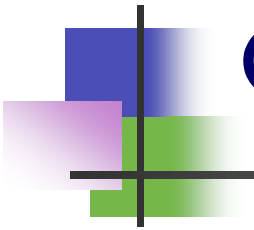
A.I.: Solving problems by searching

Chapter III: The Wrath of Exponentially-Large State Spaces

Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms

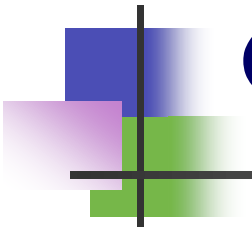


Overview

- Recall our previous discussion of **reflex agents**. Such agents cannot operate well in environments for which the state to action mapping is too large to store or would take too long to learn.

- **Problem-solving agents** use **atomic representations** (see Chapter 2), where states of the world are considered as wholes, with no internal structure visible to the problem-solving agent.

- We consider two general classes of search: (1) **uninformed search** algorithms for which the algorithm is provided no information about the problem other than its definition; (2) **informed search**, where the algorithm is given some guidance.



Overview

- Intelligent agents are supposed to maximize their **performance measure**; achieving this is sometimes simplified if the agent can adopt a **goal** and aim to satisfy it.

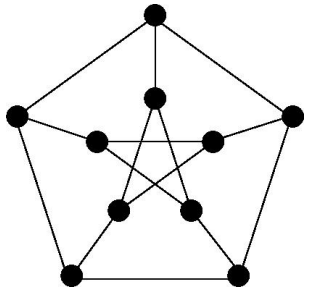
- **Goals** help organize behavior by limiting objectives. We consider a goal to be a set of world states – exactly those states in which the goal is satisfied.

- Here we consider environments that are **known**, **observable**, **discrete** and **deterministic** (i.e. each action has exactly one corresponding outcome).

- The process of looking for a sequence of actions that reaches the goal is called **search**. A **search algorithm** takes a problem as input and returns a solution in the form of an action sequence.

Well-defined problems and solutions

- A problem can be defined formally by (5) components:
- (1) The **initial state** from which the agent starts.
- (2) A description of **possible actions** available to the agent: ACTIONS(s)
- (3) A description of what each action does, i.e. the **transition model**, specified by a function RESULT (s,a)=a'.
- Together, the initial state, actions and transition model implicitly defined the state space of the problem – the set of all states reachable from the initial state by any sequence of actions.
- The state space forms a directed network or **graph** in which the nodes are states and the edges between nodes are actions. A **path** in the state space is a sequence of states connected by a sequence of actions.



Well-defined problems and solutions

- (4) The **goal test**, which determines whether a given state is a goal state. Frequently the goal test is intuitive (e.g. check if we arrived at the destination) – but note that it is also sometimes specified by an abstract property (e.g. "check mate").

- (5) A **path cost function** that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure.
- Commonly (but not always), the cost of a path is additive in terms of the individual actions along a path. Denote the step cost to take action 'a' in state s, arriving in s' as: **c(s,a,s')**.
- A key element of successful problem formulation relates to **abstraction** – the process of removing (inessential) details from a problem representation.

Problem-solving agents

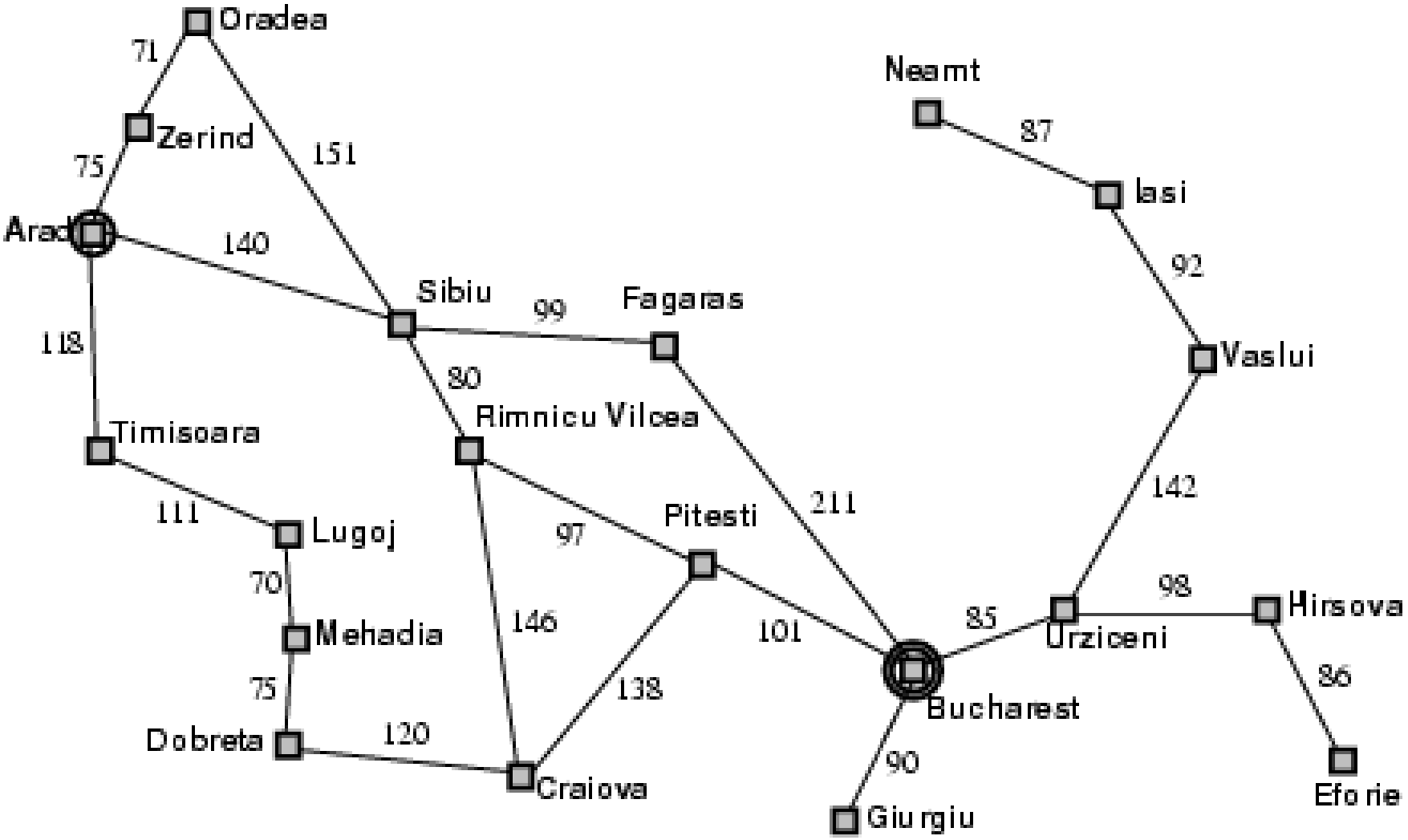
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Example: Romania

- 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

Example: Romania



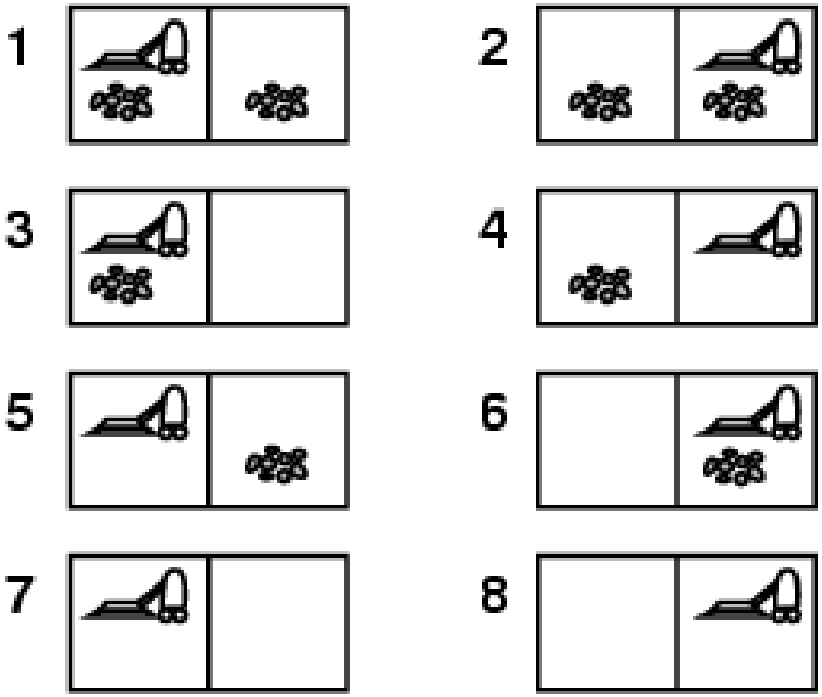
Problem types

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
  -
- Non-observable → sensorless problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence
  -
- Nondeterministic and/or partially observable → contingency problem
  - percepts provide new information about current state
  - often interleave search, execution
  -
- Unknown state space → exploration problem

Example: vacuum world

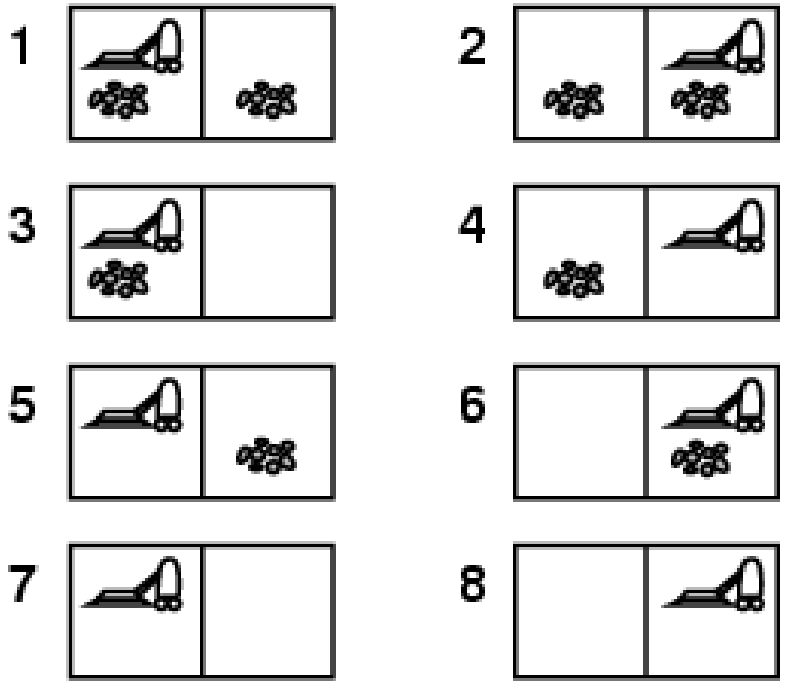
- Single-state, start in #5.

