Artificial Intelligence

Hai Thi Tuyet Nguyen

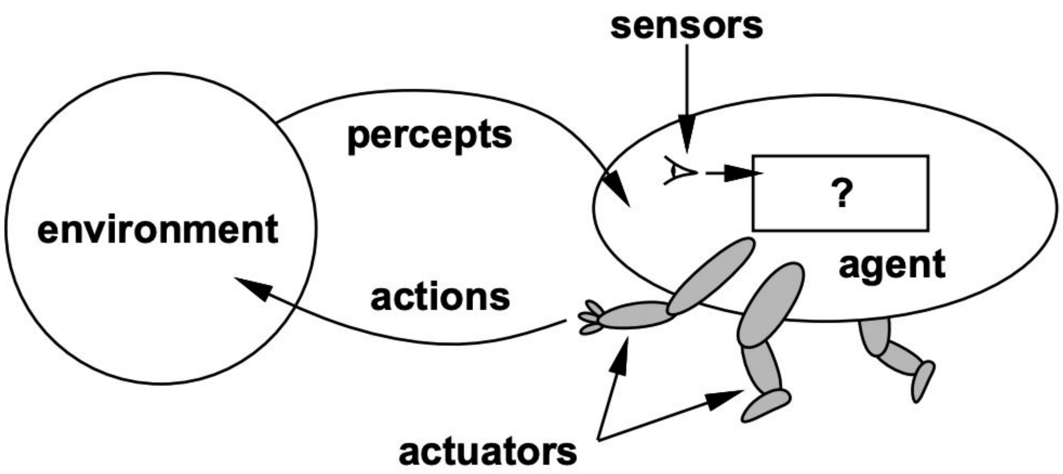
Outline

CHAPTER 3: SOLVING PROBLEMS BY SEARCHING

3.1 Problem-Solving Agents

3.1 Problem-Solving Agents

- An agent is something that
  - **perceives** its environment through **sensors**
  - **acts** upon that environment through **actuators**.
- A human agent has eyes, ears, … for sensors and hands, legs, … for actuators.



3.1 Problem-Solving Agents

A problem can be defined by 5 components:

- **initial state:** the **state** that the agent starts in.
- **possible actions** available to the agent.
  - from a state x, apply an action a, the agent reach the next state as y
- **state space**: the set of all states reachable from the initial state
  - the state space forms a **graph** in which
    the ***nodes*** are ***states***, the ***arcs*** between nodes are ***actions***.
  - a **path** in the state space is ***a sequence of states*** connected by a sequence of actions
- **goal test**: decide whether a given state is a goal state
- **path cost function**: assigns a numeric cost to each path
  - **step cost**: taking action a to go from state x to state y is denoted by $c(x,a,y)$.
  - **optimal solution**: the lowest path cost among all solutions

3.1 Problem-Solving Agents

- A simple problem-solving agent:
  - formulates a goal and a problem
  - searches for a sequence of actions to solve the problem
  - executes the actions one at a time

3.1 Single-state problem formulation

A single-state problem is defined by four items:

- initial state, e.g., "at A"
- possible actions

  $S(x)$ = set of action-state pairs

  e.g., $S(A) = \{<A \rightarrow Z, Z>, \ldots\}$
- goal test
  - explicit, e.g., x = "at B"
- path cost (additive)
  - e.g., sum of distances, number of actions executed, etc.
  - the step cost: $c(x, a, y)$, assumed to be ≥ 0

