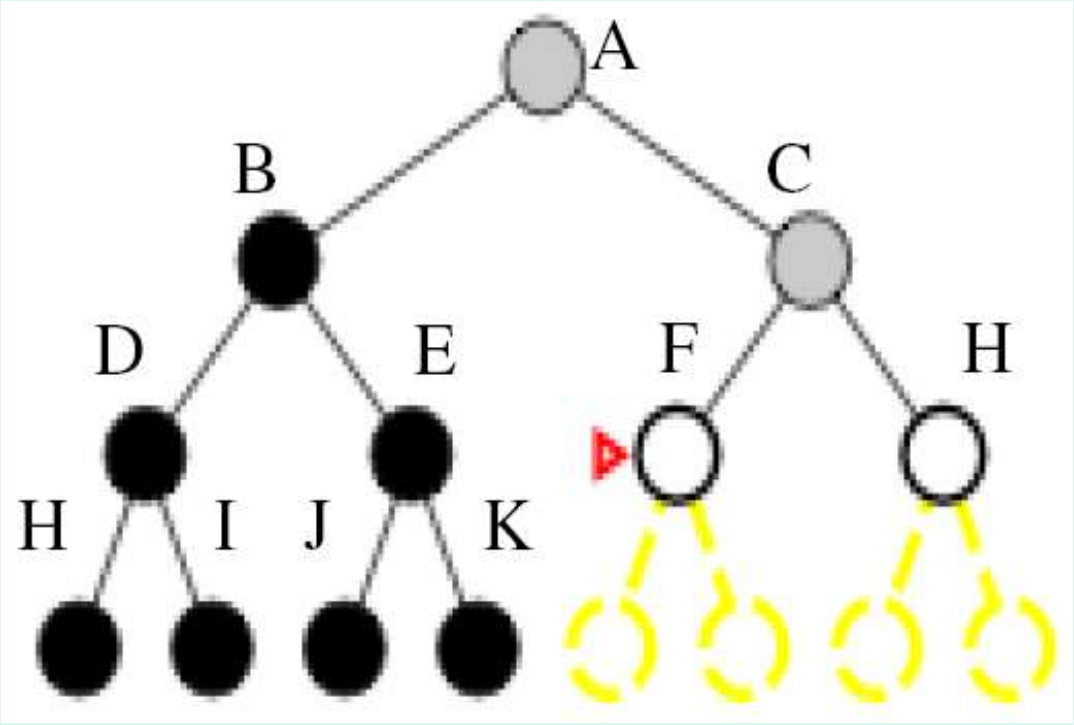
# Depth-First Search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (= stack)



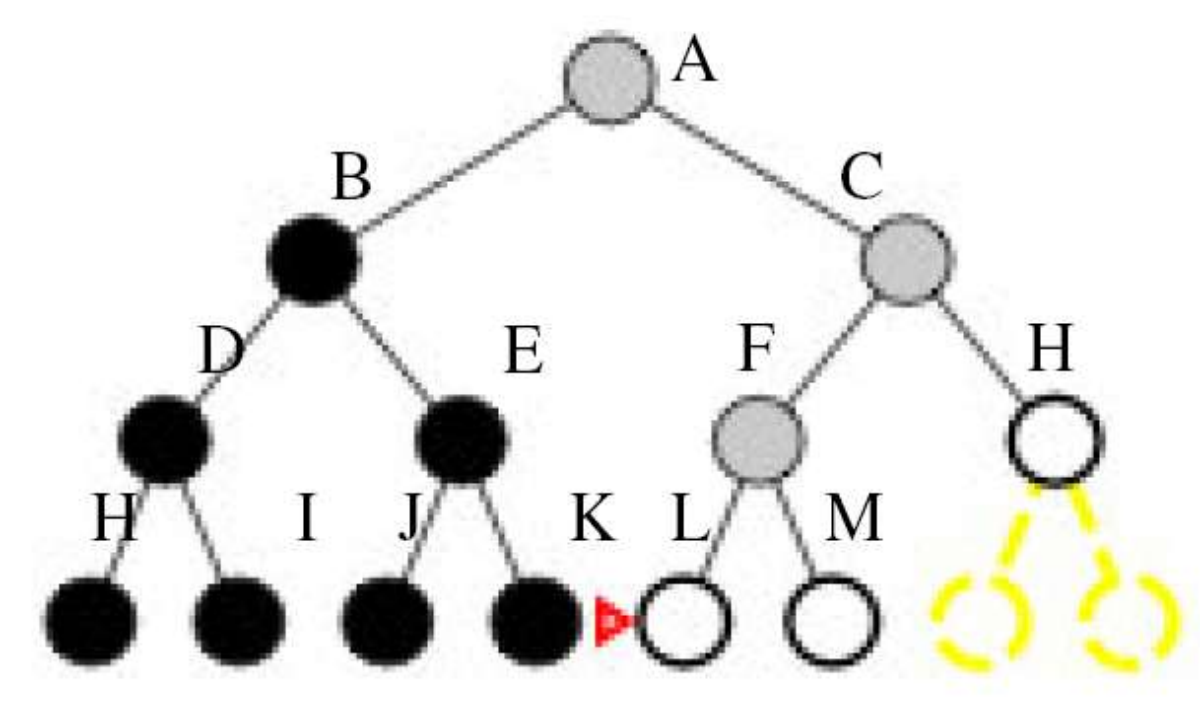
# Depth-First Search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (= stack)



