Solving problems by searching

Chapter 3

Outline

- Problem-solving agents
  - **A kind of goal-based agent**
- Problem types
  - **Search with partial information**
  - **Single state (fully observable)**
- Problem formulation

- Basic search algorithms
  - **Uninformed**

Building Goal-Based Agents

- What goal does the agent need to achieve
- How do you describe the goal?
  - **as a task to be accomplished**
  - **as a situation to be reached**
  - **as a set of properties to be acquired**
- How do you know when the goal is reached
  - **with a goal test that defines what it means to have achieved/satisfied the goal**
- Determining the goal is difficult and is usually left to the system designer or user to specify

Basic Uninformed Search

- **search: finding a path through a network/graph/tree**
  - **hopefully, a path from initial node to goal**
- **A problem defines a "search space":**

– we are at one place in the space, and we wish to find a solution destination

  - **a universe of configurations**

– different search algorithms give different ways of navigating the space

- **Blind methods: search strategies that do not use problem information to guide the search ("uninformed")**
  - **search strategy exhaustively applied until solution found (or failure)**

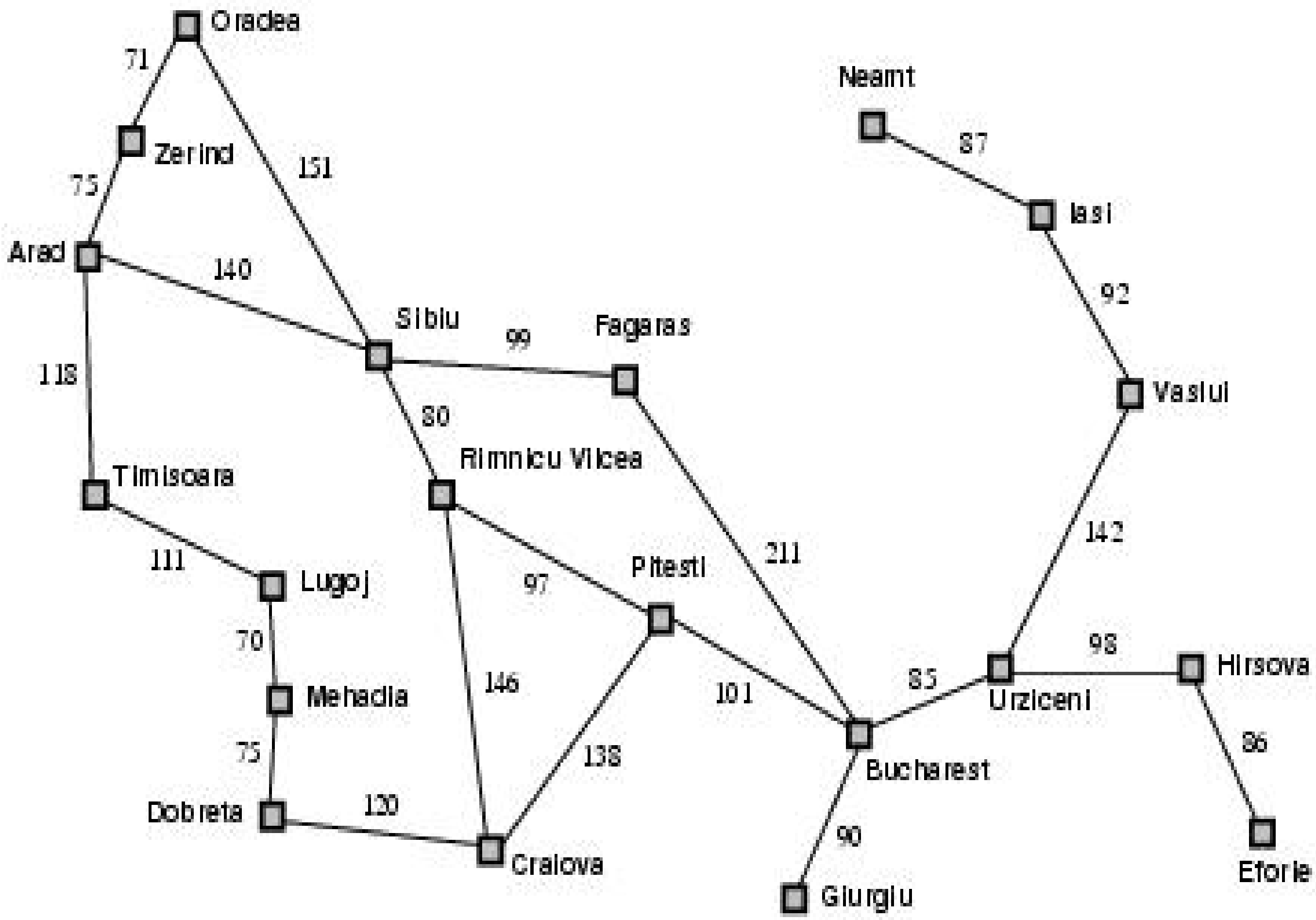
- **while searching, we normally not wish to visit the same node twice**
  - **represents a cycle - means that infinite looping may occur**
- **– solution: rewrite the network so that loops are removed**

- **search tree: tree in which each node denotes a step in a path from initial goal to target goal**

Problem-solving agent

- Four general steps in problem solving:
  - **Goal formulation**
    - What are the successful world states
  - **Problem formulation**
    - What actions and states to consider give the goal
  - **Search**
    - Determine the possible sequence of actions that lead to the states of known values and then choosing the best sequence.
  - **Execute**
    - Give the solution perform the actions.

Example: Romania



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, …

Single-State Problem formulation

- A problem is defined by:
  - An initial state, e.g. *Arad*
  - Successor function *S(X)*= set of action-state pairs
    - e.g. *S(Arad)={<Arad $\rightarrow$ Zerind, Zerind>,…}*

  intial state + successor function = state space
  - Goal test, can be
    - Explicit, e.g. *x='at bucharest'*
    - Implicit, e.g. *checkmate(x)*
  - Path cost (additive)
    - e.g. sum of distances, number of actions executed, …
    - *c(x,a,y)* is the step cost, assumed to be >= 0

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

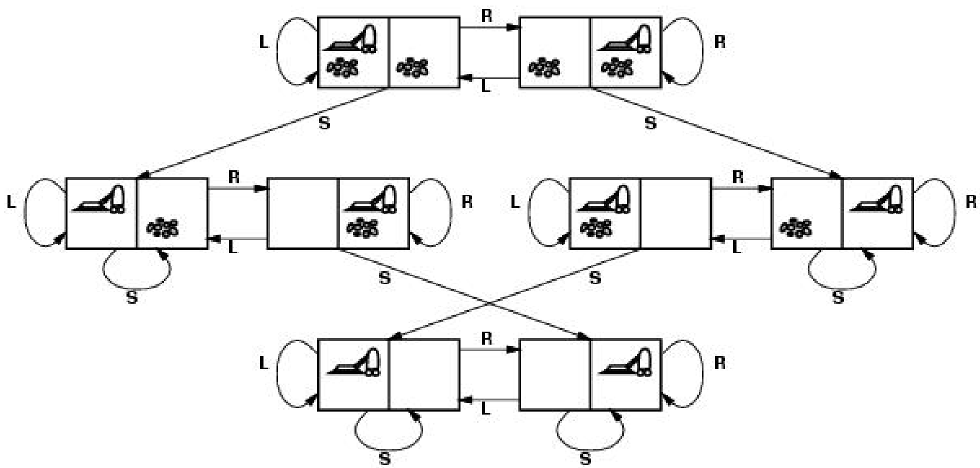
Optimal solution has the lowest path cost.

Selecting a state space

- **Real world** is absurdly complex.

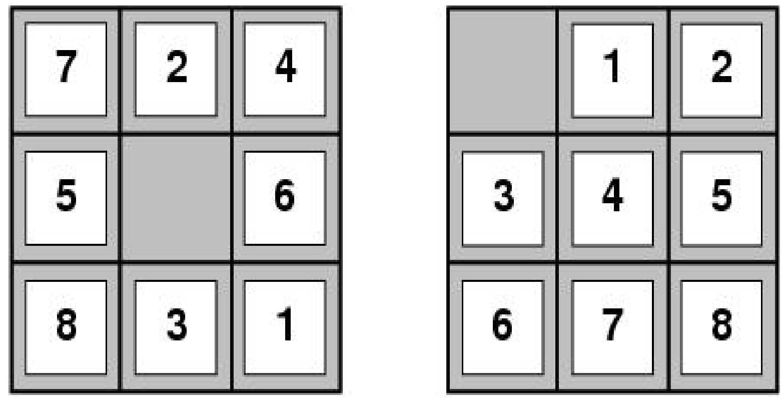
    State space must be *abstracted* for problem solving.
- (Abstract) **state** = set of real states.
- (Abstract) **action** = complex combination of real actions.
    - e.g. Arad $\rightarrow$Zerind represents a complex set of possible routes, detours, rest stops, etc.
    - The abstraction is valid if the path between two states is reflected in the real world.
- (Abstract) **solution** = set of real paths that are solutions in the real world.
- Each <u>abstract action should be “easier”</u> than the <u>real problem</u>.

