2802ICT Intelligent Systems

Uninformed search strategies

Outline

- Definition and formulation
- Optimality, Completeness, and Complexity
- Uninformed Search
  - Breadth-first
  - Uniform-cost
  - Depth-first
  - Depth-limited
  - Iterative deepening
  - Bidirectional

Read textbook: Chapter 3, pages 64-91

Representation of the world:

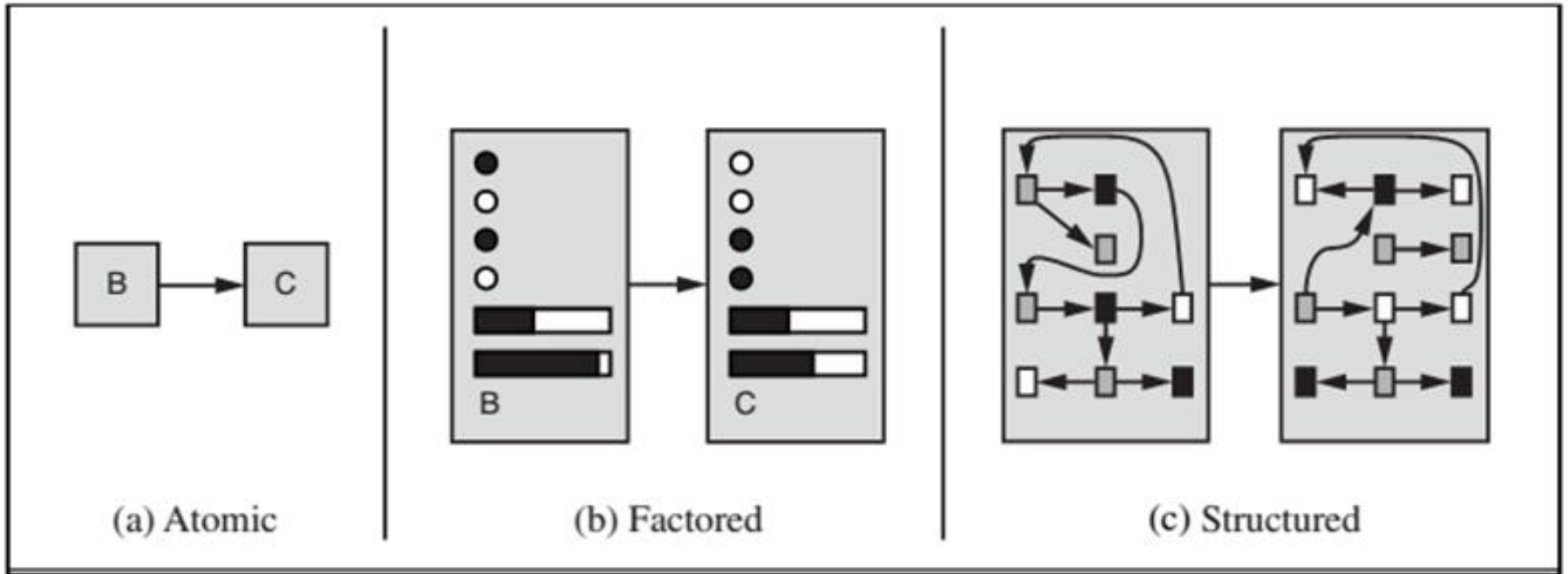


Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols. (c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.