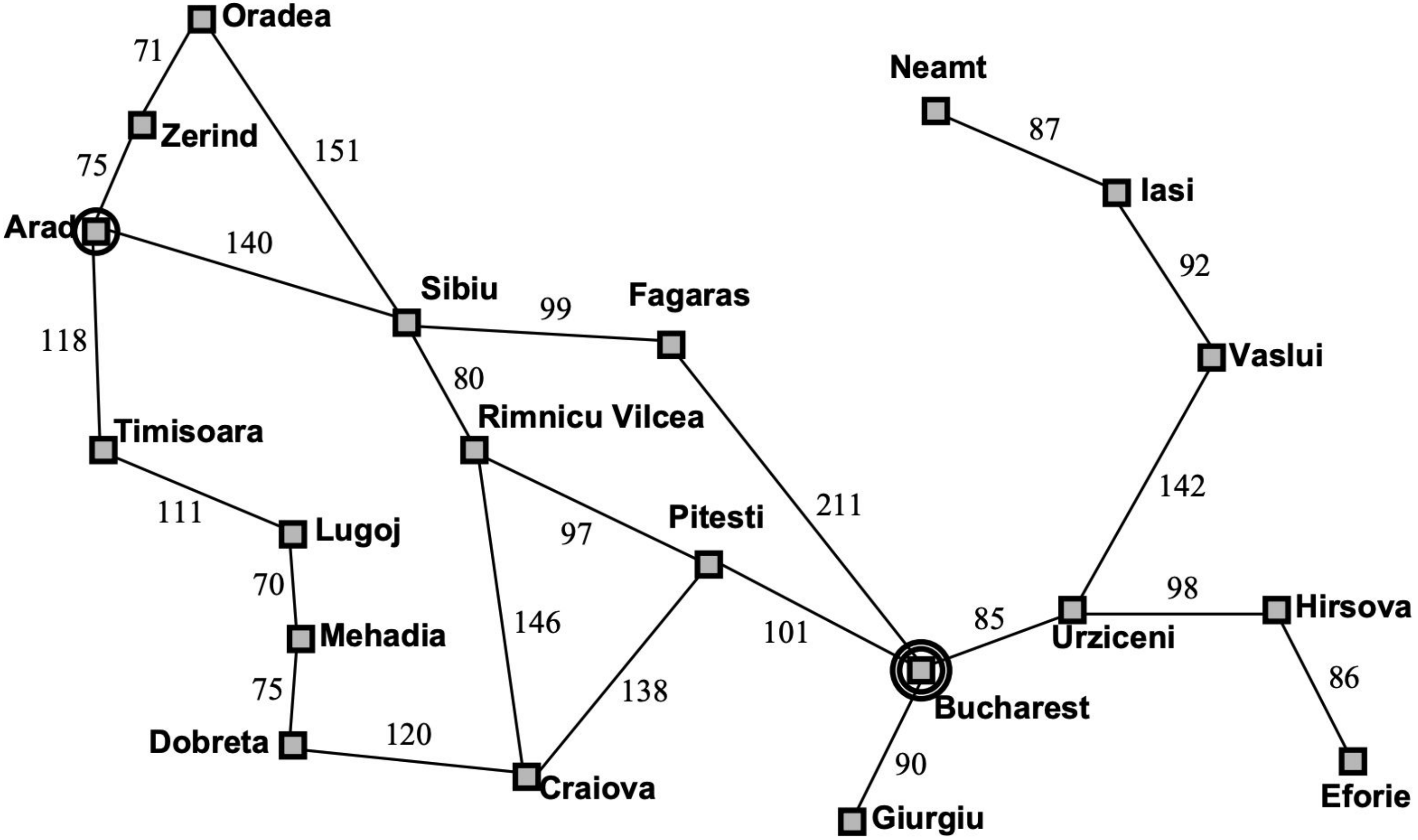
A solution is a sequence of actions leading from the initial state to a goal state.

3.2 Example

3.2 Example

- On holiday in Romania; currently 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

3.2 Example


Oradea
71
Zerind
151
75
Arad
140
Sibiu
99
Fagaras
118
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
Mehadia
146
101
75
138
Dobreta
120
Craiova
Neamt
87
Iasi
92
Vaslui
142
98
Hirsova
85
Urziceni
Bucharest
86
90
Giurgiu
Eforie

BFS

DFS

Apply BFS, DFS to find all paths from A to B.