Example: vacuum world



- **States?:** two locations with or without dirt.
- **Initial state**?: Any state can be initial
- **Actions?:** {*Left*, *Right*, *Suck*}
- **Goal test?:** No dirt at all locations.
- **Path cost?:** Number of actions to reach goal.

Example: 8-puzzle

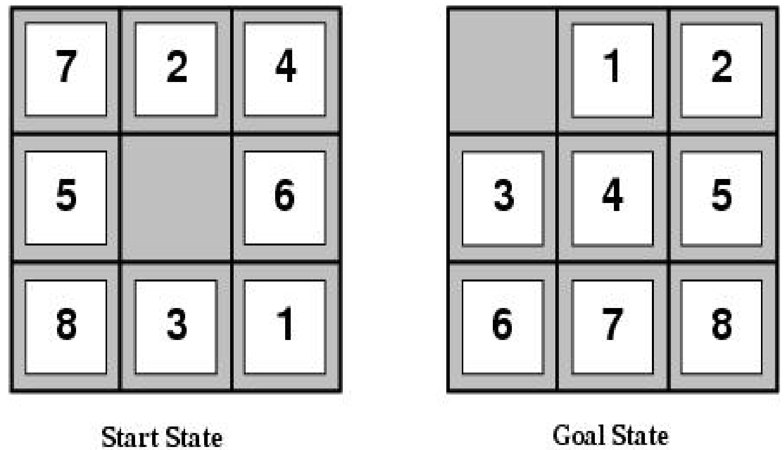


Start State

Goal State

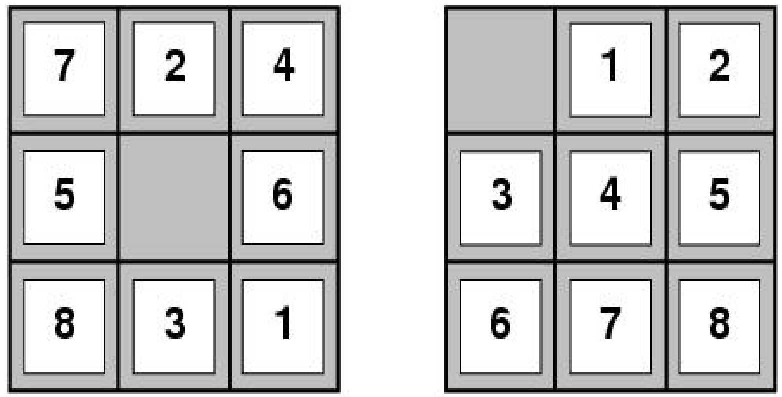
- States?
- Initial state?
- Actions?
- Goal test?
- Path cost?

Example: 8-puzzle



- **States**?: Integer location of each tile
- **Initial state**?:
- **Actions**?:
- **Goal test**?:
- **Path cost**?:

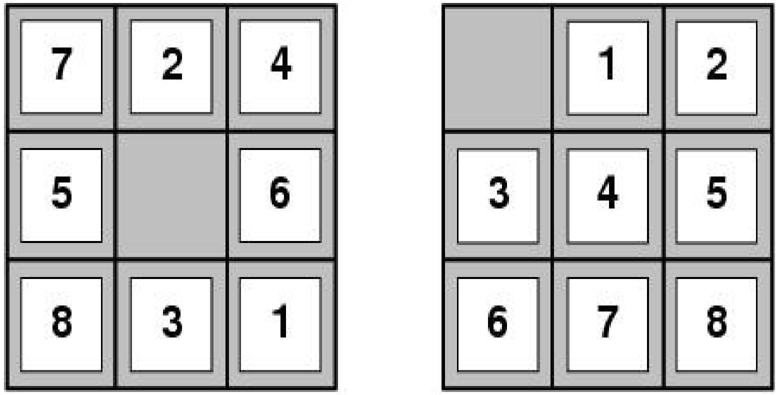
Example: 8-puzzle



Start State Goal State

- **States**?: Integer location of each tile
- **Initial state**?: Any state can be initial
- **Actions**?:
- **Goal test**?:
- **Path cost**?:

Example: 8-puzzle

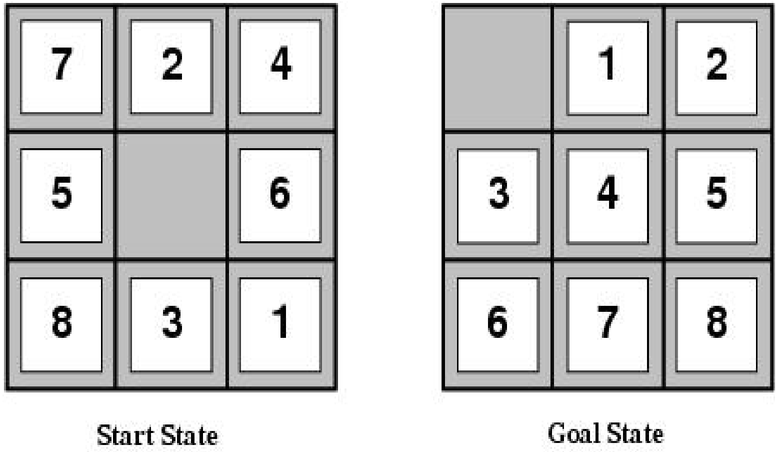


Start State

Goal State

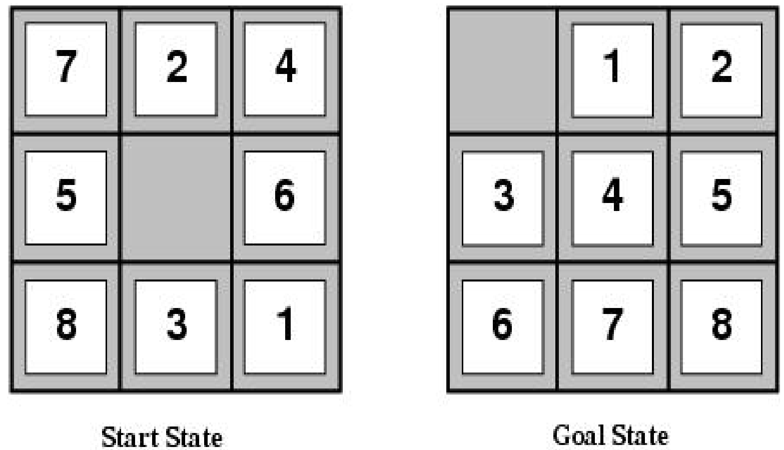
- **States**?: Integer location of each tile
- **Initial state**?: Any state can be initial
- **Actions**?: {*Left*, *Right*, *Up*, *Down*}
- **Goal test**?:
- **Path cost**?:

Example: 8-puzzle



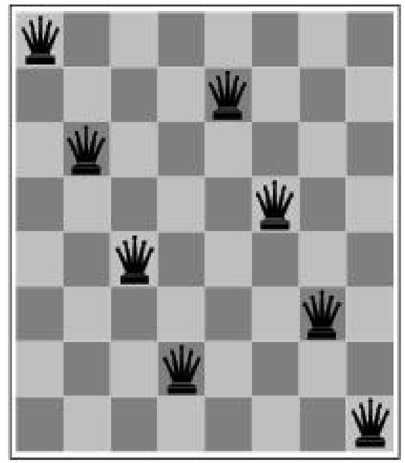
- **States**?: Integer location of each tile
- **Initial state**?: Any state can be initial
- **Actions**?: {*Left*, *Right*, *Up*, *Down*}
- **Goal test**?: goal state (given)
- **Path cost**?:

Example: 8-puzzle



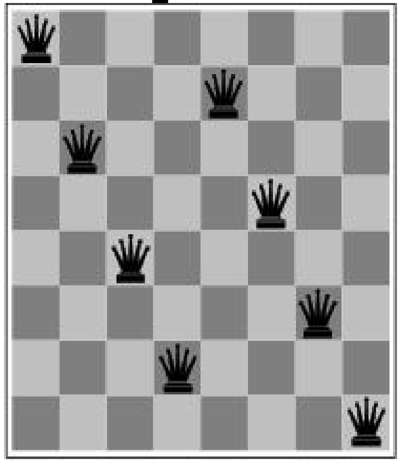
- **States**?: Integer location of each tile
- **Initial state**?: Any state can be initial
- **Actions**?: {*Left*, *Right*, *Up*, *Down*}
- **Goal test**?: goal state (given)
- **Path cost**?: Number of actions to reach goal

Example: 8-queens problem



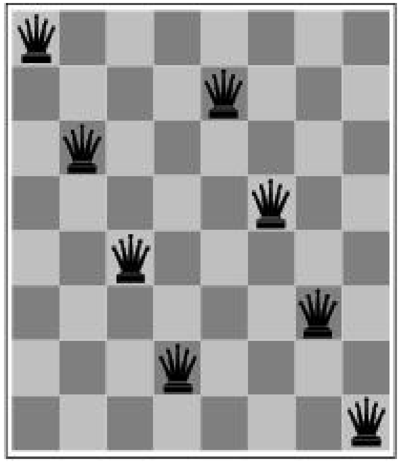
- States?
- Initial state?
- Actions?
- Goal test?
- Path cost?