  Solution?



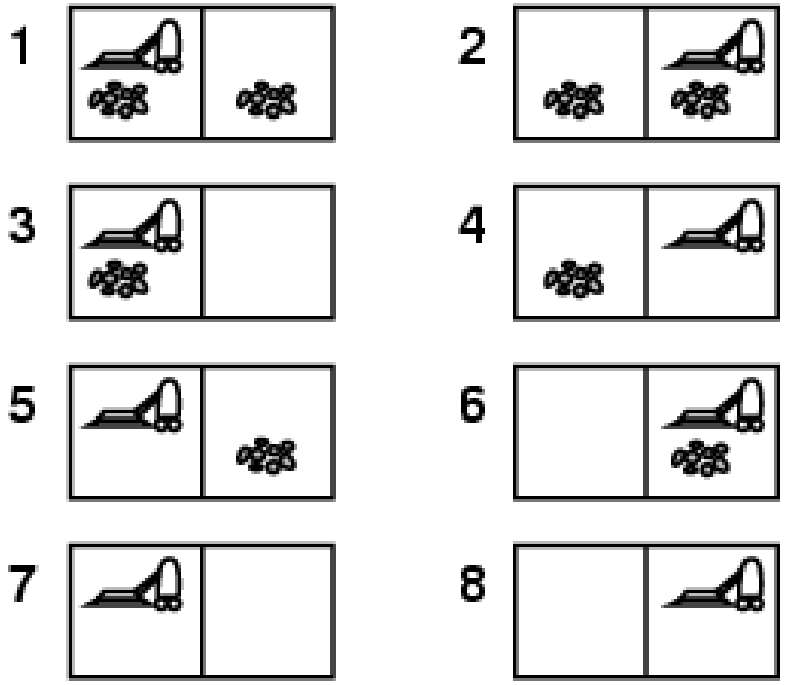
Example: vacuum world

- Single-state, start in #5.
  Solution? *[Right, Suck]*
-
- Sensorless, start in
  {1,2,3,4,5,6,7,8} e.g.,
  Right goes to *{2,4,6,8}*
  Solution?
-



Example: vacuum world

- Sensorless, start in *{1,2,3,4,5,6,7,8}* e.g., *Right* goes to *{2,4,6,8}*
  Solution?
  [Right,Suck,Left,Suck]

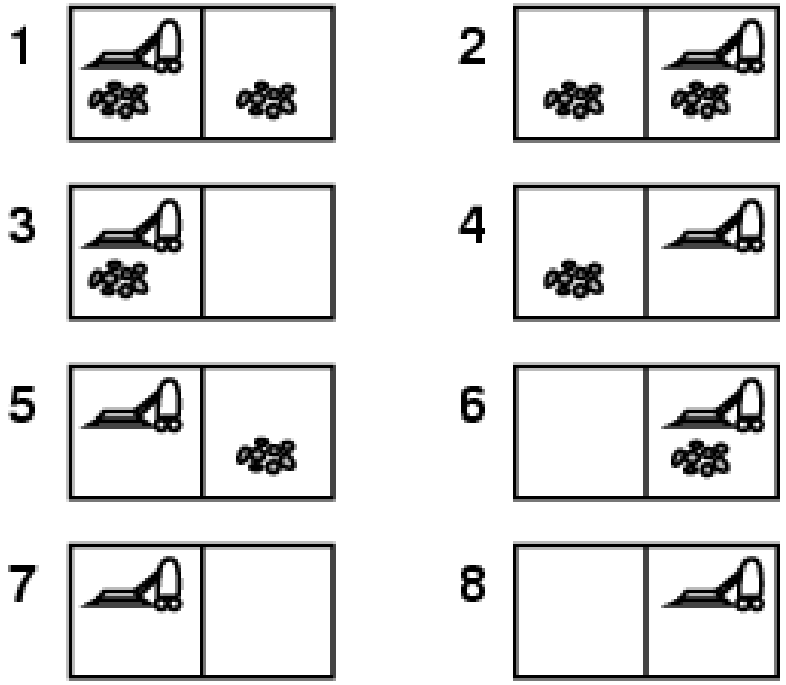


- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean]*, i.e., start in #5 or #7
    Solution?

Example: vacuum world

- Sensorless, start in *{1,2,3,4,5,6,7,8}* e.g., *Right* goes to *{2,4,6,8}*
  Solution?

  [Right,Suck,Left,Suck]



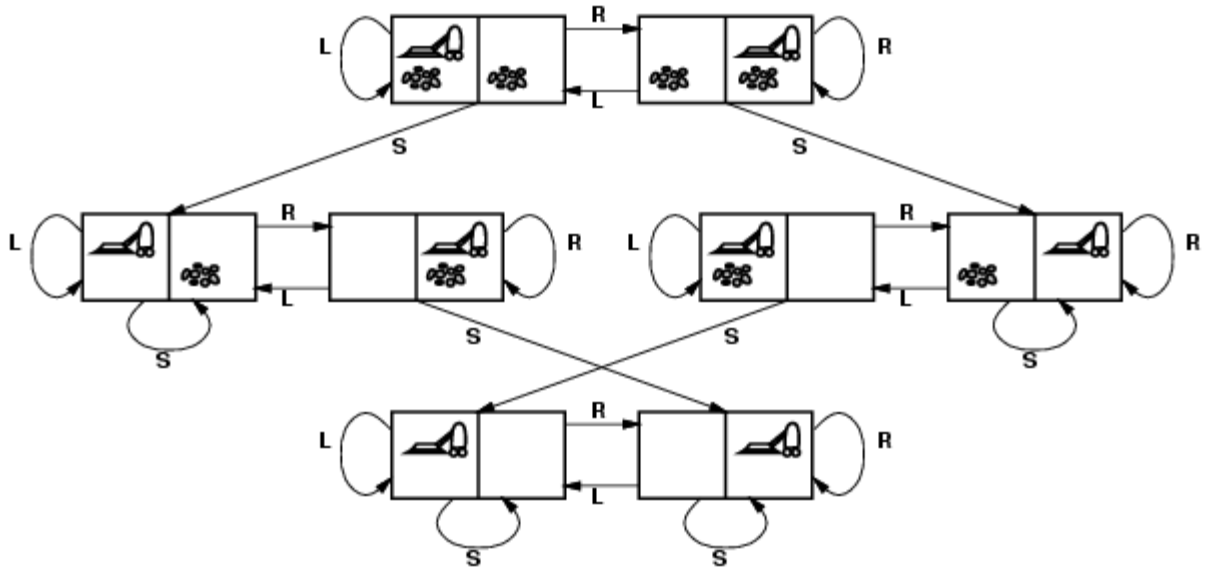
- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean]*, i.e., start in #5 or #7
    Solution? *[Right, **if** dirt **then** Suck]*

Selecting a state space

- Real world is absurdly complex
  → state space must be abstracted for problem solving

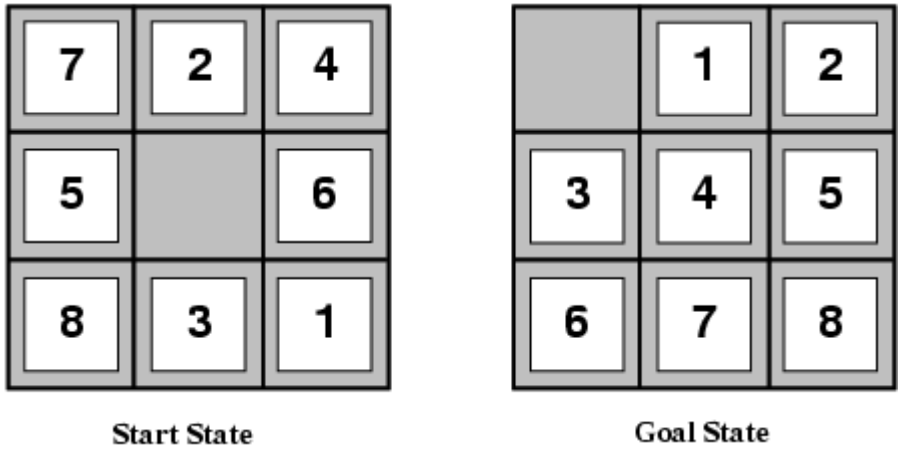
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad" must get to some real state "in Zerind"
-
- (Abstract) solution =
  - set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

Vacuum world state space graph



- States: integer dirt and robot location.
- Actions: *Left*, *Right*, *Suck*.
- Goal test: no dirt at all locations.
- Path cost: 1 per action.