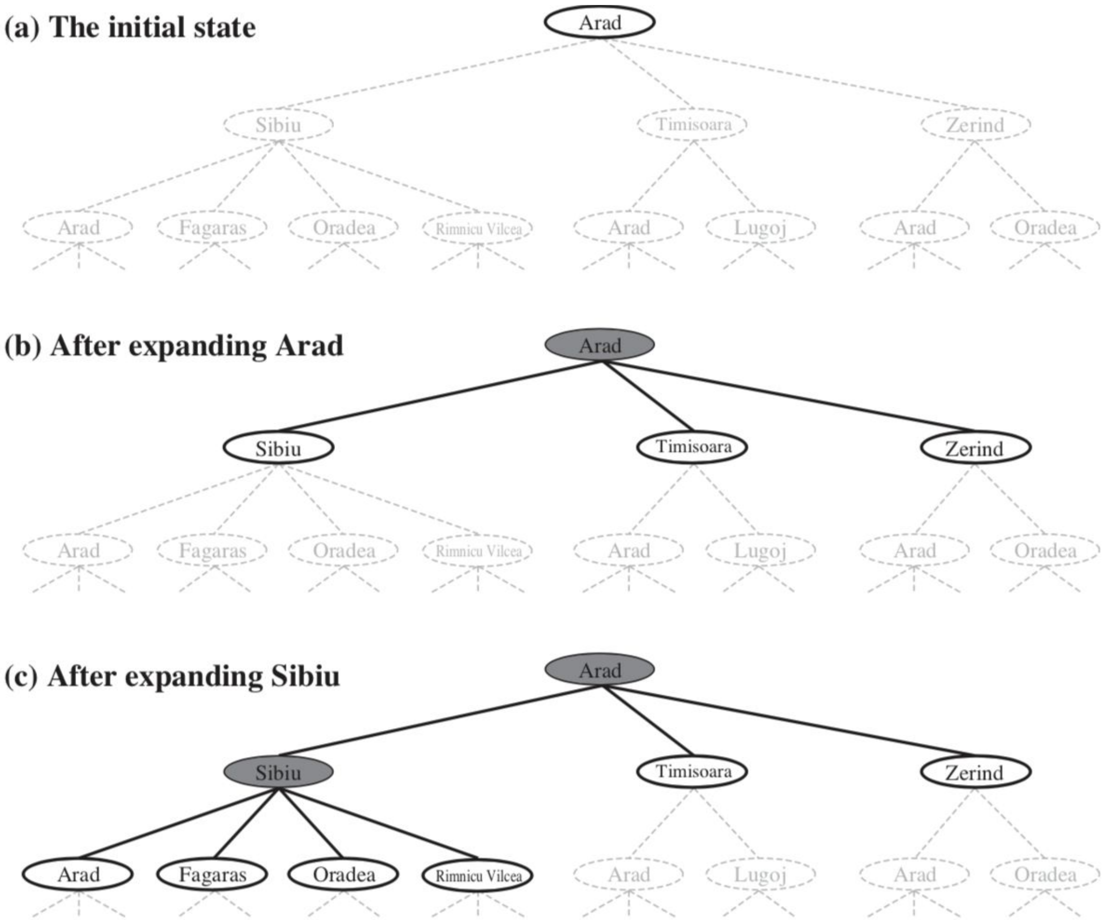
3.3 Searching For Solutions

3.3 Searching For Solutions

Tree search algorithms - basic idea

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

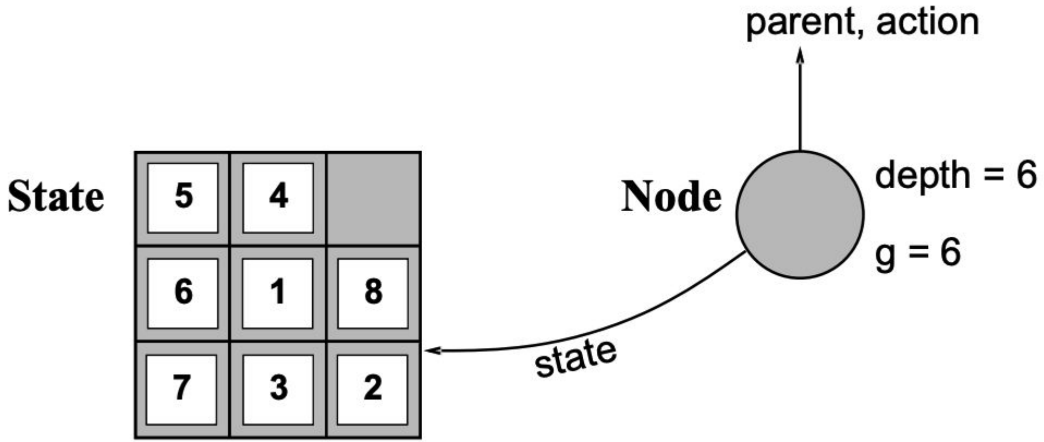
3.3 Searching For Solutions



3.3 Implementation

3.3 Implementation: states vs. nodes

- A node is a data structure which contains parent, children, depth, path cost *g(x)*
- A state is a representation of a physical configuration, states do not have parents, children, depth, or path cost



3.3 Searching For Solutions

Queue

- MAKE-QUEUE(element, …): creates a queue with the given elements.
- EMPTY?(queue): returns true only there are no more elements in the queue
- FIRST(queue): returns the first element of the queue
- **REMOVE-FRONT(queue):** returns the first element and removes it from the queue
- **INSERT(element, queue)**: inserts an element into the queue and returns the resulting queue
- **INSERT-ALL(elements, queue)**: inserts all elements into the queue and returns the resulting queue

3.3 Searching For Solutions

- fringe: the collection of left nodes that have been generated but not yet expanded
  the fringe argument must be an empty queue
  the type of the queue will affect the order of the search
- Expand(): create a set of new nodes
- SUCCESSOR-FN(x) returns a set of (action, successor): x as state, each successor is a state reached from x by applying the action

3.3 Searching For Solutions

3.3.2 Measuring problem-solving performance

- Completeness: Is the algorithm guaranteed to find a solution when there is one?
- 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?

Complexity is expressed in terms of 3 quantities:

- b: the branching factor or maximum number of successors of any node;
- d: the depth of the shallowest goal node
- m: the maximum length of any path in the state space

3.3 Implementation: general tree search

A strategy is defined by picking the order of node expansion

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

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

3.4 Uninformed search strategies

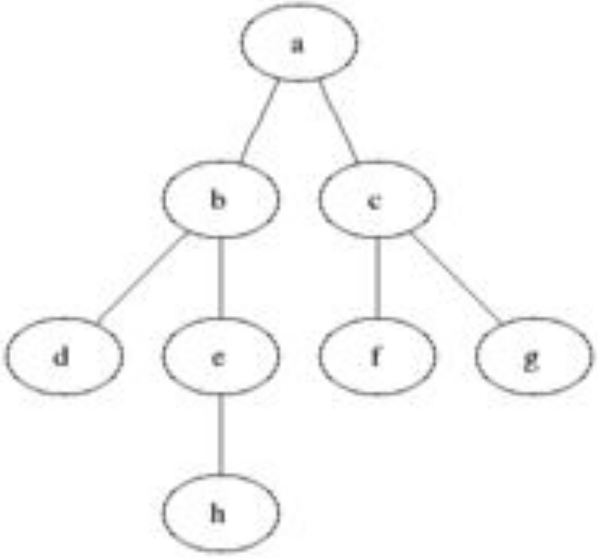
Uninformed strategies use only the information available in the problem definition

- ***Breadth-first search: BFS***
- Uniform-cost search
- ***Depth-first search: DFS***
- Depth-limited search
- Iterative deepening search

3.4 Uninformed search strategies

3.4.1 Breadth-first search

- BFS expands the shallowest nodes first
  - the root node is expanded -> its successors -> their successors, and so on.
- Implementation:
  - fringe: FIFO queue, i.e., new successors go at end



3.4 Uninformed search strategies

3.4.1 Breadth-first search

- Complete: yes if b is finite
- Time: $O(b^d)$
- Space: $O(b^d)$
- Optimal: yes if step costs all equal (shallowest path is lowest path cost)