Example: 8-queens problem



Incremental formulation
- **States**?: Any arrangement of 0 to 8 queens on the board
- **Initial state**?: No queens
- **Actions**?: Add queen in empty square
- **Goal test**?: 8 queens on board and none attacked
- **Path cost**?: None

Example: 8-queens problem



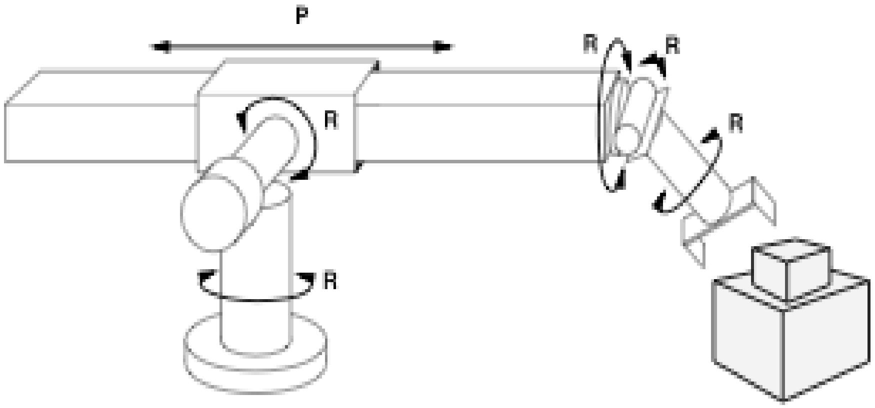
Incremental formulation (alternative)

- **States**?: *n* (0≤ *n*≤ 8) queens on the board, one per column in the *n* leftmost columns with no queen attacking another.
- **Actions**?: Add queen in leftmost empty column such that is not attacking other queens

Problem Space example: Chess

- **Problem space**:
  - **states**: all possible board positions
  - **operators**: the legal moves of chess
- **initial state**: starting board position
- **Goal**: set of all position in which opponent is checkmated

Example: robot assembly



- **States**?: Real-valued coordinates of robot joint angles; parts of the object to be assembled.
- **Initial state**?: Any arm position and object configuration.
- **Actions**?: Continuous motion of robot joints
- **Goal test**?: Complete assembly (without robot)
- **Path cost**?: Time to execute

Basic search algorithms

- How do we find the solutions of previous problems?
  - **Search the state space (remember complexity of space depends on state representation)**
  - Here: search through *explicit tree generation*
    - ROOT= initial state.
    - Nodes and leafs generated through successor function.
  - **In general search generates a graph (same state through multiple paths)**

Problem-solving agent

function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **return** an action
    static: *seq*, an action sequence
        state, some description of the current world state
        goal, a goal
        problem, a problem formulation

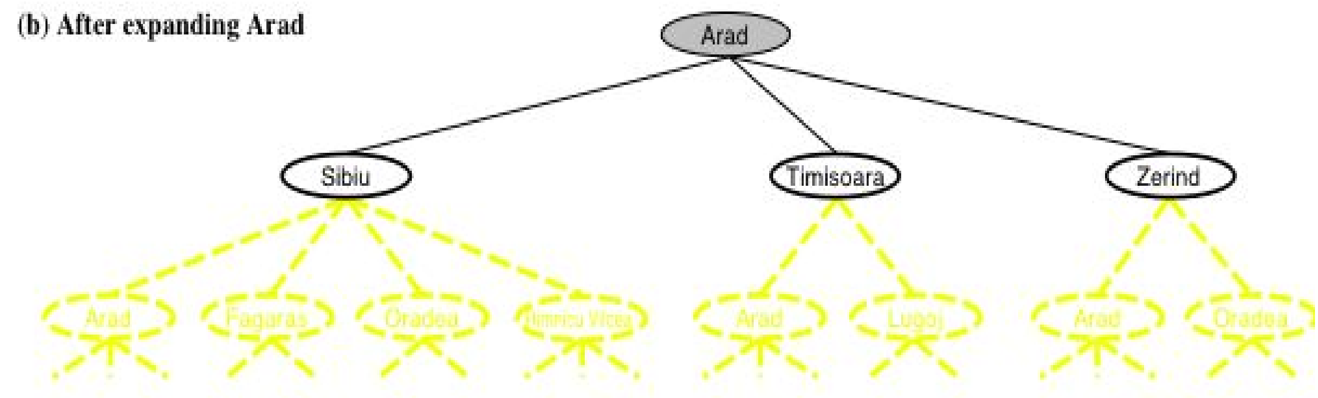
    state ← UPDATE-STATE(*state*, *percept*)
    if *seq* is empty **then**
        goal ← FORMULATE-GOAL(*state*)
        problem ← FORMULATE-PROBLEM(*state*,*goal*)
        seq ← SEARCH(*problem*)
    action ← FIRST(*seq*)
    seq ← REST(*seq*)
    return *action*

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

Simple tree search example



function TREE-SEARCH(*problem, strategy*) **return** a solution or failure

Initialize search tree to the *initial state* of the *problem*

do

if no candidates for expansion **then return** *failure*

<u>choose leaf node for expansion according to *strategy*</u>

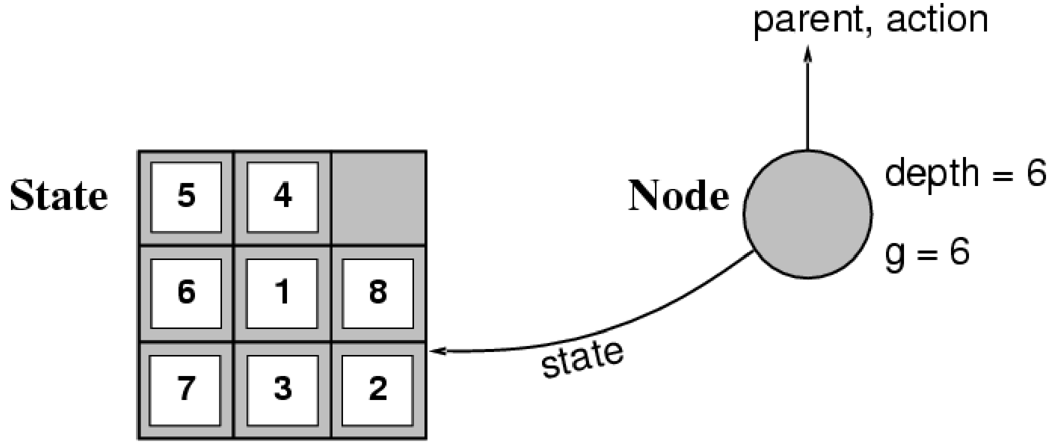
if node contains goal state **then return** *solution*

<u>**else** expand the node and add resulting nodes to the search tree</u>

enddo

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 `SuccessorFn` of the problem to create the corresponding states.

