Artificial Intelligence

Hai Thi Tuyet Nguyen

Outline

CHAPTER 1: INTRODUCTION (CHAPTER 1)

CHAPTER 2: INTELLIGENT AGENTS (CHAPTER 2)

CHAPTER 3: SOLVING PROBLEMS BY SEARCHING (CHAPTER 3)

CHAPTER 4: INFORMED SEARCH (CHAPTER 3)

CHAPTER 5: LOGICAL AGENT (CHAPTER 7)

CHAPTER 6: FIRST-ORDER LOGIC (CHAPTER 8, 9)

CHAPTER 7: QUANTIFYING UNCERTAINTY(CHAPTER 13)

CHAPTER 8: PROBABILISTIC REASONING (CHAPTER 14)

CHAPTER 9: LEARNING FROM EXAMPLES (CHAPTER 18)

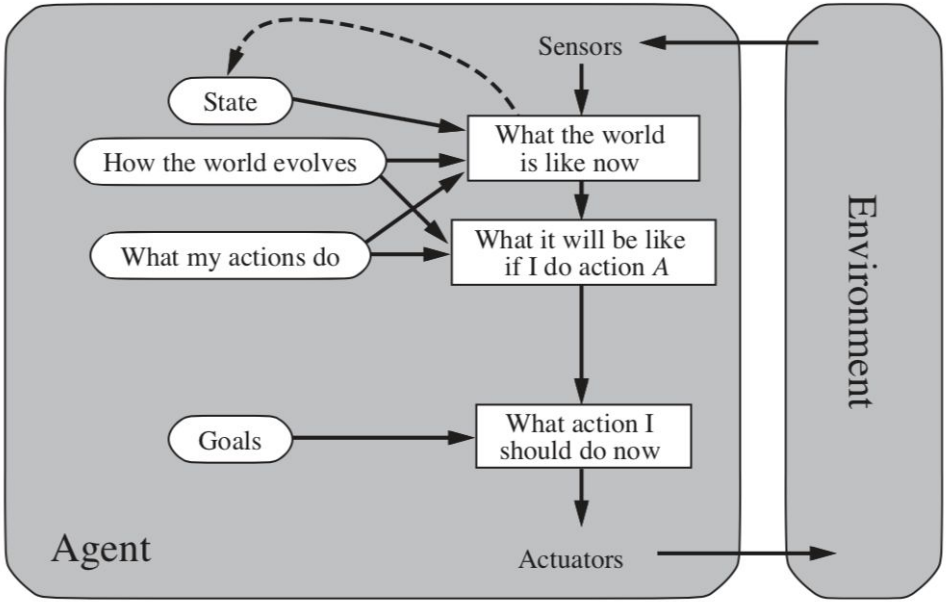
CHAPTER 3: SOLVING PROBLEMS BY SEARCHING

3.1 Problem-Solving Agents
3.2 Example Problems
3.3 Searching For Solutions
3.4 Uninformed Search Strategies

3.1 Problem-Solving Agents

3.1 Problem-Solving Agents

- A problem-solving agent is one kind of goal-based agent



3.1 Problem-Solving Agents

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

function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **returns** an action
  persistent: *seq*, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(*state*, *percept*)
  if *seq* is empty **then**
    goal ← FORMULATE-GOAL(*state*)
    problem ← FORMULATE-PROBLEM(*state*, *goal*)
    seq ← SEARCH(*problem*)
    if *seq* = *failure* **then return** a null action
  action ← FIRST(*seq*)
  seq ← REST(*seq*)
  return *action*

3.1 Problem-Solving Agents

A problem can be defined by 5 components:

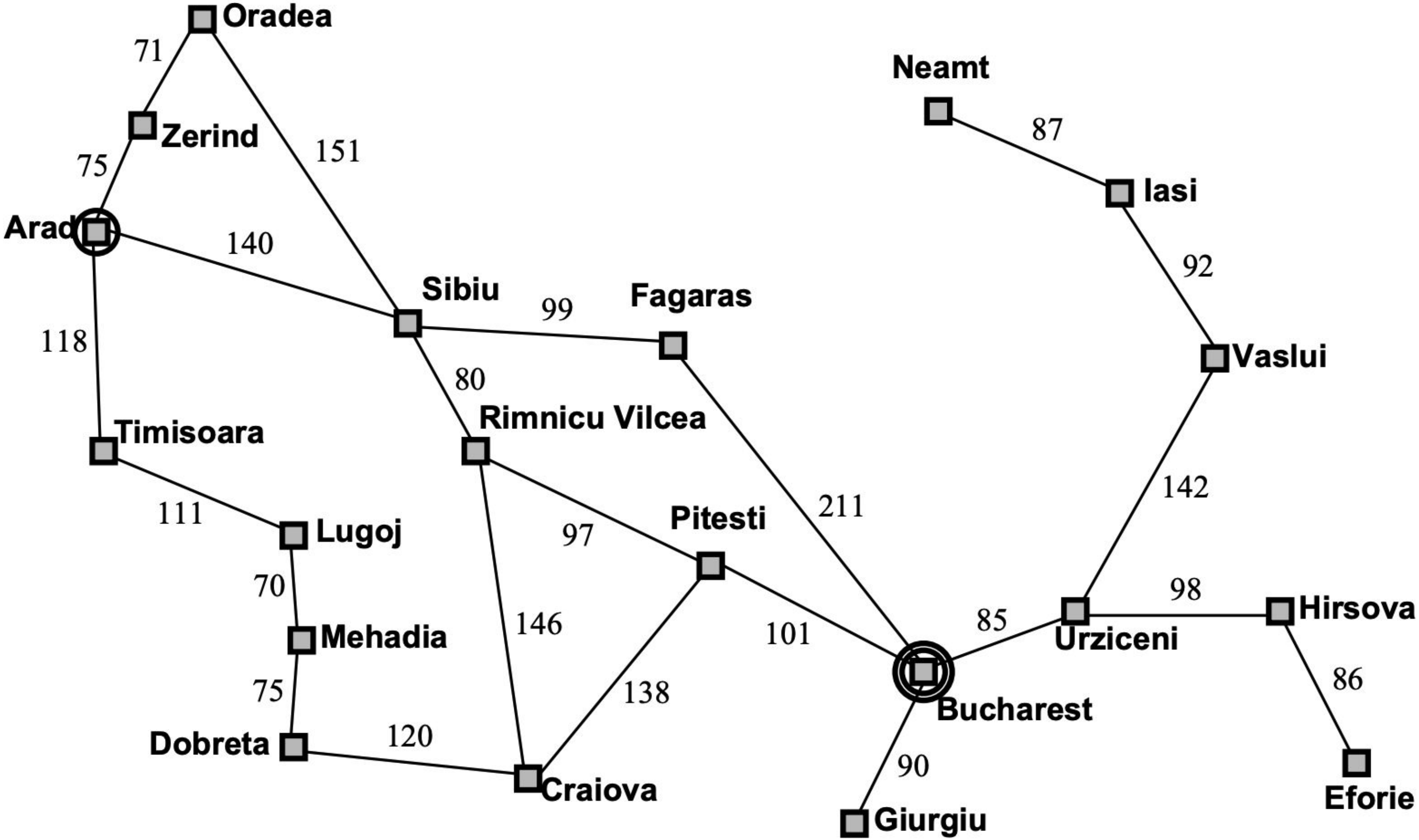
- **initial state:** the **state** that the agent starts in
- **possible actions** available to the agent.
  - Given a particular state s, **ACTIONS**(s) returns the set of actions that can be executed in s.
- **transition model:** specified by a function **RESULT**(s,a) that returns the state that results from doing action a in state s
- ***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 (~performance measure)**: assigns a numeric cost to each path
  - **step cost**: taking action a to go from state x to state y is denoted by $c(x,a,y)$.
  - **solution**: an action sequence that leads from the initial state to a goal state.
  - **optimal solution**: the lowest path cost among all solutions