3.4 Uninformed search strategies

3.4.2 Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
    - fringe = queue ordered by path cost, lowest first
- Equivalent to breadth-first if step costs all equal

3.4 Uninformed search strategies

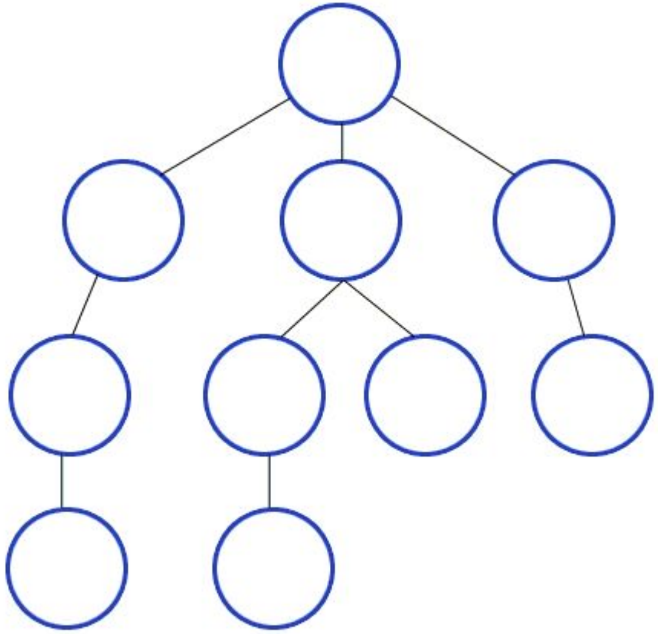
3.4.2 Uniform-cost search

- Complete: yes, if step cost $\geq \varepsilon$
- Time, space: uniform-cost search is guided by path costs rather than depths, so its complexity cannot be characterized in terms of b and d
- Optimal: yes, nodes expanded in increasing order of path cost

3.4 Uninformed search strategies

3.4.3 Depth-first search

- Expand deepest unexpanded node
- Implementation:
    - fringe = LIFO queue, i.e., put successors at front



3.4 Uninformed search strategies

3.4.3 Depth-first search

- Complete: yes in finite spaces
- Time: $O(b^m)$, terrible if m is much larger than d
  - m: the maximum length of any path
- Space: $O(bm)$, i.e., linear space!
  - store a single path from the root to a leaf node and the unexpanded sibling nodes for each node on the path.
  - when a node has been expanded and all its descendants have been explored, it can be removed from memory.
- Optimal: no, it can make a wrong choice and get stuck going down a very long path when another choice can lead to a solution near the root

3.4 Uninformed search strategies

Depth-limited search

- depth-first search with depth limit l, i.e., nodes at depth l have no successors

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
   RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
   cutoff-occurred? ← false
   if GOAL-TEST(problem, STATE[node]) then return node
   else if DEPTH[node] = limit then return cutoff
   else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
   if cutoff-occurred? then return cutoff else return failure
```

3.4 Uninformed search strategies

3.4.5 Iterative deepening depth-first search

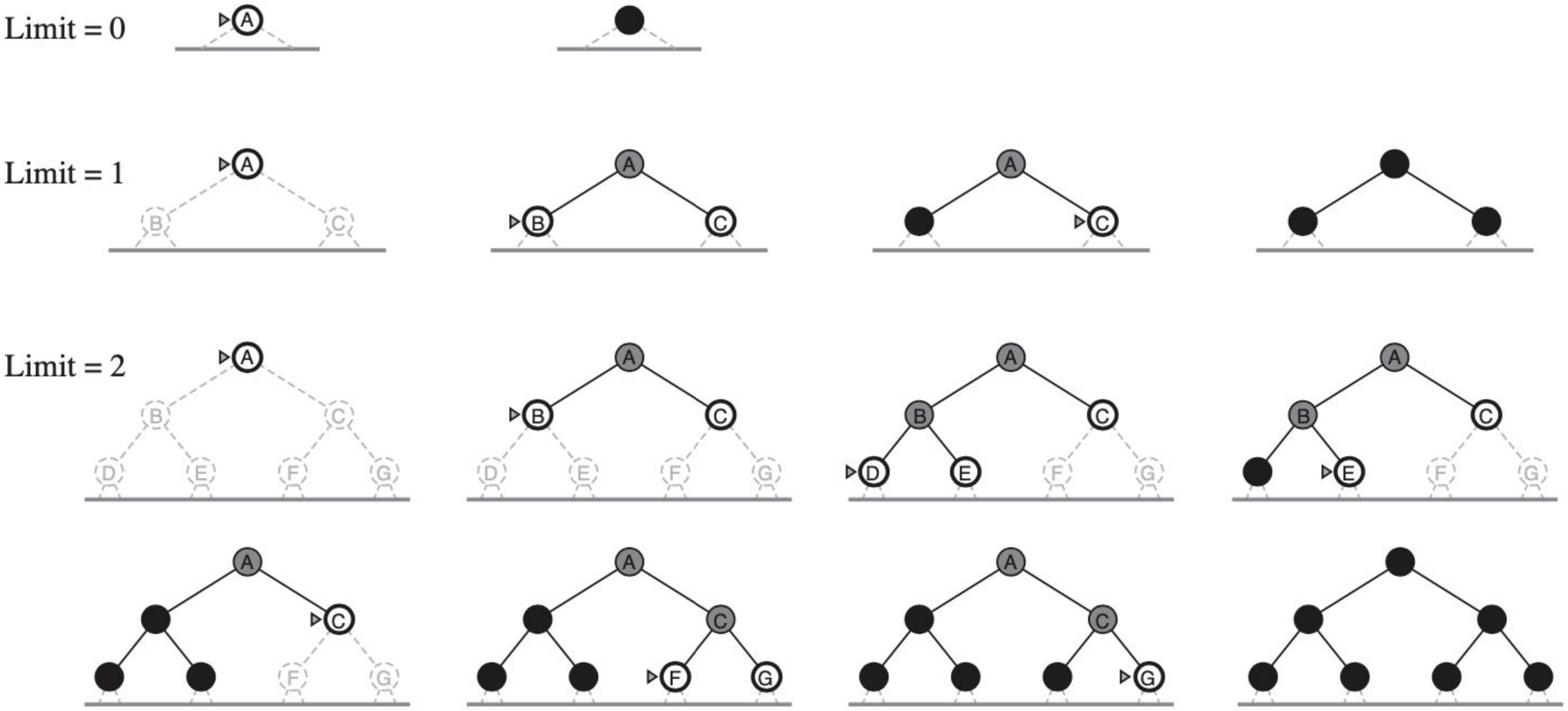
- depth-limited search with increasing limits
- terminates when a solution is found or if the depth- limited search returns failure, meaning that no solution exists.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
    end
```

3.4 Uninformed search strategies

3.4.5 Iterative deepening depth-first search



3.4 Uninformed search strategies

3.4.5 Iterative deepening depth-first search

- Complete: yes if b is finite
- Time: $O(b^d)$
    - d: the depth of the shallowest goal node
- Space: $O(bd)$
- Optimal: yes if step costs all equal

IDF is the preferred uninformed search method when there is a **large search space** and **the depth** of the solution is **unknown**.

3.4 Uninformed search strategies

3.4.7 Comparing uninformed search strategies

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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^c	Yes	No	No	Yes^c	Yes^{c,d}

Figure 3.21 Evaluation of tree-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 ϵ; ^{c} optimal if step costs are all identical; ^{d} if both directions use breadth-first search.

3.4 Avoid repeated states - Graph search

- Include the closed list, which stores every expanded node, into the general TREE-SEARCH algorithm.
- If the current node on the closed list, it is discarded instead of being expanded.

function GRAPH-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
    loop do
        if *fringe* is empty **then return** failure
        node ← REMOVE-FRONT(*fringe*)
        if GOAL-TEST(*problem*, STATE[*node*]) **then return** *node*
        if STATE[*node*] is not in *closed* **then**
            add STATE[*node*] to *closed*
            fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)
    end