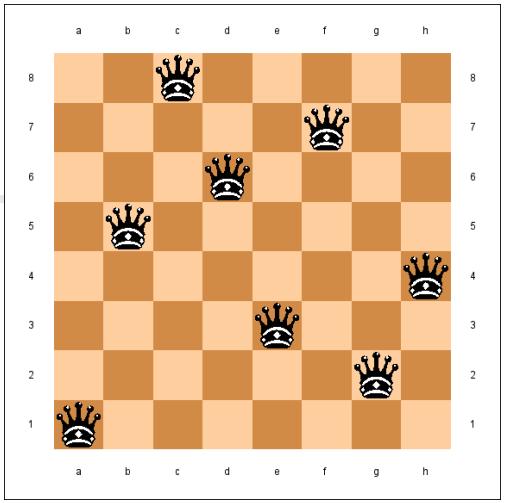
Example: The 8-puzzle



- States: Locations of tiles. (Q: How large is state space?)
- Actions: Move blank left, right, up, down.
- Goal test: s==goal state. (given)
- Path cost: 1 per move.

[Note: optimal solution of *n*-Puzzle family is NP-hard]

Example: 8-queens

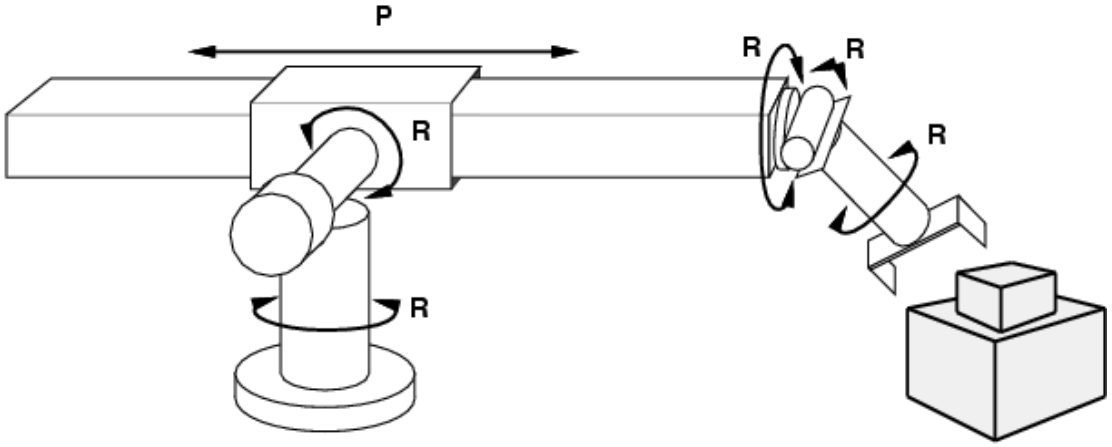


- States: Any arrangement of 0 to 8 queens on the board.
- Actions: Add a queen to any empty square.
- Transition Model: Returns the board with a queen added to the specified square.
- Goal Test: 8 queens on the board, none attacking.

- Q: How many possible sequences? (~1.8×10^{14})

- Revision to problem: arrange queens with one per column, in the leftmost n columns, with no queen attacking another.
- Actions: Add a queen to any square in leftmost empty column (with no attacking) reduction to just 2,057 states!

Example: robotic assembly

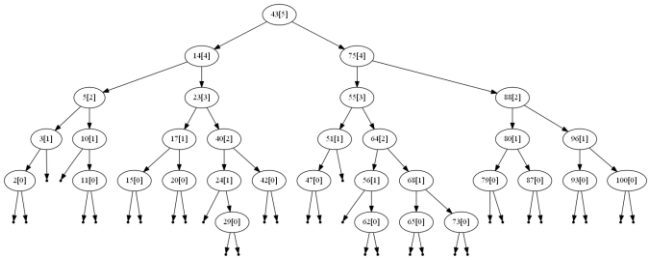


- States: Real-valued coordinates of robot joint angles parts of the object to be assembled.
- Actions: Continuous motions of robot joints.
- Goal test: Complete assembly.
- Path cost: Time to execute.

- Other examples: TSP (NP-hard), robot navigation, protein folding (unsolved).

Searching for Solutions

- Recall that a **solution is an actions sequence**; accordingly, **search algorithms** work by considering various possible action sequences.
- The possible action sequences starting at the initial state form a **search tree** with the initial state at the root; the branches are actions and the nodes correspond to state in the state space of the problem.
- To consider taking various actions, we **expand** the current state – thereby generating a new set of states.
- In this way, we add branches from the **parent node** to **children nodes**.



Searching for Solutions

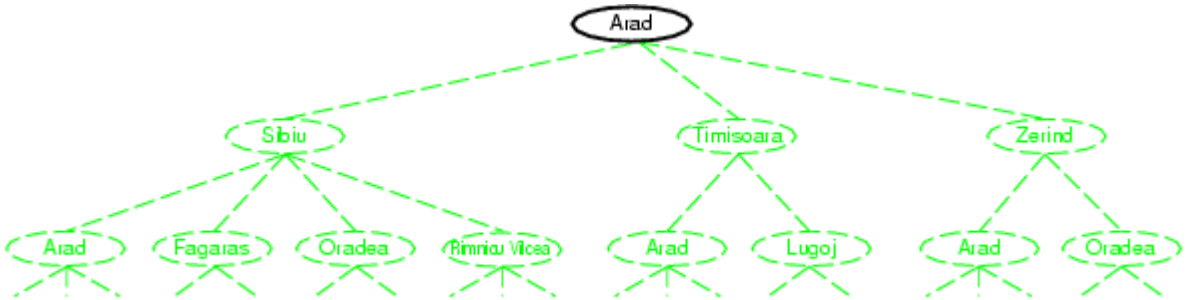
- A node with no children is called a **leaf**; the set of all leaf nodes available for expansion at any given point is called **the frontier**.
- The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.
- We consider the general TREE-SEARCH algorithm next.
- All search algorithms share this basic structure; they vary primarily according to how they choose which state to expand next – the so-called search strategy.
- In general, a TREE-SEARCH considers all possible paths (including infinite ones), whereas a GRAPH-SEARCH avoids consideration of **redundant paths**.

Tree search algorithms

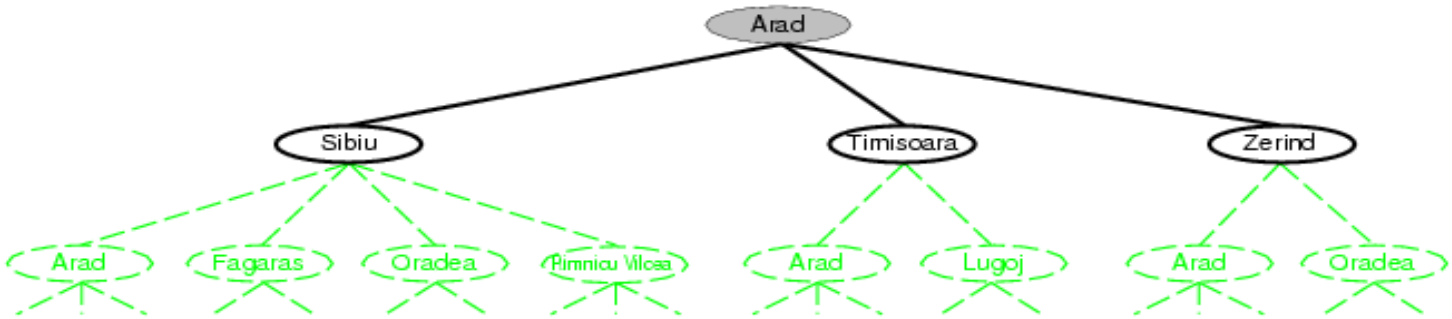
- Basic idea:
  - Offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

```
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
```

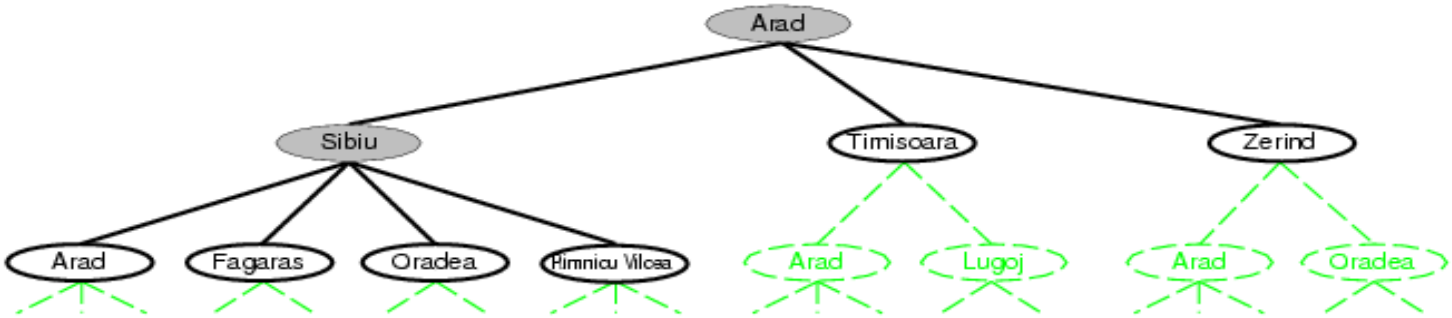
Tree search example



Tree search example



Tree search example



Implementation: general tree search

```
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 SOLUTION(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
```

Searching for Solutions

- Naturally, if we allow for **redundant paths**, then a formerly tractable problem can become intractable" "Algorithms that forget their history are doomed to repeat it."

- To avoid exploring redundant paths we can augment the TREE-SEARCH algorithm with a data structure called the **explored set**, which remembers every expanded node (we discard nodes in explored set instead of adding them to the frontier).

