Search and Sequential Action

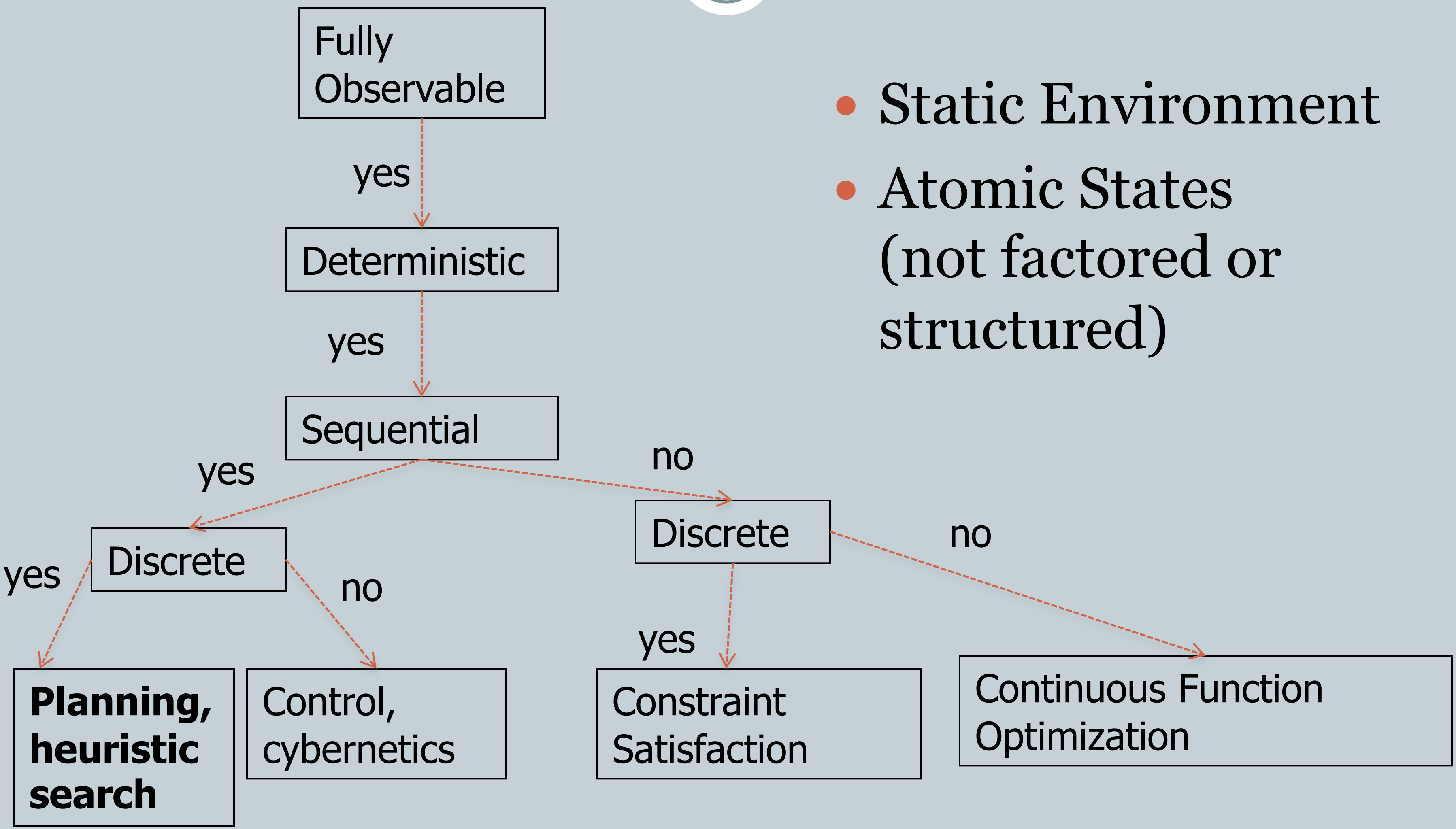
CHAPTER 3

Oliver Schulte
Simon Fraser University

Outline

- Problem formulation: representing sequential problems.
- Example problems.
- Planning for solving sequential problems without uncertainty.
- Basic search algorithms

Environment Type Discussed In this Lecture



- Static Environment
- Atomic States (not factored or structured)

Search Problems

Choice in a Deterministic Known Environment

- Without uncertainty, choice is trivial in principle: choose what you know to be the best option.
- Trivial if the problem is represented in a look-up table.

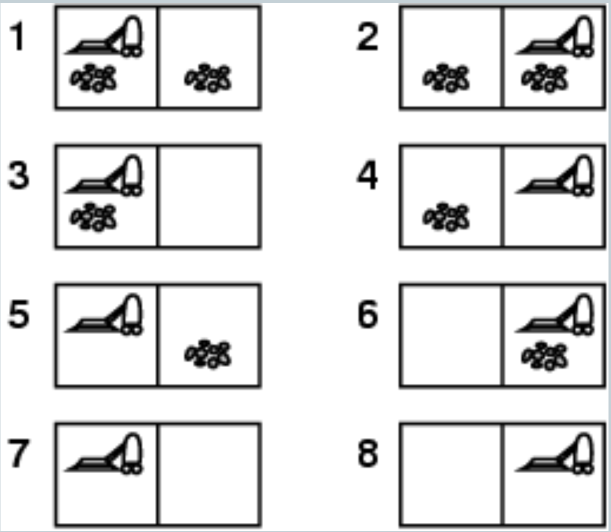
Option	Value
Chocolate	10
Wine	20
Book	15

Computational Choice Under Certainty

- But choice can be *computationally* hard if the problem information is represented differently.
- Options may be **structured** and the best option needs to be constructed.
  - E.g., an option may consist of a path, sequence of actions, plan, or strategy.
- The value of options may be given **implicitly** rather than explicitly.
  - E.g., cost of paths need to be computed from map.

Sequential Action Example

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
  - Vacuum world → everything observed
  - Romania → The full map is observed



- Single-state: Start in #5. Solution??
  - [Right, Suck]

Example: Romania

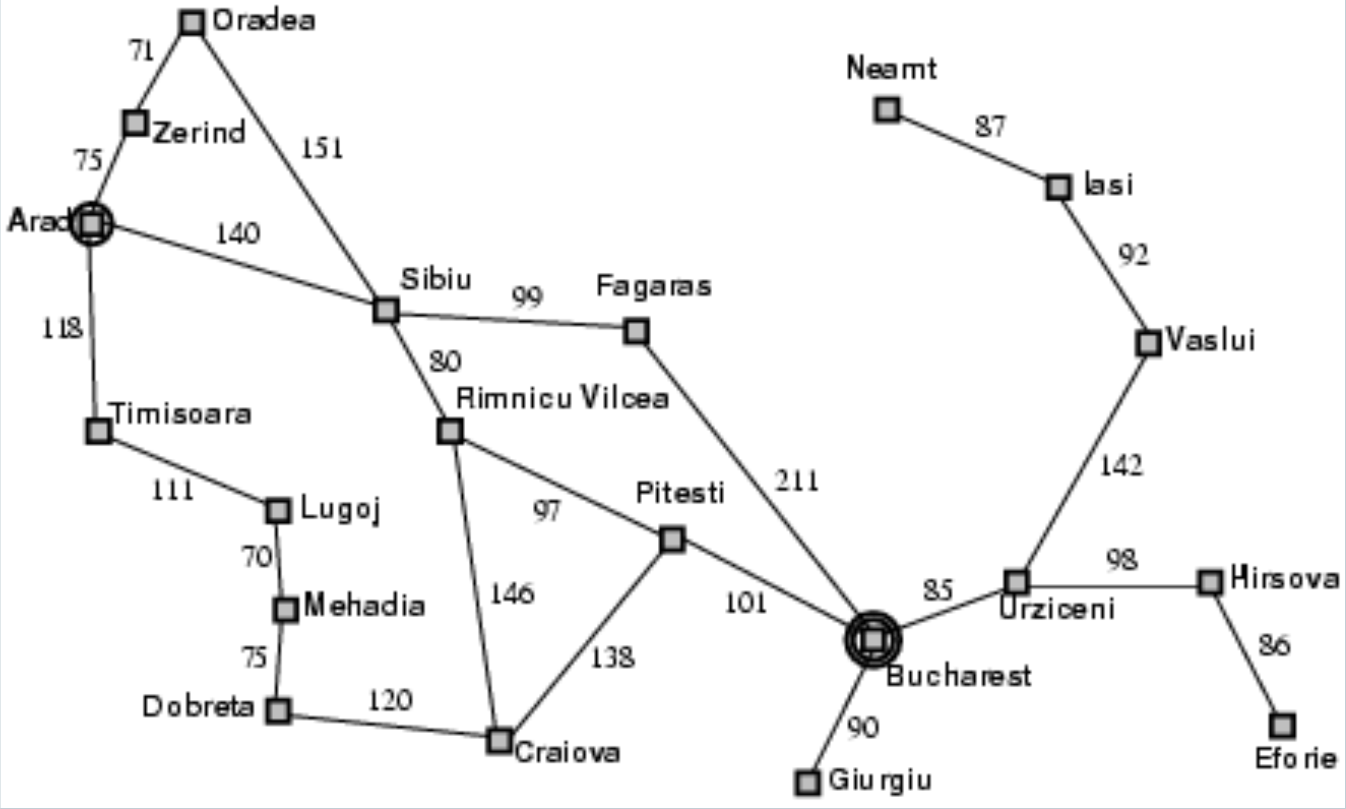
- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Buchares

1. Formulate goal: be in Bucharest
2. Formulate problem:
    - states: various cities
    - actions: drive between cities

- Find solution:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Abstraction: The process of removing details from a representation
Is the map a good representation of the problem? What is a good replacement?

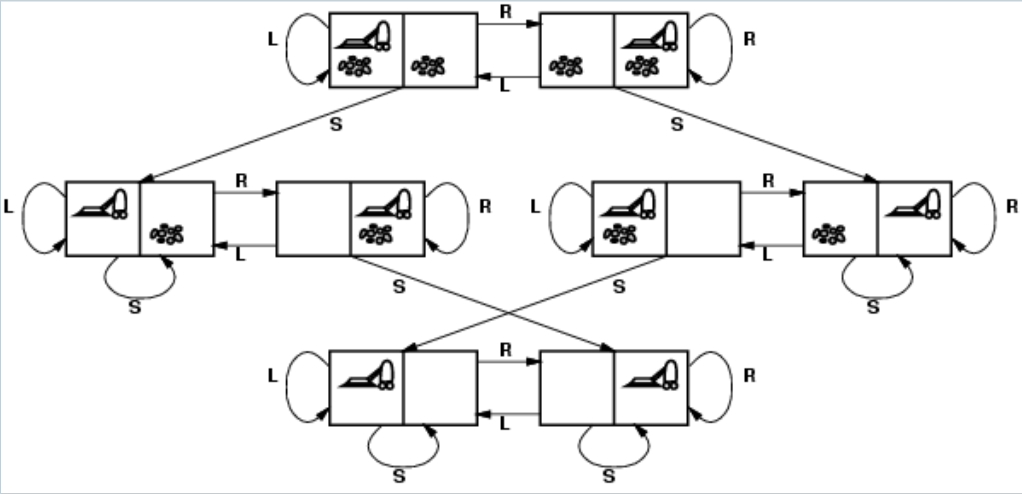
General problem formulation

A problem is defined by four items:

1. initial state e.g., "at Arad“
2. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad) = {<Arad → Zerind, Zerind>, … }*
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost, assumed to be ≥ 0

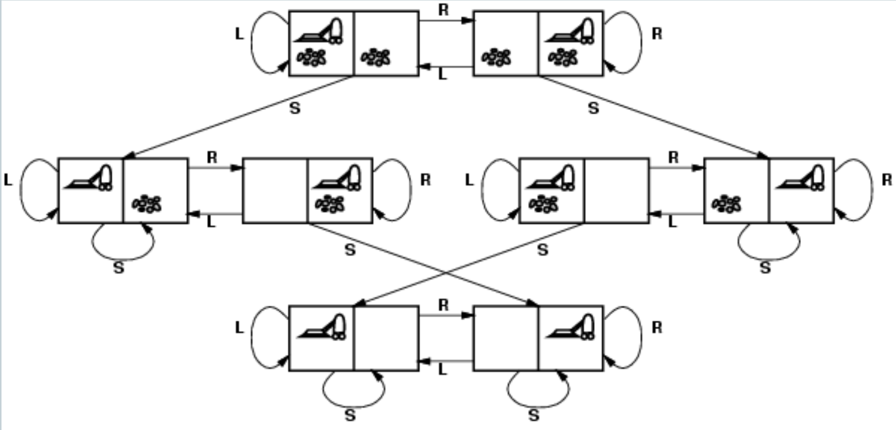
- A solution is
  - A sequence of actions leading from the initial state to a goal state
  - A sequence of actions is called a **plan**

Vacuum world state space graph



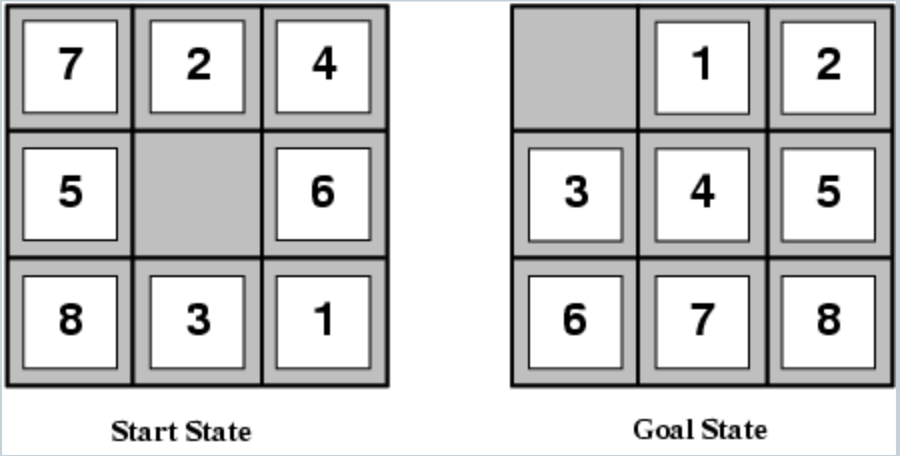
- states?
- actions?
- goal test?
- path cost?

Vacuum world state space graph



- <u>states?</u> integer dirt and robot location
- <u>actions?</u> *Left*, *Right*, *Suck*
- <u>goal test?</u> no dirt at all locations
- <u>path cost?</u> 1 per action

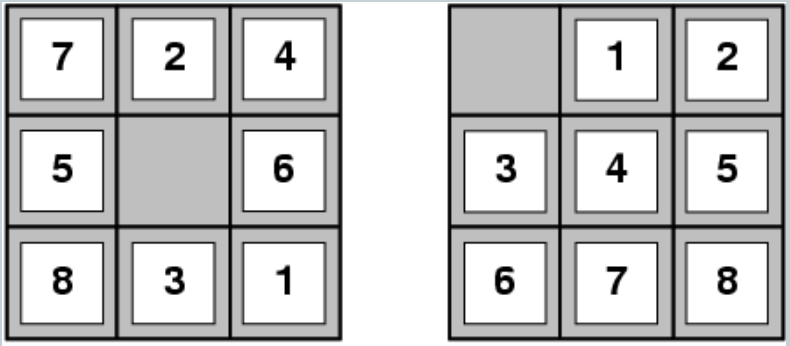
Example: The 8-puzzle



- states?
- actions?
- goal test?
- path cost?

Example: The 8-puzzle

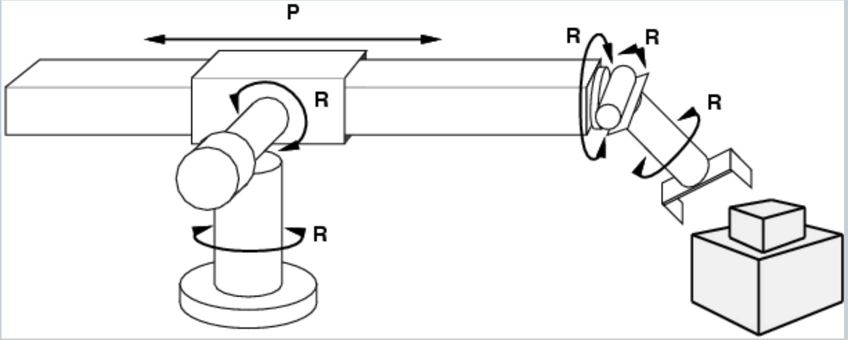
On-line Version



Start State Goal State

- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

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

Problem Solving Algorithms

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**
    goal ← FORMULATE-GOAL(*state*)
    problem ← FORMULATE-PROBLEM(*state*, *goal*)
    seq ← SEARCH(*problem*)
  action ← FIRST(*seq*)
  seq ← REST(*seq*)
  return *action*

Note: this is offline problem solving; solution executed “eyes closed.”

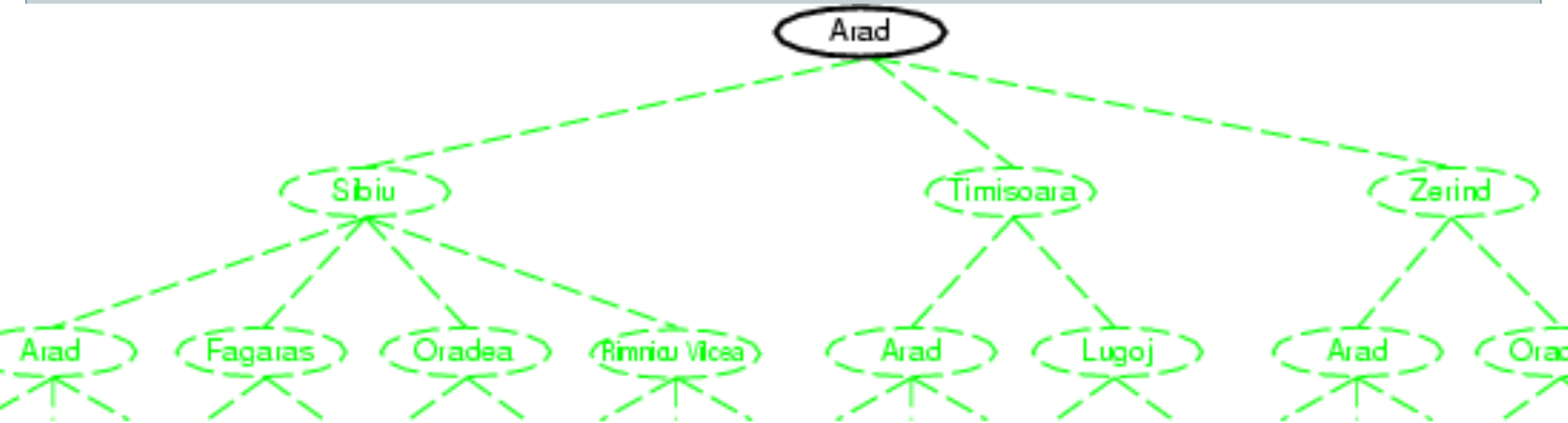
Tree search algorithms

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

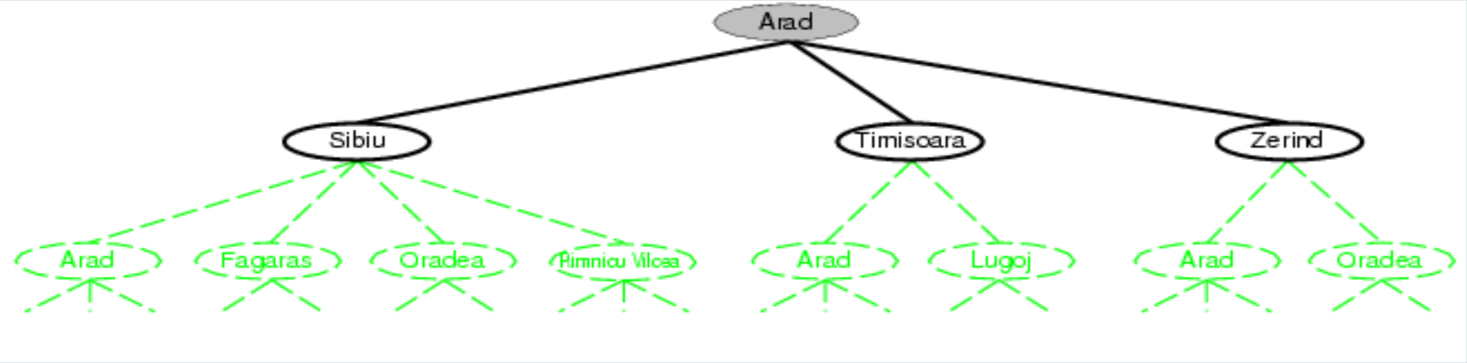
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the 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

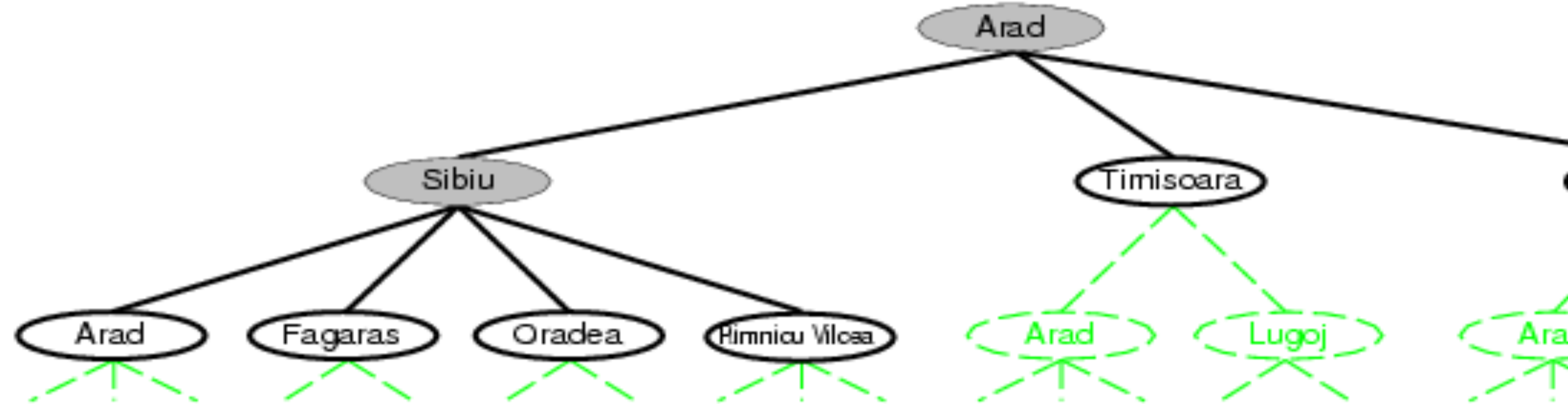
- The space of sequences can be arranged as a tree
- The search tree is a theoretical construct, not actually built



Tree search example



Tree search example



Search Graph vs. State Graph

- Be careful to distinguish
  - Search tree: Nodes are **sequences of actions.**
    - The search tree never contains a cycle.
  - State Graph: Nodes are **states of the environment**.
    - The state graph may contain a cycle.
  - *Node in Search Tree = Path in State Graph*
- Demo: http://aispace.org/search/

Evaluating Search Strategies

- A search strategy is defined by picking the order of path expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if 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 ∞)

Search Strategies

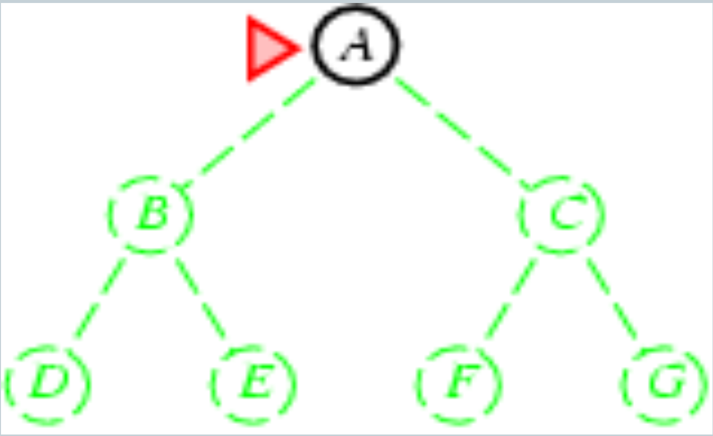
BREADTH-FIRST
DEPTH-FIRST
ITERATED DEEPENING

Uninformed search strategies

- Uninformed search strategies
  - use only the information available in the problem definition
  - No domain knowledge or expertise
- Breadth-first search
- Depth-first search
- Depth-limited search
- Iterative deepening search

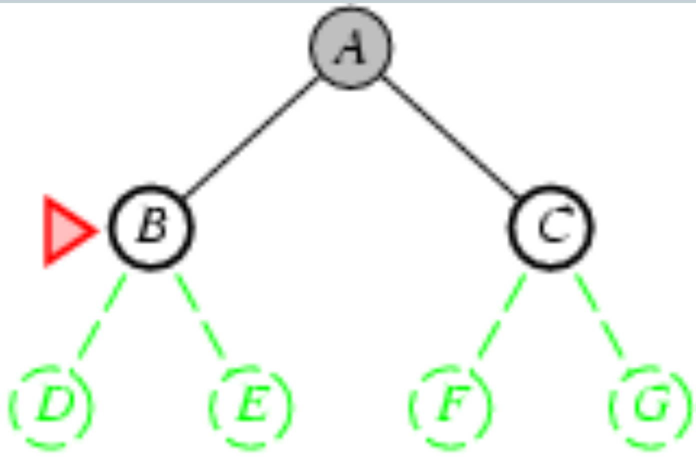
Breadth-first search

- Expand shortest paths
- **Frontier** = set of but generated paths
- Frontier = leaf nodes in the search tree.
- Implementation:
  - *Frontier* is a FIFO queue, i.e., new successors go at end



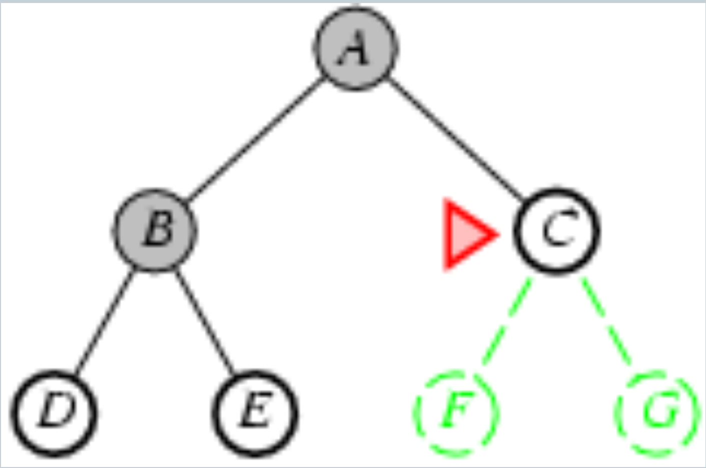
Breadth-first search

- Expand shortest paths
- Implementation:
  - *frontier* is a FIFO queue, i.e., new successors go at end



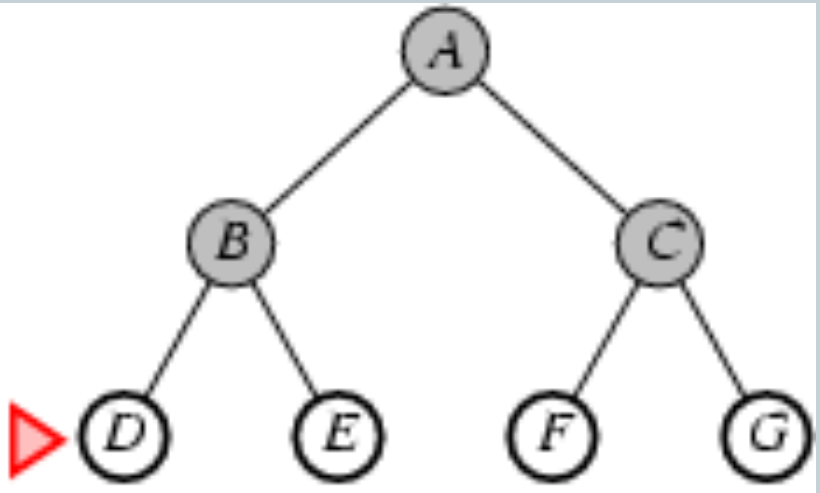
Breadth-first search

- Expand shortest paths
- Implementation:
  - *frontier* is a FIFO queue, i.e., new successors go at end



Breadth-first search

- Expand shortest paths
- http://aispace.org/search/
- Implementation:
  - *frontier* is a FIFO queue, i.e., new successors go at end



Properties of breadth-first search

- Complete? Time? Space?Optimal?
- 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

Example Numbers

Assumptions

Branching Factor b	Node Generation/sec	Node size
10	1M	1MB

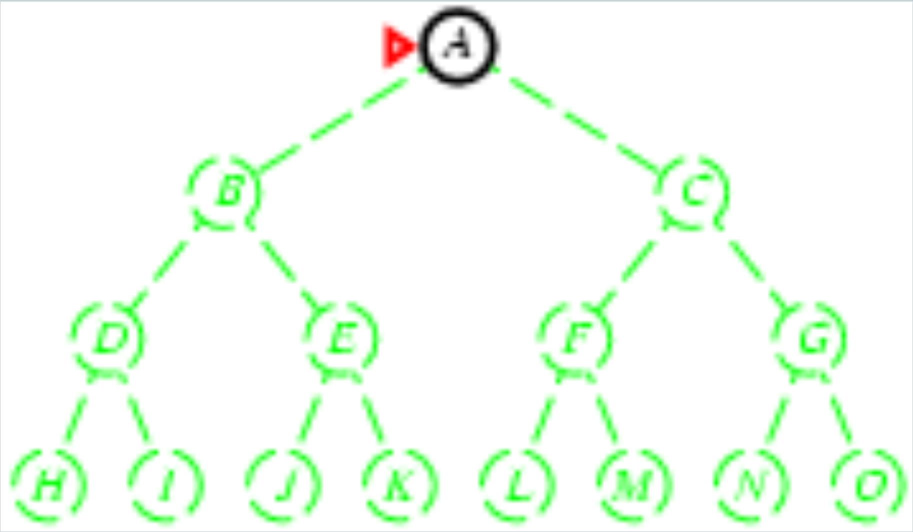
Resource Consumption

Depth	Nodes	Time	Memory
10	10^{10}	3 hours	10 TeraBytes
12	10^{12}	13 days	1 petabyte

Space is the big problem (more than time)

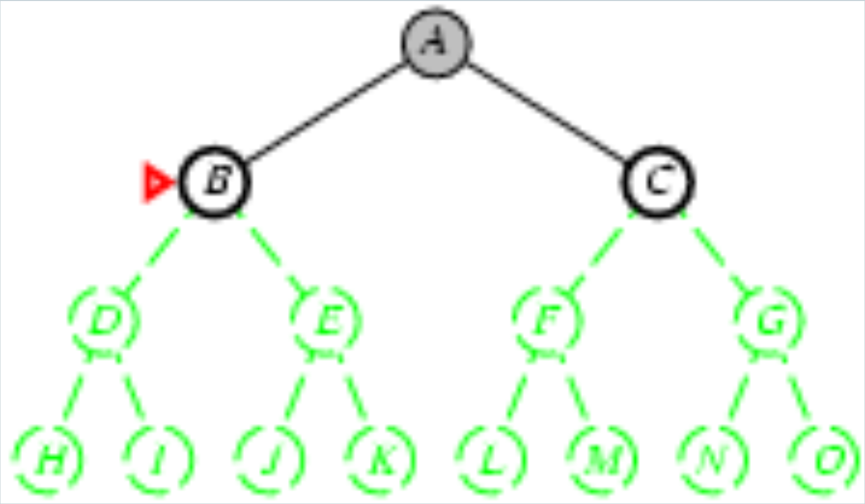
Depth-first search

- Expand longest paths
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



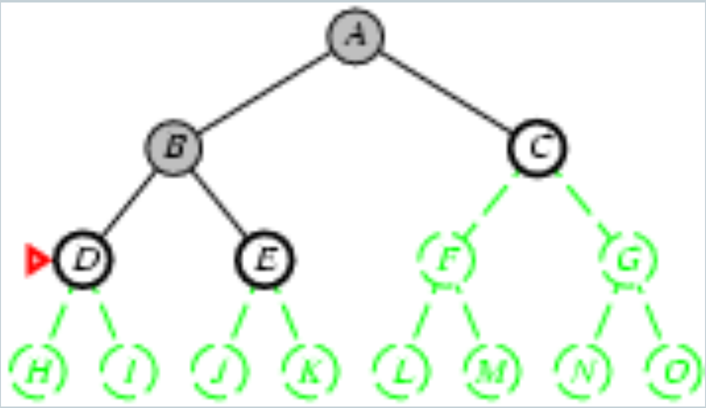
Depth-first search

- Expand longest path
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



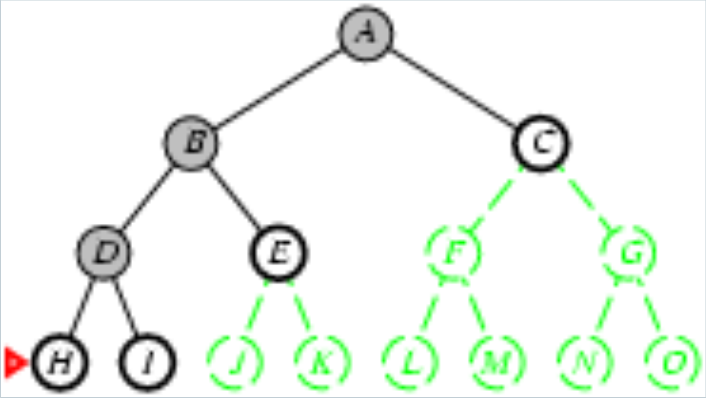
Depth-first search

- Expand longest path
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



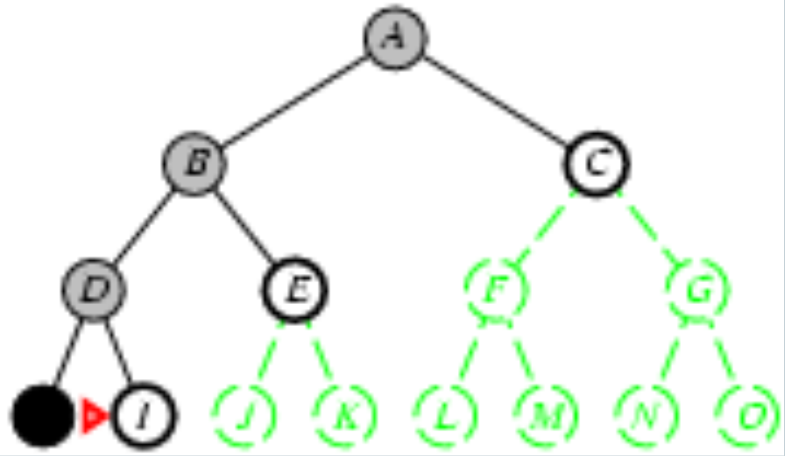
Depth-first search

- Expand longest path
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



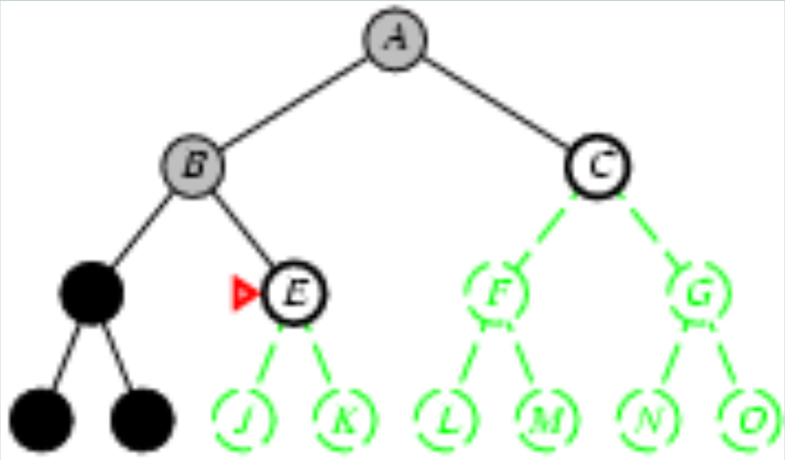
Depth-first search

- Expand longest path
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



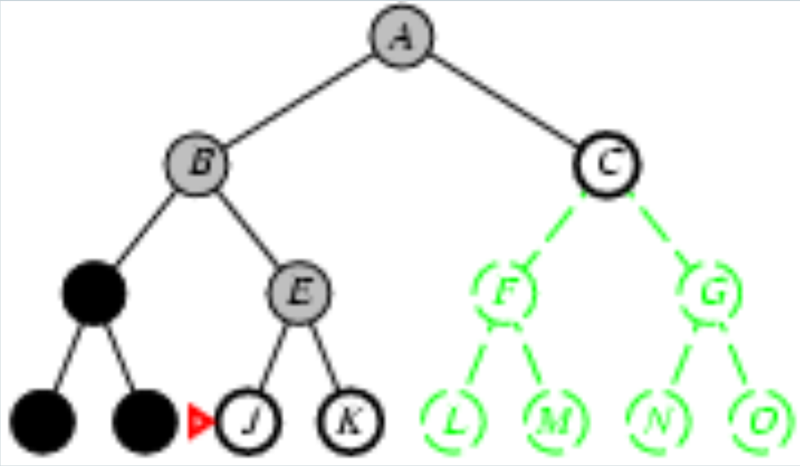
Depth-first search

- Expand longest path
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



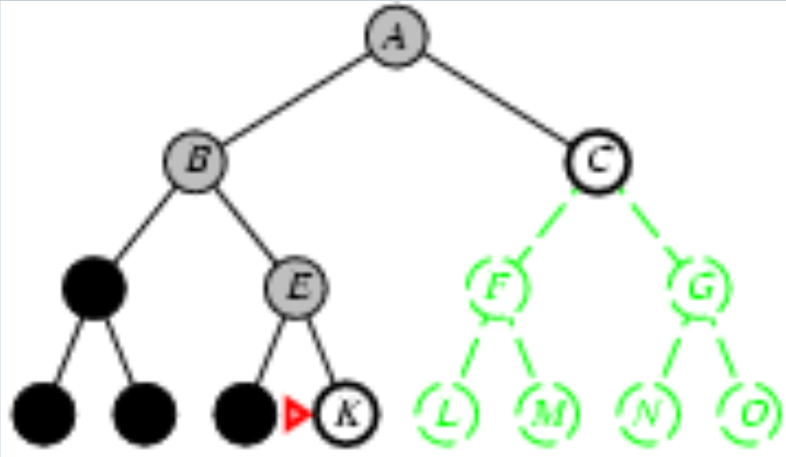
Depth-first search

- Expand longest path
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



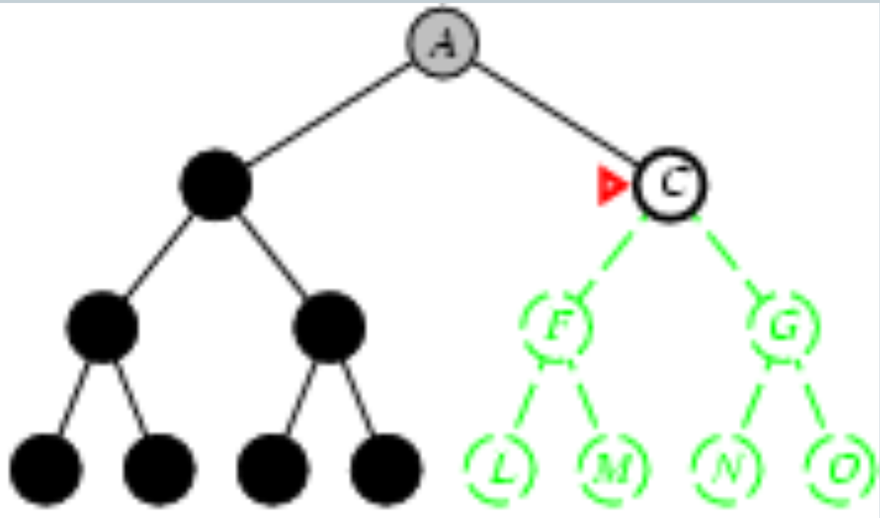
Depth-first search

- Expand longest path
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



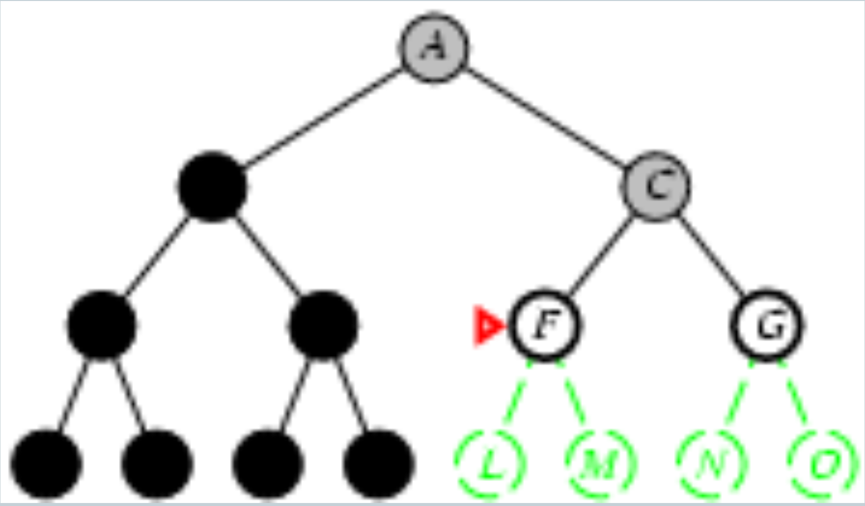
Depth-first search

- Expand longest path
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



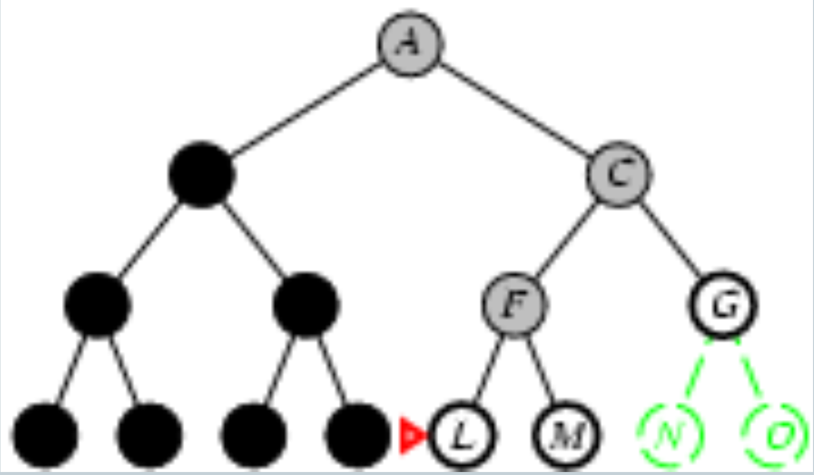
Depth-first search

- Expand longest path
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



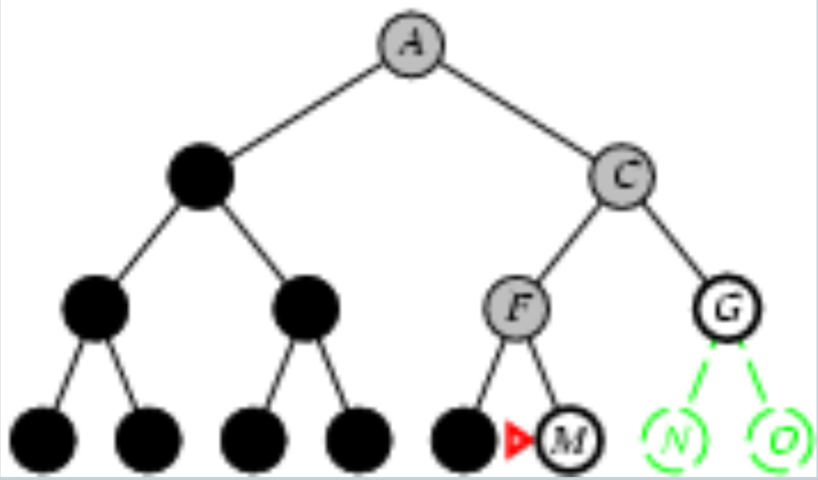
Depth-first search

- Expand longest path
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



Depth-first search

- Expand longest path
- http://aispace.org/search/
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front



Properties of depth-first search

- Complete? Time? Space?Optimal?
- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path (graph search)
  - → complete in finite spaces
- Time? $O(b^m)$: terrible if maximum depth m is much larger than solution depth d
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space! Store single path with unexpanded siblings.
  - Seems to be common in animals and humans.
- Optimal? No.
- Important for exploration (on-line search).

Depth-limited search

- depth-first search with depth limit *l*,
  - i.e., paths at depth *l* have no successors
  - Solves infinite loop problem
- Common AI strategy: let user choose search/resource bound.
- Complete? No if l < d:
- Time? $O(b^l)$ = complete tree up to depth l.
- Space? $O(bl)$, i.e., linear space!
- Optimal? No if l > d (solution needs more than limit)

Iterative deepening search

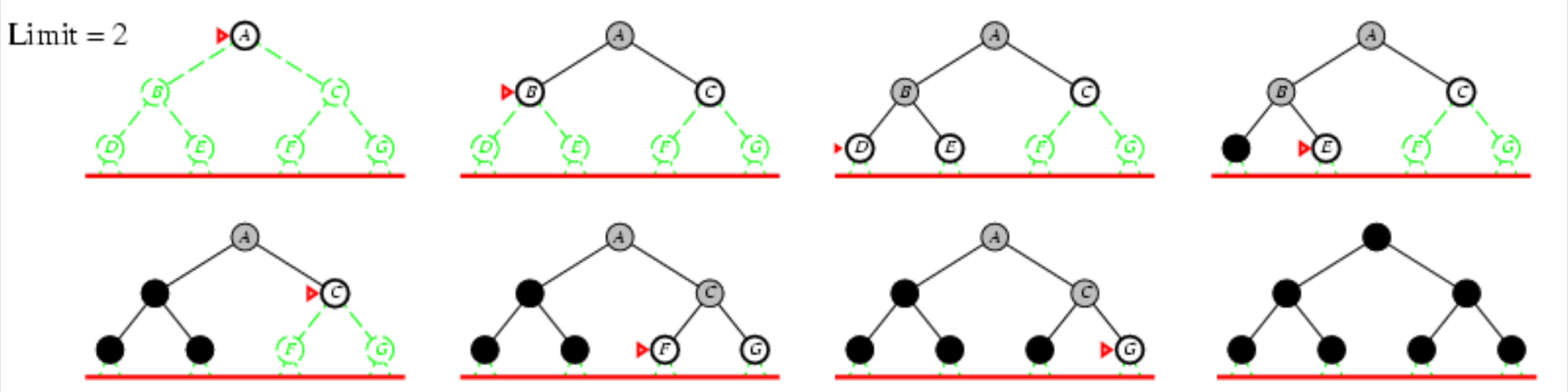

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem

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

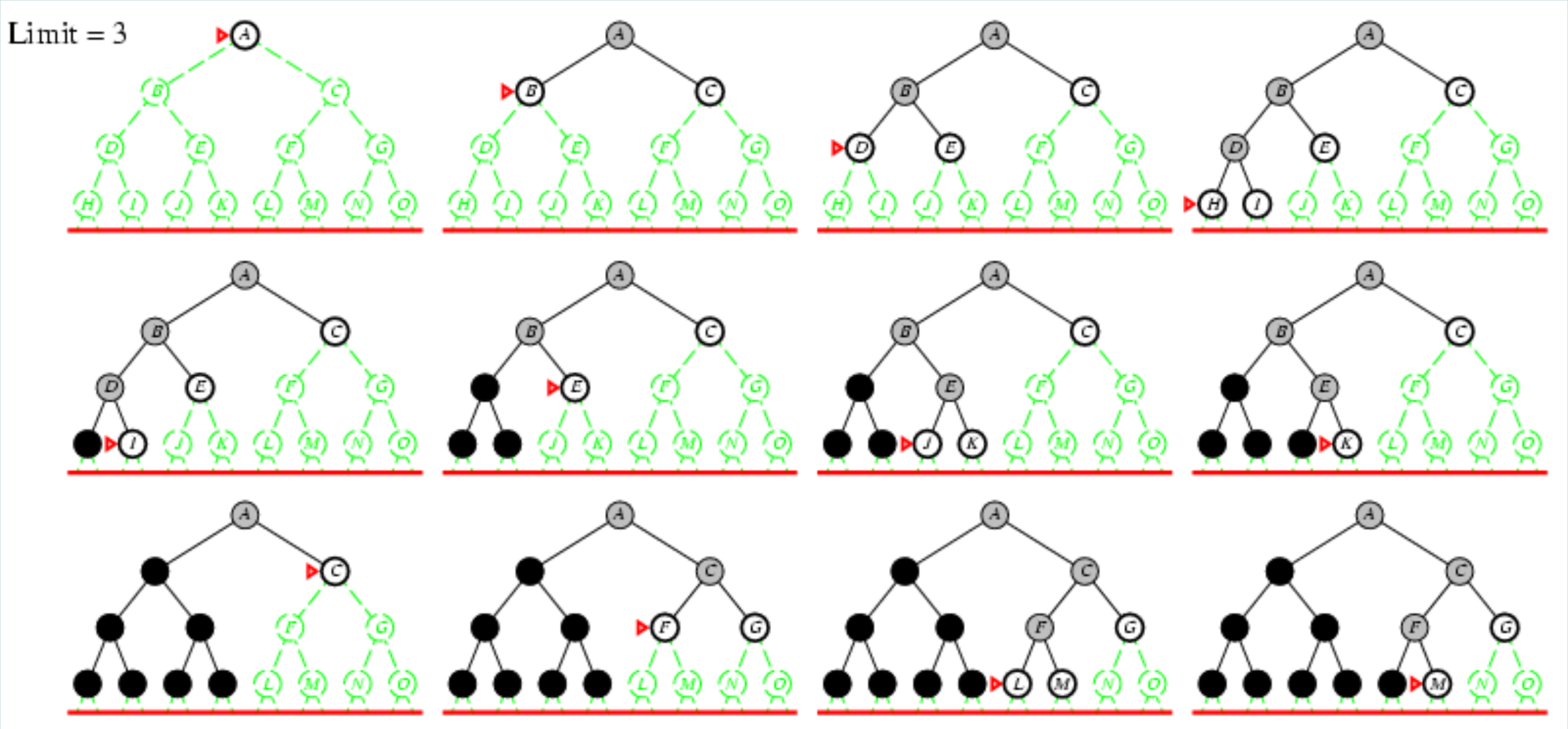
Iterative deepening search $l = 0$

Iterative deepening search l =1

Iterative deepening search l =2



Iterative deepening search l =3



Iterative deepening search overhead

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

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

- 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 overhead

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

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

- For $b = 10$, $d = 5$,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 = 11{,}106$
  - $N_{IDS} = 5 + 40 + 300 + 2{,}000 + 10{,}000 = 12{,}345$
- Overhead = (12,345 - 11,106)/11,106 = 11%

Iterative deepening search

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

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

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

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

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

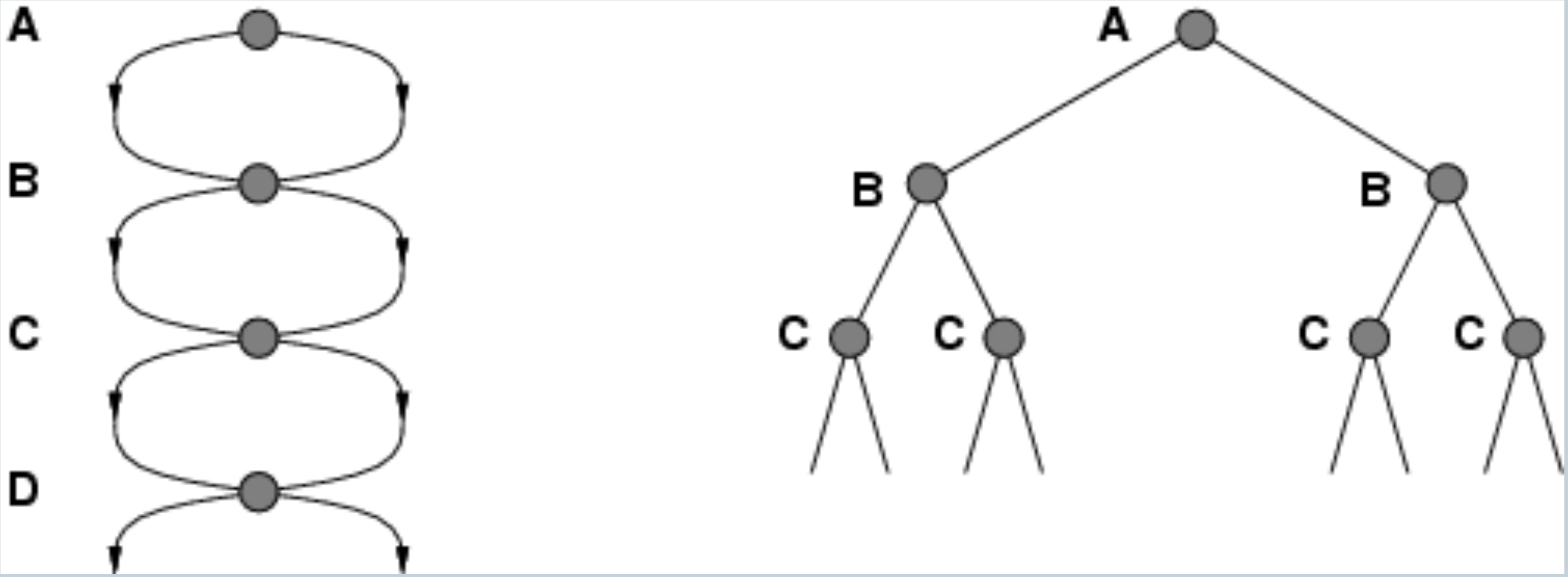
Summary of algorithms

Criterion	**Breadth-First**	**Depth-First**	**Depth-Limited**	**Iterative Deepening**
Complete?	Yes	No	No	Yes
Time	$O(b^d)$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^d)$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	No	No	Yes

Graph Search

Repeated states

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

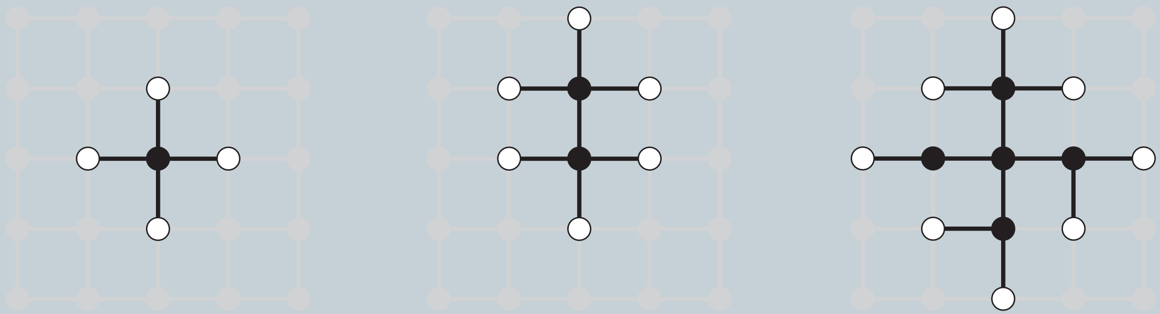


Graph search


```
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)
```

- Simple solution: just keep track of which states you have visited.
- Usually easy to implement in modern computers.

The Separation Property of Graph Search



(a) (b) (c)

- Black: expanded nodes.
- White: frontier nodes.
- Grey: unexplored nodes.

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
-