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

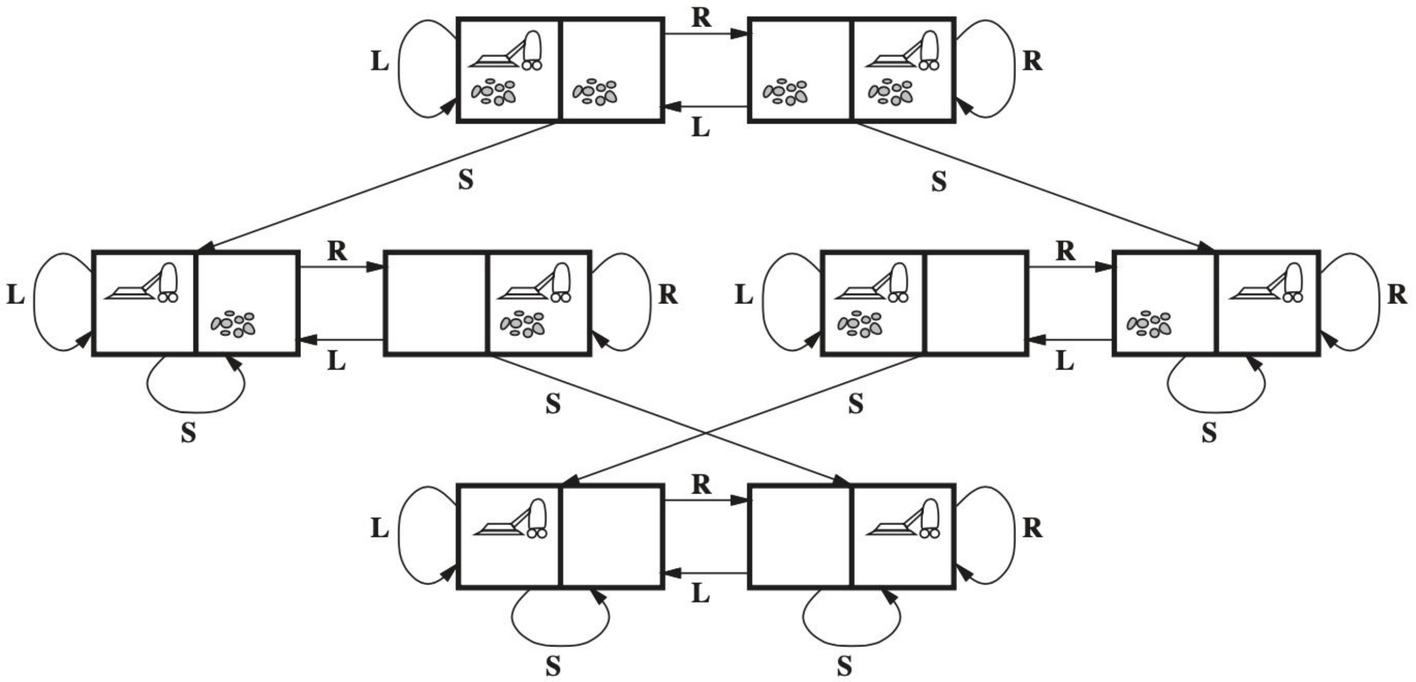

Oradea
71
Zerind
151
75
Arad
140
118
Sibiu
99
Fagaras
80
Rimnicu Vilcea
Timisoara
111
Lugoj
70
Mehadia
75
Dobreta
120
146
97
Pitesti
138
Craiova
211
101
Bucharest
90
Giurgiu
85
Urziceni
98
Hirsova
86
Eforie
142
Vaslui
92
Iasi
87
Neamt

3.2 Example

Example:

- initial state, e.g., In(Arad)
- possible actions
  - from the state In(Arad), the applicable actions are {Go(Sibiu),Go(Timisoara),Go(Zerind)}
- transition model
  - RESULT(In(Arad),Go(Zerind)) = In(Zerind) .
- goal test
  - explicit, e.g., x = In(Arad)
- path cost (additive)
  - e.g., sum of distances, number of actions executed, etc.
  - the step cost: $c(x, a, y)$, assumed to be ≥ 0

3.2 Example: vacuum world



3.2 Example: vacuum world

- States: location and contents, e.g., [A, Dirty], $2 \times 2^2 = 8$ possible world states
- Initial state: Any state can be designated as the initial state.
- Actions: 3 actions: Left, Right, and Suck.
- Transition model: The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect.
- Goal test: This checks whether all the squares are clean.
- Path cost: Each step costs 1, so the path cost is the number of steps in the path.

3.2 Real-world problems

- Route-finding problem: travel-planning, in-car systems
- Touring problems
- Traveling Salesperson Problem (TSP) is a touring problem in which each city must be visited exactly once. The aim is to find the shortest tour.
- Robot navigation

3.3 Searching For Solutions

3.3 Tree search algorithms - basic idea

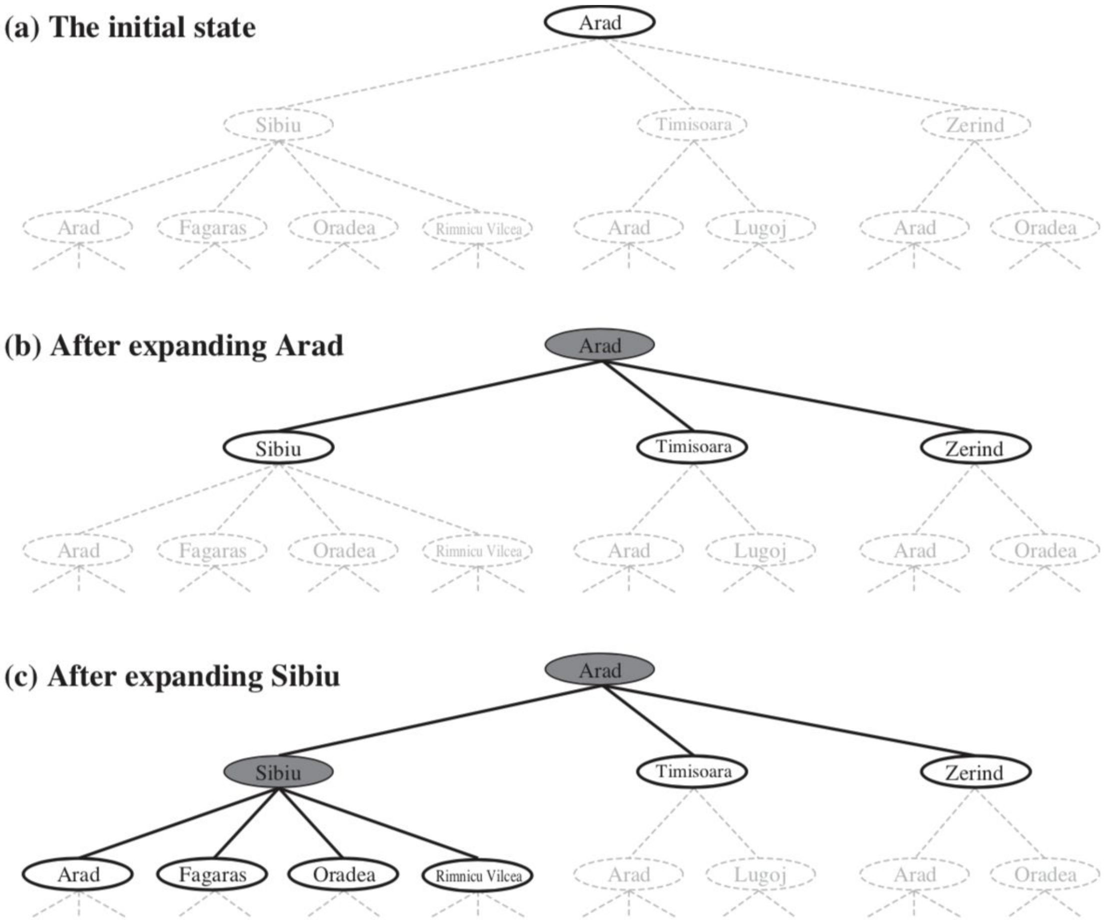
- Search tree with
  - the initial state at the root
  - the branches are actions
  - the nodes correspond to states in the state space of the problem.
- **Expanding** the current state;
  - applying each legal action to the current state, thereby generating a new set of states.
- The set of all leaf nodes available for expansion at any given point is called the **frontier** or **open list**.

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 Searching For Solutions

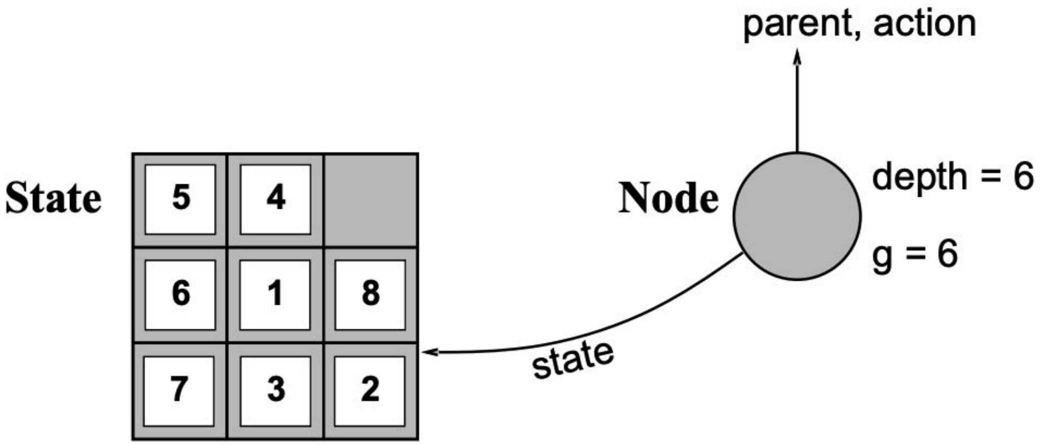
The explored set (also known as the closed list): set of all explored nodes

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

3.3.1 Data structure

For each node n of the tree, we have a structure that contains four components:

- n.STATE: the state in the state space to which the node corresponds;
- n.PARENT: the node in the search tree that generated this node;
- n.ACTION: the action that was applied to the parent to generate the node;
- n.PATH-COST: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.



3.3.2 Measuring problem-solving performance

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 4 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 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 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

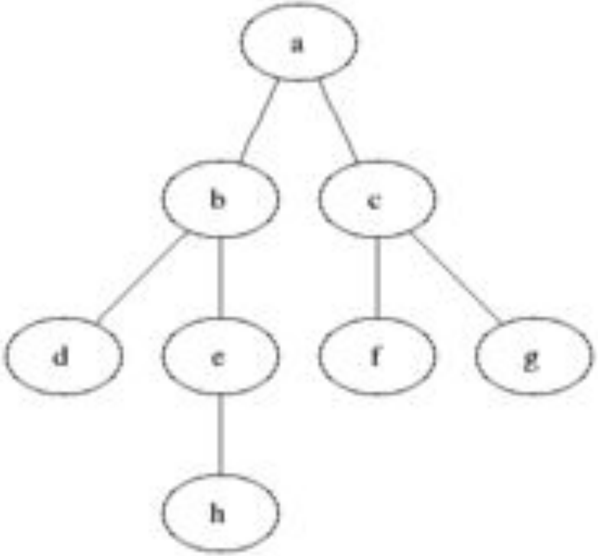
Uninformed strategies use only the information available in the problem definition

- ***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.



3.4 Uninformed search strategies

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

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)

(page 82 textbook)

Complexity is expressed in terms of 4 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

3.4 Uninformed search strategies

3.4.2 Uniform-cost search

- Expand least-cost unexpanded node
- Equivalent to breadth-first if step costs all equal

3.4 Uninformed search strategies

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

3.4 Uninformed search strategies

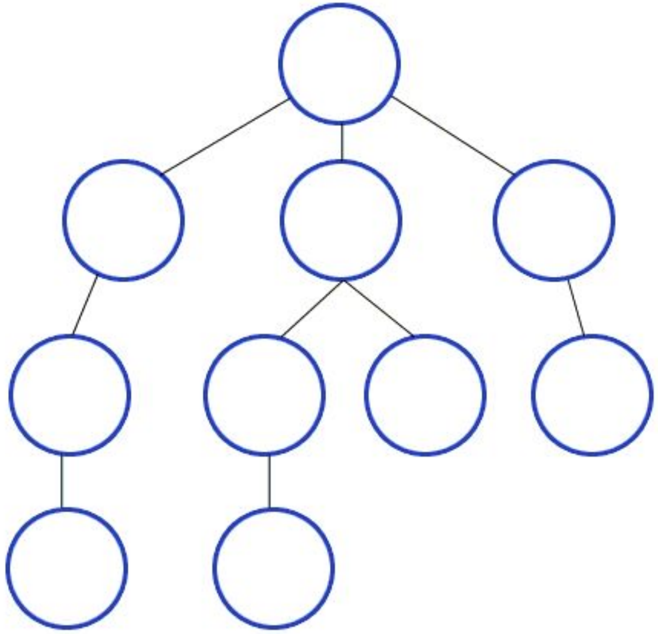
3.4.2 Uniform-cost search

- 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

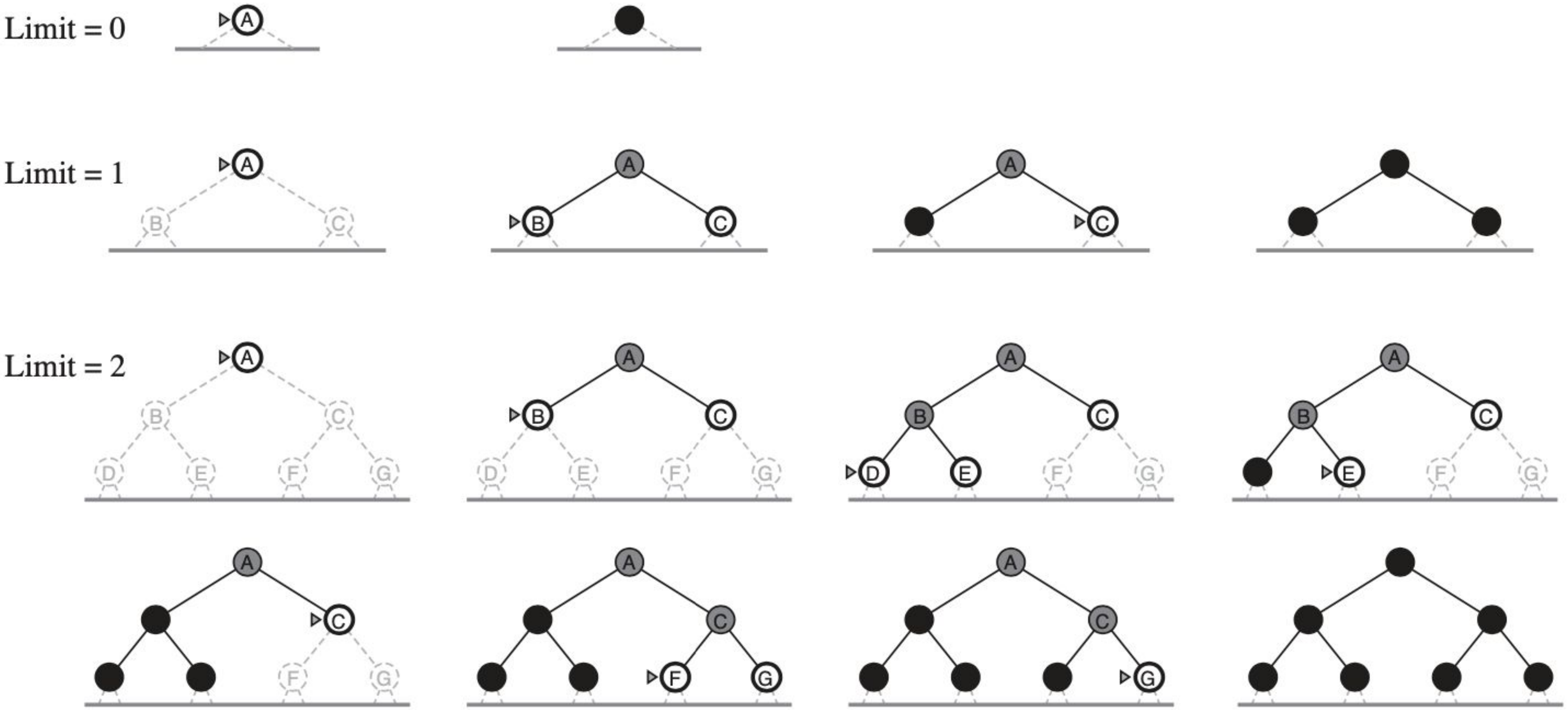
- depth-limited search with increasing limits
- terminates when a solution is found or if the depth- limited search returns failure, meaning that no solution exists.

```
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
```