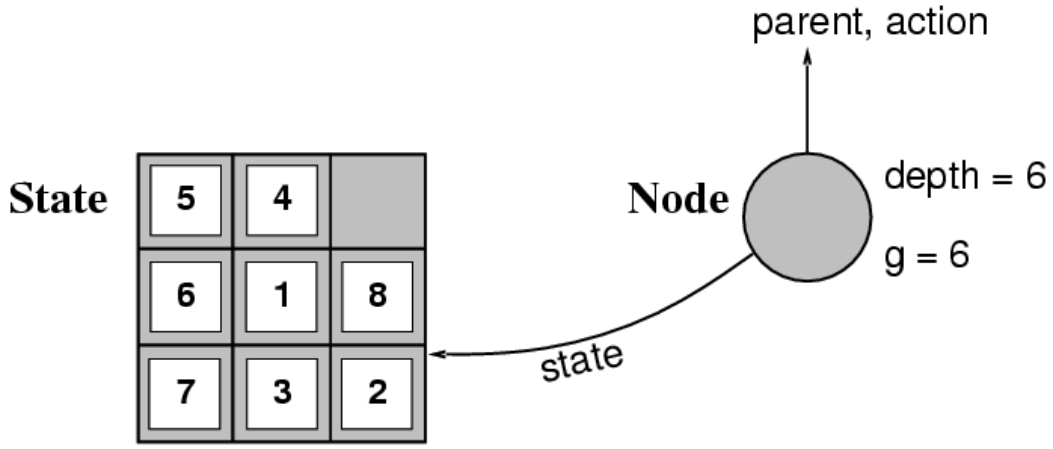
Infrastructure for search algorithms

- Search algorithms require a data structure to keep track of the search tree that is being constructed.
- Foe each **node *n* of the tree**, we have a structure with (4) components:

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

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration.
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost $g(x)$, depth.



- The **Expand** function creates new nodes, filling in the various fields and using the **Successor** function of the problem to create the corresponding states.

Infrastructure for search algorithms

- Now that we have nodes (*qua* data structures), we need to put them somewhere.
- We use a **queue**, with operations:

EMPTY?(queue): returns true only if no elements
POP(queue): removes the first element of the queue and returns it.
INSERT(element, queue): inserts an element and returns the resulting queue.

- Recall that queues are characterized by the order in which the store the inserted nodes:

FIFO (first-in, first-out): pops the oldest element of the queue.
LIFO (last-in, first-out, i.e. a **stack**) pops the newest element.
PRIORITY QUEUE: pops element with highest "priority."

Search strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution when one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?

- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)
  - The size of the state space graph (|V|+|E|)

Search strategies

- In more detail:
- **Time and space complexity** are always considered with respect to some measure of the problem difficult (e.g. |V| + |E|).
- In A.I., the state space graph is often represented *implicitly* by the initial state, actions and transition model (i.e. we don't always store it explicitly).

- Search algorithm complexity is frequently expressed in terms of:

b: branching factor (maximum number of successors of any node)

d: depth (of the shallowest goal node)

m: maximum length of any path in the state space.

- To assess the effectiveness of a search algorithm, we can consider just the **search cost**, which typically depends on time complexity (and/or memory usage); **total cost** combines search cost and path cost of the solution.

Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition.

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

Breadth-first search

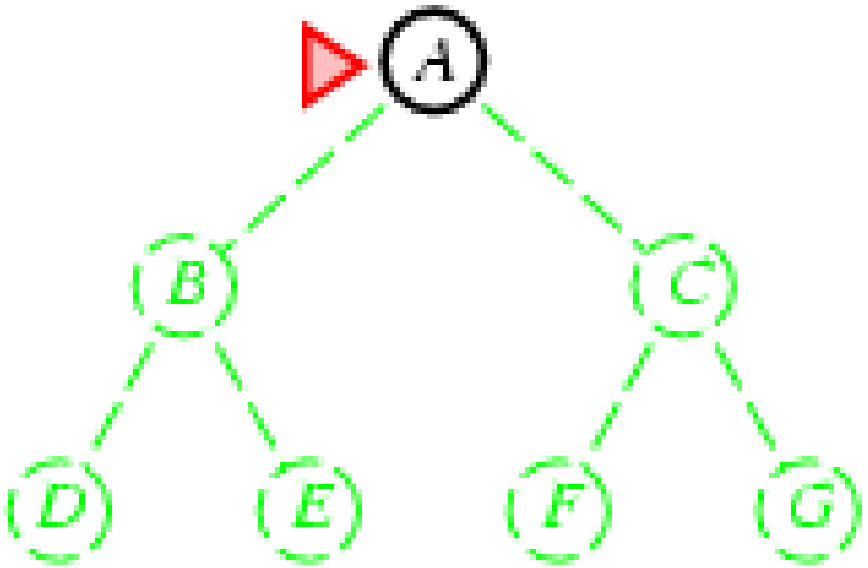
- BFS is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, etc.

- BFS is an instance of the general graph-search algorithm in which the shallowest unexpanded node is chosen for expansion.

- This is achieved by using a **FIFO queue** for the frontier. Accordingly, new nodes go to the back of the queue, and **old nodes, which are shallower than the new nodes are expanded first**.
- *NB*: The *goal test* is applied to each node when it is *generated*.

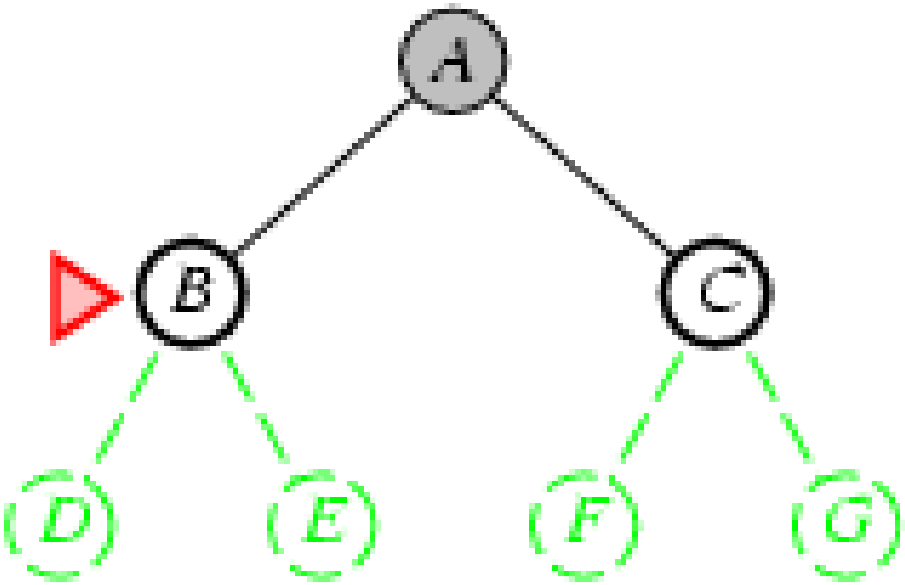
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *frontier* is a FIFO queue, i.e., new successors go at end



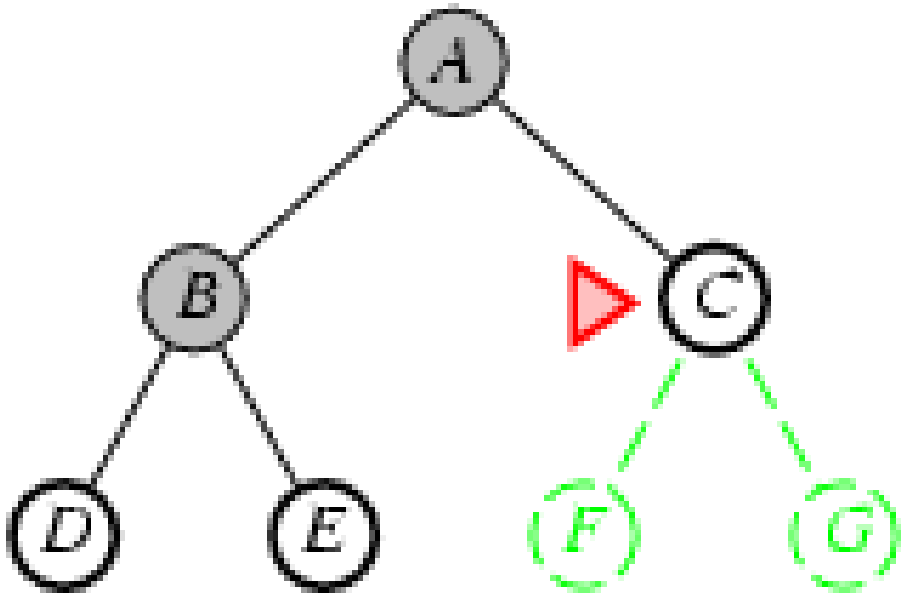
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *frontier* is a FIFO queue, i.e., new successors go at end



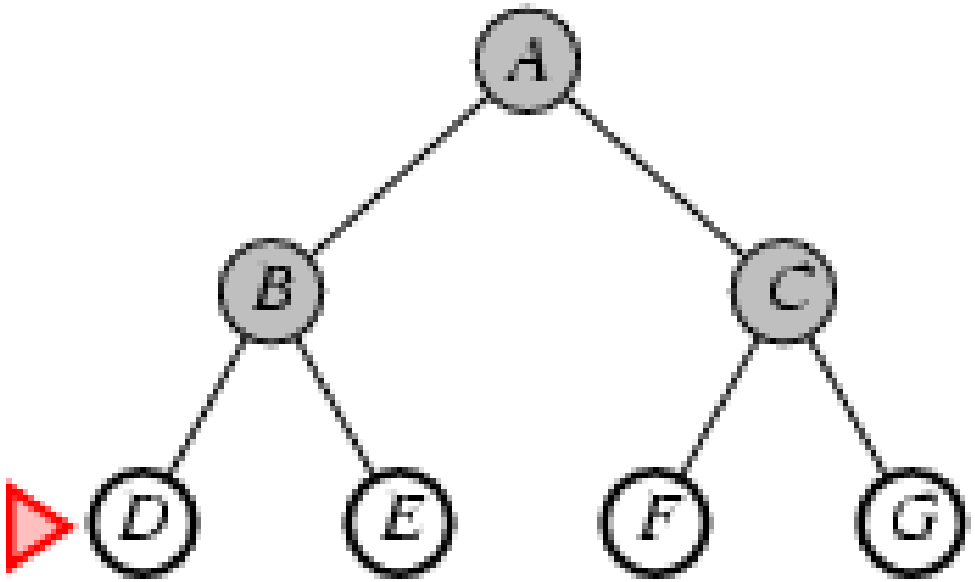
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



Properties of BFS

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

Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - *frontier* = queue ordered by path cost

- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost $\geq \varepsilon$
- Time? # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$
- Optimal? Yes – nodes expanded in increasing order of $g(n)$

Depth-first search

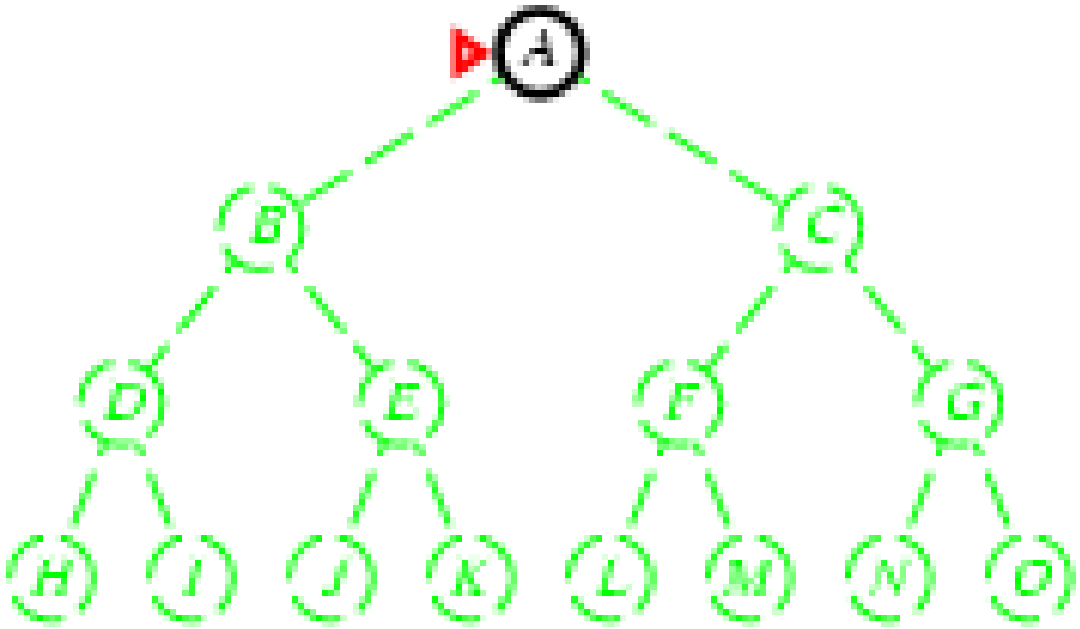
- DFS always expands the *deepest node* in the current frontier of the search tree. The search proceeds immediately to the deepest level fo the search tree, where the nodes have no successors.

- As those nodes are expended, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors.

- DFS is an instance of the general graph-search algorithm which uses a LIFO queue. This means that the most recently generated node is chosen for expansion.