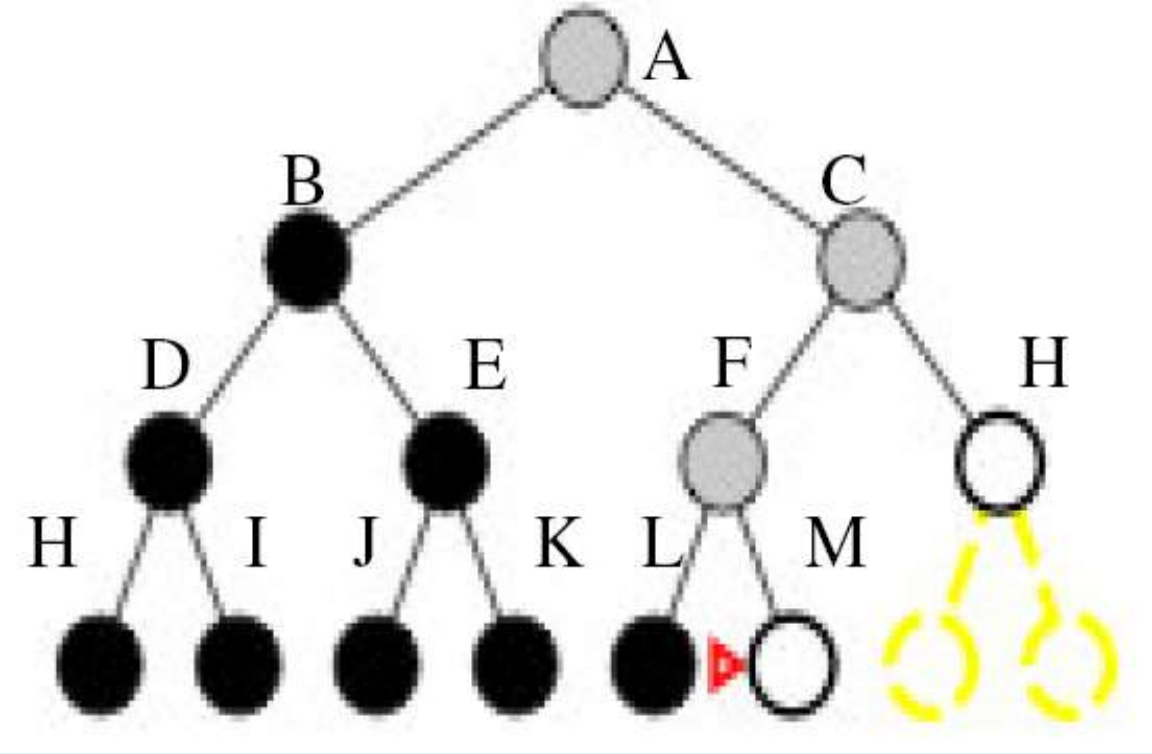
# Depth-First Search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (= stack)



# Depth-First Search

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (= stack)



# DF-Search: Evaluation

- Completeness:
  - Is a solution always found if one exists?
  - No (unless search space is finite and no loops are possible)
- Optimality:
  - Is the least-cost solution always found?
  - No

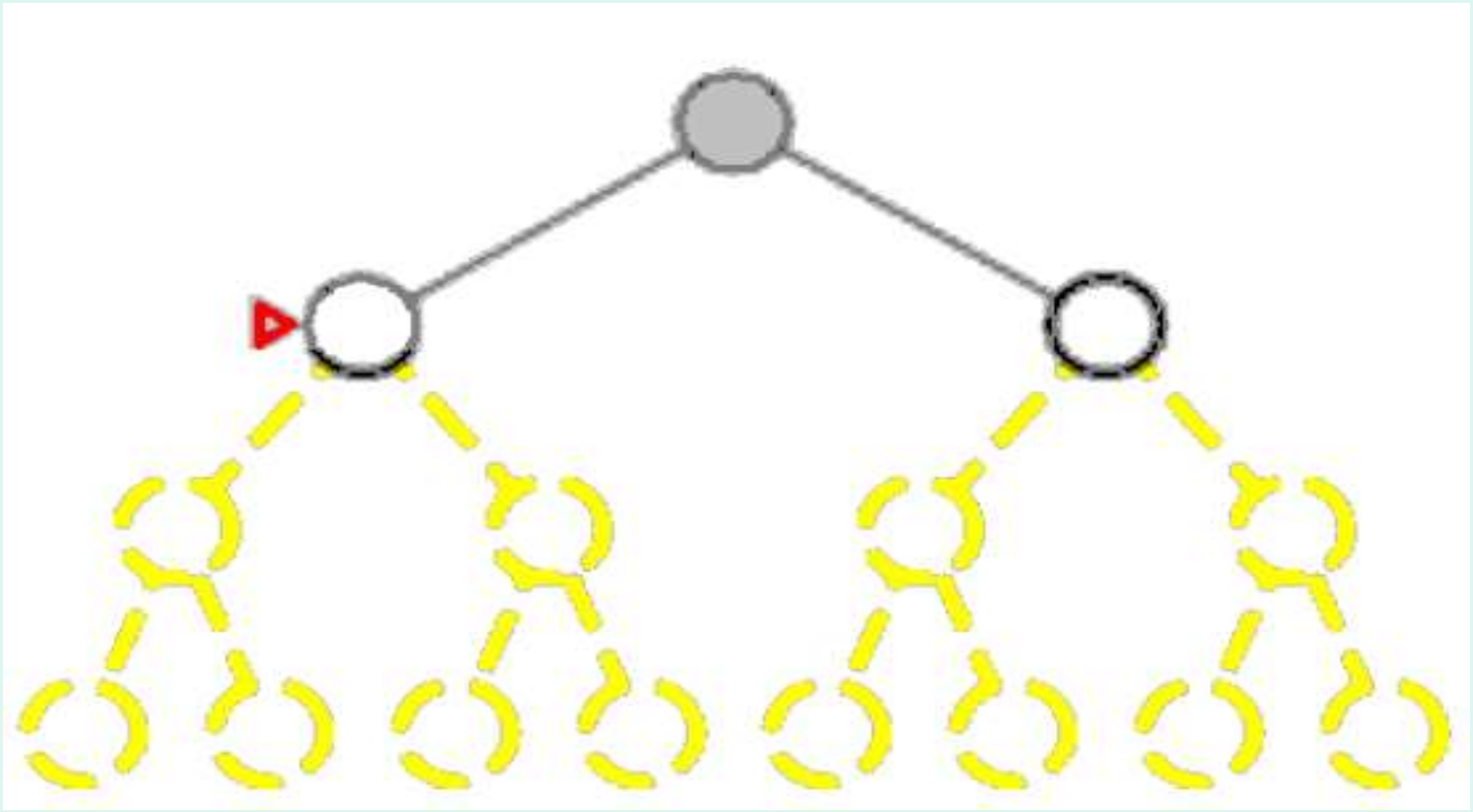
# DF-Search: Evaluation

- Time complexity: $O(b^m)$
- In general, time is terrible if $m$ (maximal depth) is much larger than $d$ (depth of shallowest solution)
  - But if there exist many solutions then faster than BF-search
- Space complexity: $O(bm)$
  - Backtracking search uses even less memory (one successor instead of all $b$)