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
```

Search strategies

- A **strategy** is defined by picking the order of node expansion.
- **Problem-solving performance is measured** in four ways:
  - **Completeness**; *Does it always find a solution if one exists?*
  - **Optimality**; *Does it always find the least-cost solution?*
  - **Time Complexity**; *Number of nodes generated/expanded?*
  - **Space Complexity**; *Number of nodes stored in memory during search?*
- Time and space complexity are measured in terms of problem difficulty defined by:
  - *b - maximum branching factor of the search tree*
  - *d - depth of the least-cost solution*
  - *m - maximum depth of the state space (may be ∞)*

Uninformed search strategies

- (a.k.a. **blind search**) use only information available in problem definition.
- Categories defined by expansion algorithm:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search.
  - Bidirectional search

Basic Uninformed Search

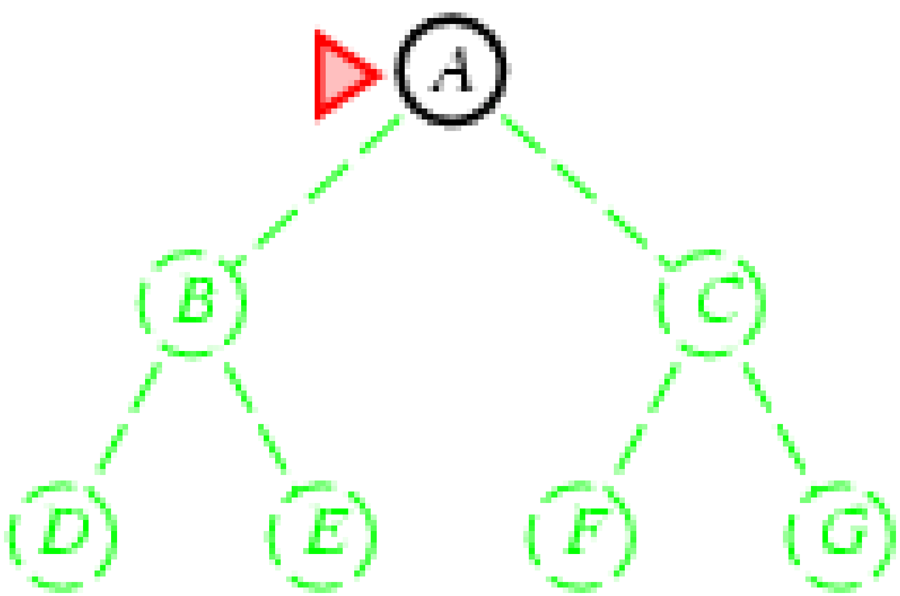
- **Blind methods: search strategies that do not use problem information to guide the search ("uninformed")**
  - **search strategy exhaustively applied until solution found (or failure)**

- **while searching, we normally not wish to visit the same node twice**
  - **represents a cycle - means that infinite looping may occur**
- **– solution: rewrite the network so that loops are removed**

- **search tree: tree in which each node denotes a step in a path from initial goal to target goal**

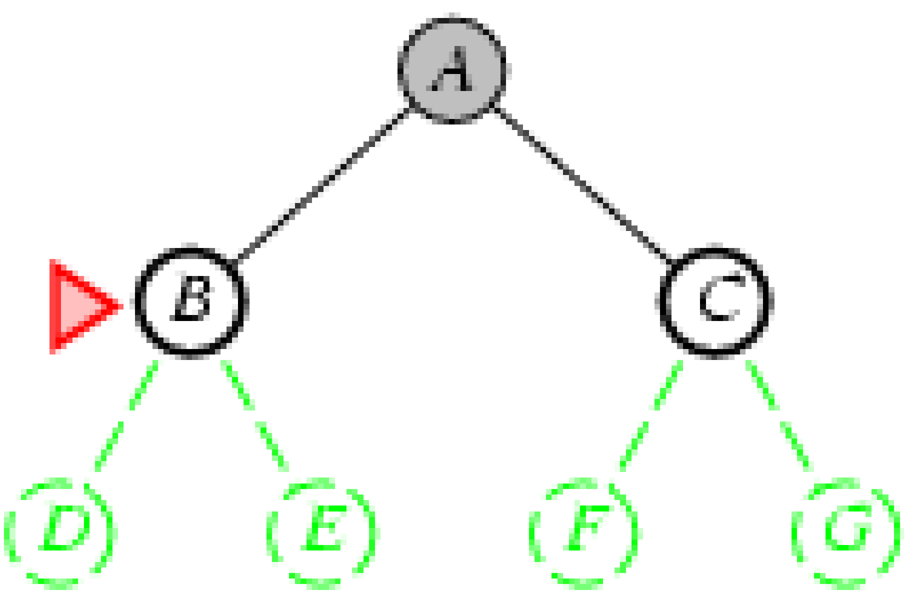
Breadth-first search

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



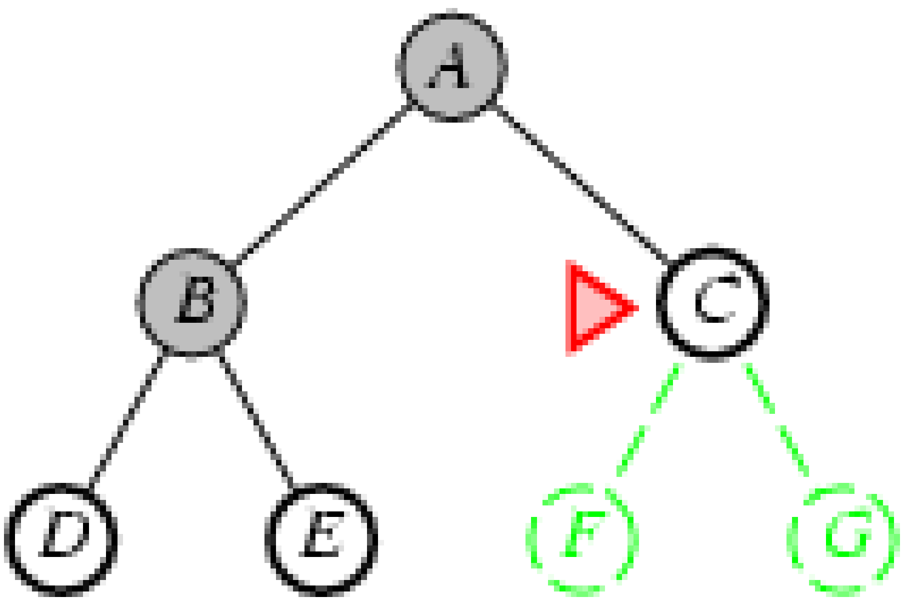
Breadth-first search

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



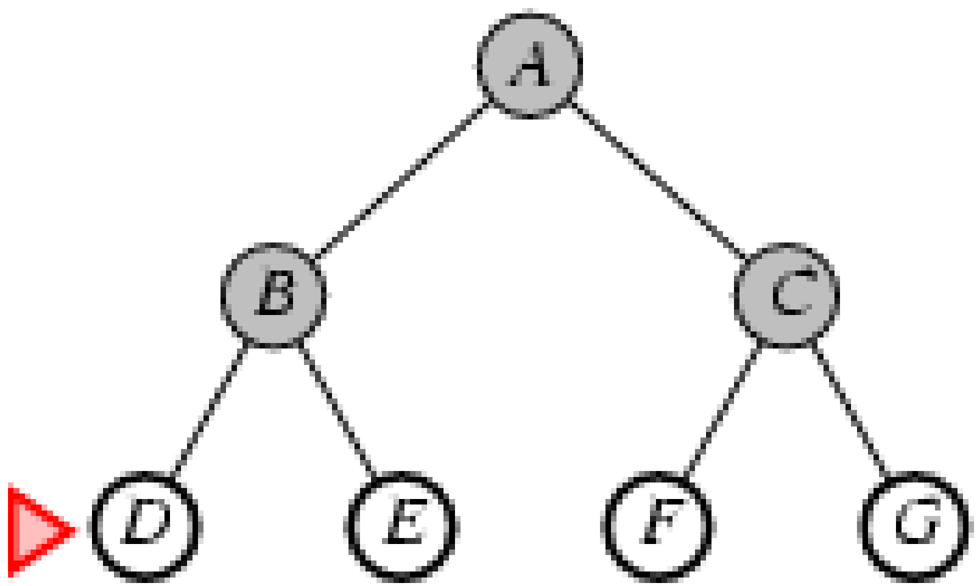
Breadth-first search

- 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 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)

BF-search; evaluation

- Two lessons:
  - Memory requirements are a bigger problem than its execution time.
  - Exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances.

DEPTH2	NODES	TIME	MEMORY
2	1100	0.11 seconds	1 megabyte
4	111100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3523 years	1 exabyte

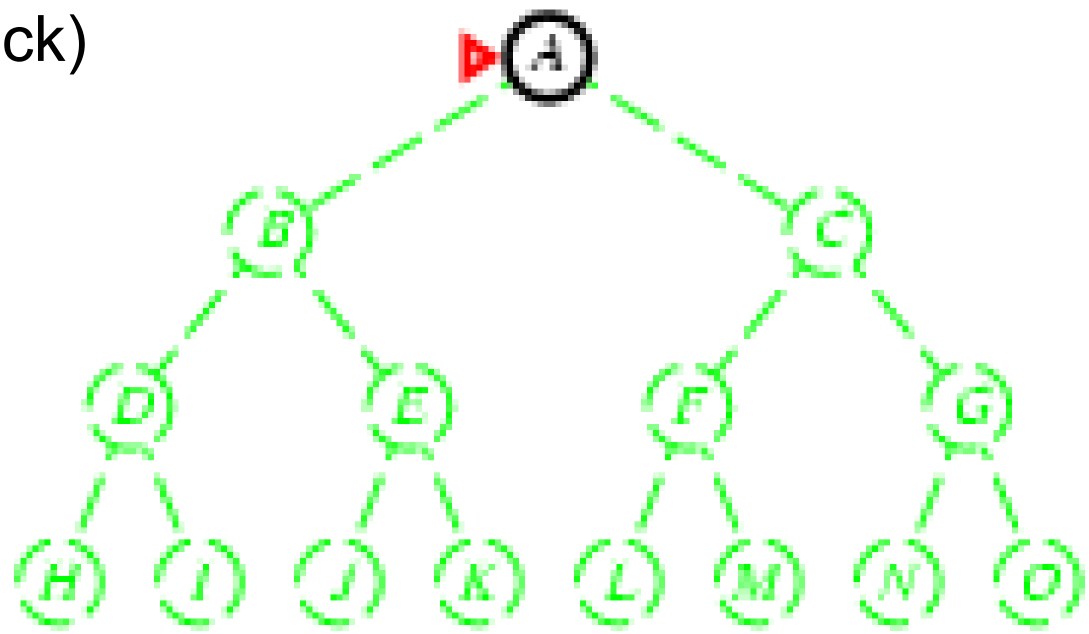
Uniform-cost search

- **Extension of BF-search**:
  - Expand node with *lowest path cost*
- Implementation: *fringe* = queue ordered by path cost.

- UC-search is the same as BF-search when all step-costs are equal.

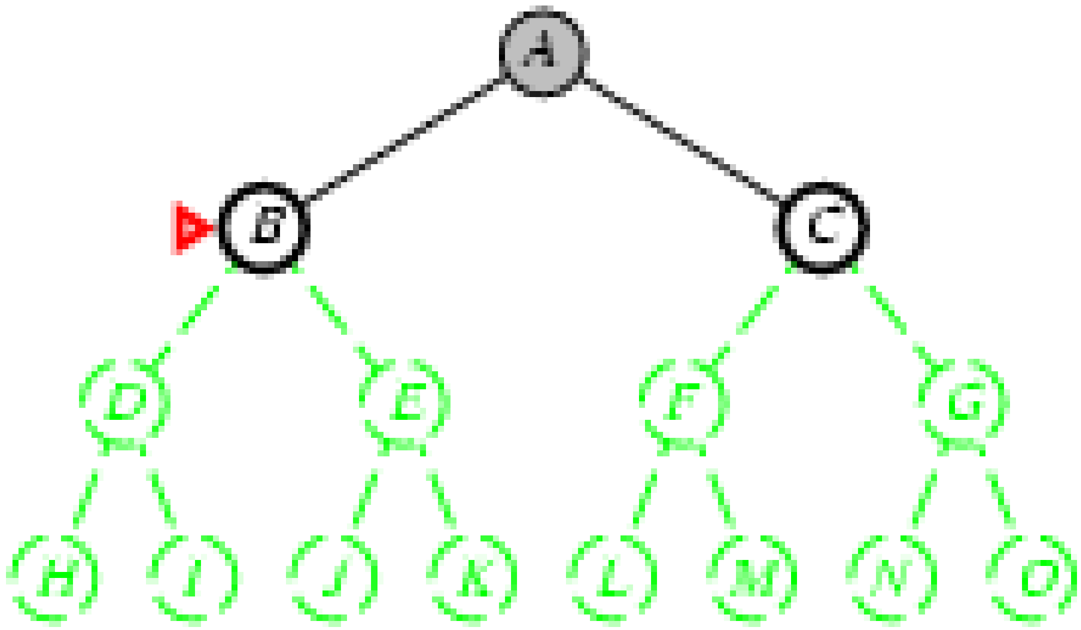
Depth-first search

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



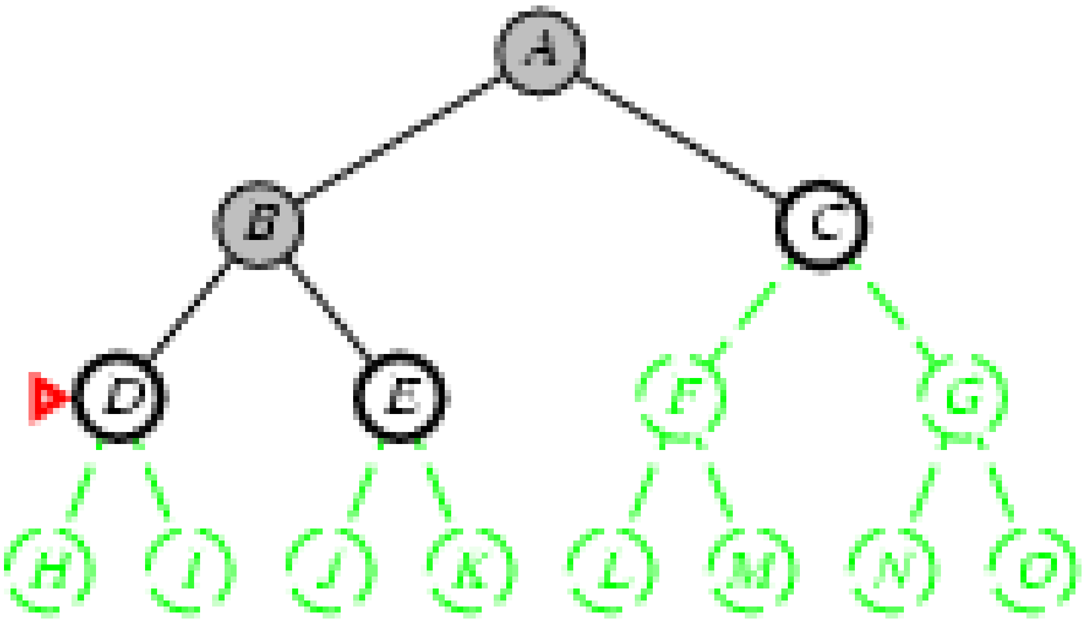
Depth-first search

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



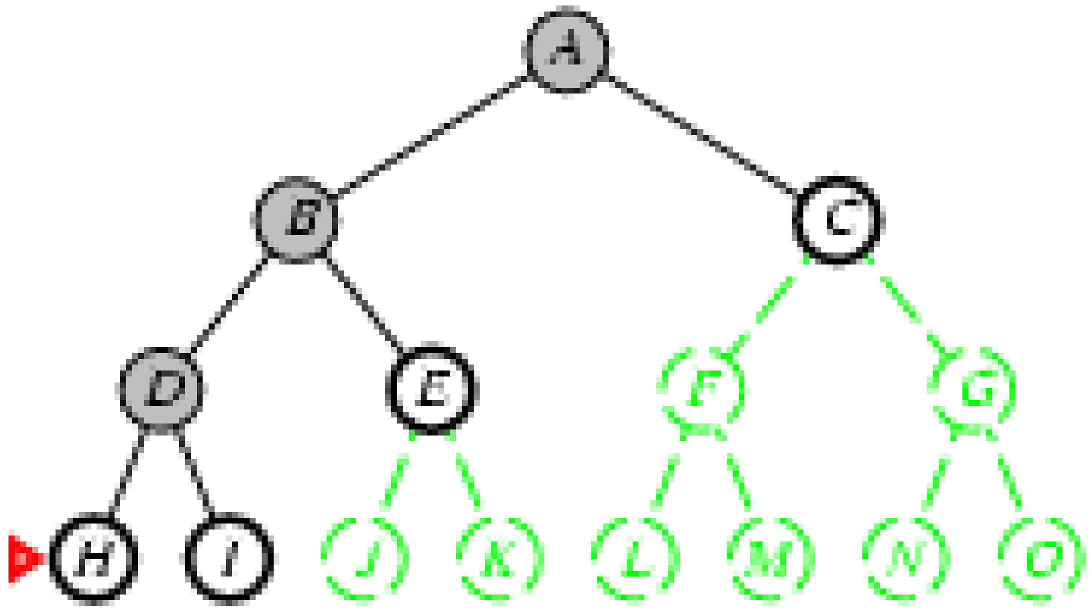
Depth-first search

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



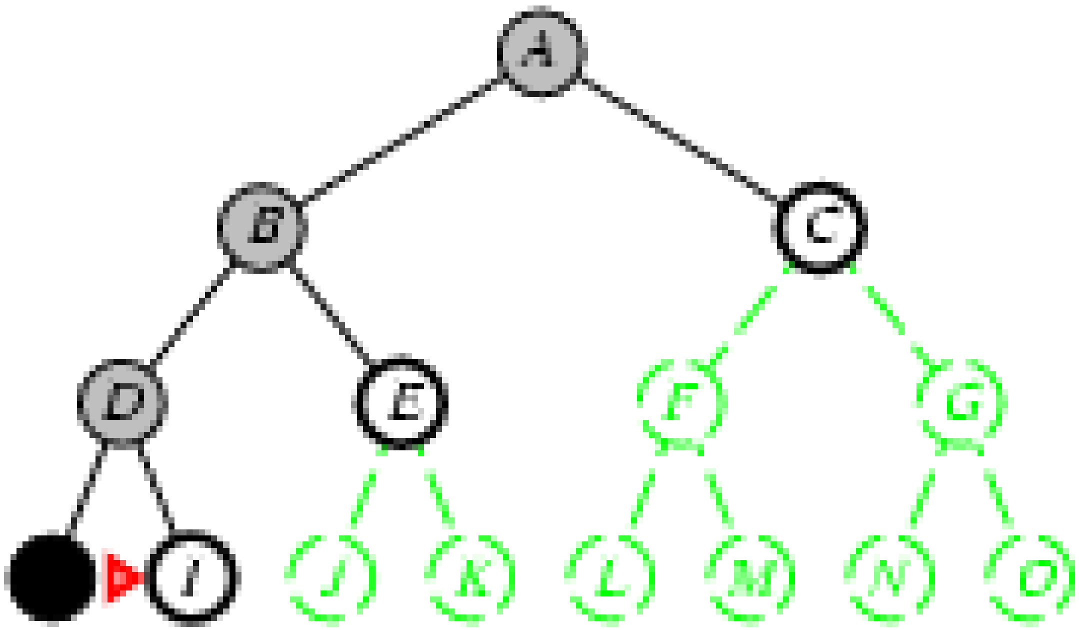
Depth-first search

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