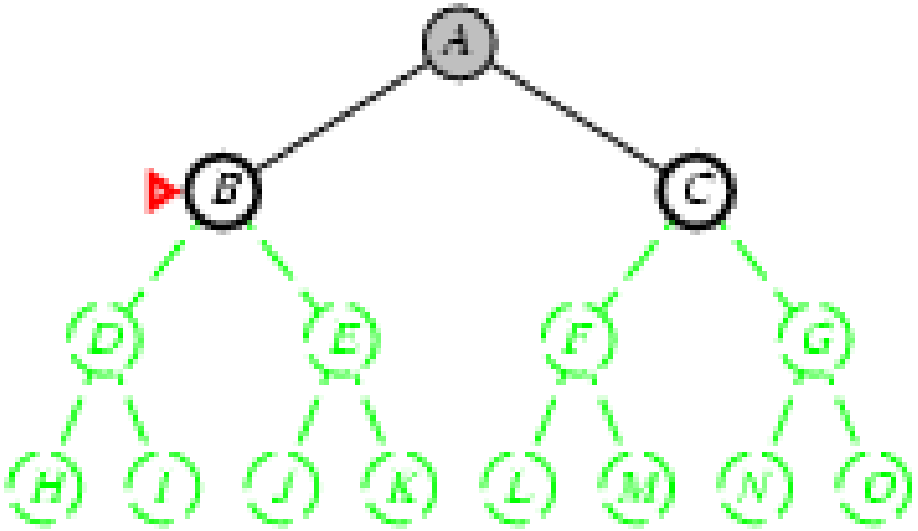
Depth-first search

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



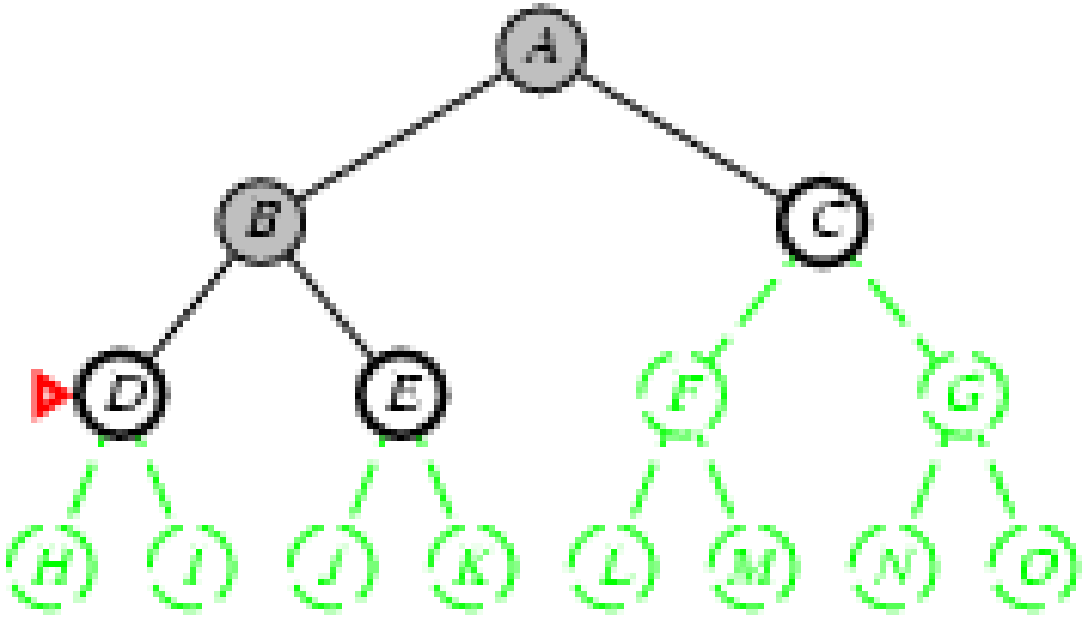
Depth-first search

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



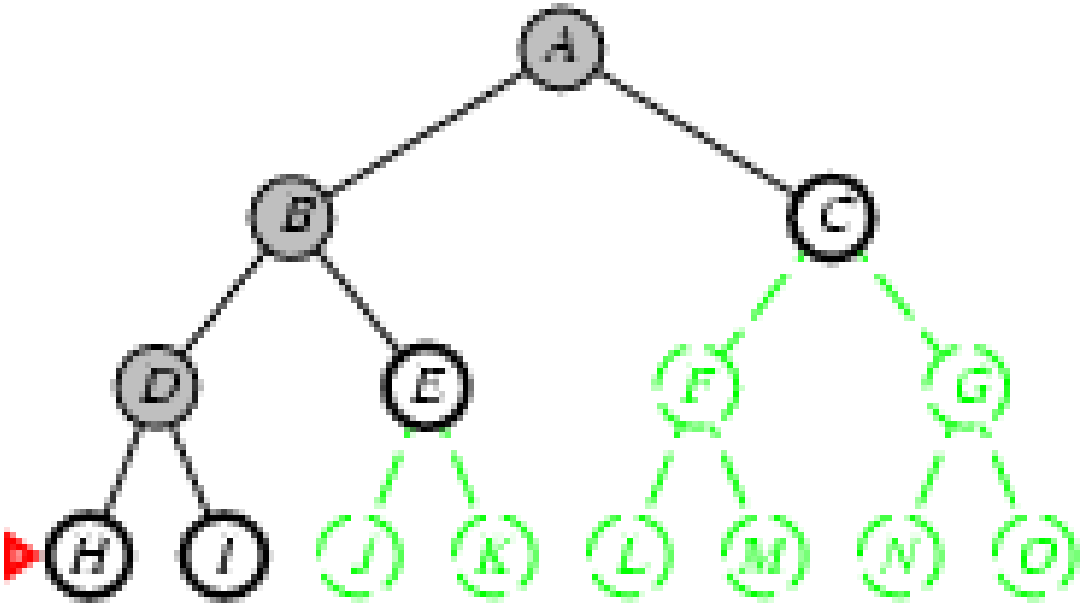
Depth-first search

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



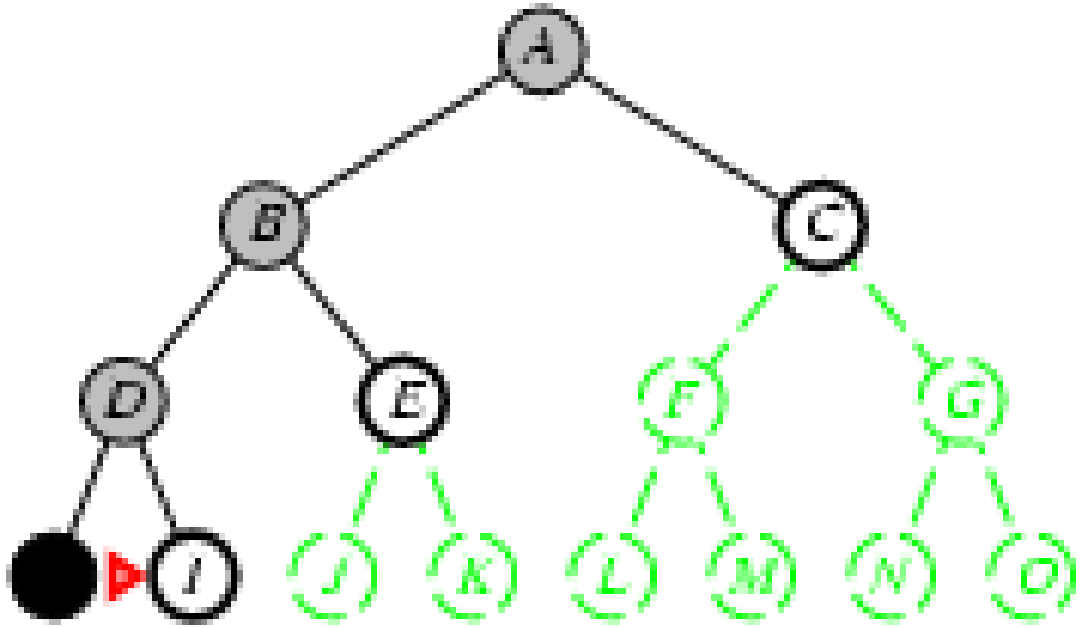
Depth-first search

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



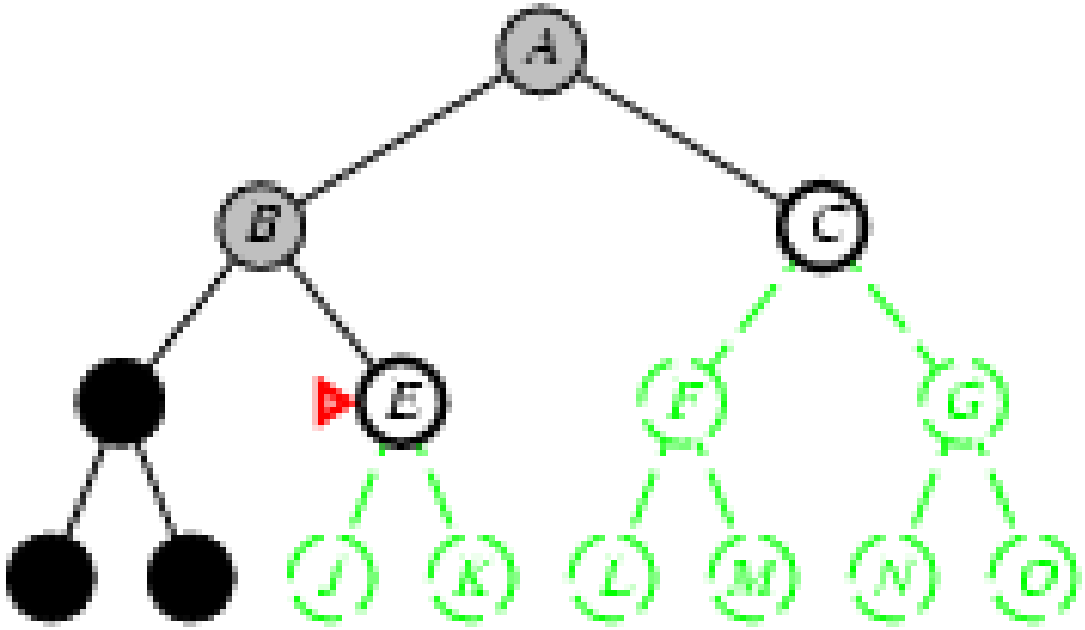
Depth-first search

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



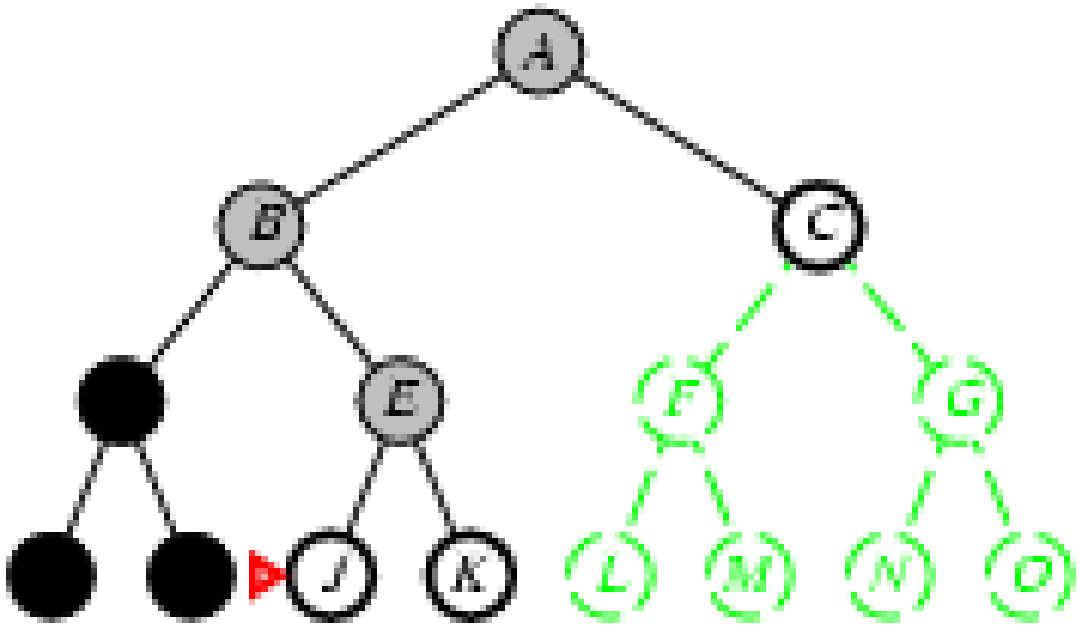
Depth-first search

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



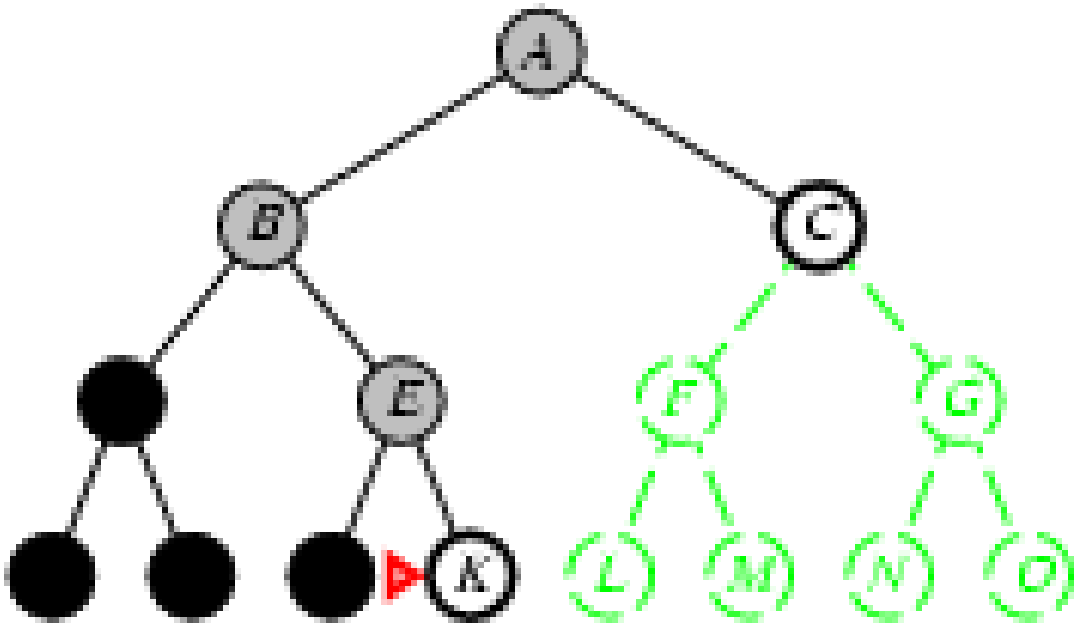
Depth-first search

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



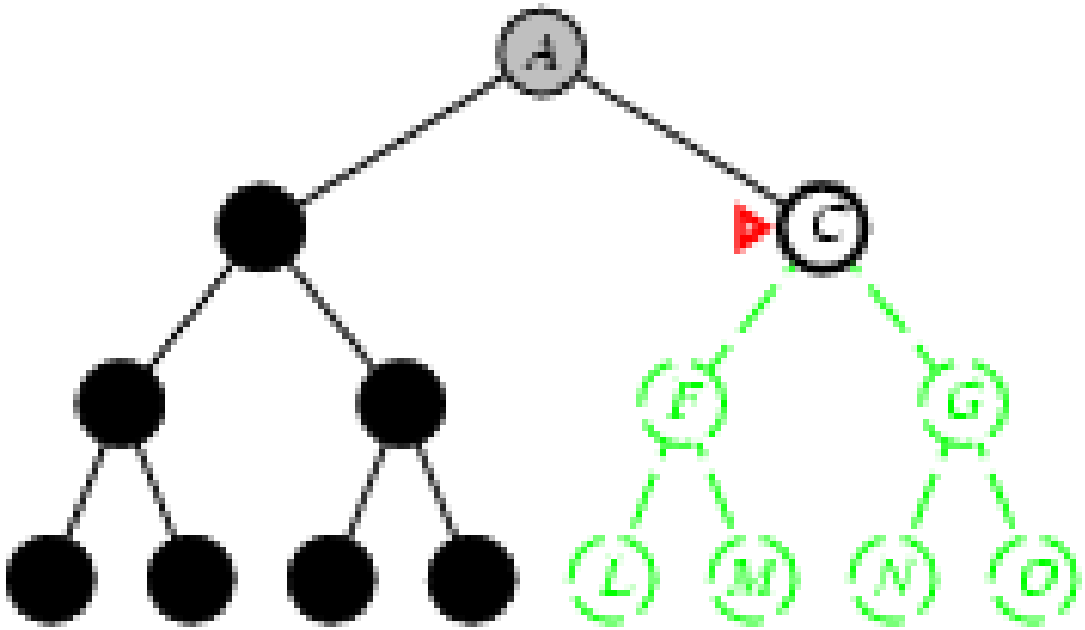
Depth-first search

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



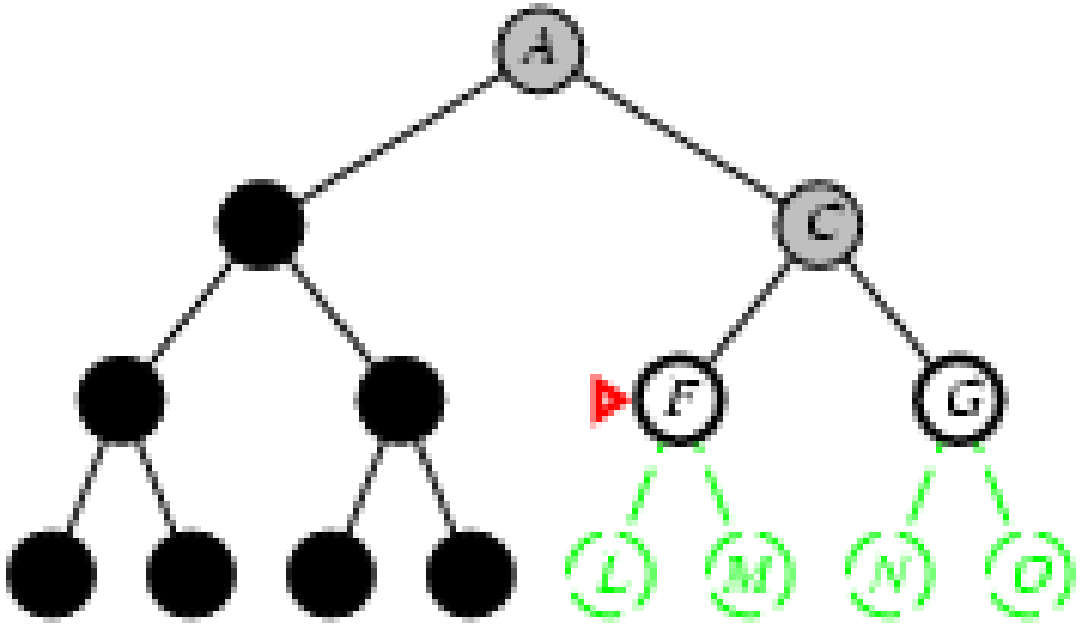
Depth-first search

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



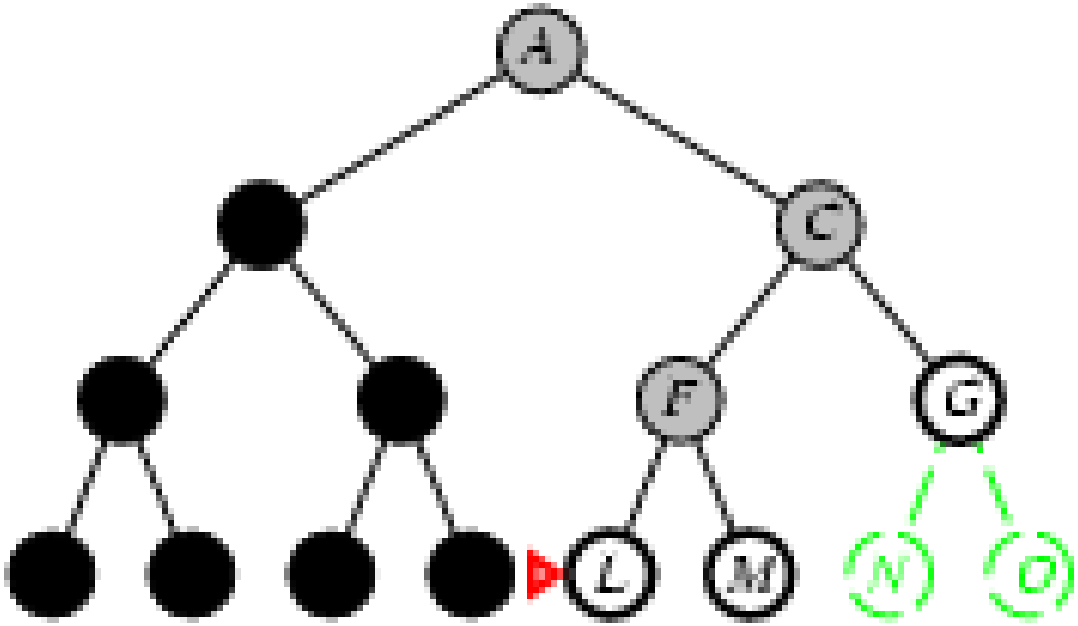
Depth-first search

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



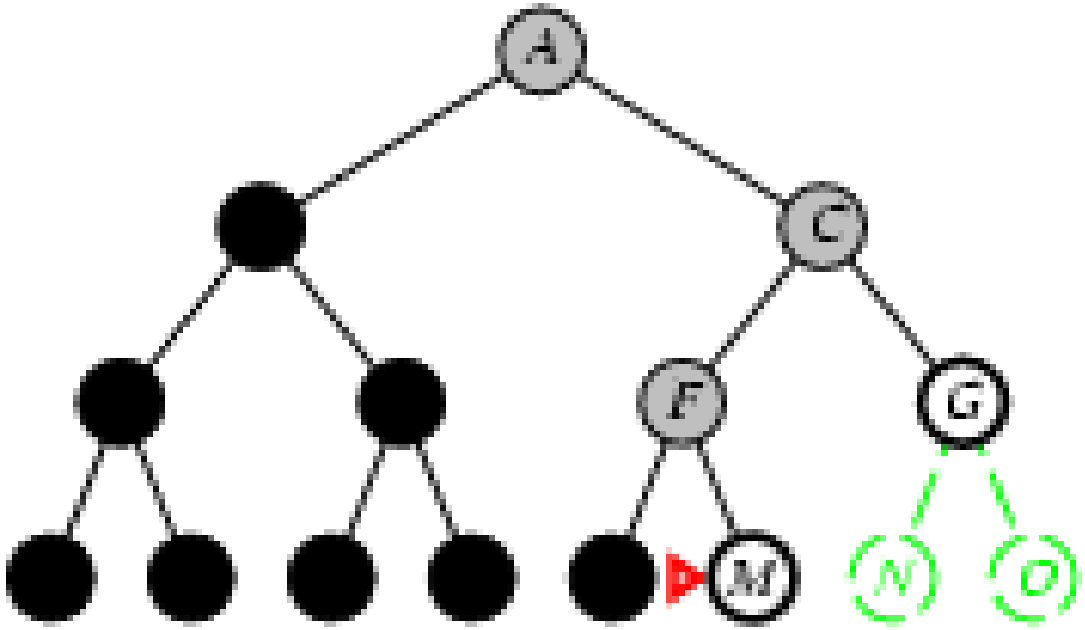
Depth-first search

- 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



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

Depth-limited search

- The failure of DFS in infinite state spaces can be alleviated by suppling DFS with a pre-determined **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 SOLUTION(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
```

Iterative deepening search

- **Iterative deepening search** is a general strategy, often used in combination with DFS tress 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; this will occur when the depth limit reaches d, the depth of the shallowest goal node.

- Note that iteratively deepening search may seem wasteful because states are generated multiple times, but this, in fact, turns out not to be too costly (the reason is that most of the nodes are in the bottom level for a constant branching factor).

Iterative deepening search

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

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

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

$$N_{IDS} = (d+1)b^0 + d\,b\text{^}1 + (d-1)b\text{^}2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For $b = 10$, $d = 5$,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

- Overhead = (123,456 - 111,111)/111,111 = 11%

Iterative deepening search

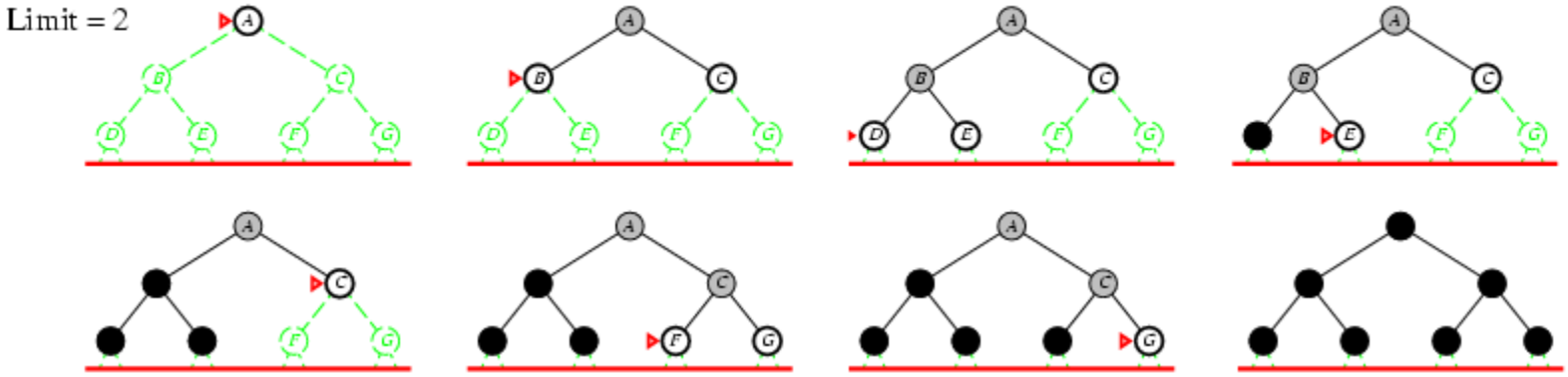
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

Iterative deepening search $l = 0$

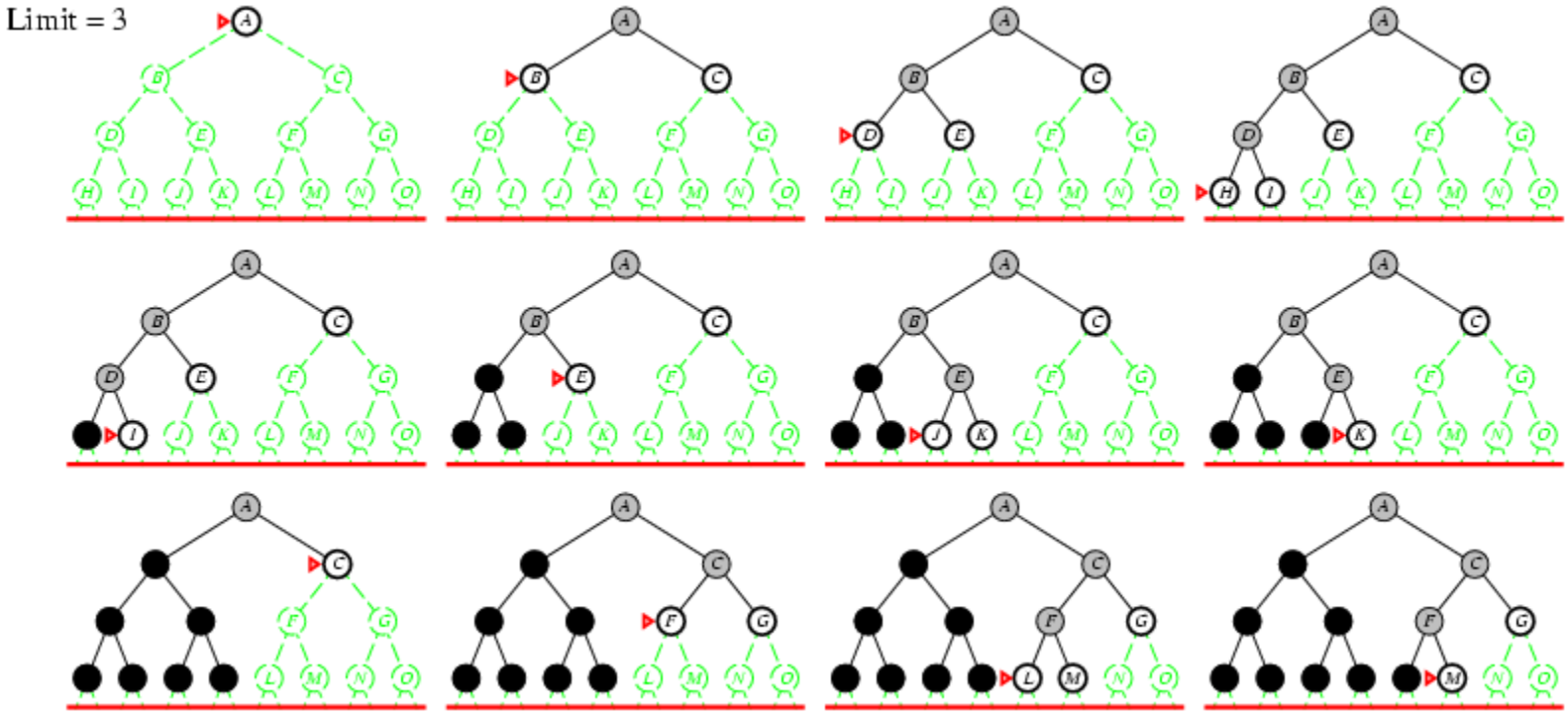
Limit = 0

Iterative deepening search $l = 1$

Iterative deepening search $l = 2$



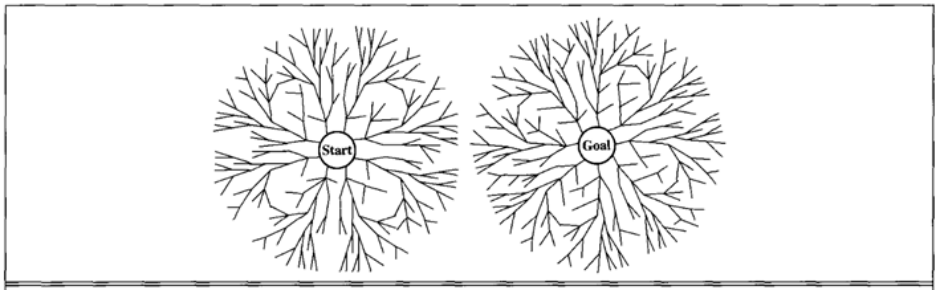
Iterative deepening search $l = 3$



Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

Bidirectional search



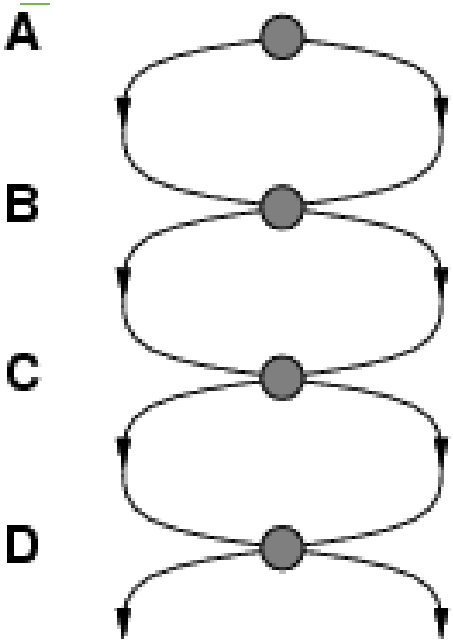
- The main idea with **bidirectional search** is to run two simultaneous searches – one forward from the initial state and the other backward from the goal – hope that the two searches meet in the middle.
- **Key**: $b^{d/2}+b^{d/2} << b^{d.}$
- Replace goal test with check to see whether frontiers intersect.
- Time complexity (with BFS in both directions): $O(b^{d/2})$; space complexity: $O(b^{d/2})$; space requirement is a serious weakness.
- Also, it is not always a simple matter to "search backward" – goal state could be abstract.

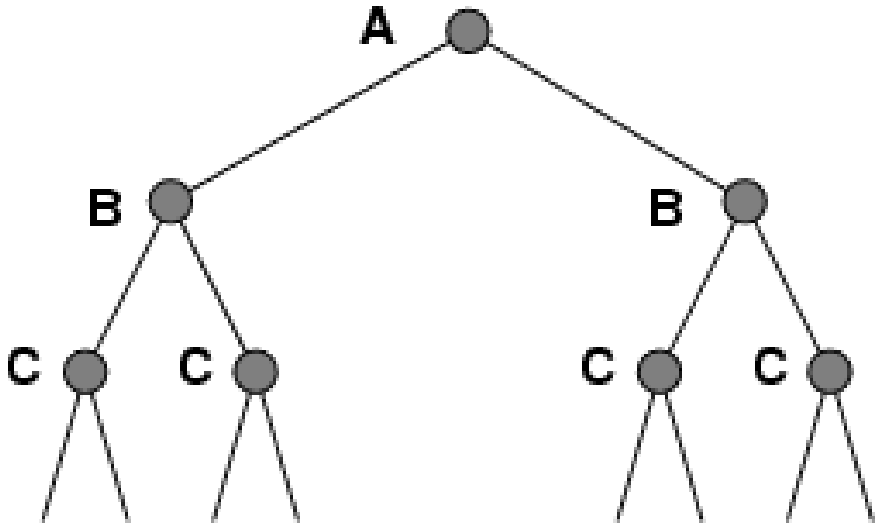
Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!



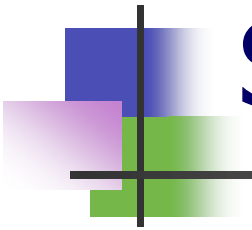


Graph search

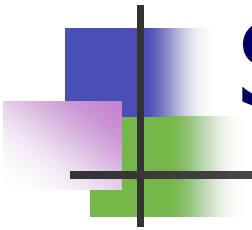
```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

    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 SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Summary



- Before an agent can start searching for solutions, a **goal** must be identified and a well-defined problem formulated.
- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- A **problem** consists of (5) parts: **initial state**, **actions**, **transition model**, **goal test** function and **path cost** function.
- The **environment** of the problem is represented by the state space. A **path** through the state space from the initial state to a goal state is a solution.
- TREE-SEARCH considers all possible paths; GRAPH-SEARCH avoids consideration of redundant paths.



Summary

- Search algorithms are judged on the basis of **completeness**, **optimality**, **time complexity** and **space complexity**. Complexity depends on **b**, the branching factor in the state space and **d**, the depth of the shallowest solution.
- Uniformed search methods have access only the problem definition, including:

BFS – expands the shallowest nodes first

Uniform-cost search – expands the node with the lowest path cost, g(n).

DFS – expands the deepest unexpanded node first (depth-limited search adds a depth bound).

Iterative Deepening Search – calls DFS with increasing depth limits until a goal is found.

Bidirectional Search – can reduce time complexity but not always applicable.