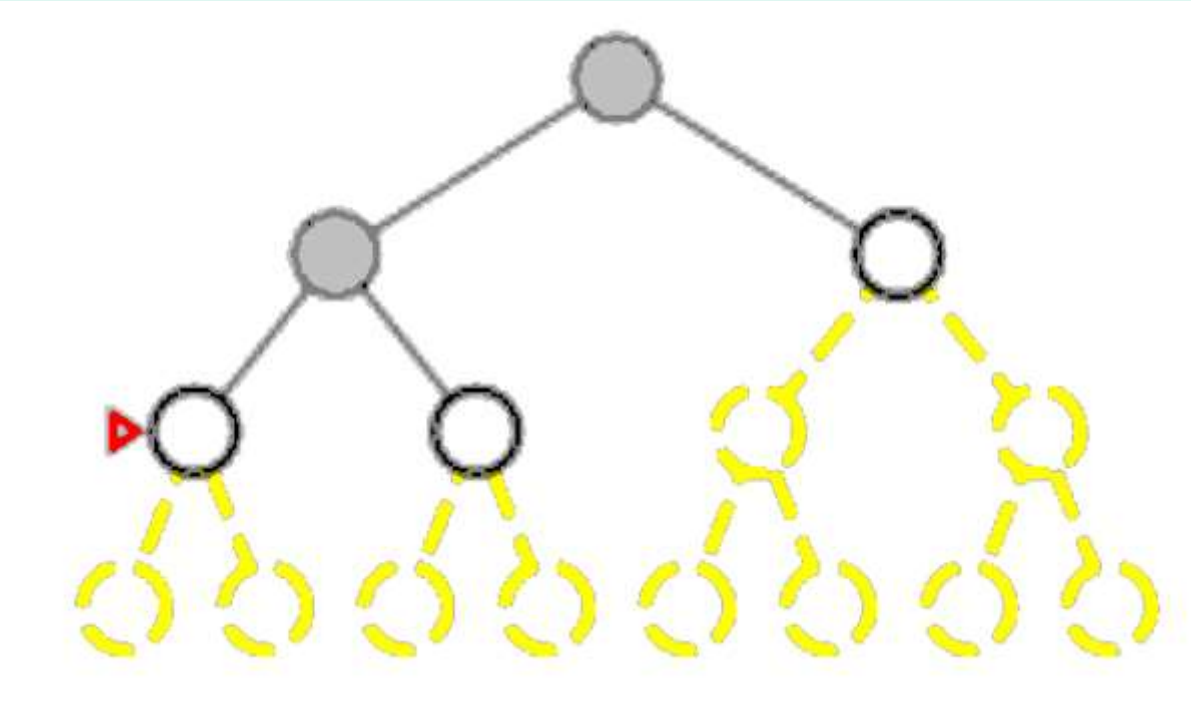
# Depth-Limited Search

- DF-search with depth limit $l$
  - i.e. nodes at depth $l$ have no successors
  - Problem knowledge can be used
- Solves the infinite-path problem
- If $l < d$ then incompleteness results
- Time complexity: $O(b^l)$
- Space complexity: $O(bl)$
- Depth-first search can be viewed as a special case of depth-limited search with $l = \infty$.

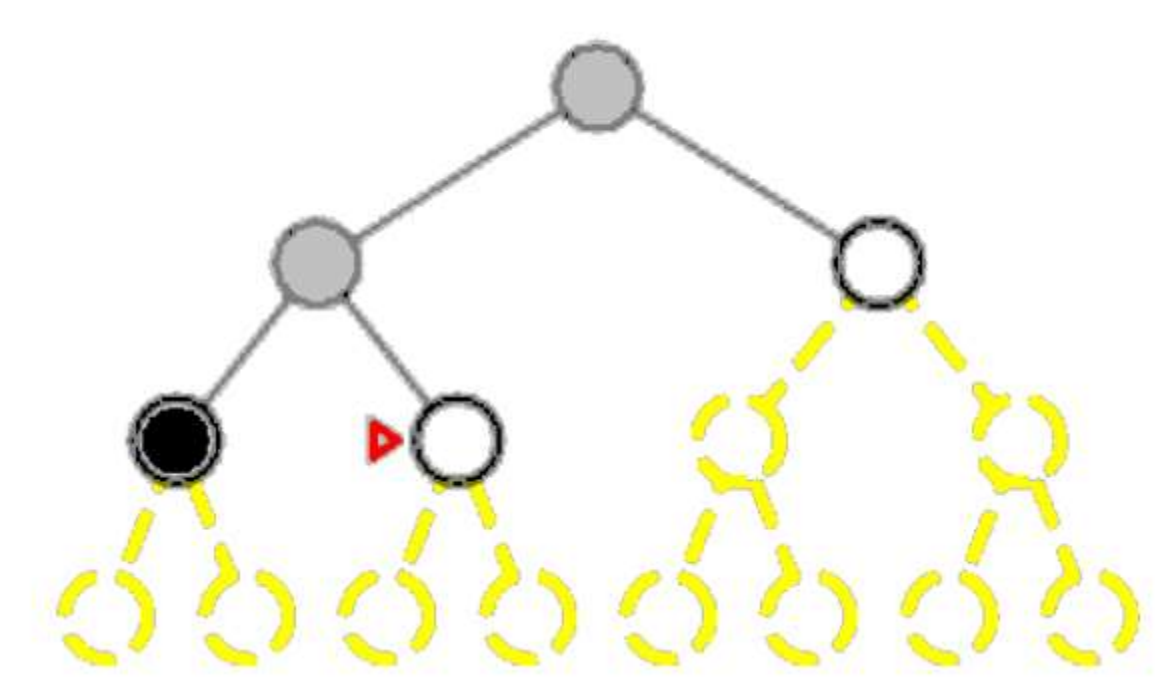
# Depth-Limited Search with *l* = 2



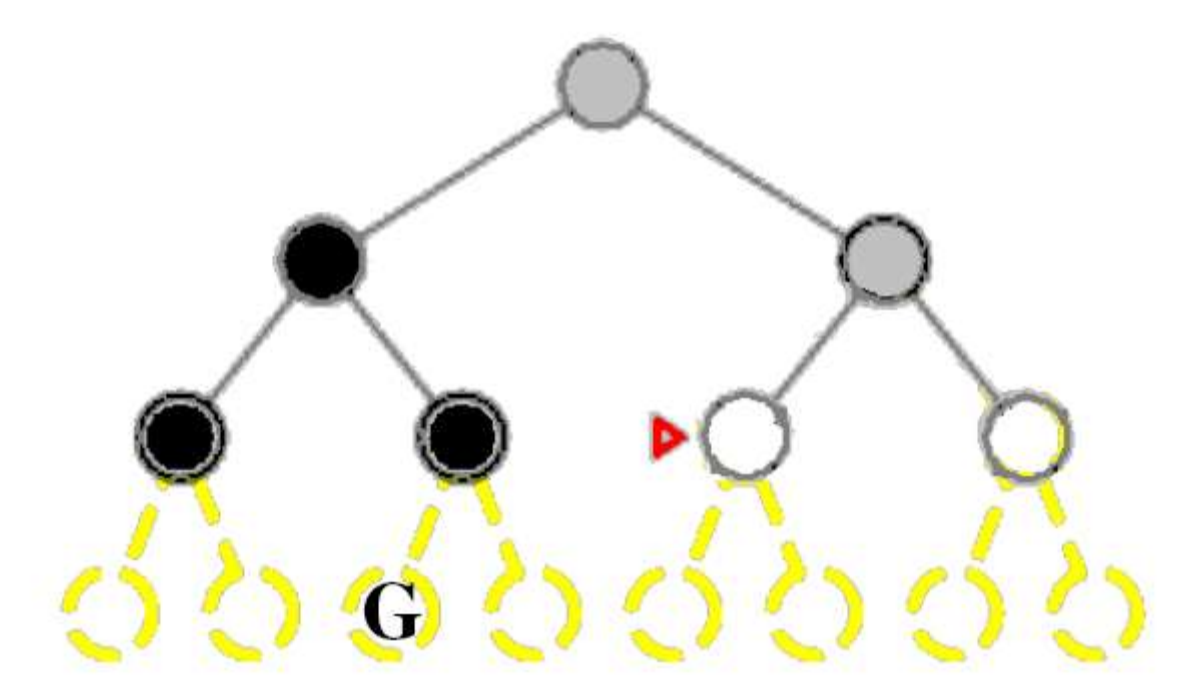
# Depth-Limited Search with *l* = 2



# Depth-Limited Search with *l* = 2



# Depth-Limited Search with *l* = 2



# Depth-Limited Search

- DF-search with depth limit $l$
  - i.e. nodes at depth $l$ have no successors
  - Problem knowledge can be used
- Sometimes, depth limits can be based on knowledge of the problem.
- For example, on the map of Romania there are 20 cities.
- Therefore, we know that if there is a solution, it must be of length 19 at the longest, so $l$ = 19 is a possible choice.

# Depth-Limited Search

- But in fact if we studied the map carefully, we would discover that any city can be reached from any other city in at most 9 steps.
- This number, known as the **diameter** of the state space, gives us a better depth limit, which leads to a more efficient depth-limited search.
- ***Diameter*** of state space is the maximum distance from one state to another in the state space.
- For most problems, however, we will not know a good depth limit until we have solved the problem.

# Iterative Deepening Search

- A general strategy to find best depth limit *l*
  - Goal is found at depth *d*, the depth of the shallowest goal-node
  - Often used in combination with DF-search
- Combines benefits of DF- and BF-search
- Like depth-first search, its memory requirements are very modest: $O(bd)$ to be precise.
- Like breadth-first search, it is complete when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node.

# Iterative Deepening Search

- Iterative deepening search may seem wasteful, because states are generated multiple times.
- It turns out this is not very costly.
- The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times.

# Iterative Deepening Search

- In an iterative deepening search, the nodes on the bottom level (depth $d$) are generated once, those on the next to bottom level are generated twice, and so on, up to the children of the root, which are generated $d$ times.
- So the total number of nodes generated is

$$N(IDS) = d.b + (d - 1).b^2 + ...+ 1.b^d = O(b^d)$$

- We can compare this to the nodes generated by a breadth-first search:

$$N(BFS) = b + b^2 + ...+ b^d + (b^{d+1} - b) = O(b^{d+1})$$

- Notice that breadth-first search generates some nodes at depth $d$+1, whereas iterative deepening does not.
- The result is that iterative deepening is actually *faster* than breadth-first search, despite the repeated generation of states.

# Iterative Deepening Search

- For example, if $b = 10$ and $d = 5$,

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

# ID-Search: Example

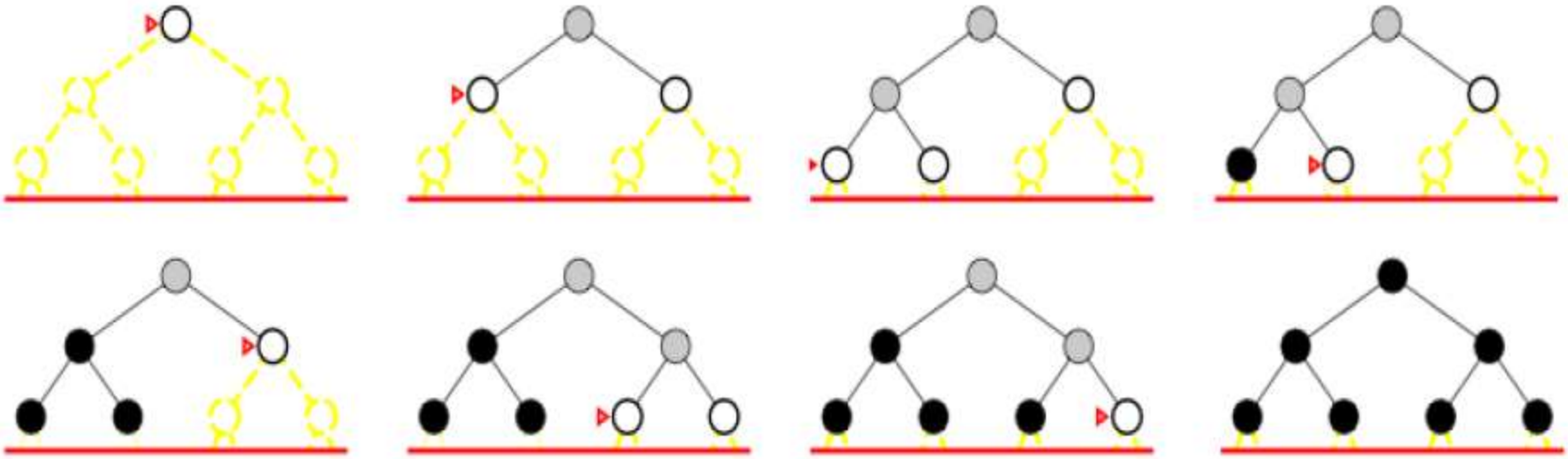
- Limit=0

# ID-Search: Example

- Limit=1

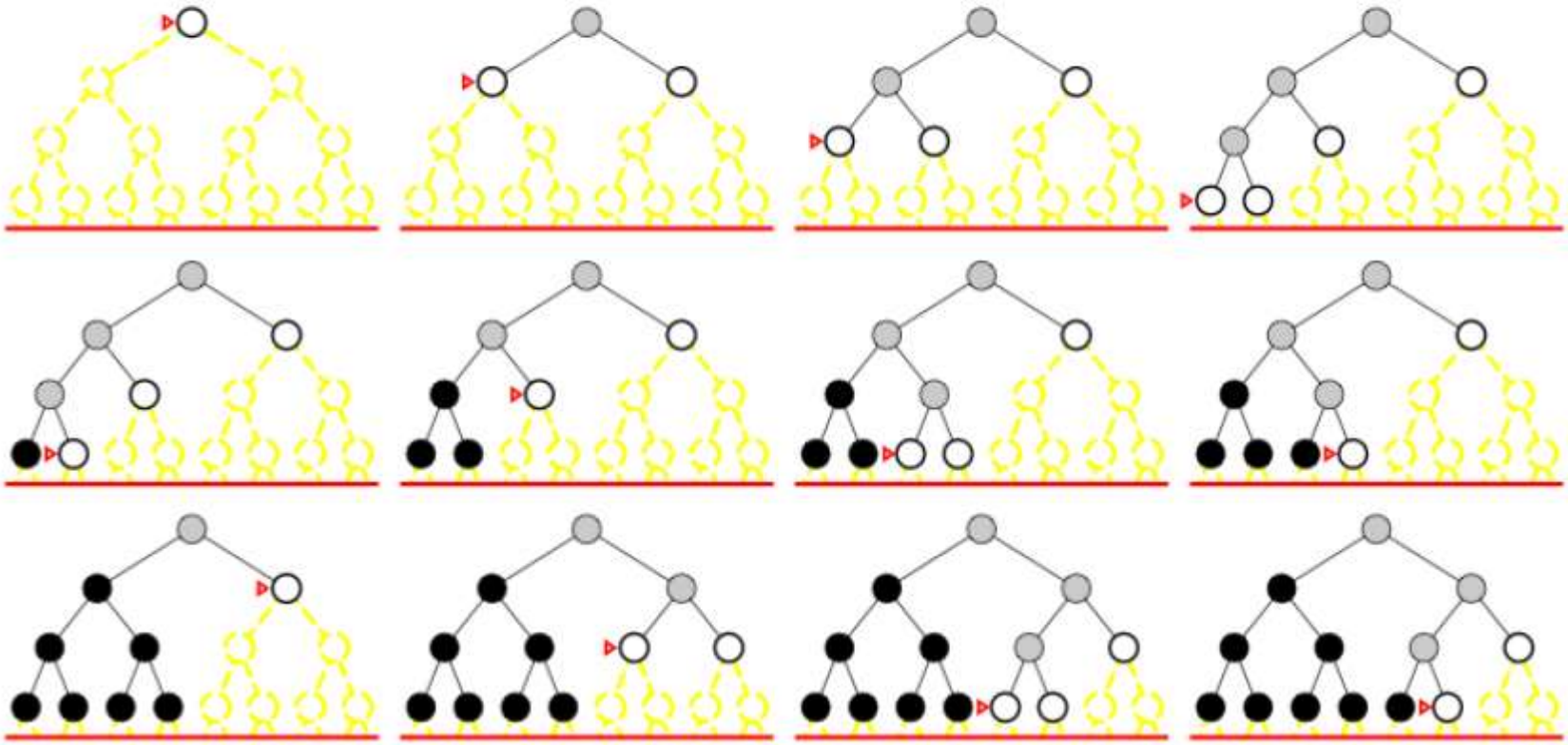
# ID-Search: Example

- Limit=2



# ID-Search: Example

- Limit=3

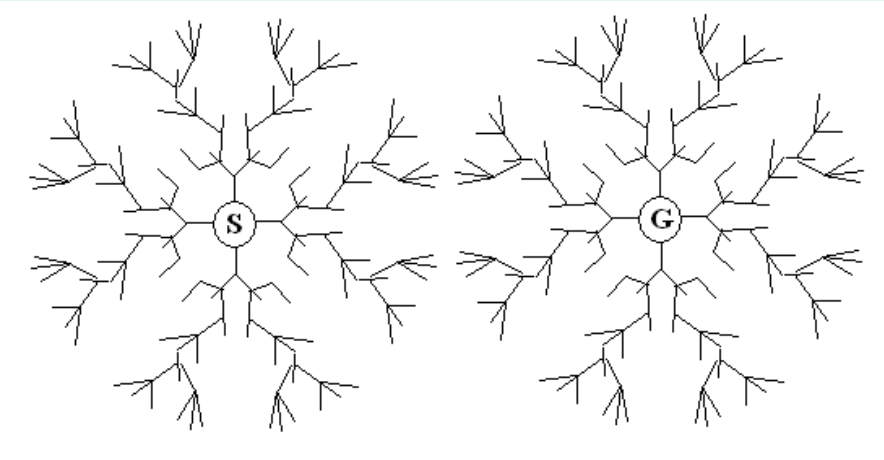


# Bidirectional Search

- Two simultaneous searches run from start and goal.
  - Motivation:

$$b^{d/2} + b^{d/2} \neq b^{d}$$

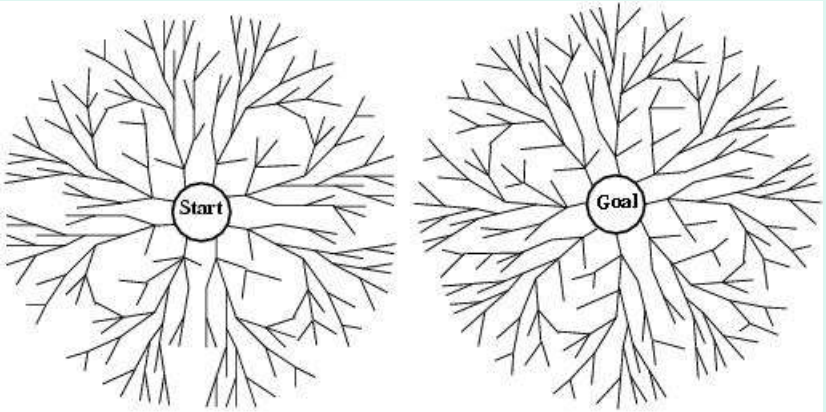
  - One forward from the initial state
  - Other backward from goal
  - Stops when the two searches meet in the middle
    - Check whether the node belongs to the other fringe before expansion



# Bidirectional Search

- Time complexity: $O(b^{d/2})$
- Space complexity: $O(b^{d/2})$
  - This space complexity is the most significant weakness of bidirectional search.
- Completeness and Optimality: Yes
  - If step costs are uniform
  - If both searches are breadth-first search.

# Bidirectional Search




- The reduction in time complexity makes bidirectional search attractive, but *how do we search backwards*?
- The predecessor of each node should be efficiently computable.
  - When actions are easily reversible.

# Summary of Algorithms

| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening | Bidirectional (if applicable) |
|---|---|---|---|---|---|---|
| Complete? | Yes[a] | Yes[a,b] | No | No | Yes[a] | Yes[a,d] |
| Time | $O(b^{d+1})$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(b^m)$ | $O(b^\ell)$ | $O(b^d)$ | $O(b^{d/2})$ |
| Space | $O(b^{d+1})$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(bm)$ | $O(b\ell)$ | $O(bd)$ | $O(b^{d/2})$ |
| Optimal? | Yes[c] | Yes | No | No | Yes[c] | Yes[c,d] |

**Figure 3.17** Evaluation of search strategies. $b$ is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if $b$ is finite; [b] complete if step costs $\geq \epsilon$ for positive $\epsilon$; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.