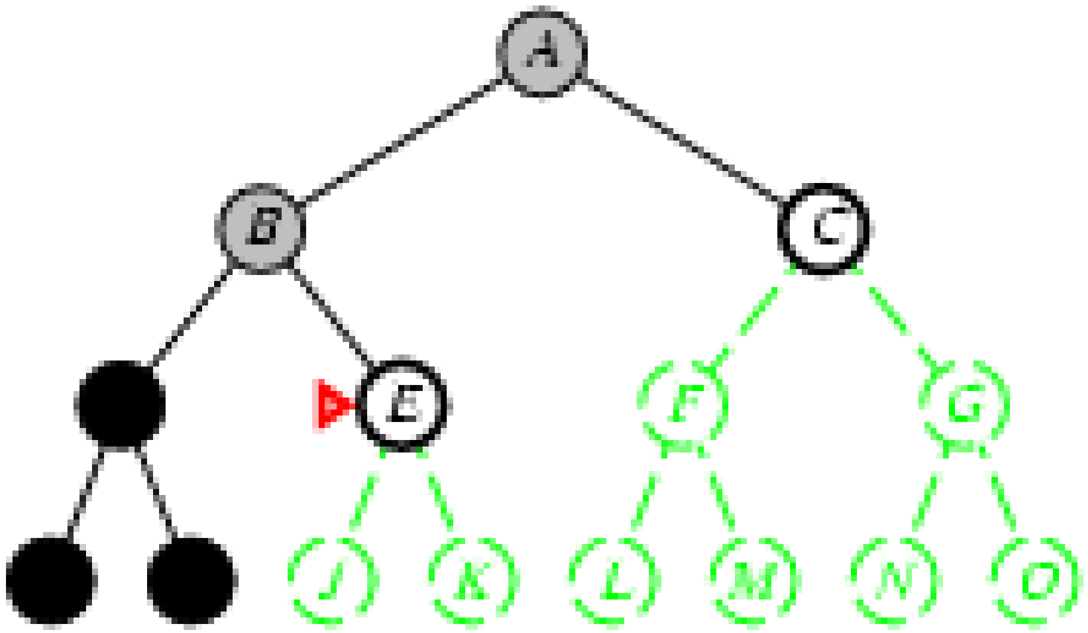
Depth-first search

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



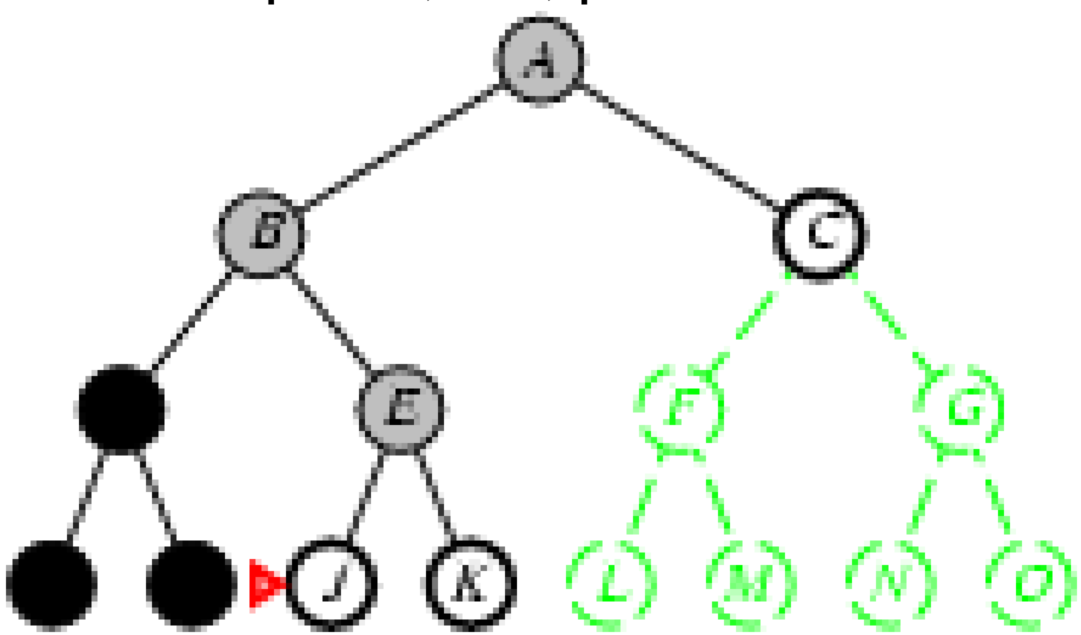
Depth-first search

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



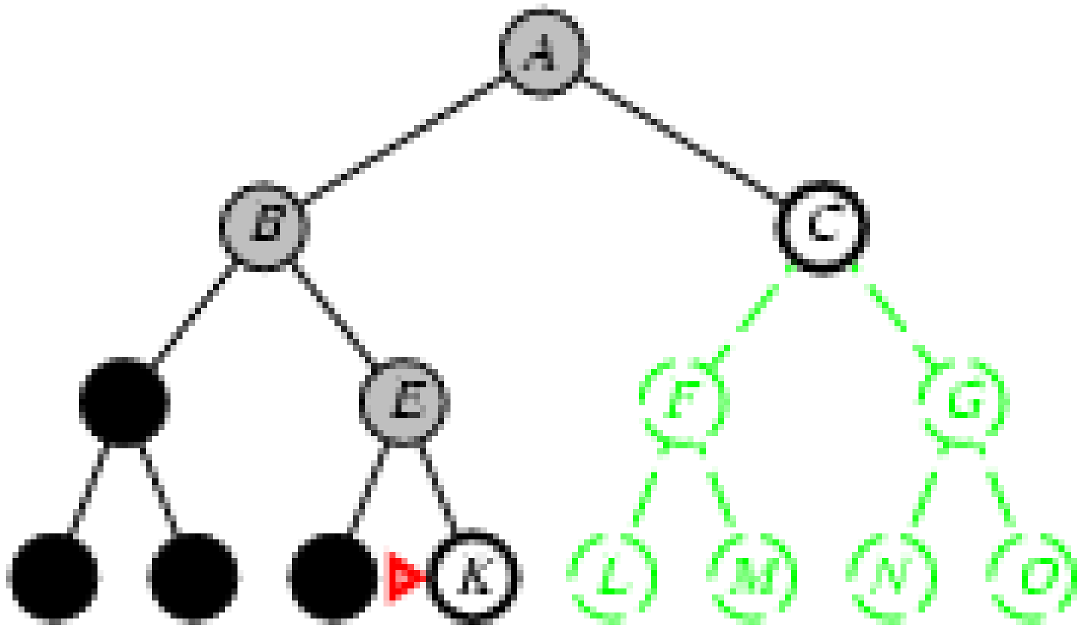
Depth-first search

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



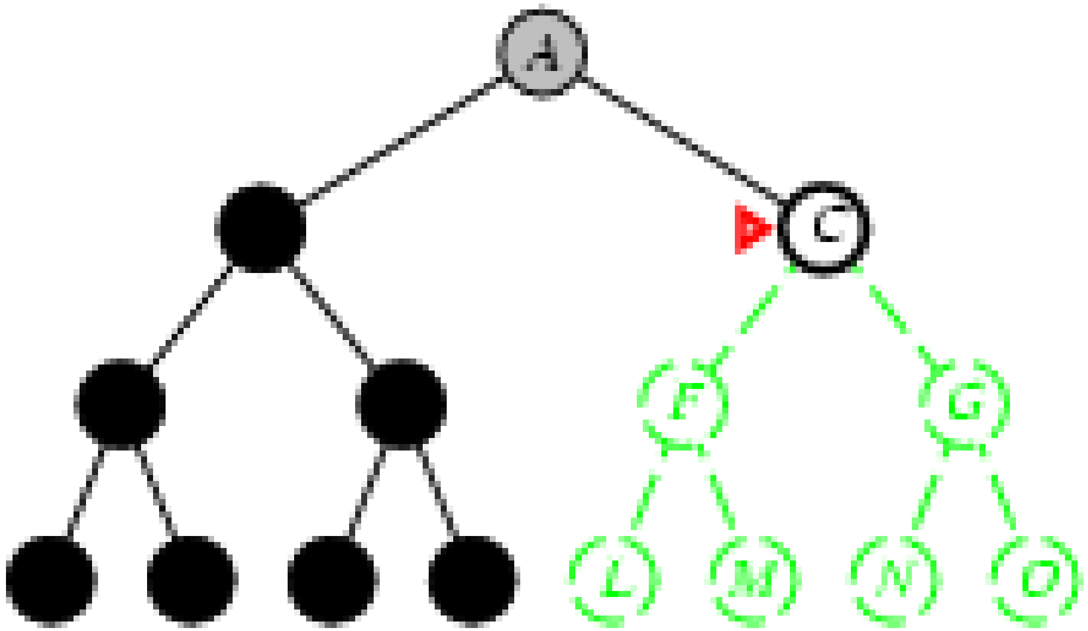
Depth-first search

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



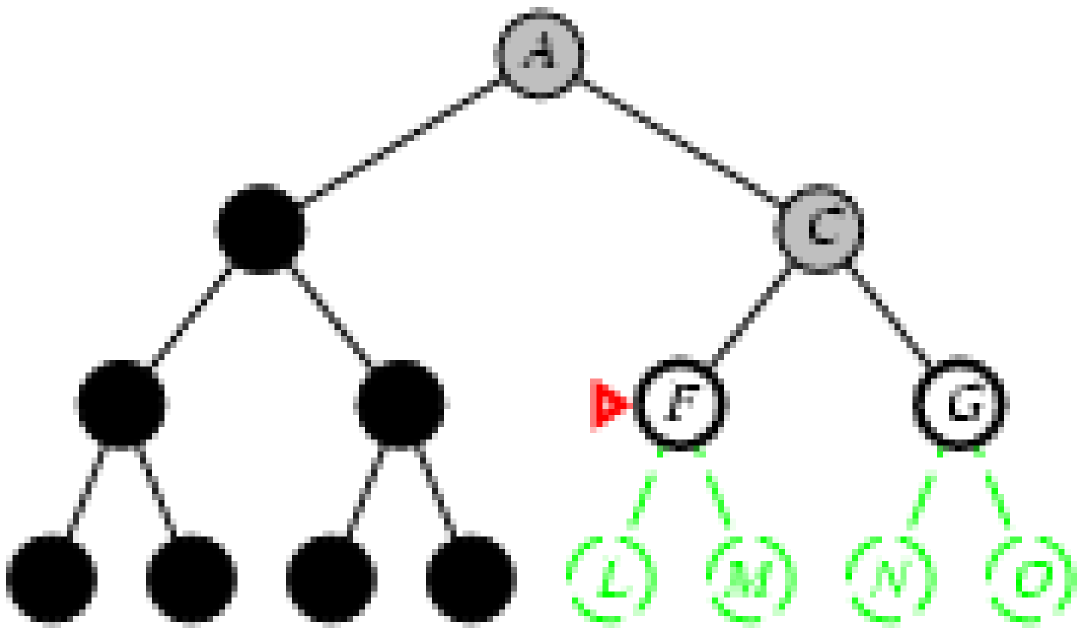
Depth-first search

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



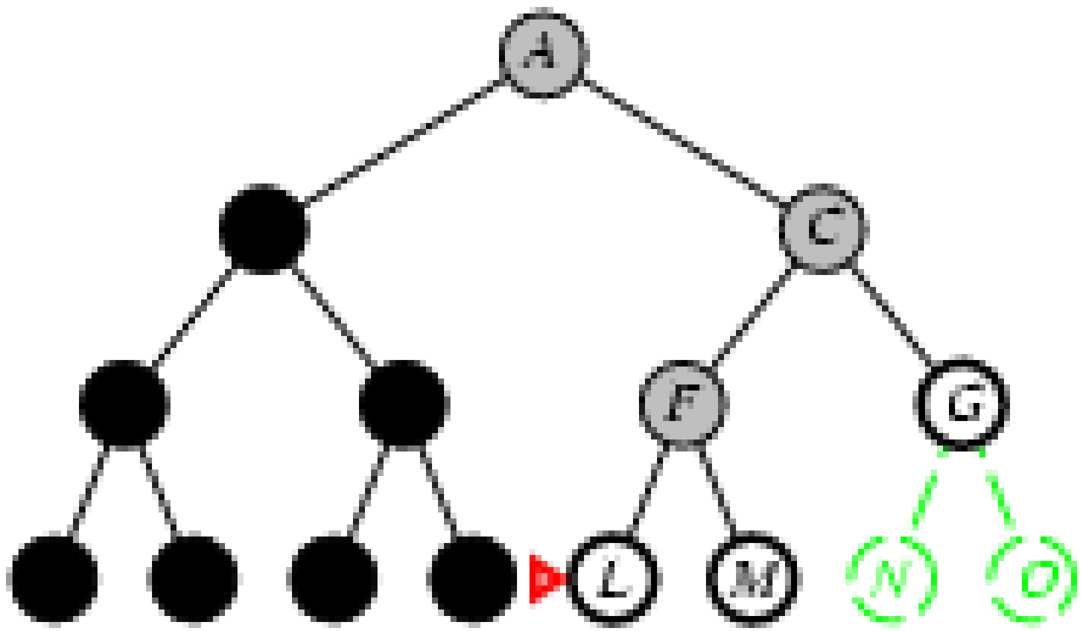
Depth-first search

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



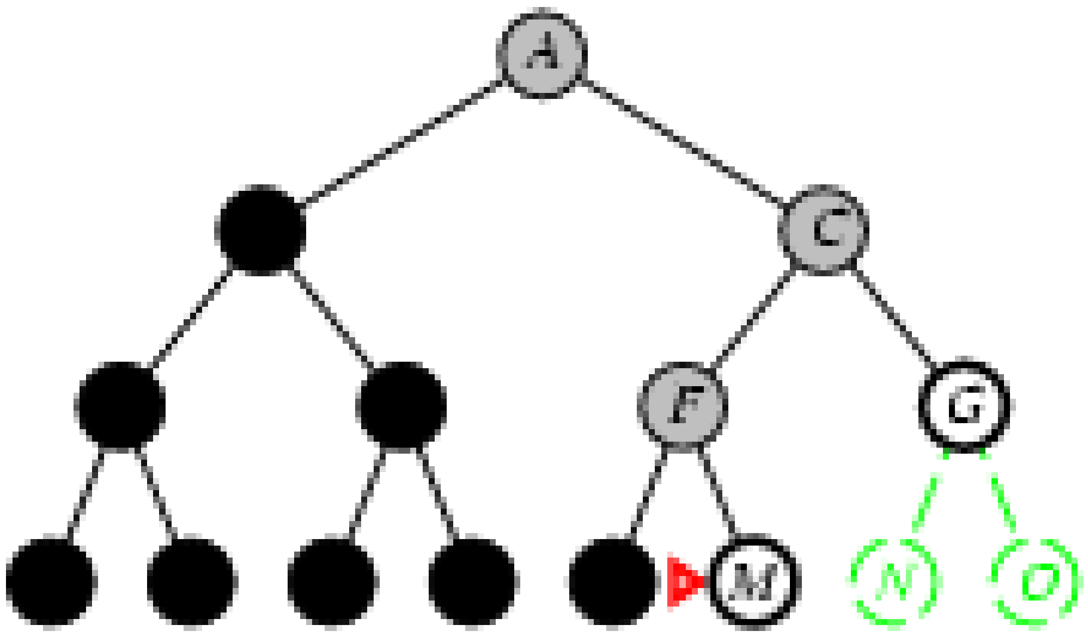
Depth-first search

- 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



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

= depth-first search with depth limit *l*,

i.e., nodes at depth *l* have no successors

- Recursive implementation:

```
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

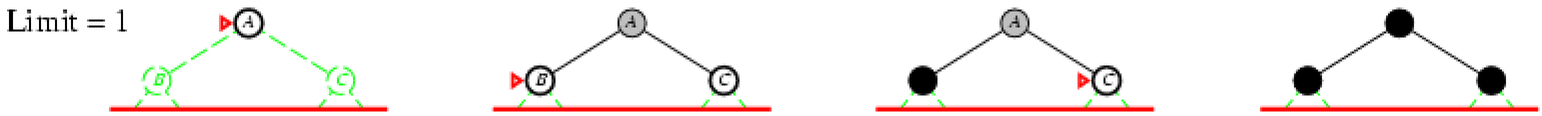
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
    inputs: problem, a problem

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

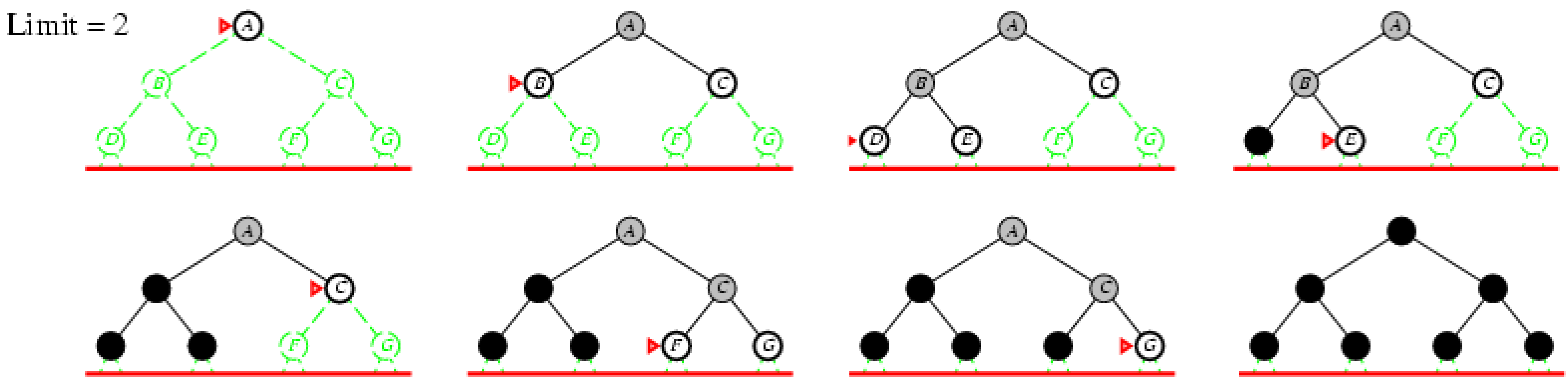
Iterative deepening search *l* =0

Limit = 0

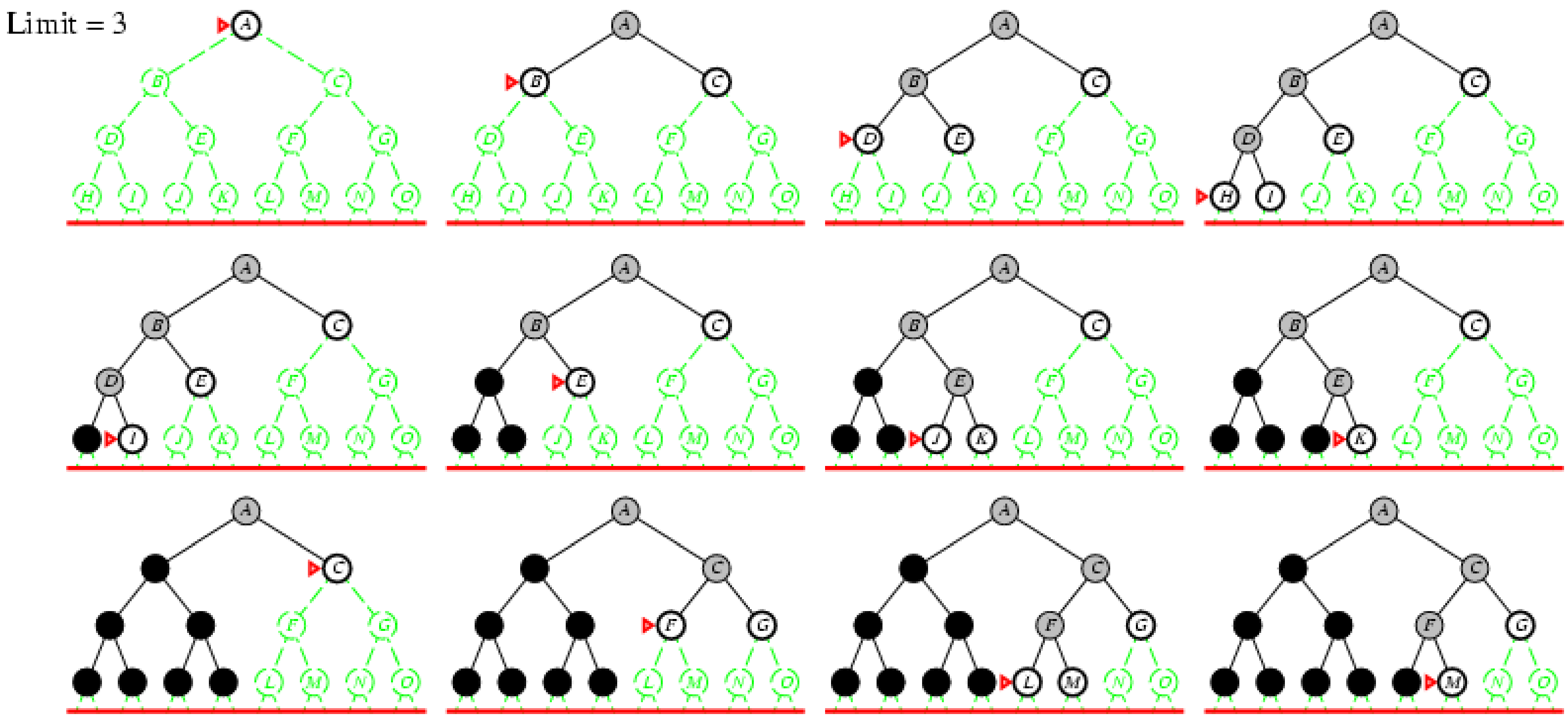
Iterative deepening search *l* =1



Iterative deepening search *l* =2



Iterative deepening search *l* =3



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%

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

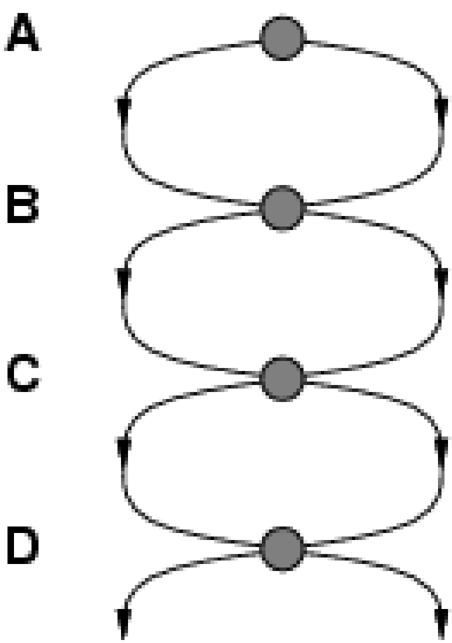
Summary of algorithms

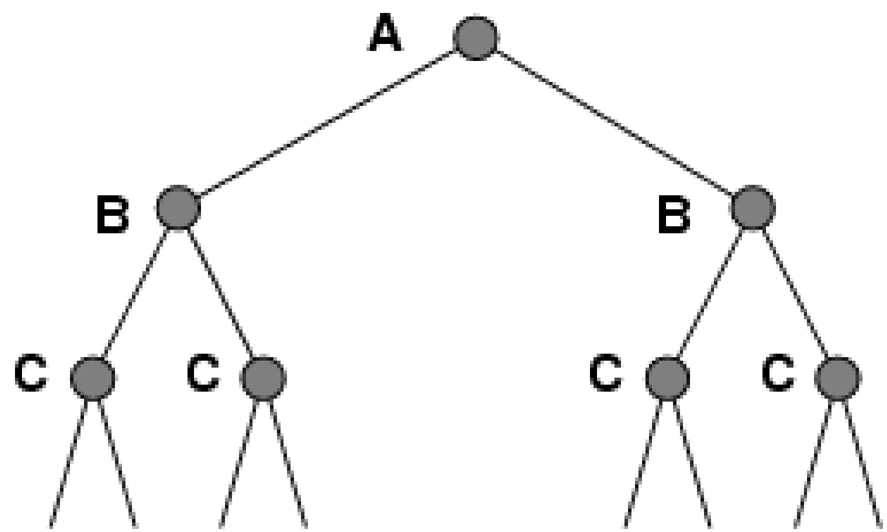
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Avoiding Repeated states

(Read Section 3.5 (class textbook)

- 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

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms