Search

Search

- Because there are many ways to achieve the same goal
  - Those ways are together expressed as a tree
  - Multiple options of unknown value at a point,
    - the agent can examine different possible sequences of actions, and choose the best
  - This process of looking for the best sequence is called ***search***
  - The best sequence is then a list of actions, called ***solution***

Search algorithm

- Defined as
  - taking a *problem*
  - and returns a *solution*
- Once a solution is found
  - the agent follows the solution
  - and carries out the list of actions – execution phase
- Design of an agent
  - "Formulate, search, execute"

Well-defined problems and solutions

A problem is defined by 5 components:

- ***Initial state***
- ***Actions***
- ***Transition model or***

(Successor functions)

- ***Goal Test.***
- ***Path Cost.***

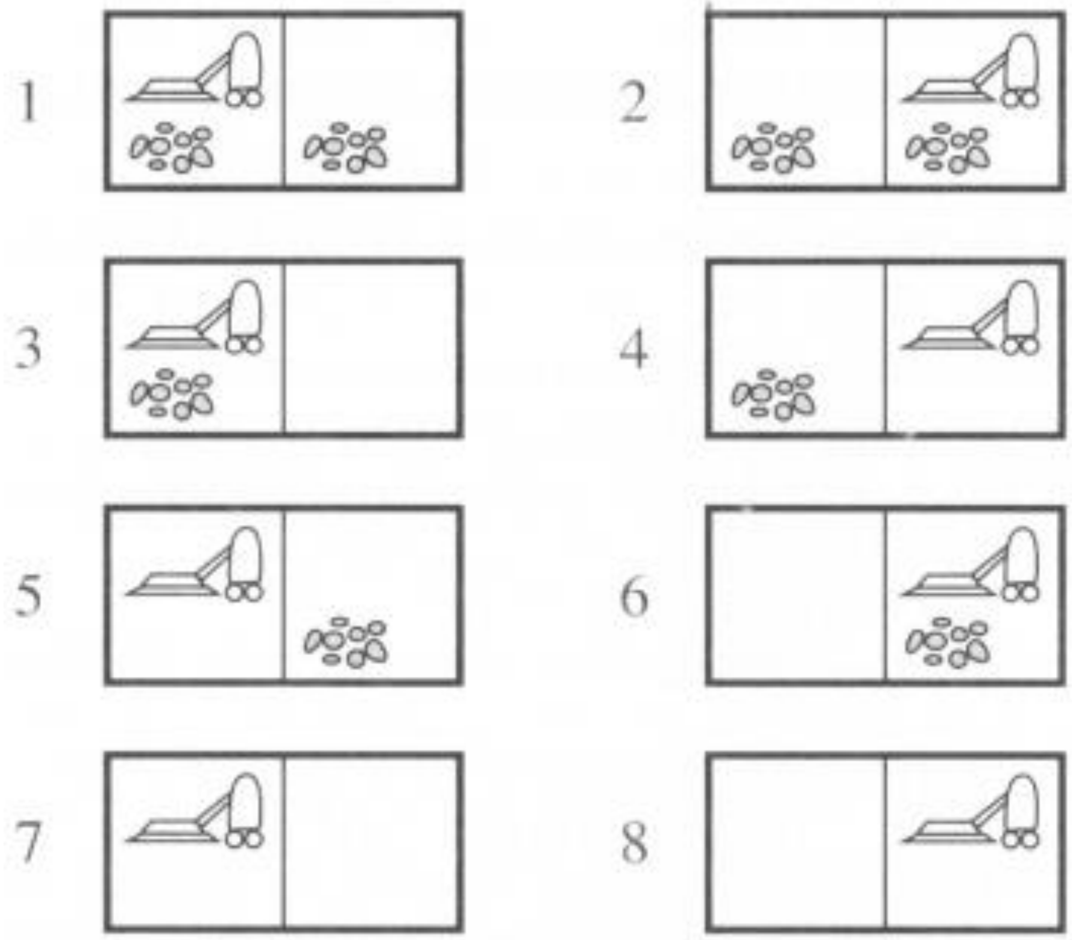
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  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
```

Example problems

- *Toy problems*
  - Those intended to illustrate or exercise various problem-solving methods
  - E.g., puzzle, chess, etc.
- *Real-world problems*
  - Tend to be more difficult and whose solutions people actually care about
  - E.g., Design, planning, Traveling salesperson problem (TSP), Touring problems (Visit every city at least once, starting and ending in Bucharest) etc.

Toy problems

- Example: vacuum world



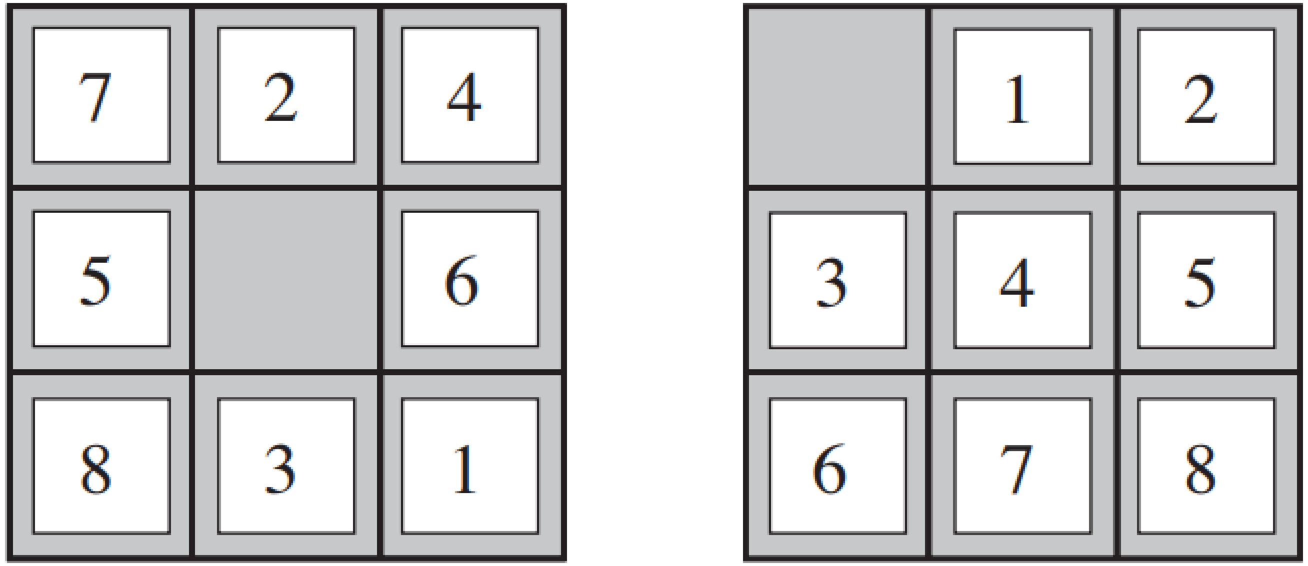
- Number of states: 8
- Initial state: Any
- Number of actions: 4
  - *left, right, suck, noOp*
- Goal: clean up all dirt
  - Goal states: {7, 8}
- Path Cost:
  - Each step costs 1

- Compared with the real world, this toy problem has discrete locations, discrete dirt, reliable cleaning, and it never gets any dirtier

The 8-puzzle

- **States**:
  - a state description specifies the location of each of the eight tiles and blank in one of the nine squares
- **Initial State**:
  - Any state in state space
- **Successor function**:
  - the blank moves *Left*, *Right*, *Up*, or *Down*
- **Goal test**:
  - current state matches the goal configuration
- **Path cost**:
  - each step costs 1, so the path cost is just the length of the path

- The 8-puzzle has 9!/2 = 181, 440 reachable states and is easily solved
- The 15-puzzle (on a 4×4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms
- The 24-puzzle (on a 5 × 5 board) has around 10^25 states, and random instances take several hours to solve optimally



Start State Goal State

The 8-queens

- There are two ways to formulate the problem
- All of them have the common followings:
    - Goal test: 8 queens on board, not attacking to each other
    - Path cost: zero

The 8-queens (Incremental formulation)

- States: Any arrangement of 0 to 8 queens on the board is a state
- Initial state: No queens on the board
- Actions: Add a queen to any empty square
- Transition model: Returns the board with a queen added to the specified square
- Goal test: 8 queens are on the board, none attacked
- In this formulation, we have $64 \cdot 63 \cdots 57 \approx 1.8 \times 10^{14}$ possible sequences to investigate
- States: All possible arrangements of n queens ($0 \le n \le 8$), one per column in the leftmost n columns, with no queen attacking another
- Actions: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen

The 8-queens (Complete-state formulation)

- starts with all 8 queens on the board
- move the queens individually around
- States:
  - any arrangement of 8 queens, one per column in the leftmost columns
- Operators: move an attacked queen to a row, not attacked by any other

- This formulation reduces the 8-queens state space from 1.8×10^{14} to just 2,057, and solutions are easy to find
- On the other hand, for 100 queens the reduction is from roughly 10^{400} states to about 10^{52} states
- A big improvement, but not enough to make the problem tractable

Searching for solutions

Searching for solutions

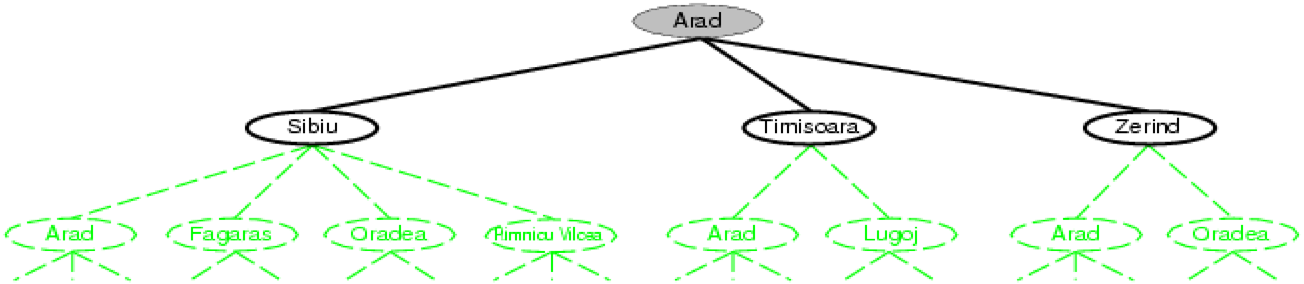
- Finding out a solution is done by
  - searching through the state space
- All problems are transformed
  - as a search tree
  - generated by the initial state and successor function

Search tree

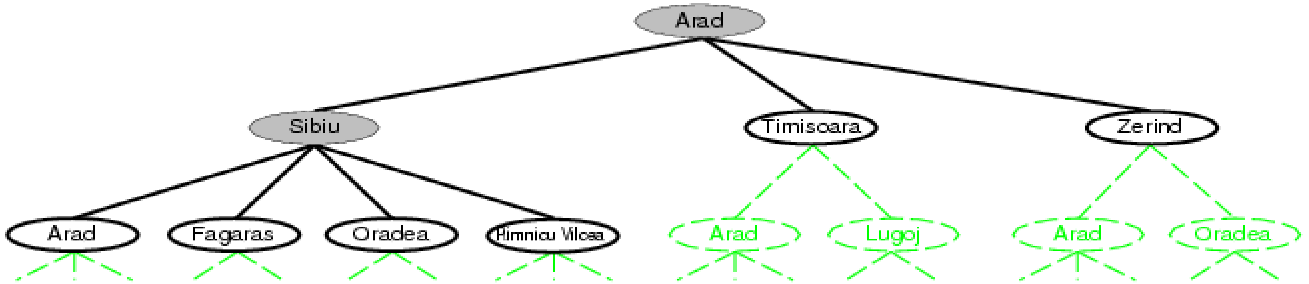
- **Initial state**
  - The root of the search tree is a **search node**
- **Expanding**
  - applying successor function to the current state
  - thereby generating a new set of states
- **leaf nodes**
  - the states having no successors

  Fringe : Set of search nodes that have not been expanded yet.
- Refer to next figure

Tree search example



Tree search example



Search tree

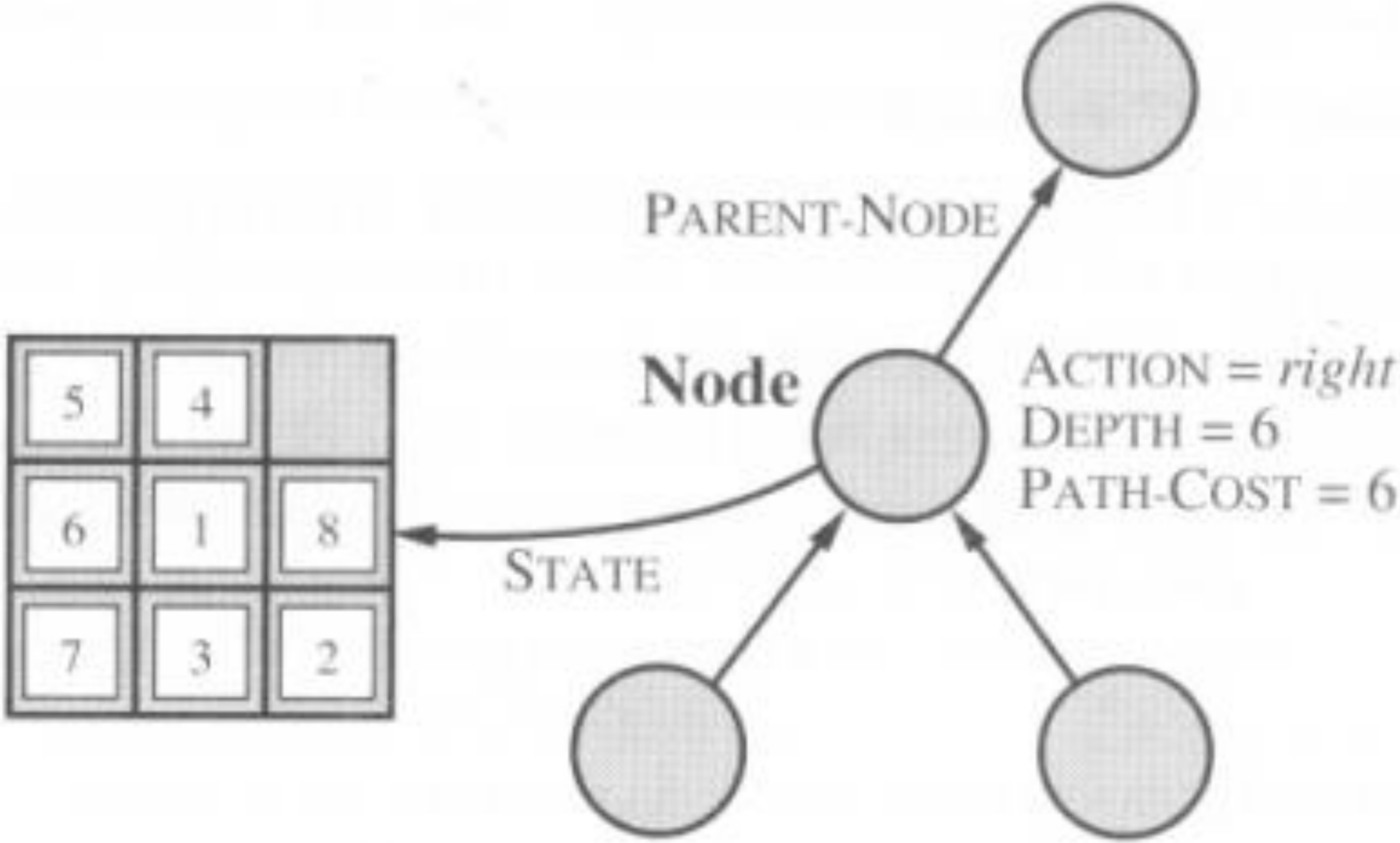
- The **essence** of searching
  - in case the first choice is not correct
  - choosing one option and keep others for later inspection
- Hence, we have the **search strategy**
  - which determines the choice of which state to expand
  - good choice → fewer work → faster
- Important:
  - state space ≠ search tree

Search tree

- State space
  - has unique states {A, B}
  - while a search tree may have cyclic paths: A-B-A-B-A-B- …
- A good search strategy should avoid such paths

Search tree

- A node is having five components:
  - STATE: which state it is in the state space
  - PARENT-NODE: from which node it is generated
  - ACTION: which action applied to its parent-node to generate it
  - PATH-COST: the cost, g(n), from initial state to the node n itself
  - DEPTH: number of steps along the path from the initial state



PARENT-NODE
Node
ACTION = right
DEPTH = 6
PATH-COST = 6
STATE
5
4
6
1
8
7
3
2

Measuring problem-solving performance

- The evaluation of a search strategy
  - **Completeness:**
    - is the strategy guaranteed to find a solution when there is one?
  - **Optimality**:
    - does the strategy find the highest-quality solution when there are several different solutions?
  - **Time complexity**:
    - how long does it take to find a solution?
  - **Space complexity**:
    - how much memory is needed to perform the search?

Measuring problem-solving performance

- In AI, complexity is expressed in
  - **b**, branching factor, 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
- Time and Space is measured in
  - number of nodes generated during the search
  - maximum number of nodes stored in memory

Measuring problem-solving performance

- For effectiveness of a search algorithm
  - we can just consider the total cost
  - **The total cost** = path cost (g) of the solution found **+** search cost
    - search cost = time necessary to find the solution
- Tradeoff:
  - (long time, optimal solution with least g)
  - vs. (shorter time, solution with slightly larger path cost g)

Uninformed search strategies

Uninformed search strategies

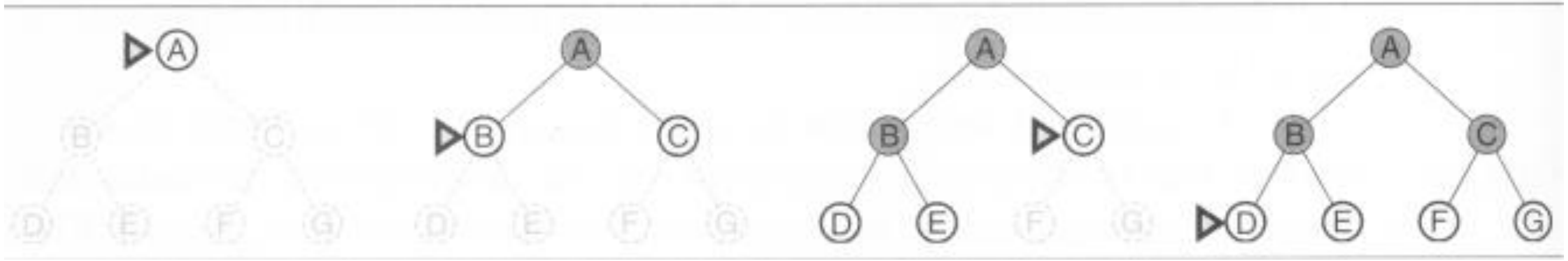
- **Uninformed search**
  - no information about the number of steps
  - or the path cost from the current state to the goal
  - search the state space **blindly**
- **Informed search, or heuristic search**
  - a cleverer strategy that searches toward the goal,
  - based on the information from the current state so far

Uninformed search strategies

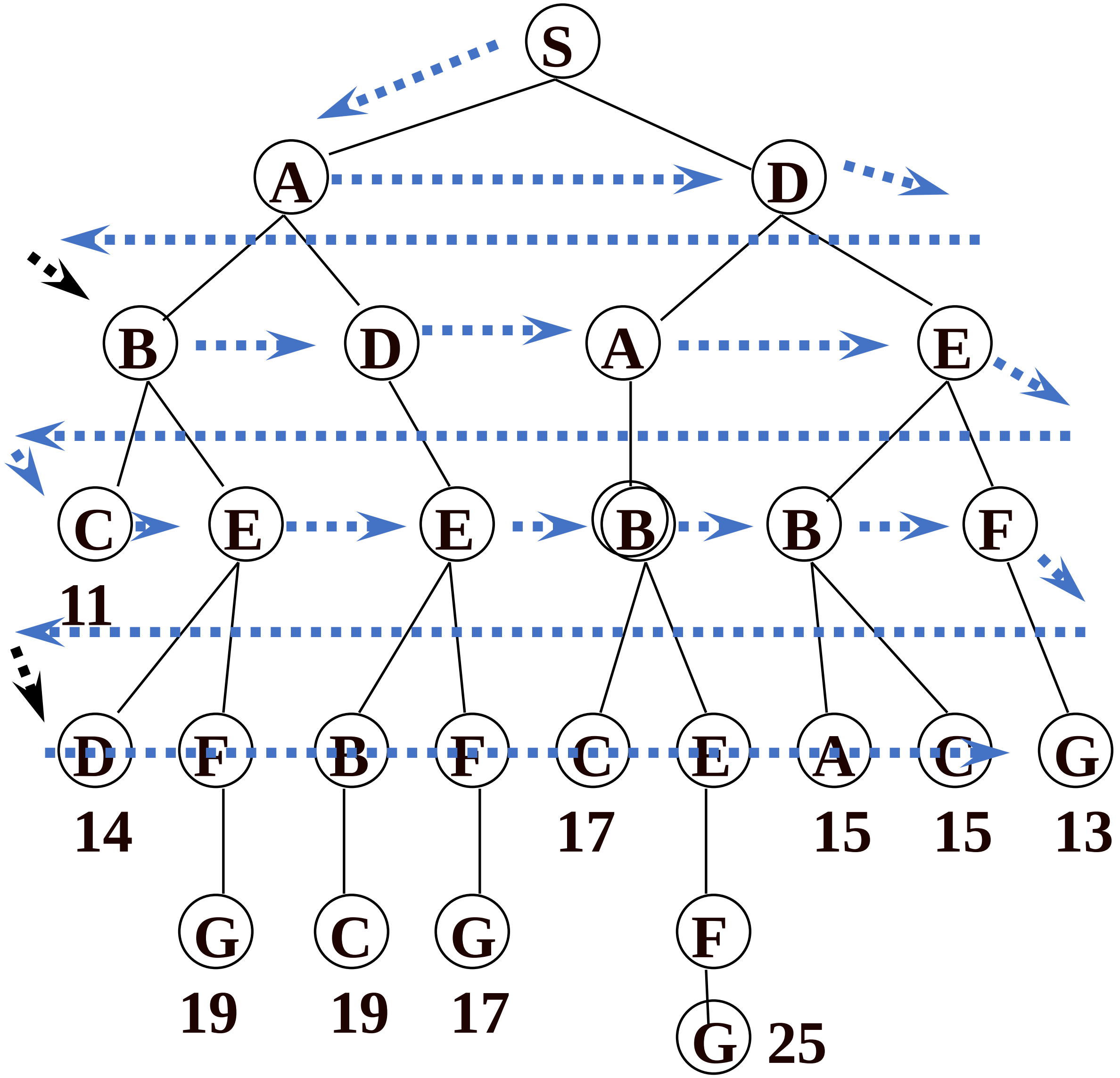
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening depth-first search
- Bidirectional search

Breadth-first search

- The root node is expanded first (FIFO)
- All the nodes generated by the root node are then expanded
- And then their successors and so on

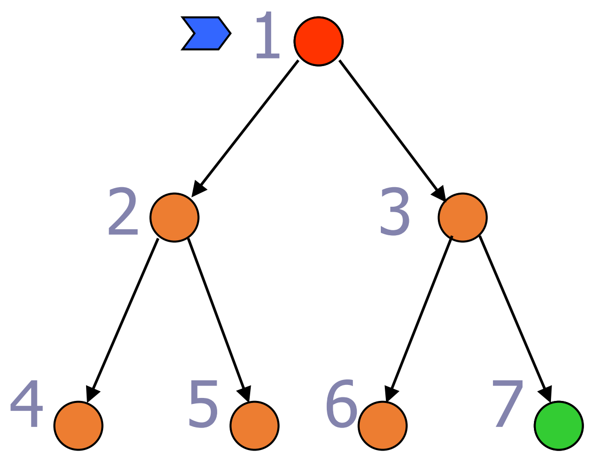


Breadth-first search



Breadth-First Strategy

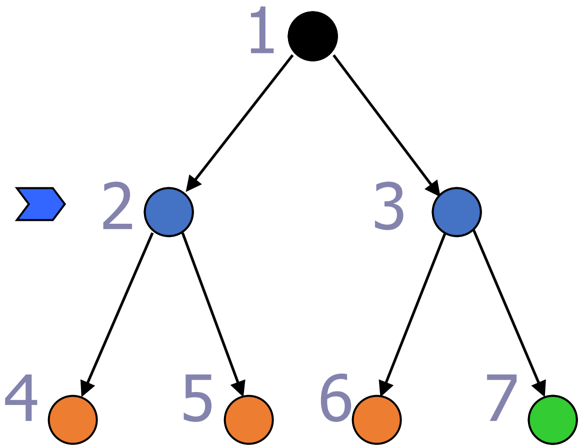
New nodes are inserted at the end of FRINGE



FRINGE = (1)

Breadth-First Strategy

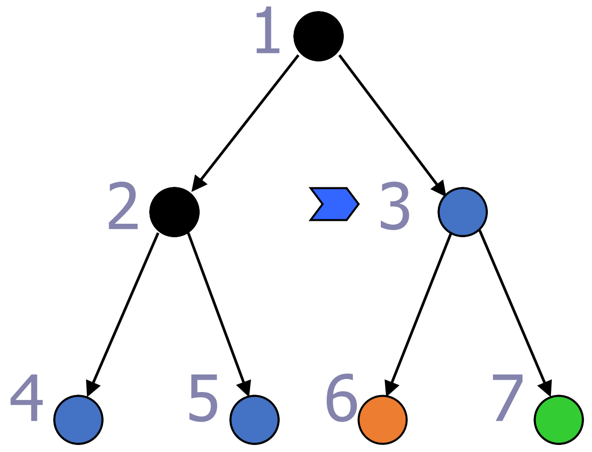
New nodes are inserted at the end of FRINGE



FRINGE = (2, 3)

Breadth-First Strategy

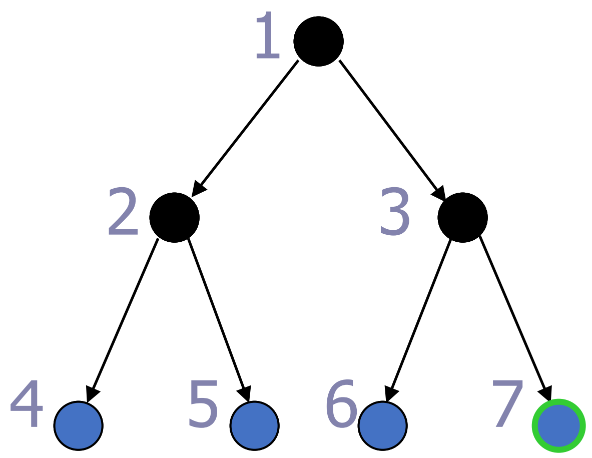
New nodes are inserted at the end of FRINGE



FRINGE = (3, 4, 5)

Breadth-First Strategy

New nodes are inserted at the end of FRINGE



FRINGE = (4, 5, 6, 7)

Breadth-first search (Analysis)

- Breadth-first search
  - Complete – find the solution eventually
  - Optimal, if step cost is 1 (breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node. The most common such scenario is that all actions have the same cost.)

  The disadvantage

  - if the branching factor of a node is large,
  - for even small instances (e.g., chess)
    - the ***space complexity*** and the ***time complexity*** are enormous

Time and memory requirements for breadth-first search

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

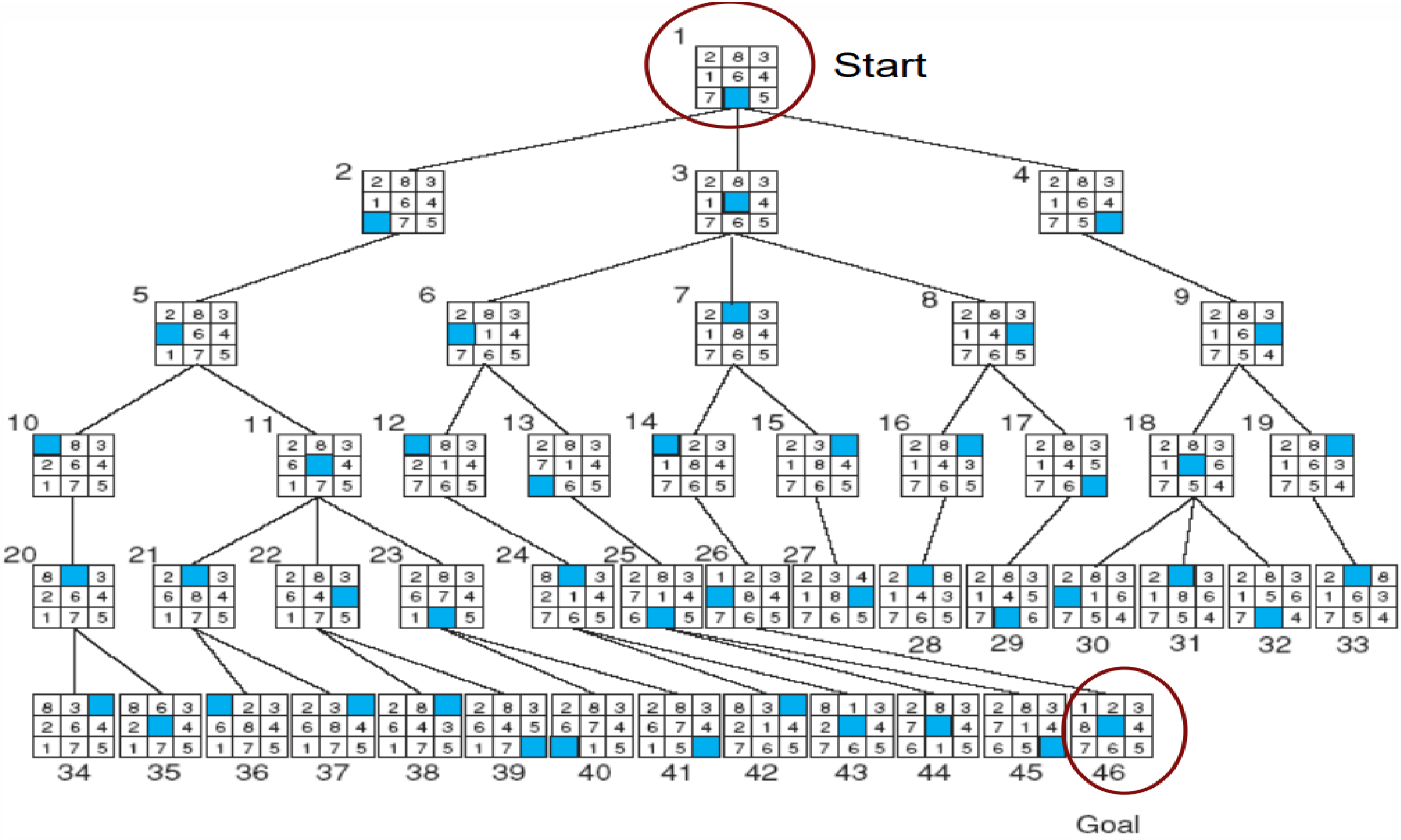
Note: The numbers shown assume branching factor b = 10; 1 million nodes/second; 1000 bytes/node

Properties of breadth-first search

- Complete? Yes (if b is finite)
- Time? $1+b+b^2+b^3+\ldots +b^d = b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)
- Space is the bigger problem (more than time)

Learning

- First, the memory requirements are a bigger problem for breadth-first search than is the execution time
- One might wait 13 days for the solution to an important problem with search depth 12, but no personal computer has the petabyte of memory it would take
- Second, time is still a major factor
- If your problem has a solution at depth 16, then (given our assumptions) it will take about 350 years for breadth-first search (or indeed any uninformed search) to find it
- In general, exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances

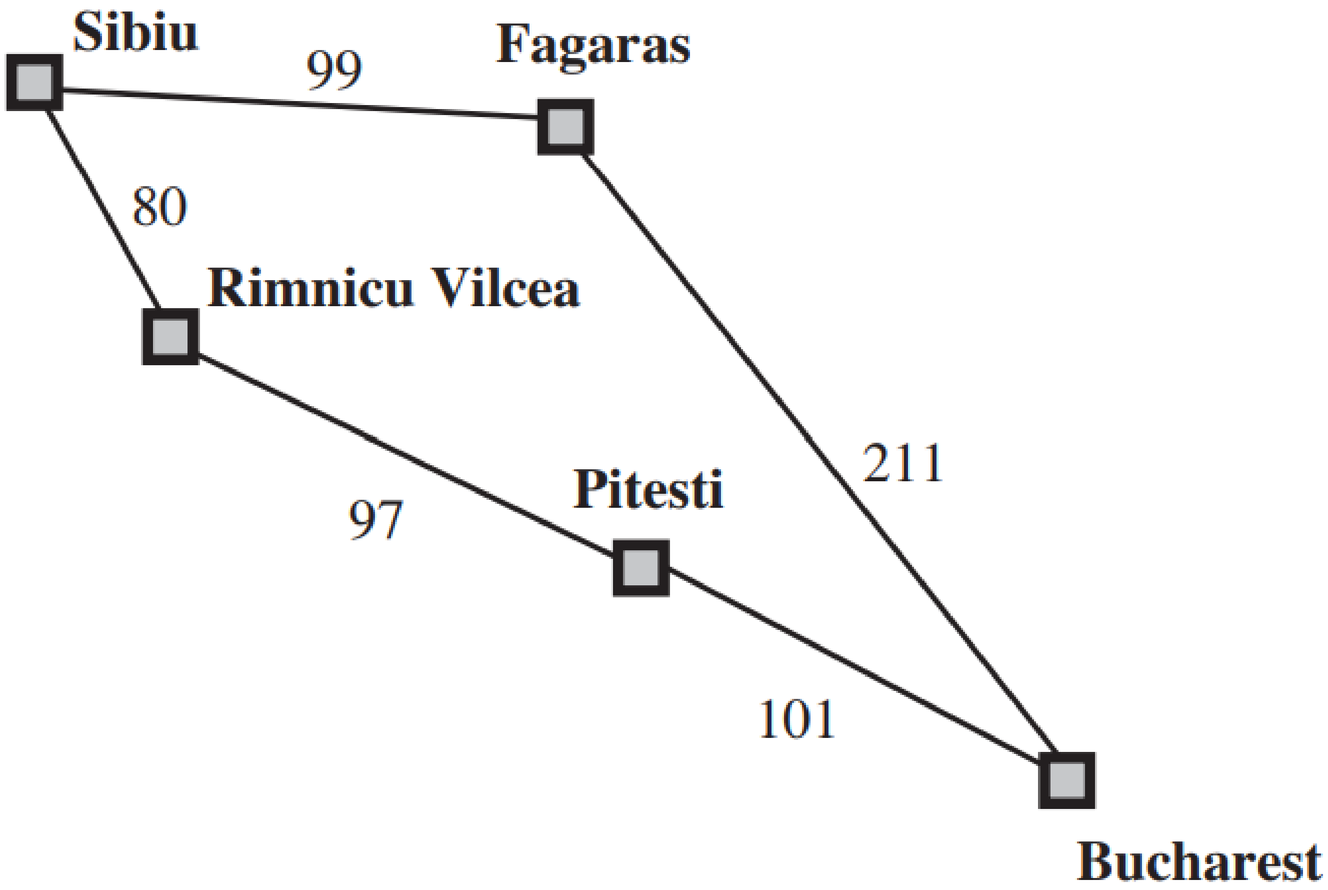

Start
Goal

Uniform-cost search

Uniform-cost search

- When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node
- Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost g(n)
- This is done by storing the frontier as a priority queue ordered by g
- The goal test is applied to a node when it is selected for expansion rather than when it is first generated (difference with BFS)
- A test is added in case a better path is found to a node currently on the frontier (difference with BFS)

Part of the Romania state space, selected to illustrate uniform-cost search



The problem is to get from Sibiu to Bucharest

- The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively
- The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97 = 177
- The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99 + 211 = 310
- Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+ 97+ 101 = 278
- Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded
- Bucharest, now with g-cost 278, is selected for expansion and the solution is returned

Time & space complexity

- Then the algorithm's worst-case time and space complexity is $O(b^{(1+C*/ε)})$, which can be much greater than b^d
- Where $C*$ is the cost of the optimal solution and assume that every action costs at least $ε$
- When all step costs are equal, $b^{(1+C*/ε)}$ is just b^{d+1}

Learning

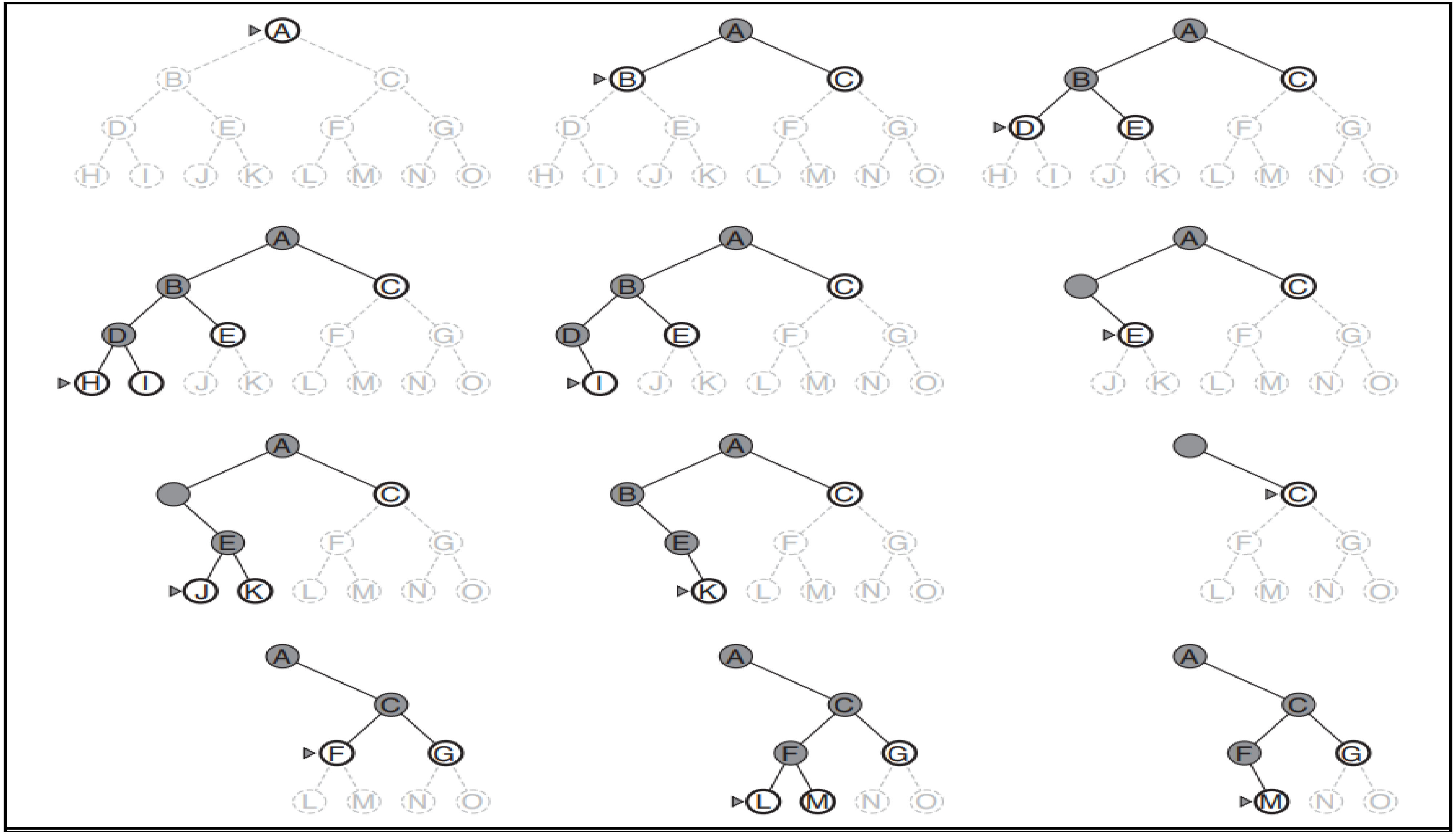
- It is easy to see that uniform-cost search is optimal in general
- Uniform-cost search does not care about the number of steps a path has, but only about their total cost
- Therefore, it will get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions—for example, a sequence of NoOp actions
- Completeness is guaranteed provided the cost of every step exceeds some small positive constant
- When all step costs are the same, uniform-cost search is like breadth-first search, except that the latter stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal's depth to see if one has a lower cost

Depth-first Search

Depth-first search

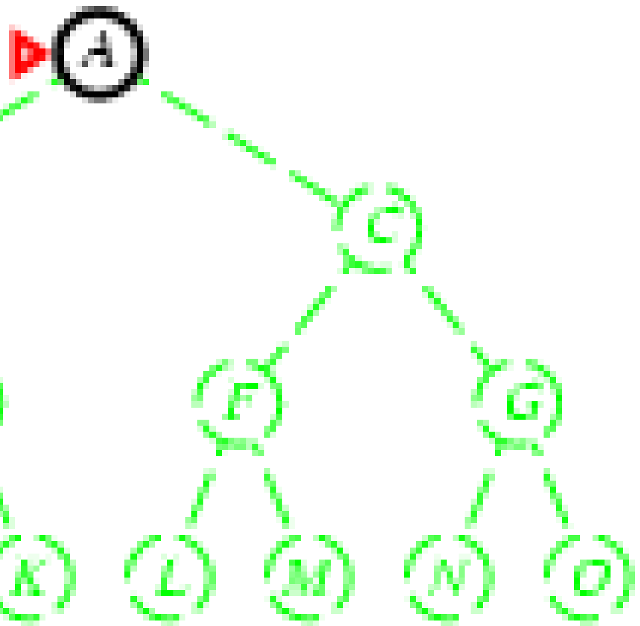
- Always expands one of the nodes at the ***deepest*** level of the tree
- Only when the search hits a dead end
  - goes back and expands nodes at shallower levels
  - Dead end → leaf nodes but not the goal
- Backtracking search
  - only one successor is generated on expansion
  - rather than all successors
  - fewer memory

Depth-first search on a binary tree M is the only goal node



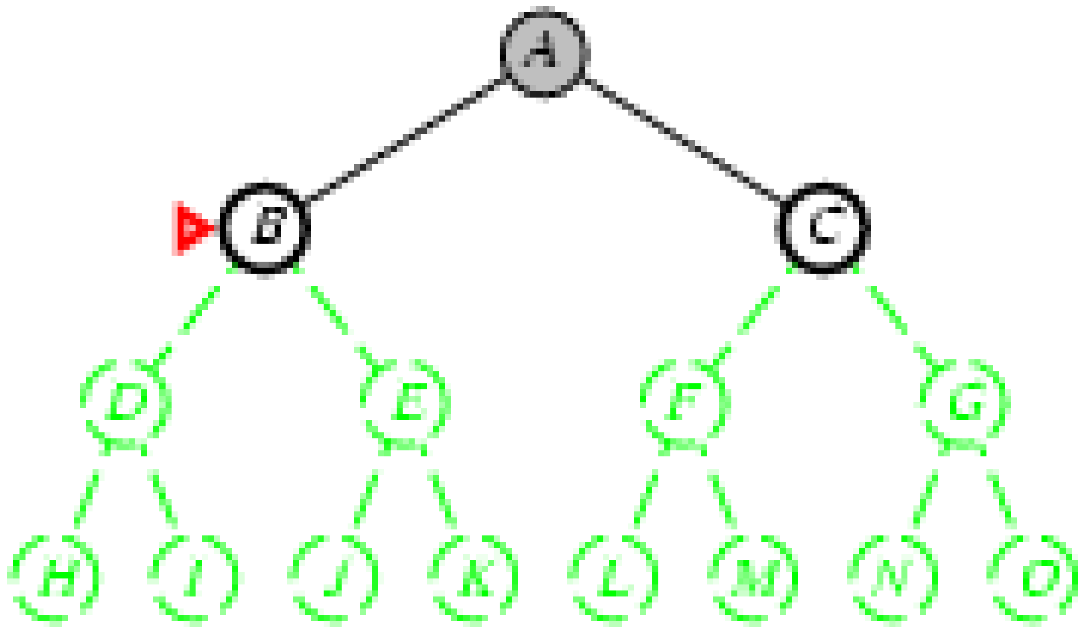
Depth-first search

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



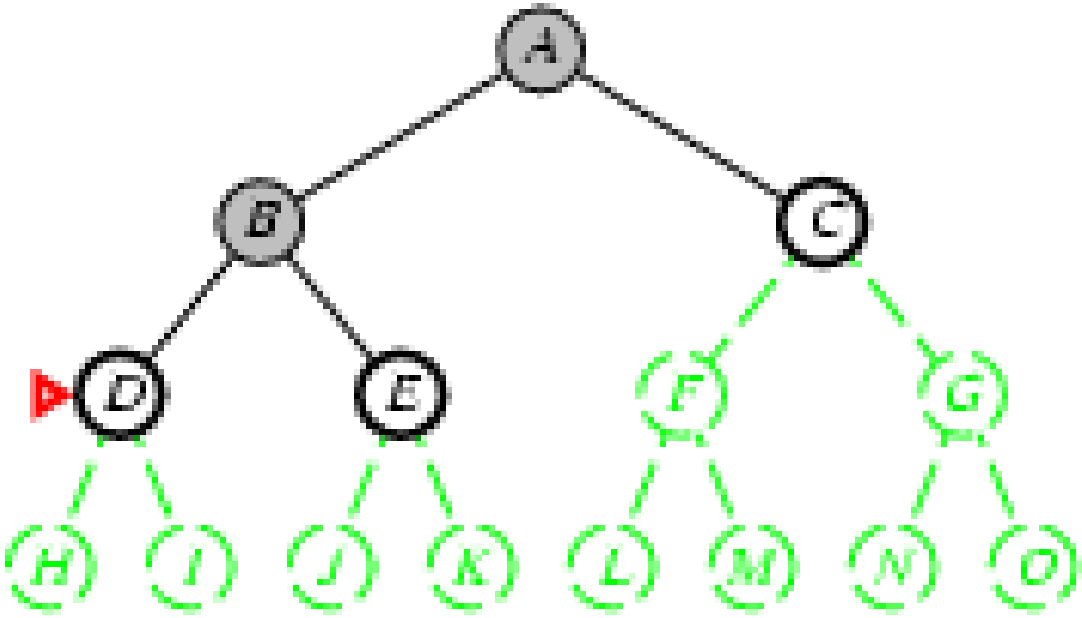
Depth-first search

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



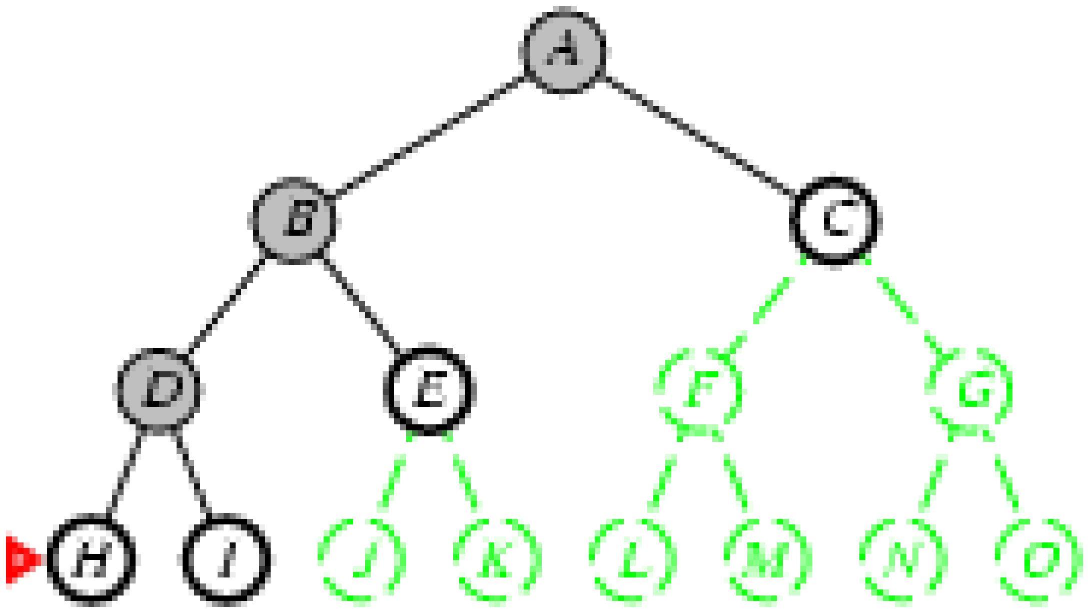
Depth-first search

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



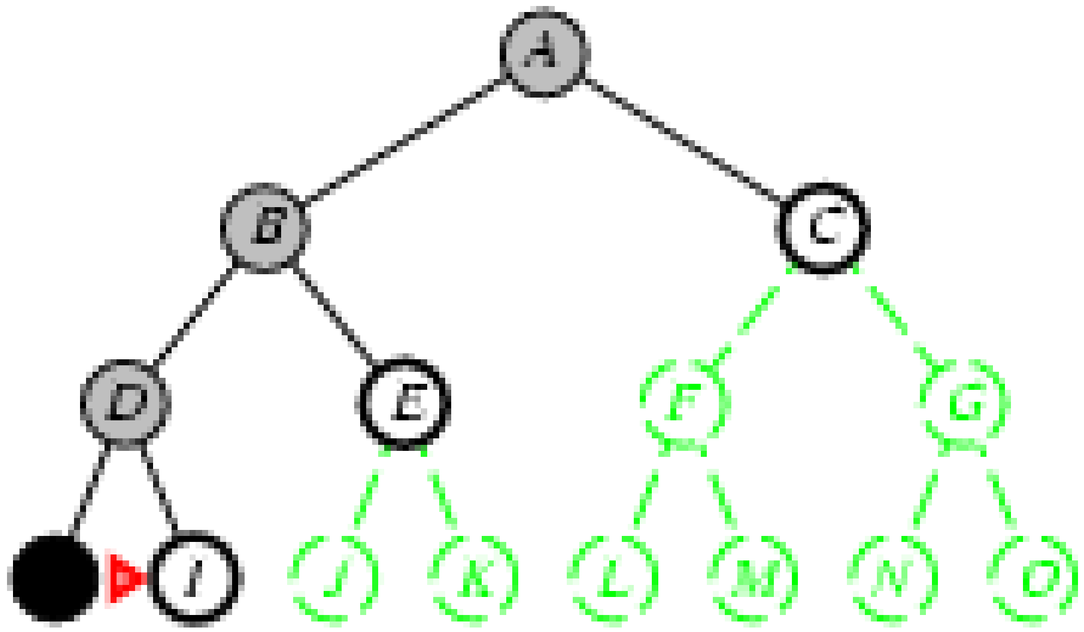
Depth-first search

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



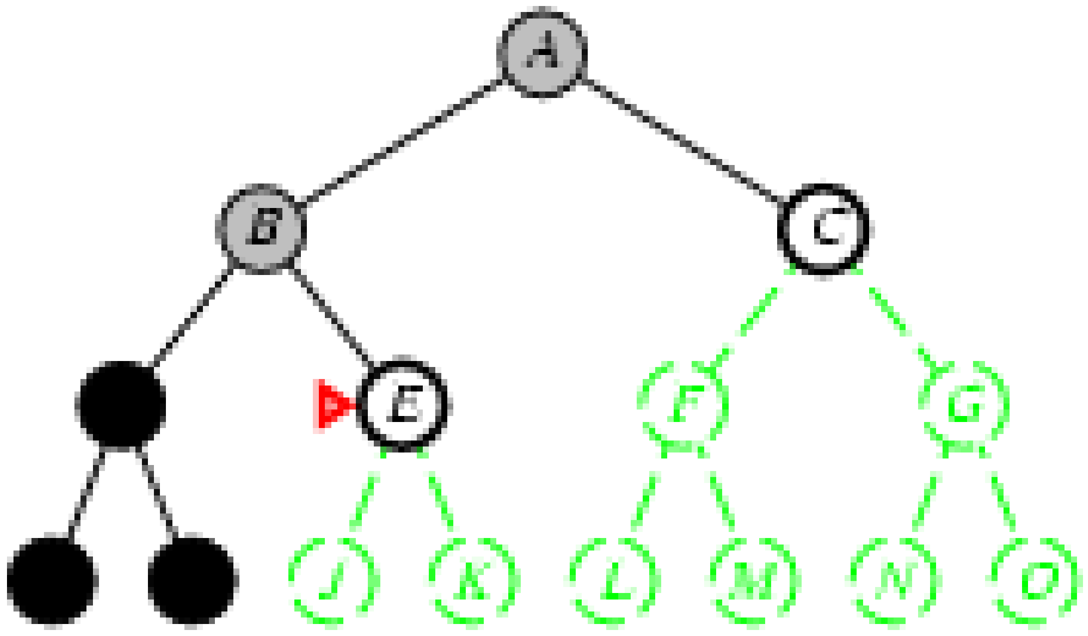
Depth-first search

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



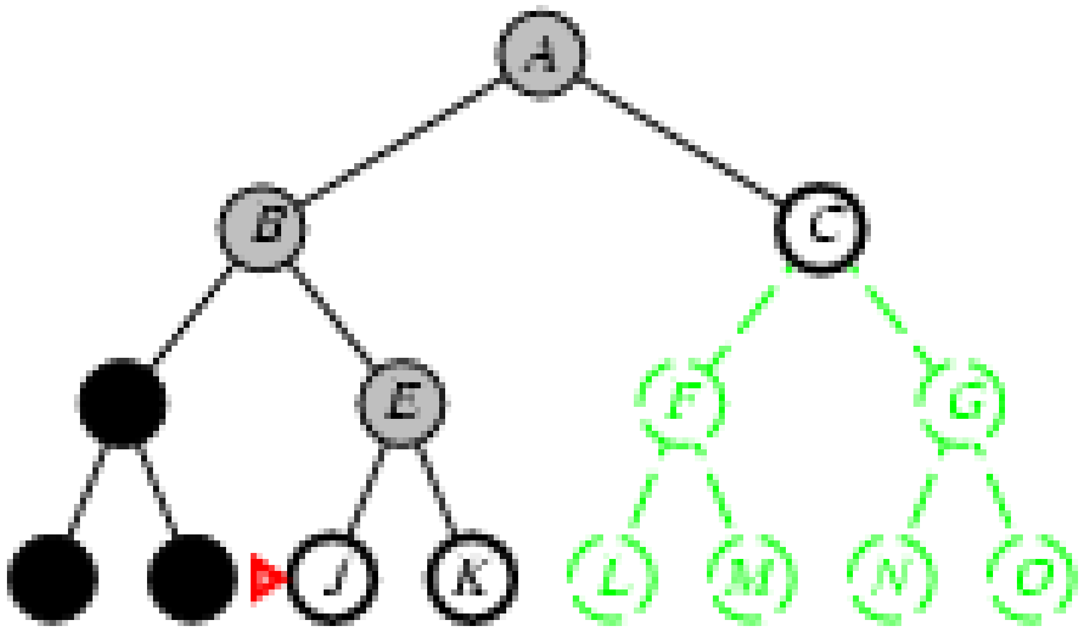
Depth-first search

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



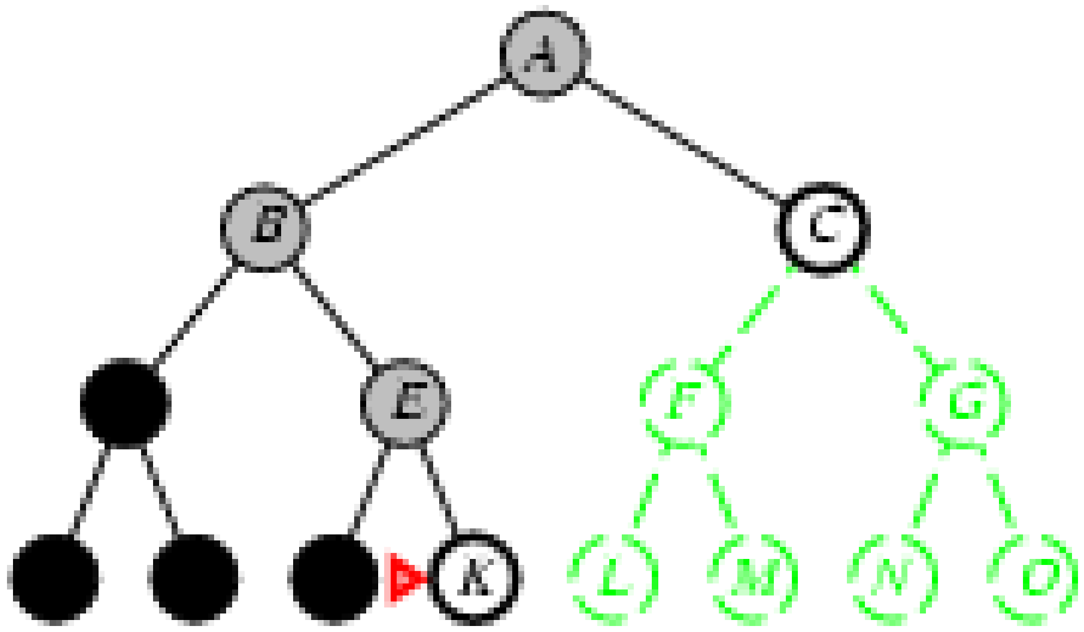
Depth-first search

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



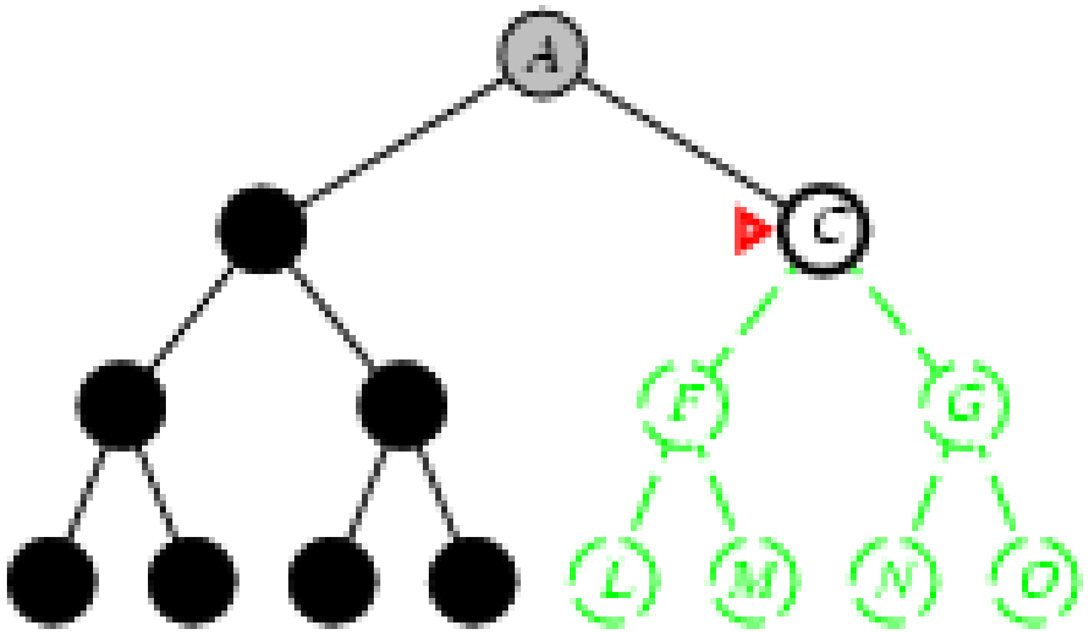
Depth-first search

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



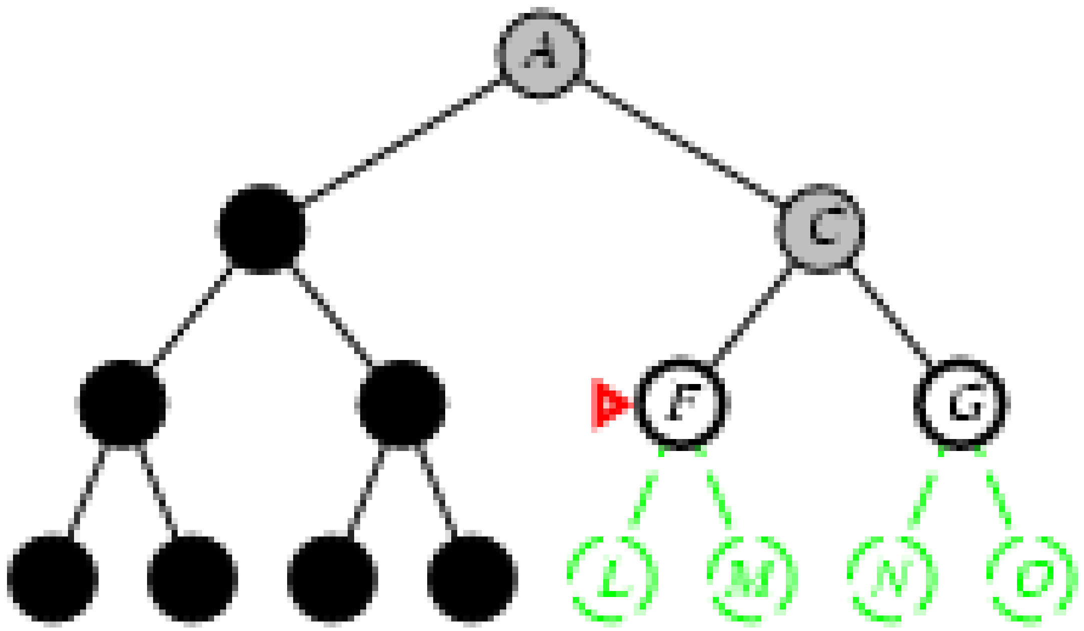
Depth-first search

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



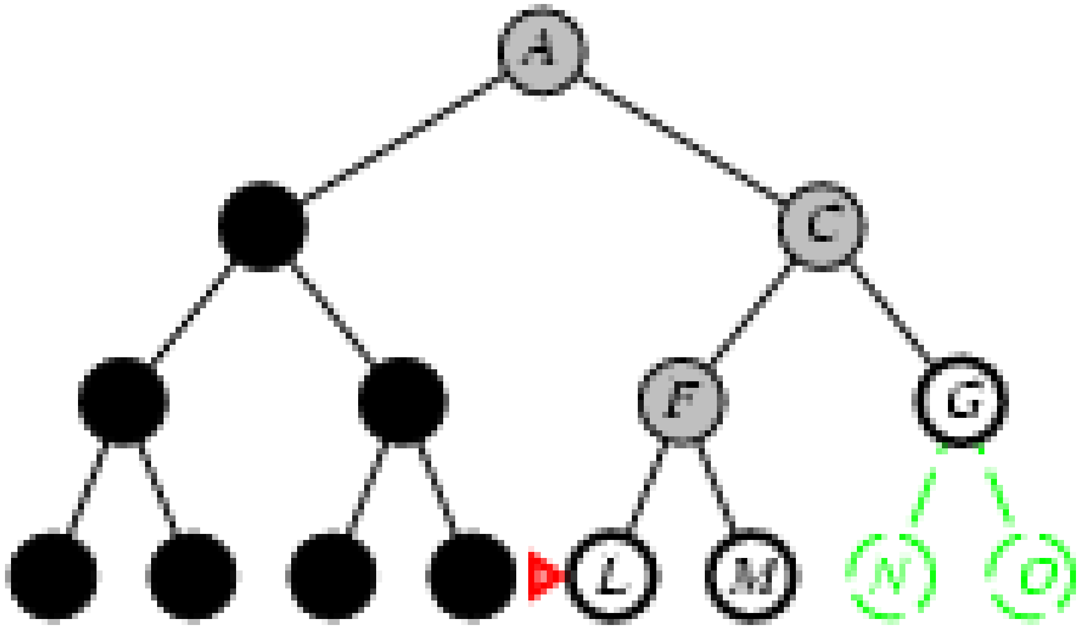
Depth-first search

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

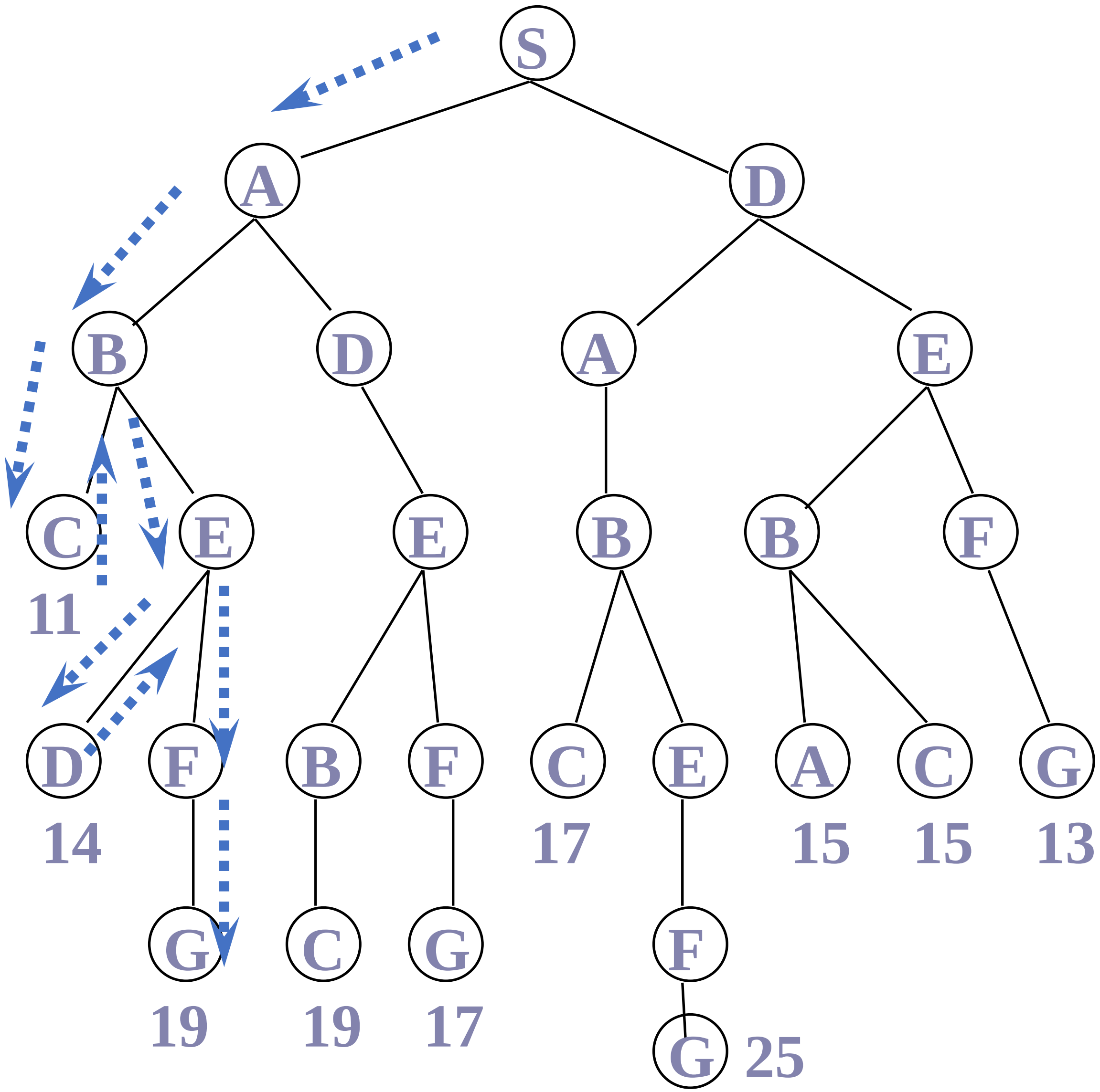


Depth-first search

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



Depth-first search



Discussions

- The properties of depth-first search depend strongly on whether the graph-search or tree-search version is used
- The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will eventually expand every node
- The tree-search version, on the other hand, is not complete
- Depth-first tree search can be modified at no extra memory cost so that it checks new states against those on the path from the root to the current node
- This avoids infinite loops in finite state spaces
- In infinite state spaces, both versions fail if an infinite non-goal path is encountered
- For similar reasons, both versions are nonoptimal

Non optimality: A case

- DFS will explore the entire left subtree even if node C is a goal node
- If node J were also a goal node, then depth-first search would return it as a solution instead of C, which would be a better solution; hence, depth-first search is not optimal

Time & space complexity

- Time complexity: A depth-first tree search, may generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node
- Space complexity: For a state space with branching factor b and maximum depth m, depth-first search requires storage of only O(bm) nodes

Advantages of DFS

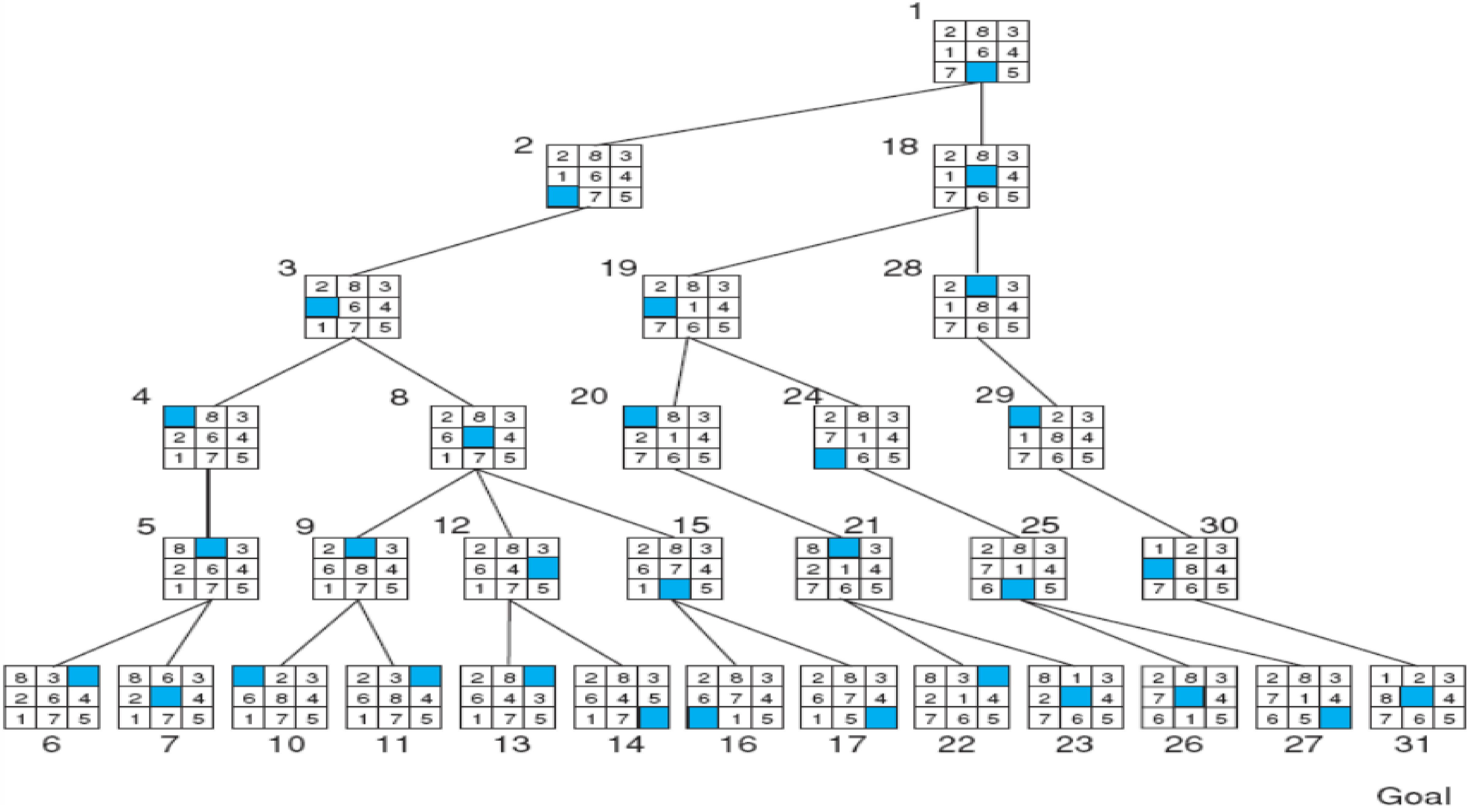
- For a graph search, there is no advantage
- But a depth-first tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path
- Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored
- For a state space with branching factor b and maximum depth m, depth-first search requires storage of only O(bm) nodes
- Using the same assumptions as in the slide 38 above and assuming that nodes at the same depth as the goal node have no successors, we find that depth-first search would require 156 kilobytes instead of 10 exabytes at depth d = 16, a factor of 7 trillion times less space
- For the same reason, depth-first tree search is adopted as the basic workhorse of many areas of AI, including constraint satisfaction, propositional satisfiability, and logic programming

Depth-first search (Analysis)

- Not complete
  - because a path may be infinite or looping
  - then the path will never fail and go back try another option
- Not optimal
  - it doesn't guarantee the best solution
- It overcomes
  - the time and space complexities

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
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space!
- Optimal? No


1
2
18
3
19
28
4
8
20
24
29
5
9
12
15
21
25
30
6
7
10
11
13
14
16
17
22
23
26
27
31
Goal

Depth-limited search

Introduction

- Depth-first search fails in the case of infinite state spaces
- This can be alleviated by supplying depth-first search with a predetermined depth limit L
- That is, nodes at depth L are treated as if they have no successors
- This approach is called depth-limited search
- The depth limit solves the infinite-path problem

- Depth-limited search will also be nonoptimal if we choose L>d
- Its time complexity is O(b^L) and its space complexity is O(bL)
- Depth-first search can be viewed as a special case of depth-limited search with L=∞
- Notice that depth-limited search can terminate with two kinds of failure: the standard failure value indicates no solution; the cutoff value indicates no solution within the depth limit

How to assign the best depth limit

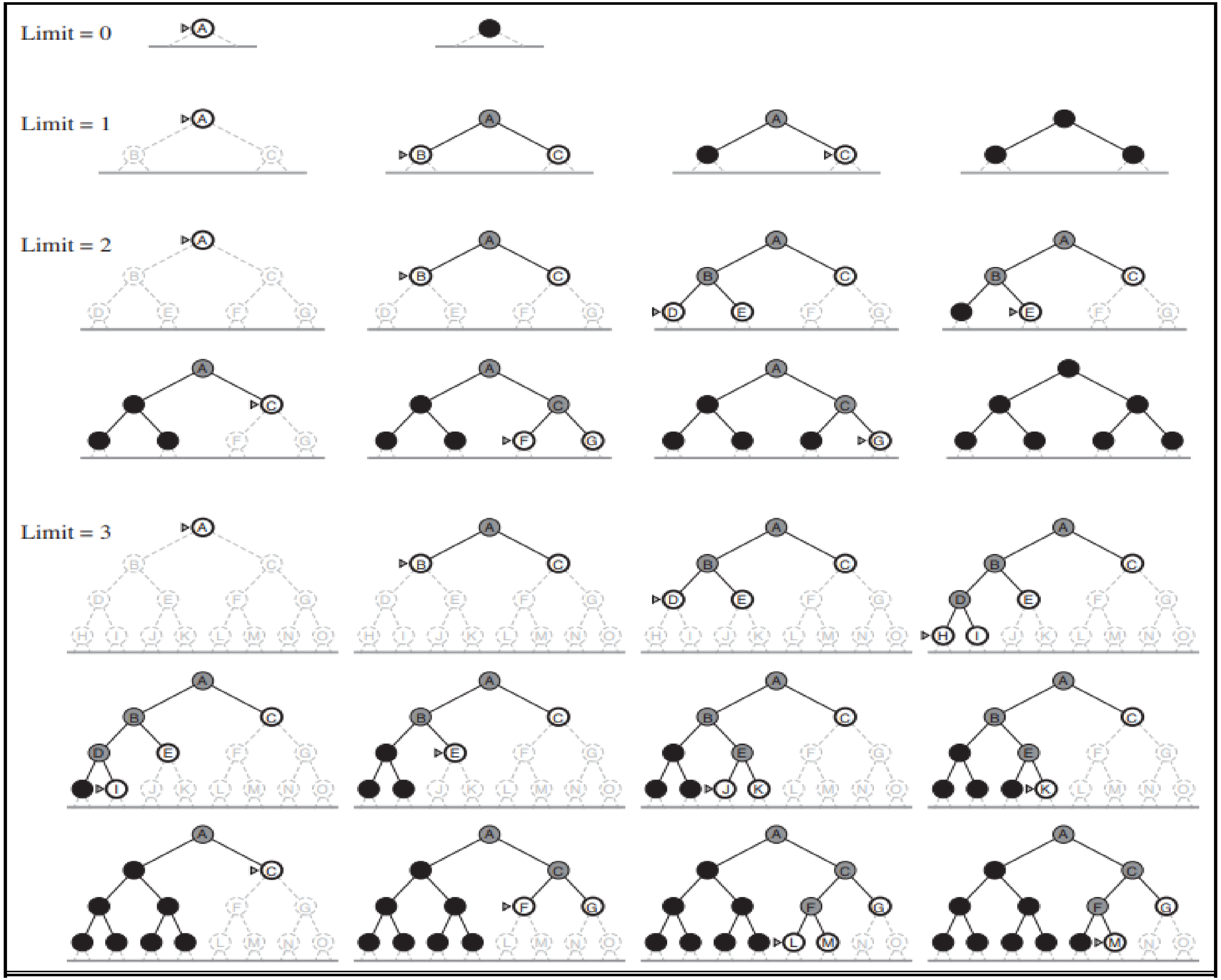
- Sometimes, depth limits can be based on knowledge of the problem
- For example, on the map of Romania there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so = 19 is a possible choice
- But if we see the map carefully, we would discover that any 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
- For most problems, however, we will not know a good depth limit until we have solved the problem

Iterative deepening depth-first search

Introduction

- Iterative deepening search (or iterative deepening depth-first search) is a general strategy, often used in combination with depth-first tree search, that finds the best depth limit
- It does this by gradually increasing the limit—first 0, then 1, then 2, and so on—until a goal is found
- Iterative deepening combines the benefits of depth-first and breadth-first search
- Like depth-first search, its memory requirements are modest: O(bd)
- 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

Four iterations of iterative deepening search on a binary tree



- Iterative deepening search may seem wasteful because states are generated multiple times
- But this is not too costly
- 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 in the worst case is $N(IDS)=(d)b + (d - 1)b^2 + \cdots + (1)b^d$

- Its time complexity is O(b^d)—asymptotically the same as breadth-first search
- There is some extra cost for generating the upper levels multiple times, but it is not large
- For example, if b = 10 and d = 5, the numbers are
- N(IDS) = 50 + 400 + 3, 000 + 20, 000 + 100, 000 = 123, 450
- N(BFS) = 10 + 100 + 1, 000 + 10, 000 + 100, 000 = 111, 110
- Therefore, in general, iterative deepening is the preferred uninformed search method when the search space is large, and the depth of the solution is not known

Search strategies in terms of the four evaluation criteria

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

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 ≥ for positive ; **c:** optimal if step costs are all identical