Problem-Solving Agents

- Use **atomic** representations

```
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
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Properties of environment

- **Observable**: always know the current state
- **Discrete**: only finitely many actions to choose from
- **Known**: knows which states are reached by each action
- **Deterministic**: each action has one outcome

Example: Romania

- On holiday in Romania, currently in Arad
- Goal: to be in Bucharest
- States: Various cities
- Optimal solution: Find sequence of cities from initial state (Arad) to goal state (Bucharest) that has the lowest path cost

Example: Romania

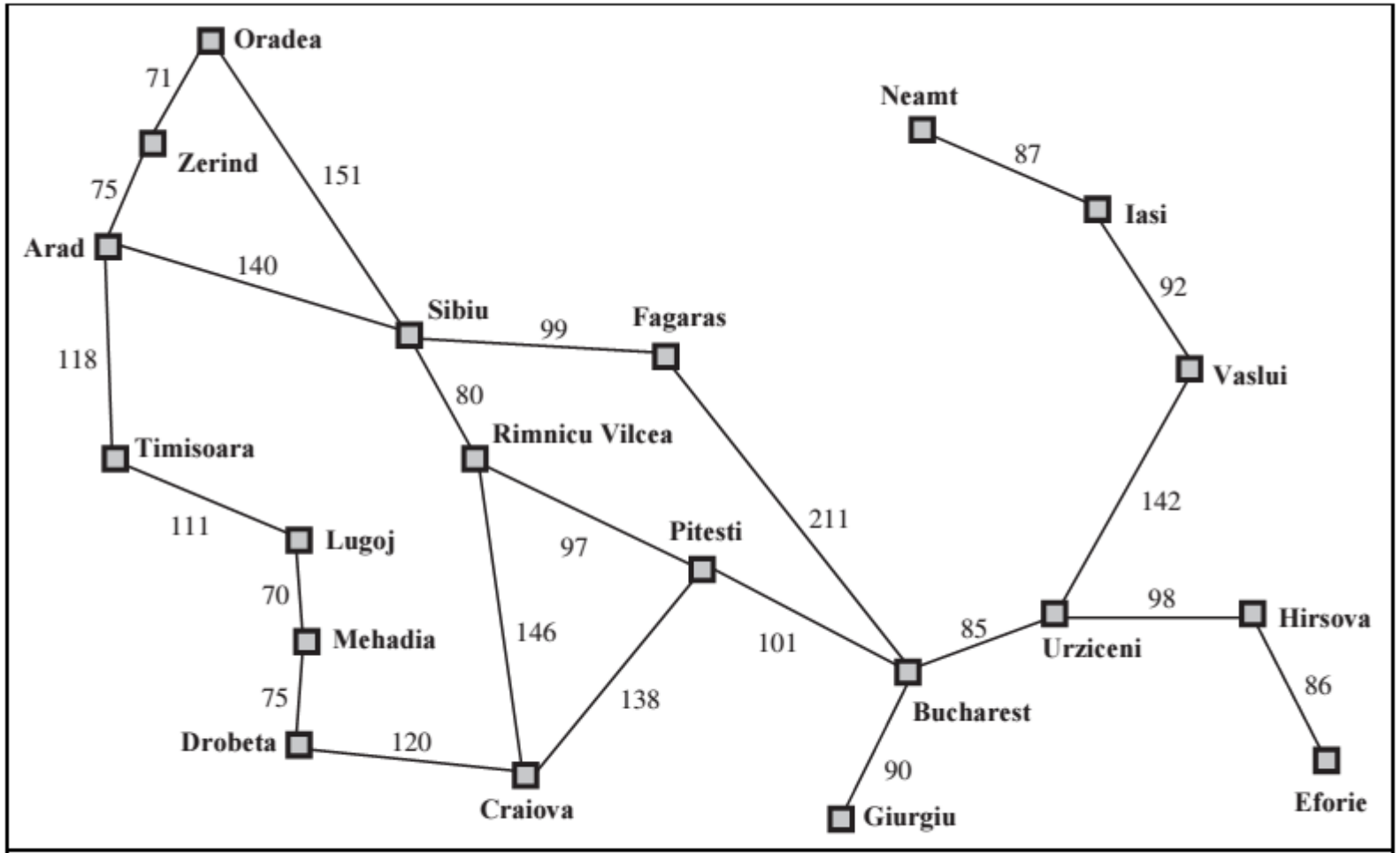


Figure 3.2 A simplified road map of part of Romania.

Problem formulation

- A problem is defined by five components:
  - **Initial state**: e.g. *In*(Arad)
  - **Actions**: Given a particular state *s*, *ACTIONS*(*s*) returns the set of actions that can be executed in *s*. E.g. the applicable actions for *In*(Arad) are {*Go*(Sibiu), *Go*(Timisoara), *Go*(Zerind)}.
  - **Transition model**: specified by a function *RESULT*(*s*, *a*). Also called ***successor*** which refers to any state reachable from a given state by a single action. E.g. *RESULT*(*s*, *a*)

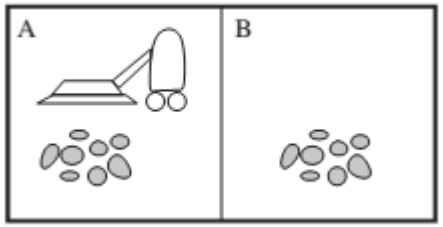
    RESULT(*In*(Arad), *Go*(Zerind)) = *In*(Zerind)

    Together, the initial state, actions, and transition model define the *state space* of the problem.

    A *path* in the state space is a sequence of states connected by a sequence of actions.
  - **Goal test**: determines whether a given state is a goal state.
  - **Path cost**: assigns a numeric cost to each path. It is given by the sum of *step cost*.

Example: Vacuum world

A vacuum-cleaner world with just two locations:



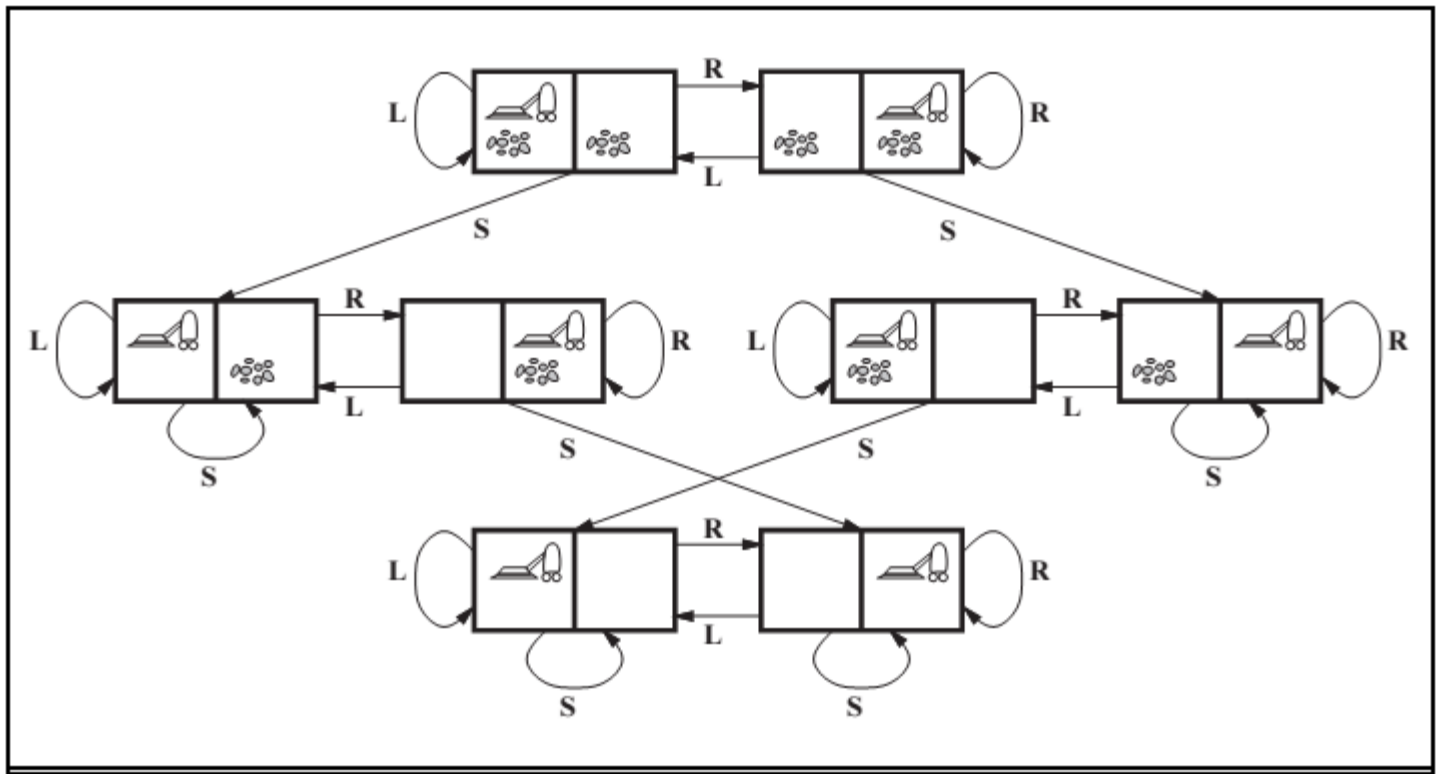


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

Example: Vacuum world

- **States**: Agent in one of two locations, each of which might or might not contain dirt. Hence, there are $2x2^2=8$ possible world states
- **Initial state**: Can be any of the world state
- **Actions**: *Left*, *Right*, *Suck*
- **Transition model**: Actions have their expected effects, except for move *left* in leftmost square, move *right* in rightmost square, and *suck* in clean square
- **Goal test**: Check whether all squares are clean
- **Path cost**: Step cost is 1, so path cost = number of steps in the path

Example: 8-puzzle

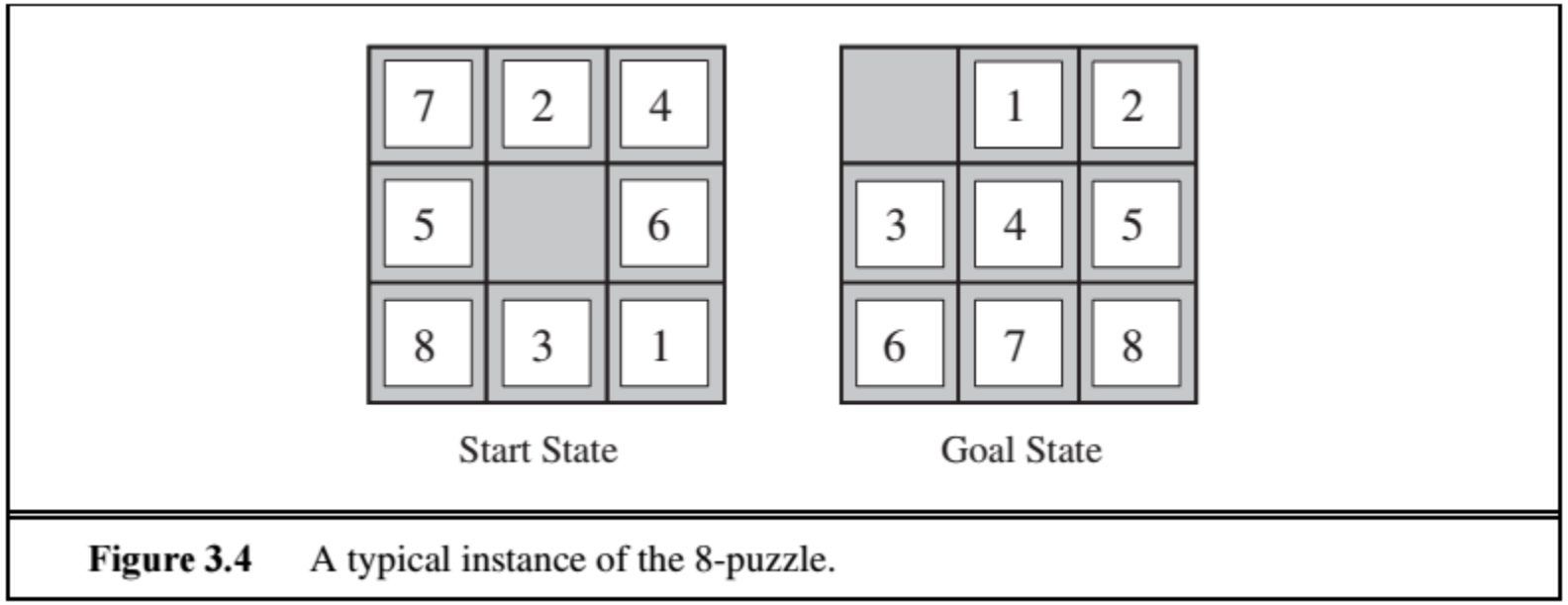


Figure 3.4 A typical instance of the 8-puzzle.

Example: 8-puzzle

- **States**: Specifies the location of each of the eight tiles and the blank, has 9!/2 reachable states
- **Initial state**: Can be any of the world state
- **Actions**: move blank space *Left*, *Right*, *Up*, *Down*
- **Transition model**: E.g. apply *Left* to start state results in the blank and ‘5’ switched
- **Goal test**: Check whether the goal state configuration is achieved
- **Path cost**: Step cost is 1, so path cost = number of steps in the path

Search Algorithm

- Basic idea: exploration of search space by generating successors of already explored states

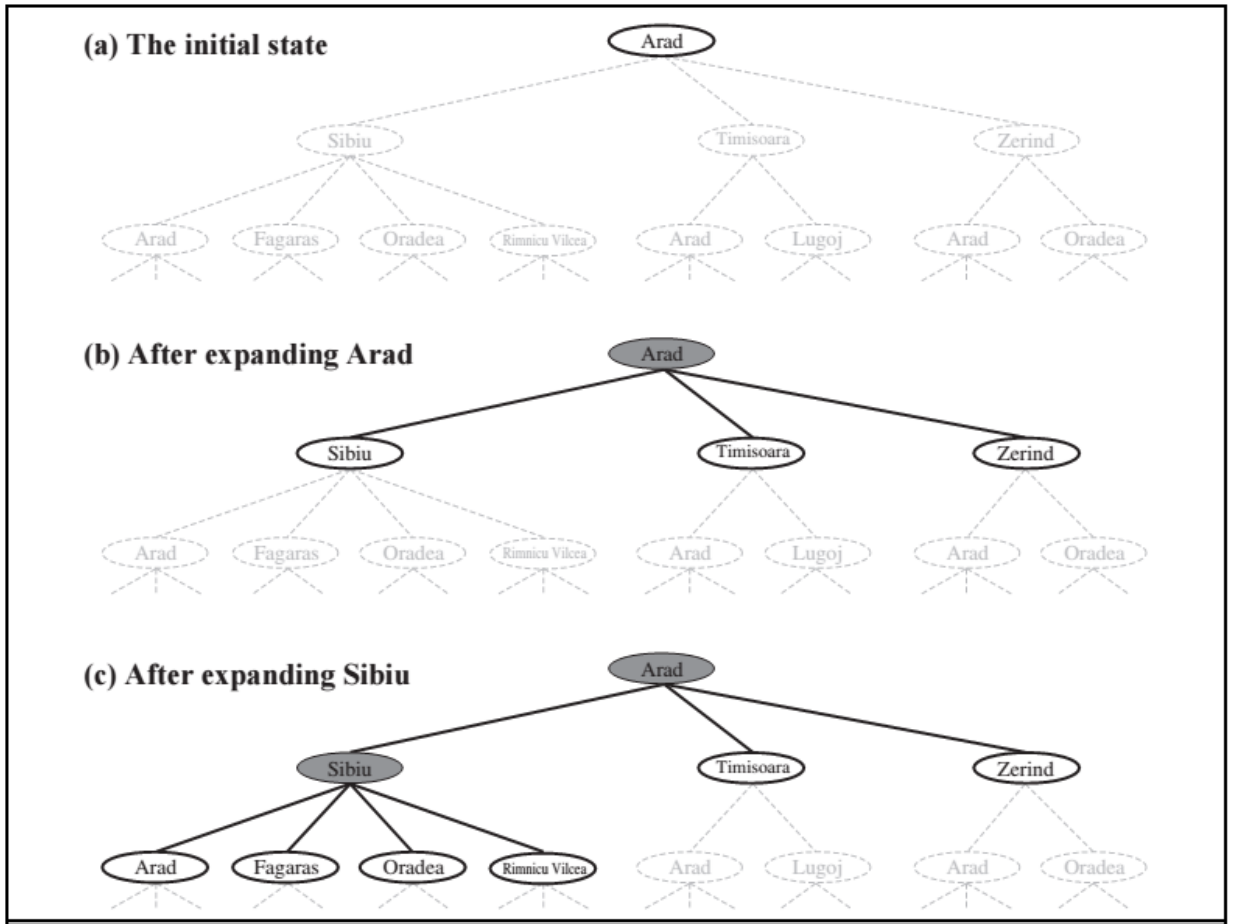


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Search Algorithm

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  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
    expand the chosen node, adding the resulting nodes to the frontier

---

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

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Example: Romania

Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

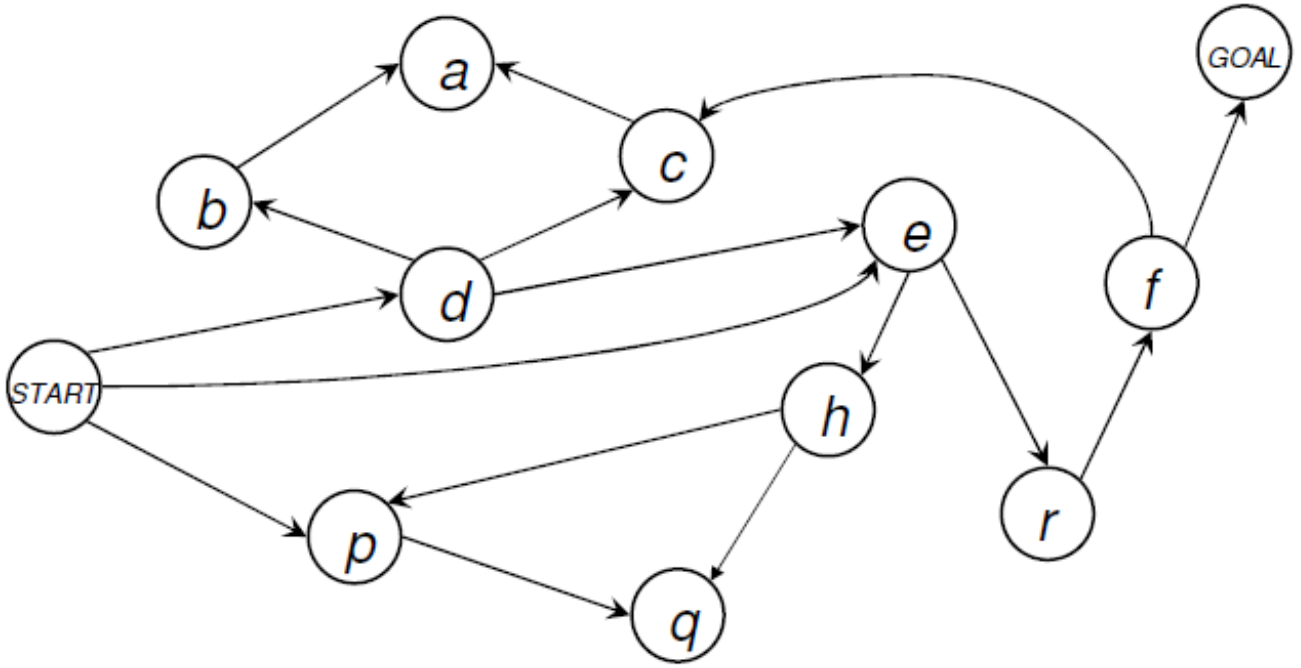
Formulation

- Q: Finite set of states
- $S \subseteq Q$: Non-empty set of start states
- $G \subseteq Q$: Non-empty set of goal states
- **succs**: function $Q \rightarrow \mathcal{P}(Q)$
  succs(s) = Set of states that can be reached from s in one step
- **cost**: function $Q \times Q \rightarrow$ Positive Numbers
  cost(s,s') = Cost of taking a one-step transition from state s to state s'

- Problem: Find a sequence $\{s_1,\ldots,s_K\}$ such that:

1. $s_1 \in S$
2. $s_K \in G$
3. $s_{i+1} \in$ **succs**(s_i)
4. $\sum$ **cost**(s_i, s_{i+1}) is the smallest among all possible sequences (desirable but optional)

Formulation



- $Q = \{START, GOAL, a, b, c, d, e, f, h, p, q, r\}$
- $S = \{START\}$ $G = \{GOAL\}$
- **succs**(d) = $\{b,c\}$
- **succs**($START$) = $\{p,e,d\}$
- **succs**(a) = NULL
- **cost**(s,s') = 1 for all transitions

Data structure of search tree

- 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.

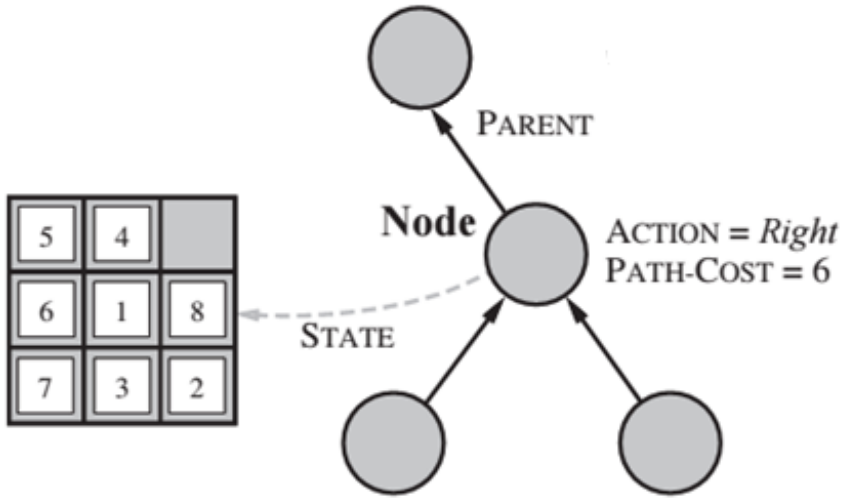


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Desirable properties

- ***Completeness*:** Is the algorithm guaranteed to find a solution if one exists?
- ***Optimality*:** Does the strategy find the optimal solution (e.g. lowest path cost)?
- ***Time Complexity*:** How long does it take to find the solution?
- ***Space Complexity*:** How much memory is needed to perform the search?

Time and space complexity are measured in terms of

- *b* : branching factor of the search tree, i.e. maximum number of successors of any node
- *d* : depth of the shallowest goal node
- *m* : maximum length of any path in the state space

Breadth-first search

- Expand shallowest unexpanded node
- Implementation:

  Use a FIFO queue for the *frontier*, i.e., new successors go to end of queue

```
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)
```

Figure 3.11 Breadth-first search on a graph.

Breadth-first search

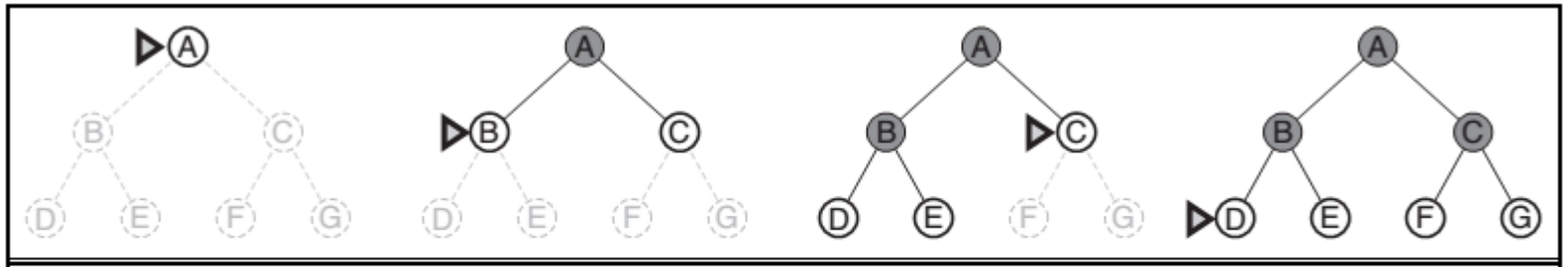


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Properties of breadth-first search

- Complete? Yes (if b is finite)
- Time? Total number of nodes generated is $b+b^2+b^3+\ldots+b^d = (b^{k+1}-1)/(b-1) = O(b^d)$
- Space? $O(b^{d-1})$ nodes in explored set and $O(b^d)$ nodes in frontier (keeps every node in memory), so space complexity is $O(b^d)$
- Optimal? Yes (if step costs are all equal)

Space is the bigger problem (more than time)

Time and memory cost for BFS

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Uniform-cost search

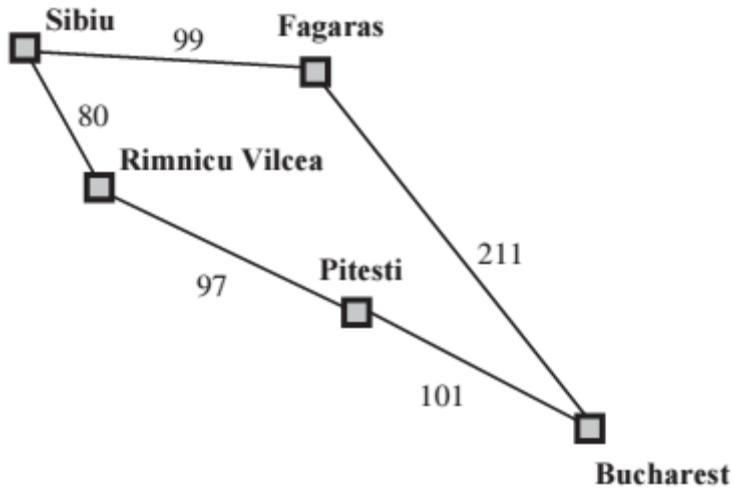
- Optimal for any step-cost function
- Expand least-cost unexpanded node
- Implementation:
  - *frontier* = priority queue ordered by path cost
- Equivalent to breadth-first if step costs all equal

Uniform-cost search

```
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
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

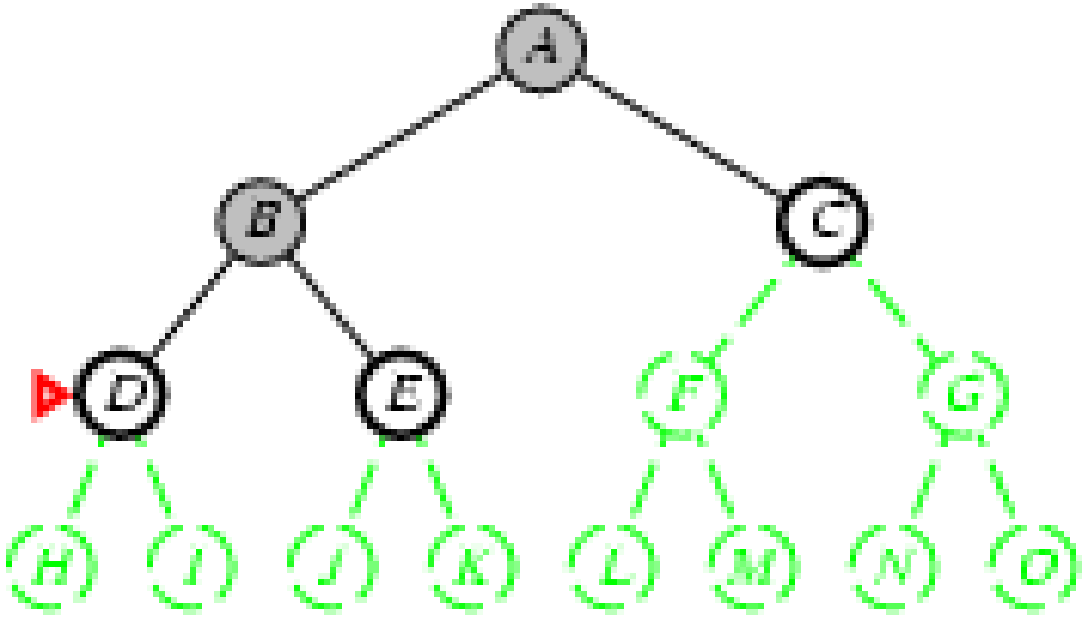


Uniform-cost search

- Complete? Yes, if step cost ≥ ε (a small positive constant, to prevent stuck in infinite loop)
- Time? $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$ where C^* is the cost of the optimal solution
  - i.e. taking steps at distances 0, ε, 2ε, 3ε, …, (C*/ε) ε
- Space? $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
- Optimal? Yes – nodes expanded in increasing order of *g(n)*

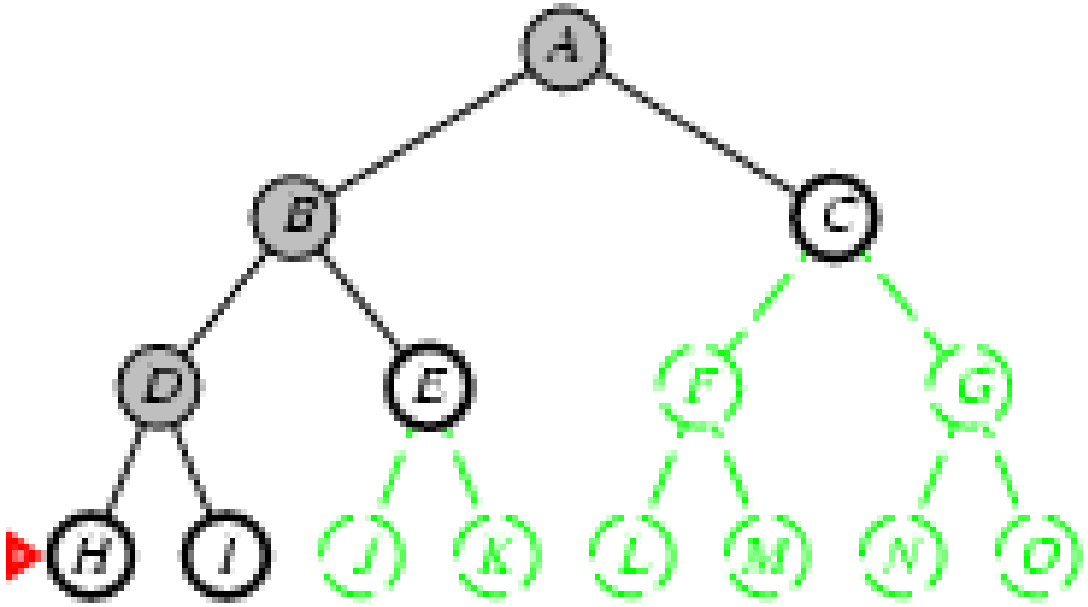
Depth-first search

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



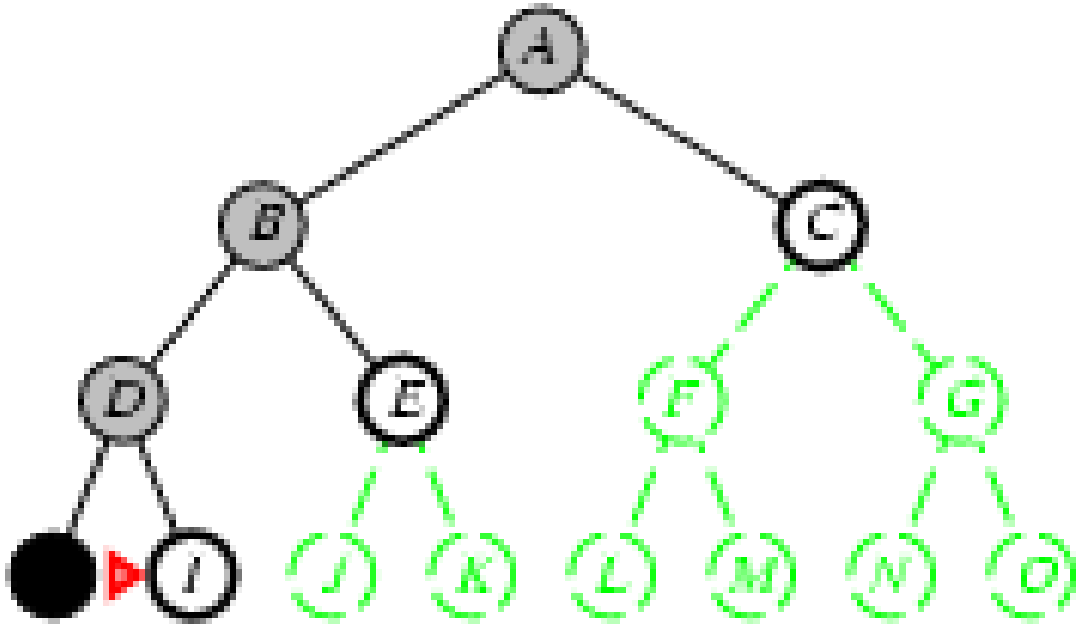
Depth-first search

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



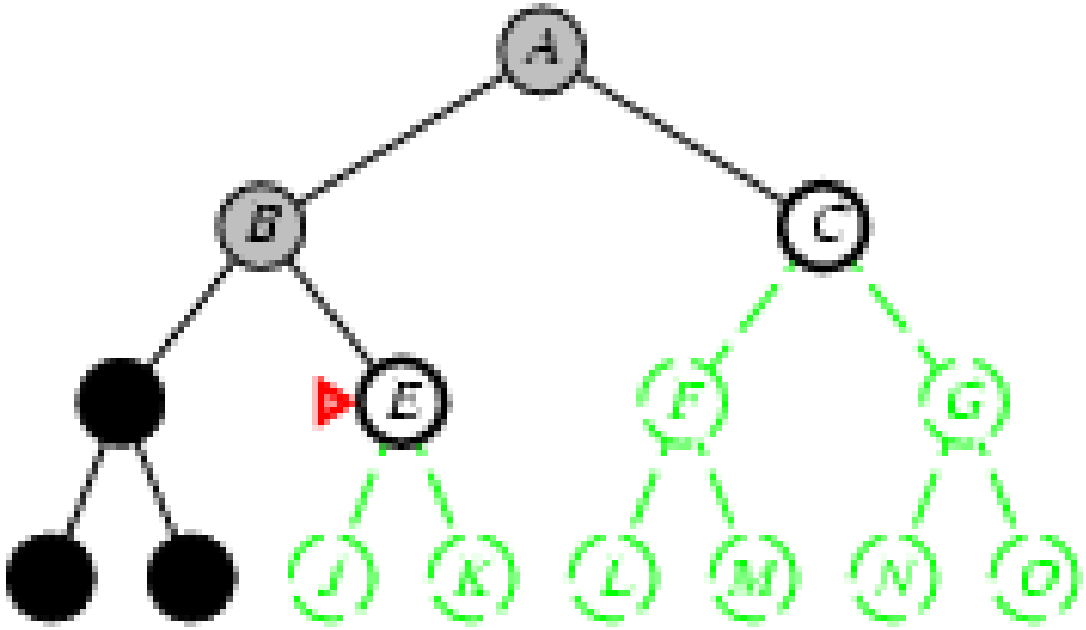
Depth-first search

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



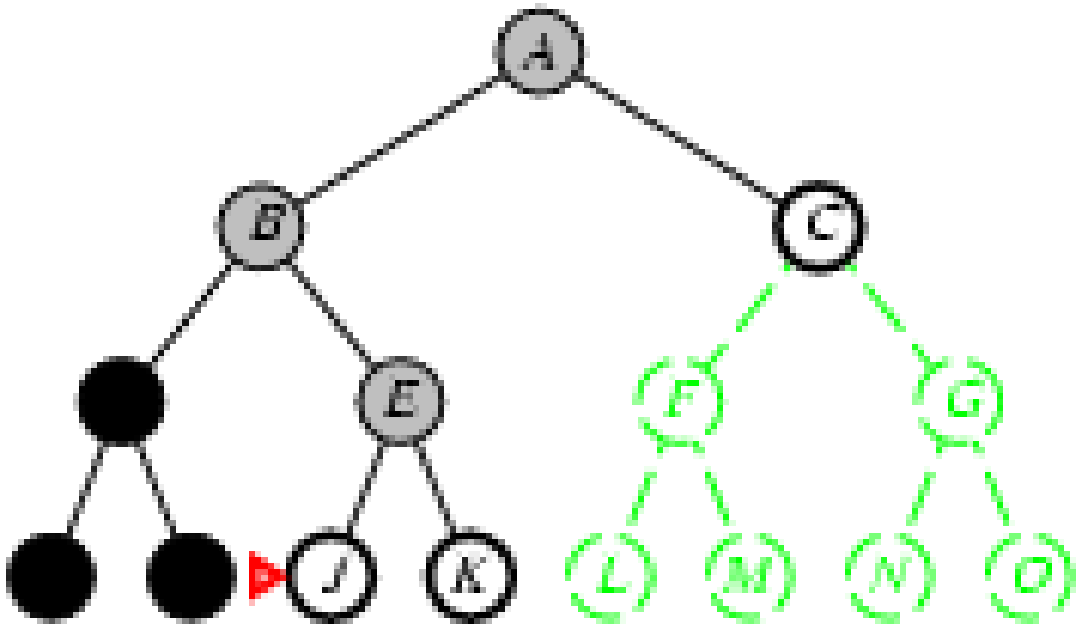
Depth-first search

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



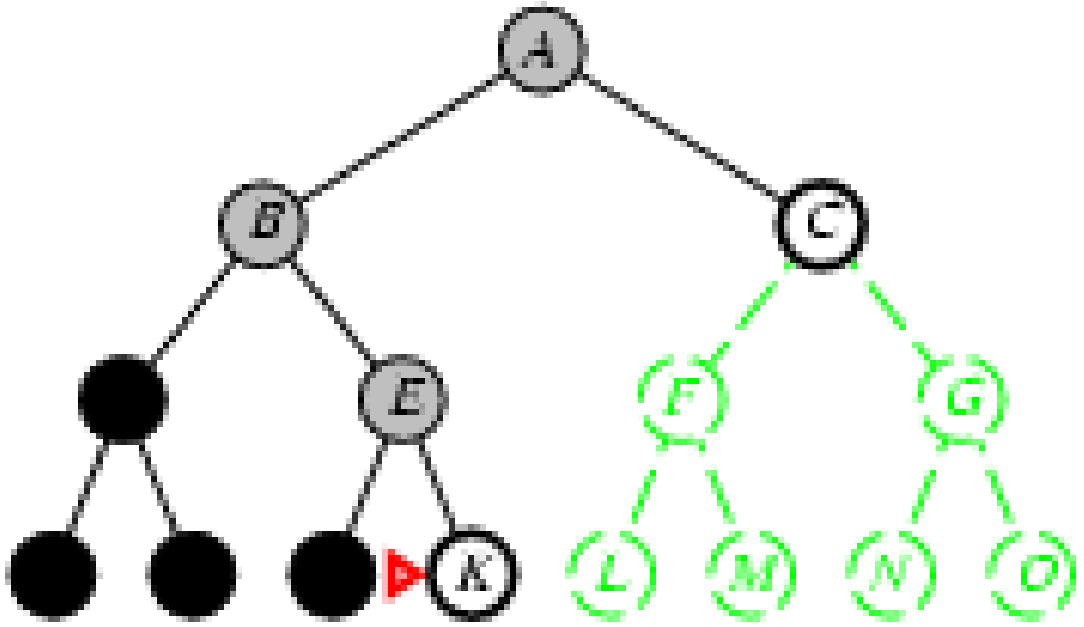
Depth-first search

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



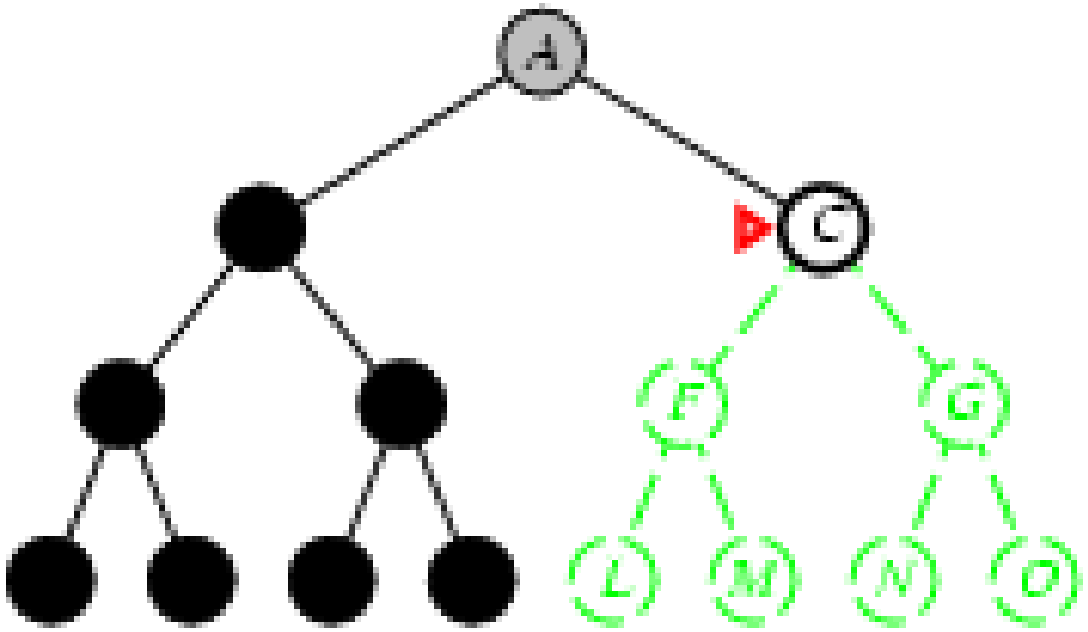
Depth-first search

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



Depth-first search

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



Depth-first search

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

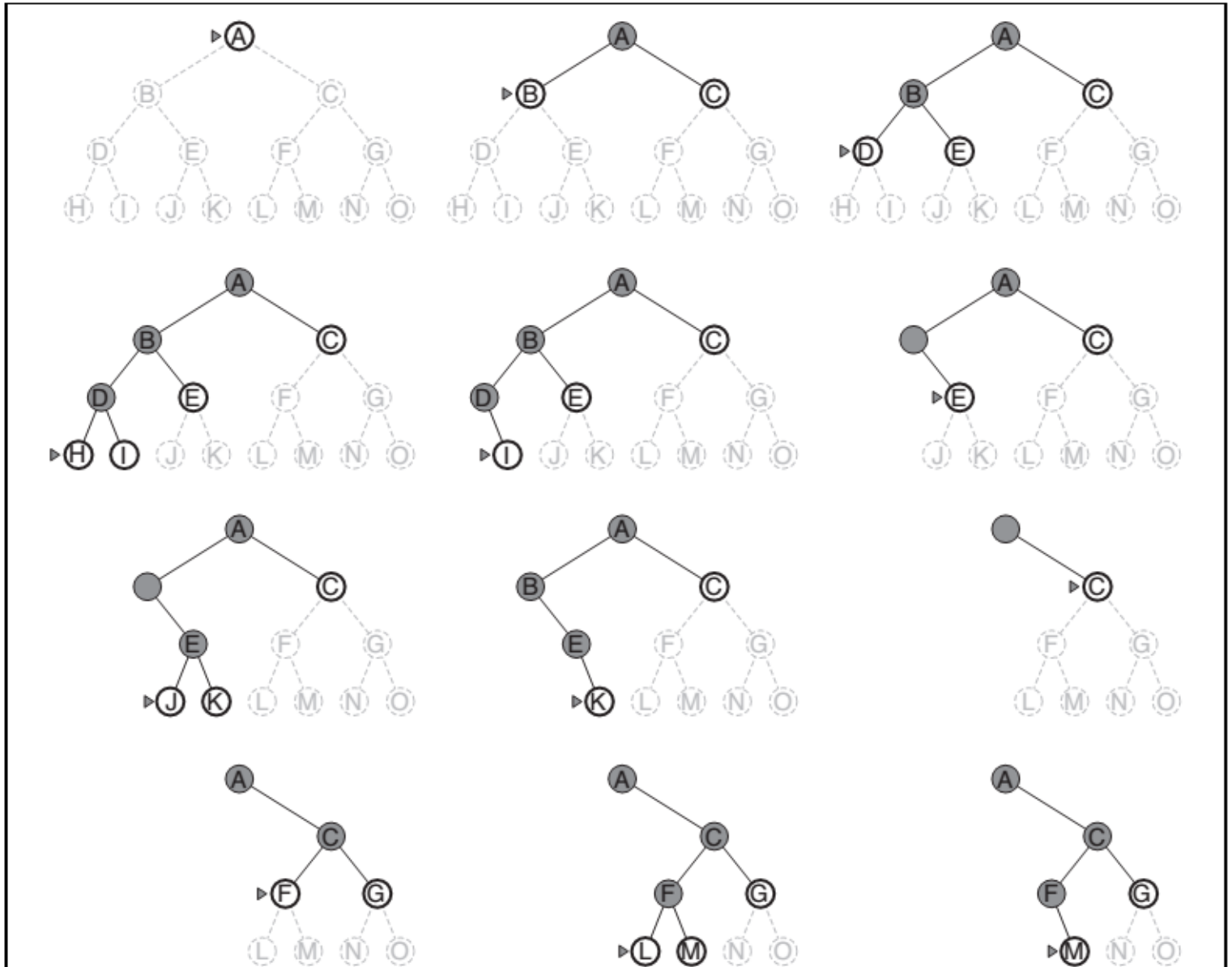


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    → complete in finite spaces
- Time? $O(b^m)$: terrible if m is much larger than d
- Space? $O(bm)$, i.e., linear space! i.e. once a node has been expanded, it can be removed from memory as soon as all its dependents have been explored.
- Optimal? No. If C and J are both goal nodes, DFS will return J as solution instead of C

Depth-limited search

- Equal to DFS with depth limit *l*, i.e., nodes at depth *l* have no successors

- For most problems, we will not know a good depth limit until we solved the problem!

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

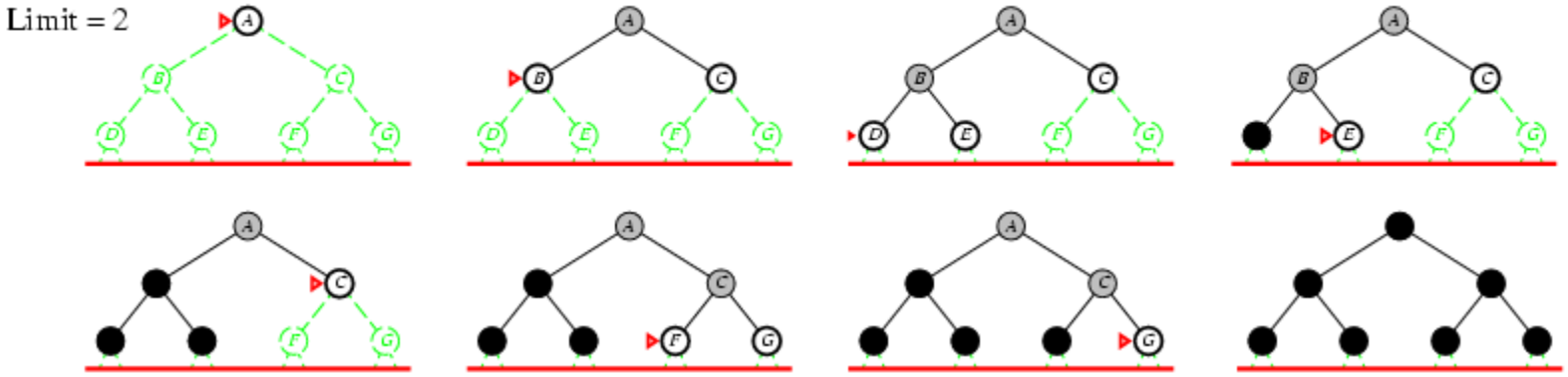
Figure 3.18 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

Iterative deepening search / =0

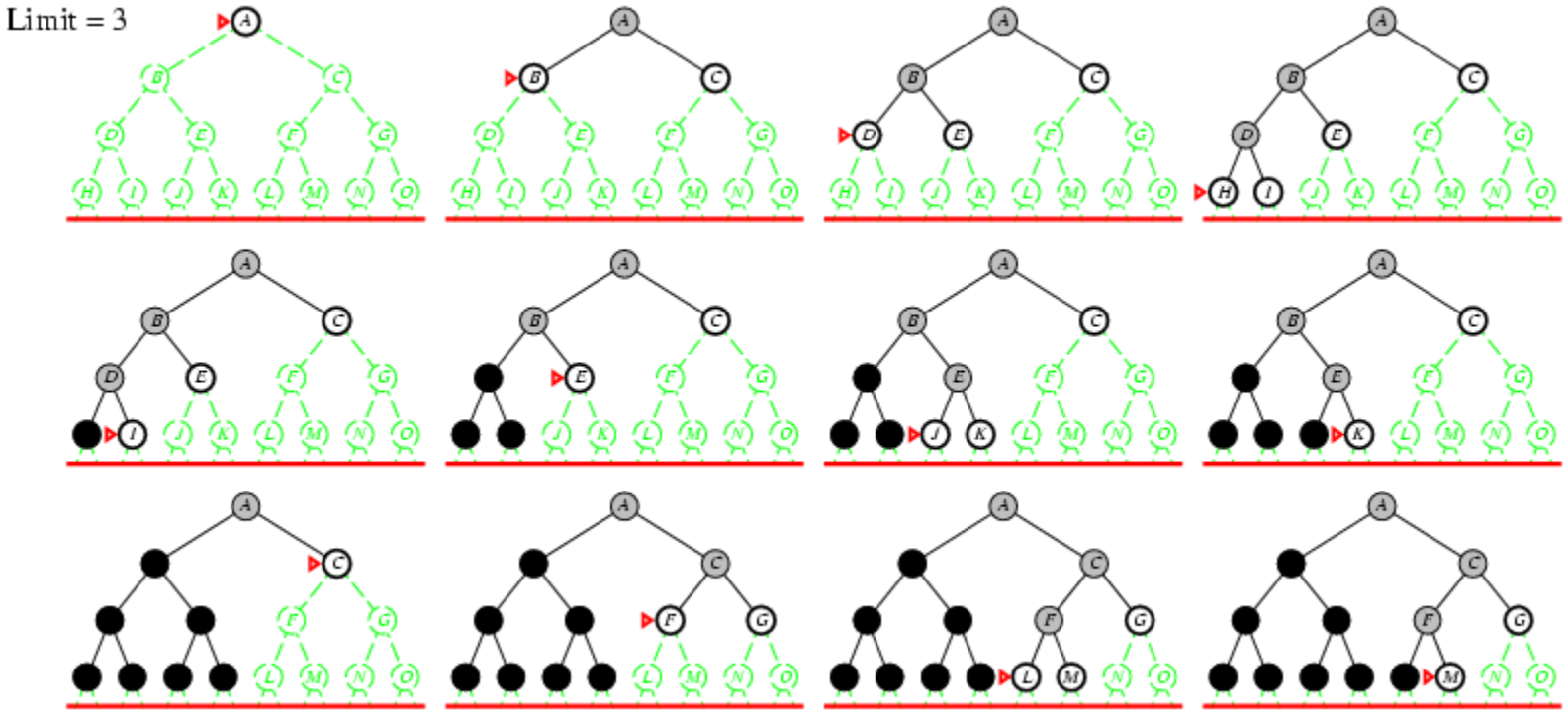
Limit = 0

Iterative deepening search / =1

Iterative deepening search l =2



Iterative deepening search / =3



Iterative deepening search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

$$N_{DLS} = b + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = (d)\ b + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For *b = 10*, *d = 5*,
-
  - $N_{DLS} = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}110$
  -
  - $N_{IDS} = 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}450$
  -

- Overhead = (123,450 - 111,110)/111,110 = 11%

Properties of iterative deepening search

- Complete? Yes
- Time? $d\,b + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step costs are equal

Bidirectional search

- Run two simultaneous searches – one forward from initial state and the other backward from goal state.
- Motivation: $b^{d/2} + b^{d/2} << b^{d}$
- Implemented by replacing goal test with a check to see whether the frontiers of the two searches intersect.
- Time and space complexity is $O(b^{d/2})$ (if BFS is used in both direction)

Bidirectional search

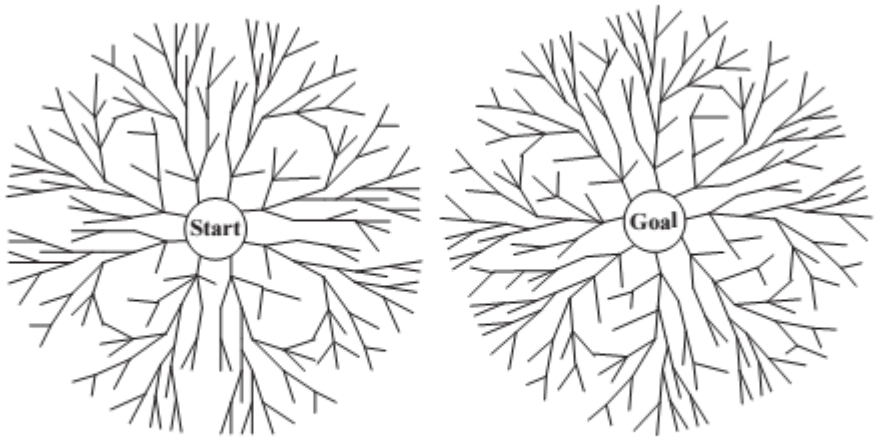


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

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	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
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.

Note: For graph searches, the main differences are that depth-first search is complete for finite state spaces and that the space and time complexities are bounded by the size of the state space

Next…

- INFORMED (HEURISTIC) SEARCH STRATEGIES
  - Greedy best-first search
  - A* search
  - Memory-bounded heuristic search
- Local search
  - Hill climbing
  - Simulated annealing
  - Genetic algorithms
- Stochastic search
  - Monte-Carlo algorithms