Solving problems by searching

Chapter 3

Outline

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

Solving Problems by Searching

- Reflex agent is simple
  - base their actions on a direct mapping from states to actions
  - but cannot work well in environments
    - which this mapping would be too large to store
    - and would take too long to learn
- Hence, goal-based agent is used

Problem-solving agent

- Problem-solving agent
  - A kind of goal-based agent
  - It solves problem by
    - finding sequences of actions that lead to desirable states (goals)
  - To solve a problem,
    - the first step is the ***goal formulation***, based on the current situation

Goal formulation

- The goal is formulated
  - as a set of world states, in which the goal is satisfied
- Reaching from initial state → goal state
  - Actions are required
- *Actions* are the operators
  - causing transitions between world states
  - **Actions** should be abstract enough at a certain degree, instead of very detailed
  - E.g., turn left VS turn left 30 degree, etc.

Problem formulation

- The process of deciding
  - what actions and states to consider
- E.g., driving Amman → Zarqa
  - in-between states and actions defined
  - States: Some places in Amman & Zarqa
  - Actions: Turn left, Turn right, go straight, accelerate & brake, etc.

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***
- The process SEARCH of looking for a sequence of actions that reaches the goal is called search

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"

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

  $state \leftarrow$ UPDATE-STATE($state, percept$)
  if *seq* is empty **then**
    $goal \leftarrow$ FORMULATE-GOAL($state$)
    $problem \leftarrow$ FORMULATE-PROBLEM($state, goal$)
    $seq \leftarrow$ SEARCH($problem$)
    if $seq = failure$ **then return** a null action
  $action \leftarrow$ FIRST(seq)
  $seq \leftarrow$ REST(seq)
  return *action*

Well-defined problems and solutions

A problem is defined by 5 components:

- ***<u>Initial state</u>***
- ***<u>Actions</u>***
- ***<u>Transition model</u> or***

<u>(Successor functions)</u>

- ***<u>Goal Test.</u>***
- ***<u>Path Cost.</u>***

Well-defined problems and solutions

- A problem is defined by 5 components:
  - The ***initial state***
    - that the agent starts in
  - The set of possible **actions**
  - ***Transition model:*** *description of what each action does.*

  (***successor functions***): refer to any state reachable from given state by a single action
  - Initial state, actions and Transition model define the ***state space***
    - the set of all states reachable from the initial state by any sequence of actions.
  - A ***path*** in the state space:
    - any sequence of states connected by a sequence of actions.

Well-defined problems and solutions

- The ***goal test***
  - Applied to the current state to test
    - if the agent is in its goal

  -Sometimes there is an explicit set of possible goal states. (example: in Amman).

  -Sometimes the goal is described by the properties
    - instead of stating explicitly the set of states
  - Example: Chess
    - the agent wins if it can capture the KING of the opponent on next move (checkmate).
    - no matter what the opponent does

Well-defined problems and solutions

- A ***path cost*** function,
  - assigns a numeric cost to each path
  - = performance measure
  - denoted by ***g***
  - to distinguish the best path from others
- Usually the path cost is
  - the sum of the **step costs** of the individual actions (in the action list)

Well-defined problems and solutions

- Together a problem is defined by
  - Initial state
  - Actions
  - Successor function
  - Goal test
  - Path cost function
- The ***solution*** of a problem is then
  - *a path from the initial state to a state satisfying the goal test*
- ***Optimal*** solution
  - the solution with lowest path cost among all solutions

Formulating problems

- Besides the four components for problem formulation
  - anything else?
- ***Abstraction***
  - the process to take out the irrelevant information
  - leave the most essential parts to the description of the states

  (Remove detail from representation)
  - **Conclusion**: Only the most important parts *that are contributing to searching* are used

Evaluation Criteria

- formulation of a problem as search task
- basic search strategies
- important properties of search strategies
- selection of search strategies for specific tasks

(The ordering of the nodes in FRINGE defines the search strategy)

Problem-Solving Agents

- agents whose task is to solve a particular problem (steps)
  - goal formulation
    - what is the goal state
    - what are important characteristics of the goal state
    - how does the agent know that it has reached the goal
    - are there several possible goal states
      - are they equal or are some more preferable
  - problem formulation
    - what are the possible states of the world relevant for solving the problem
    - what information is accessible to the agent
    - how can the agent progress from state to state

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

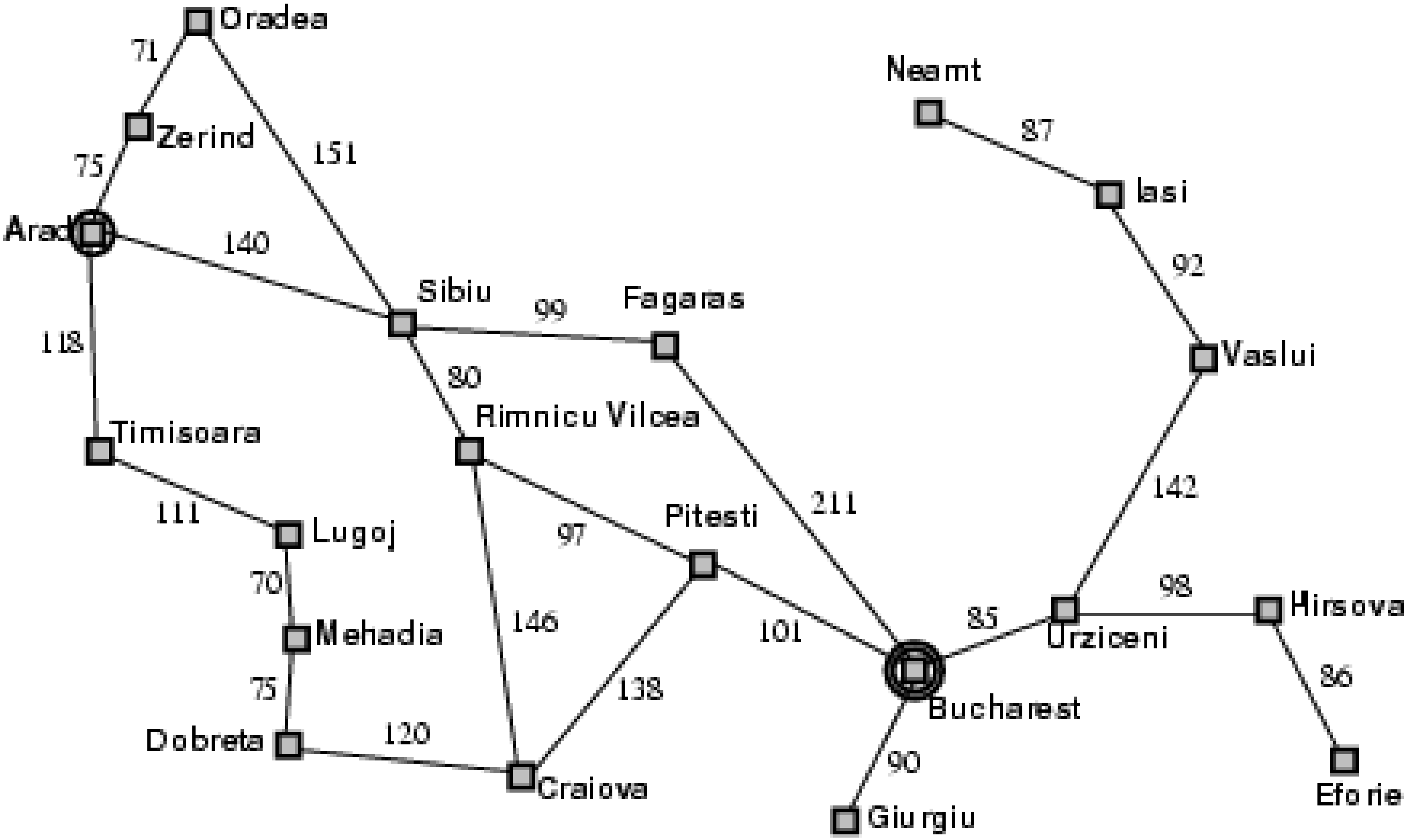
- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest

- 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

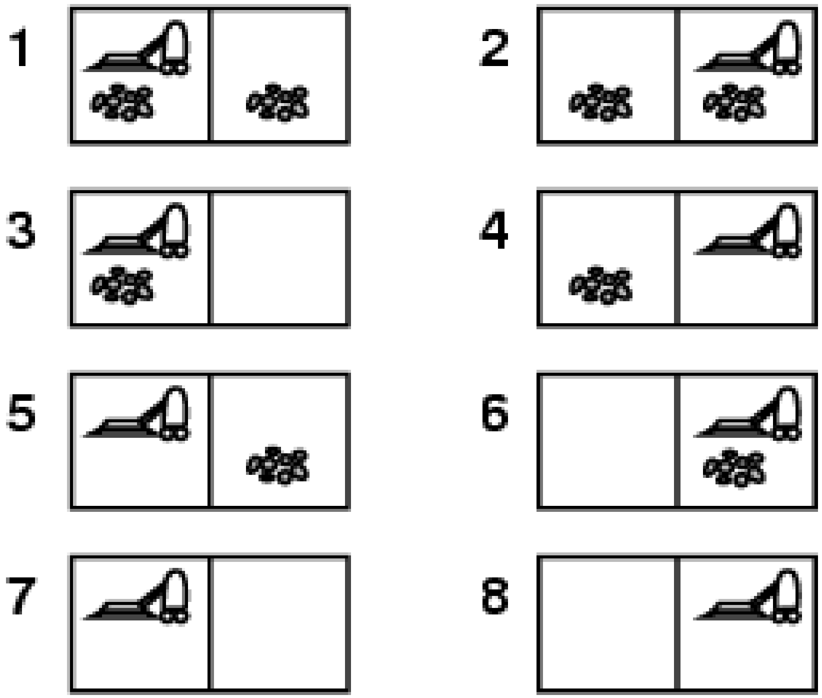


Problem types

- 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

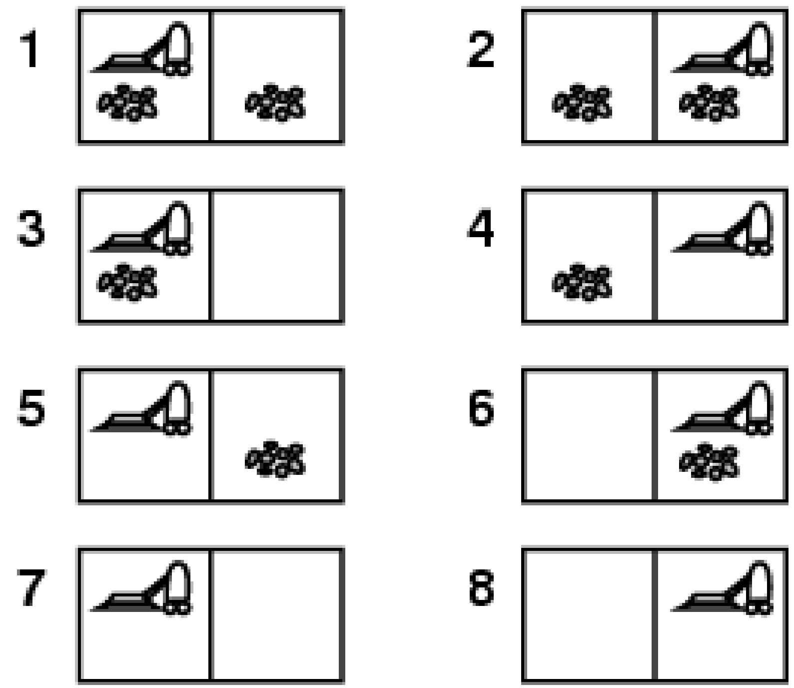
- Single-state, start in #5.
  Solution?
-



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?
-

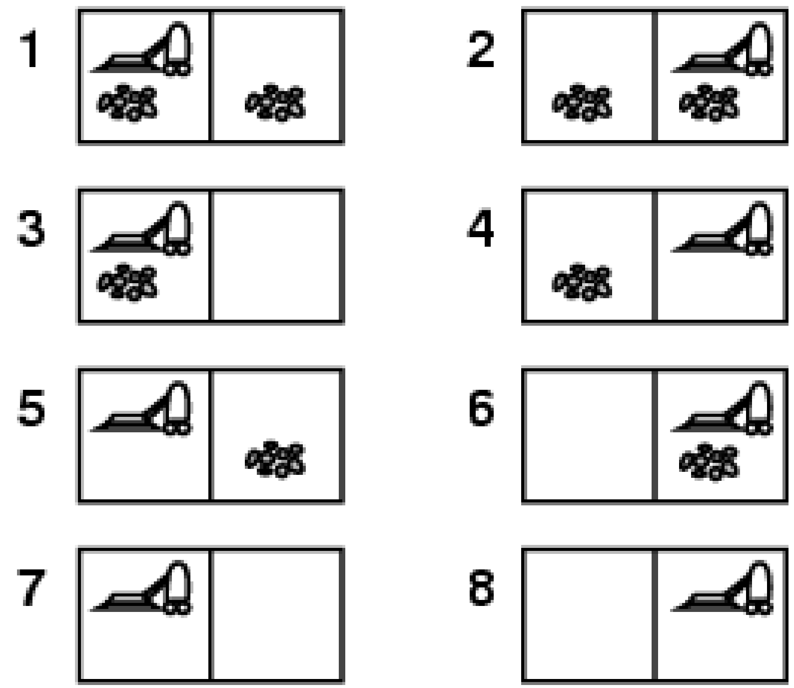


Example: vacuum world

- 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]

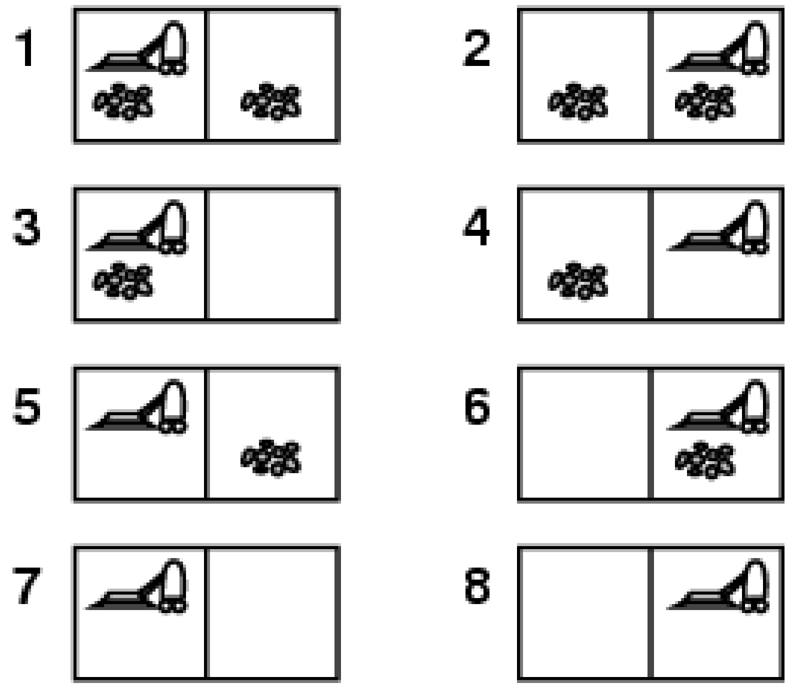
-



- 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]*



Single-state 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)*

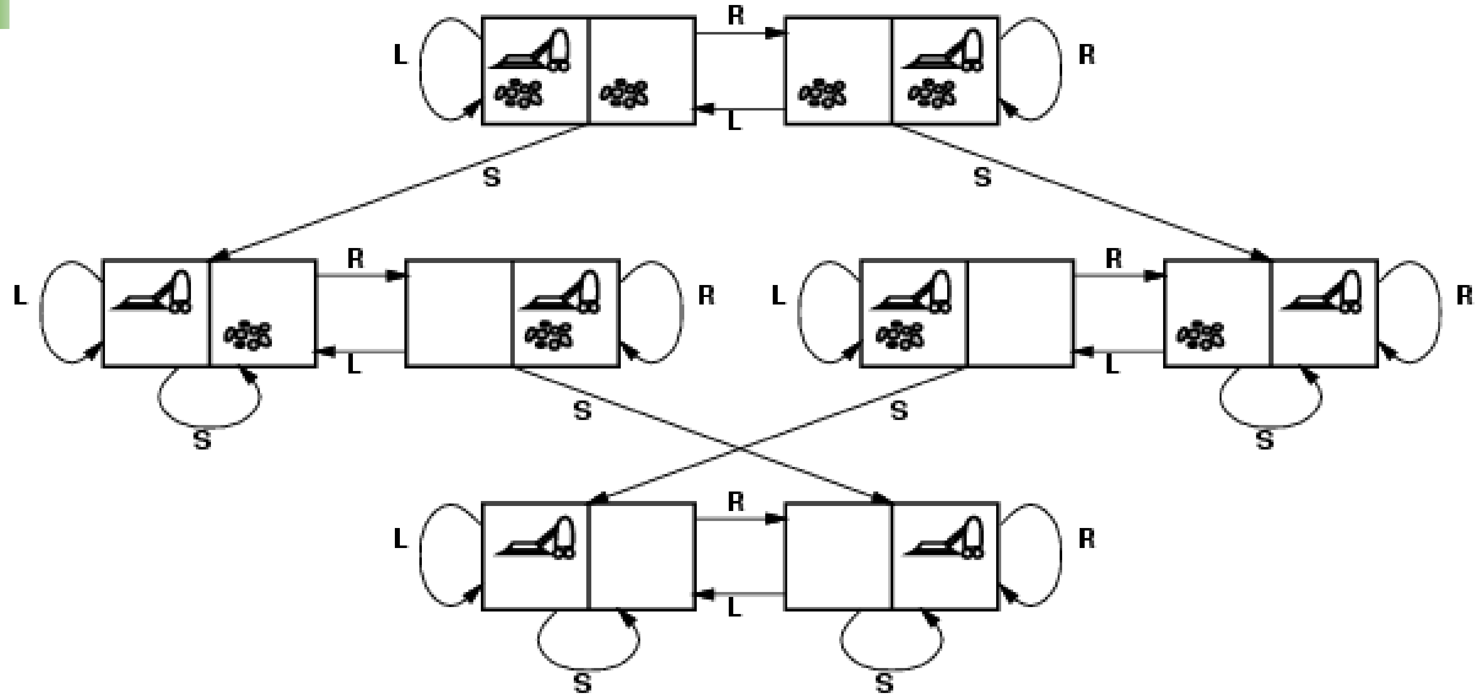
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost, assumed to be ≥ 0
   - A solution is a sequence of actions leading from the initial state to a goal state

Selecting a state space

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

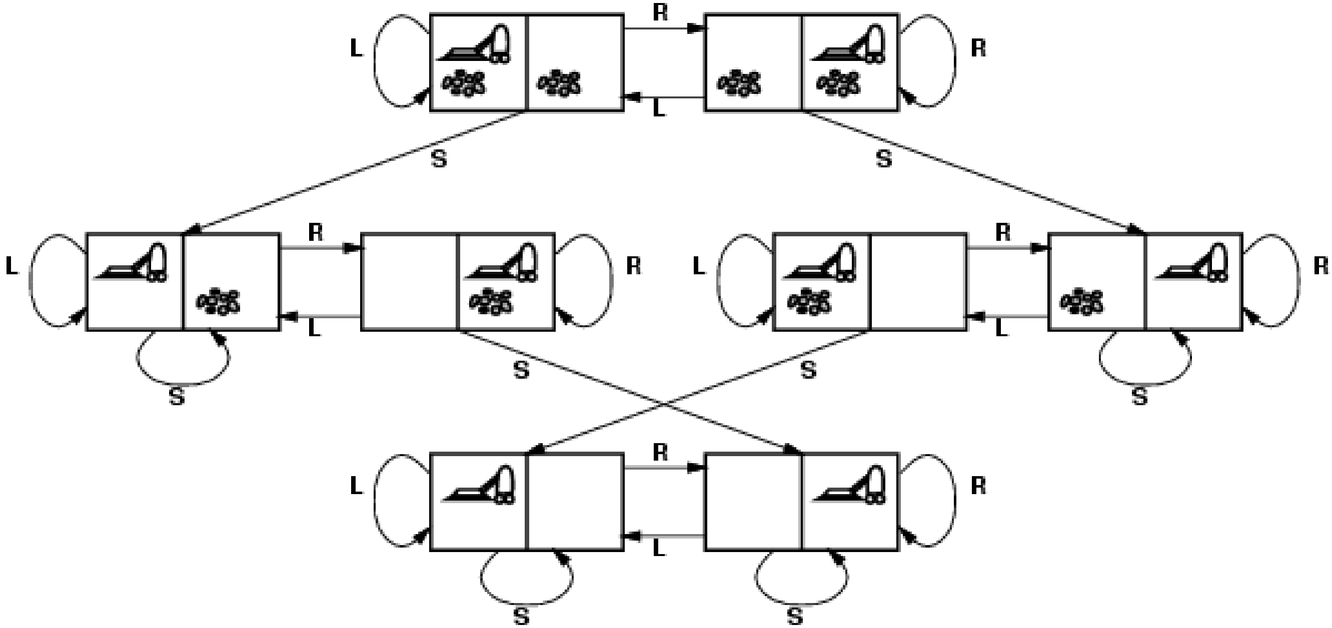
- (Abstract) state = set of real states
-
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad" must get to some real state "in Zerind"
-
- (Abstract) 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



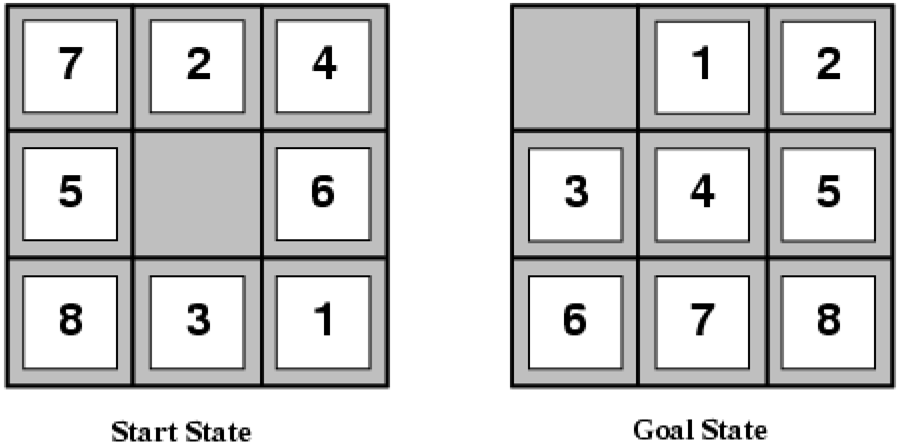
- states?
- actions?
- goal test?
- path cost?

Vacuum world state space graph



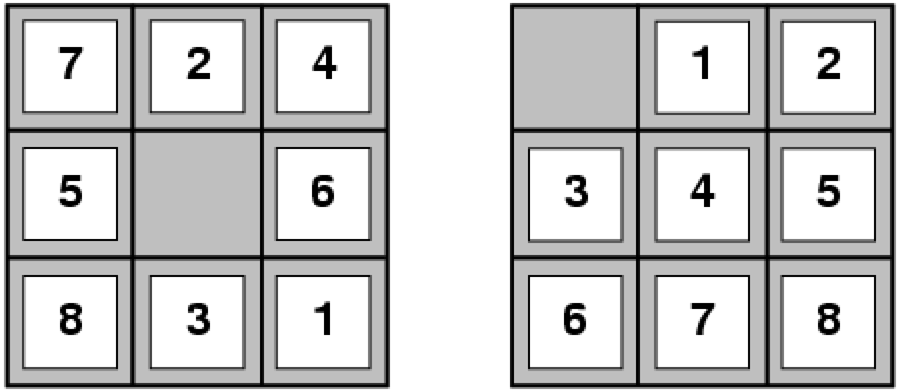
- states? integer dirt and robot location
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

Example: The 8-puzzle



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

Example: The 8-puzzle



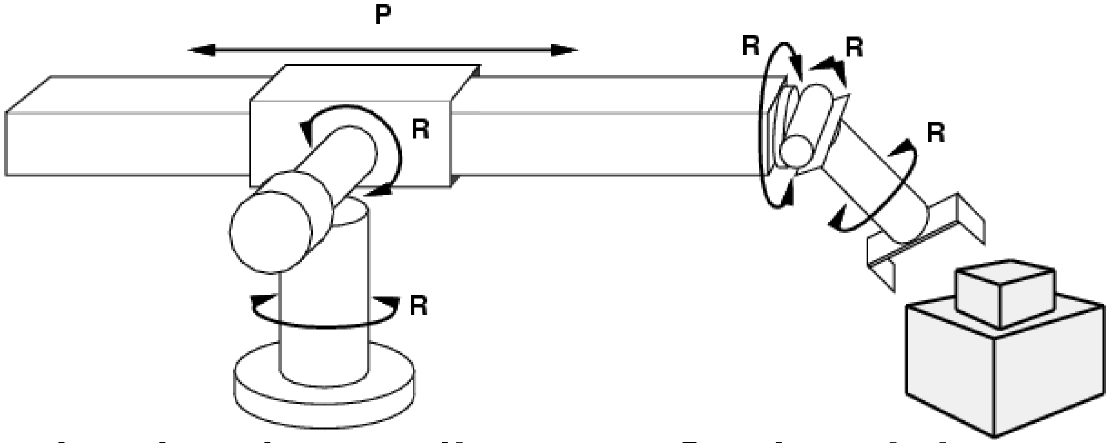
Start State

Goal State

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

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

Example: robotic assembly



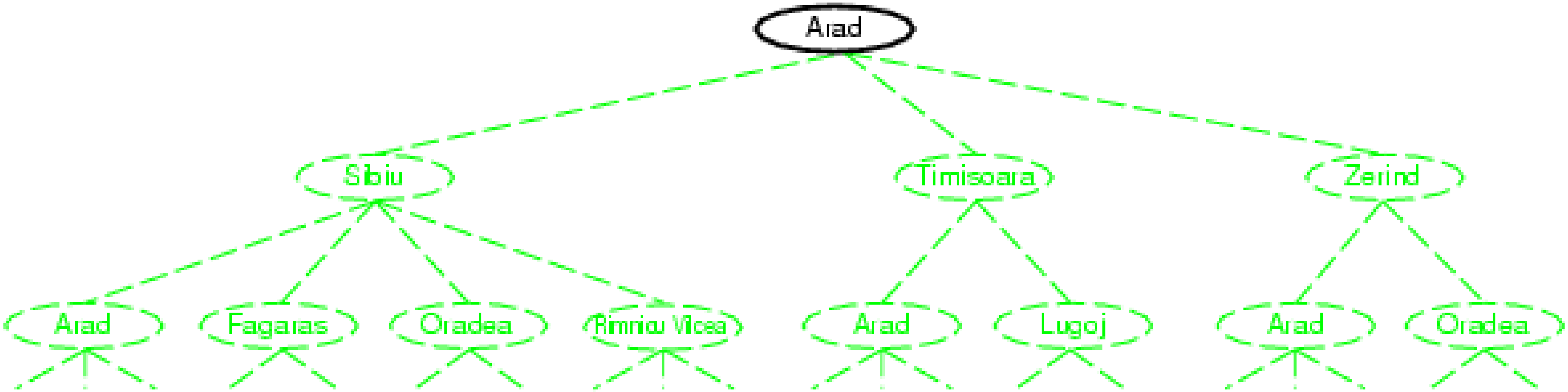
- states?: real-valued coordinates of robot joint angles parts of the object to be assembled
- actions?: continuous motions of robot joints
- goal test?: complete assembly
- path cost?: time to execute

Tree search algorithms

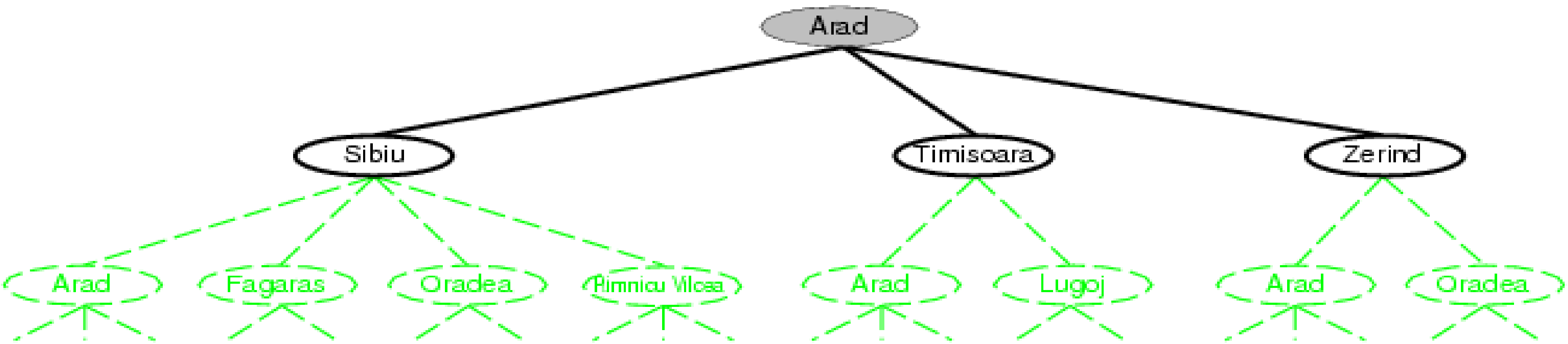
- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

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

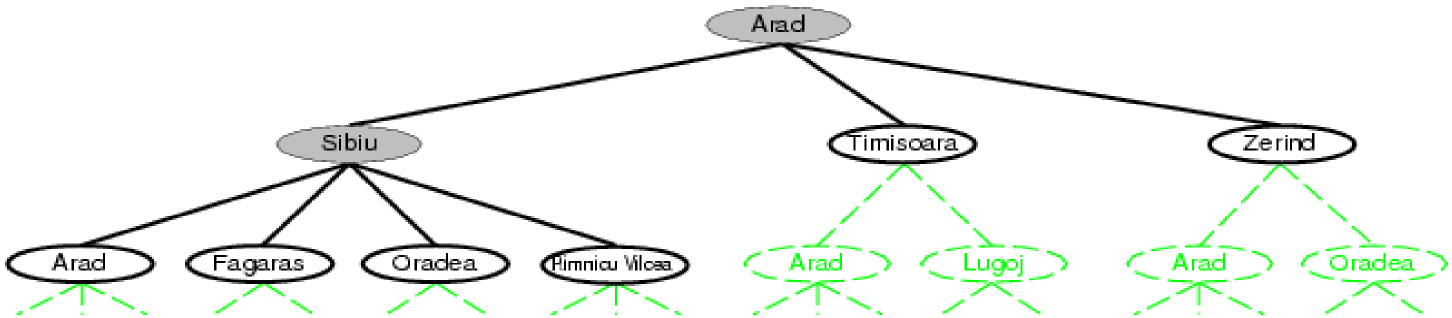
Tree search example



Tree search example



Tree search example



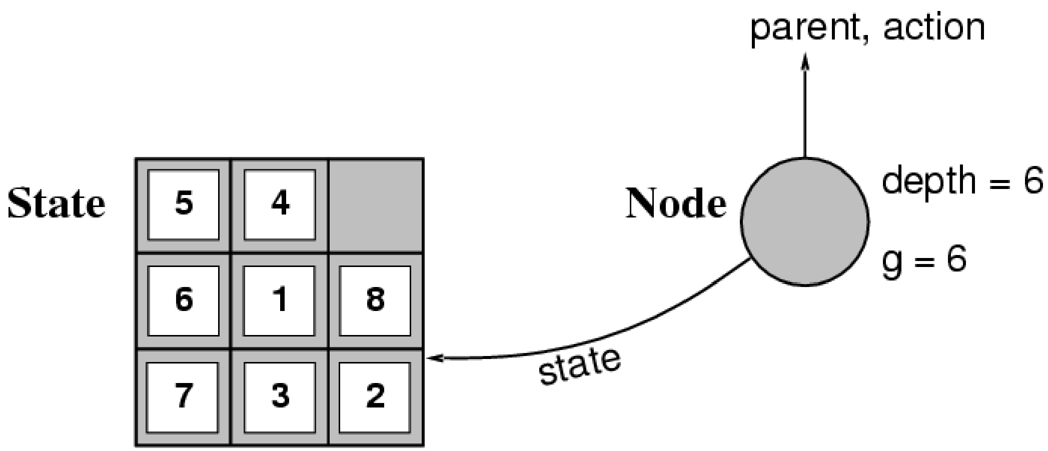
Implementation: general tree search

```
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
```

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.
-

Search strategies

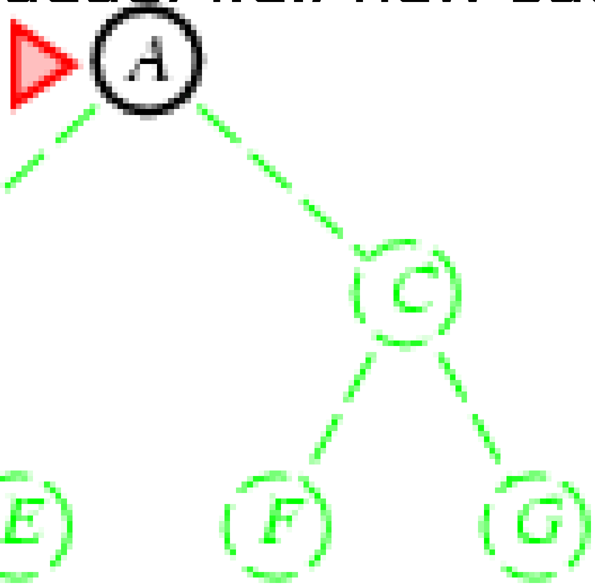
- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: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

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

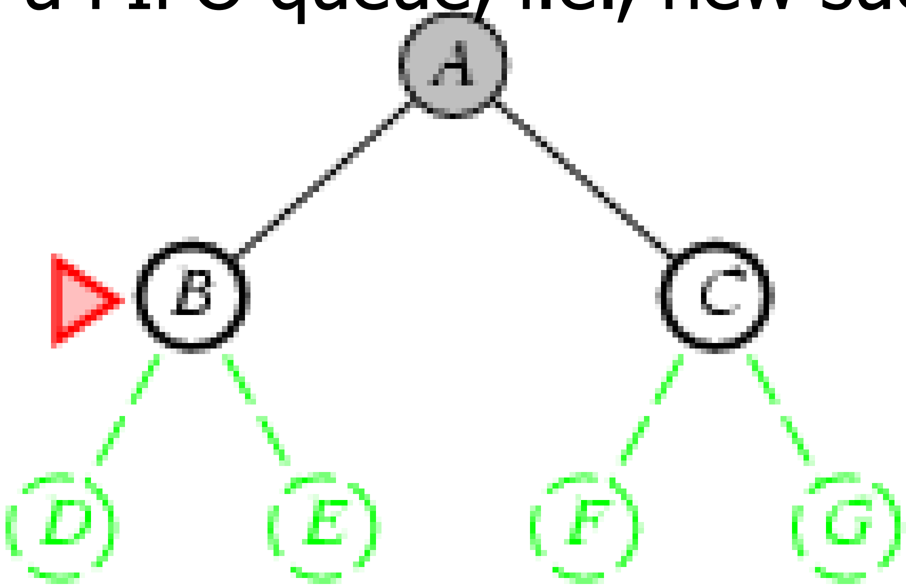
Breadth-first search

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



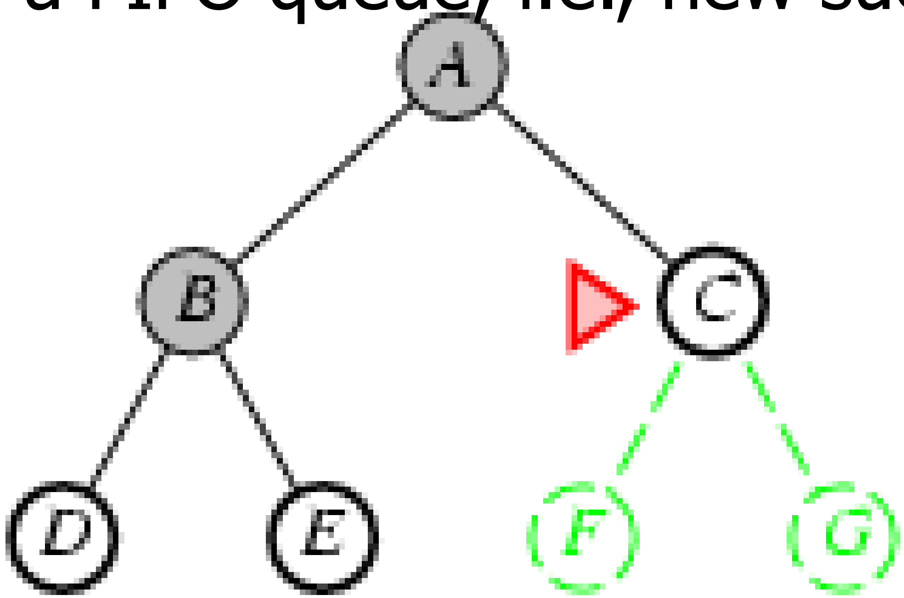
Breadth-first search

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



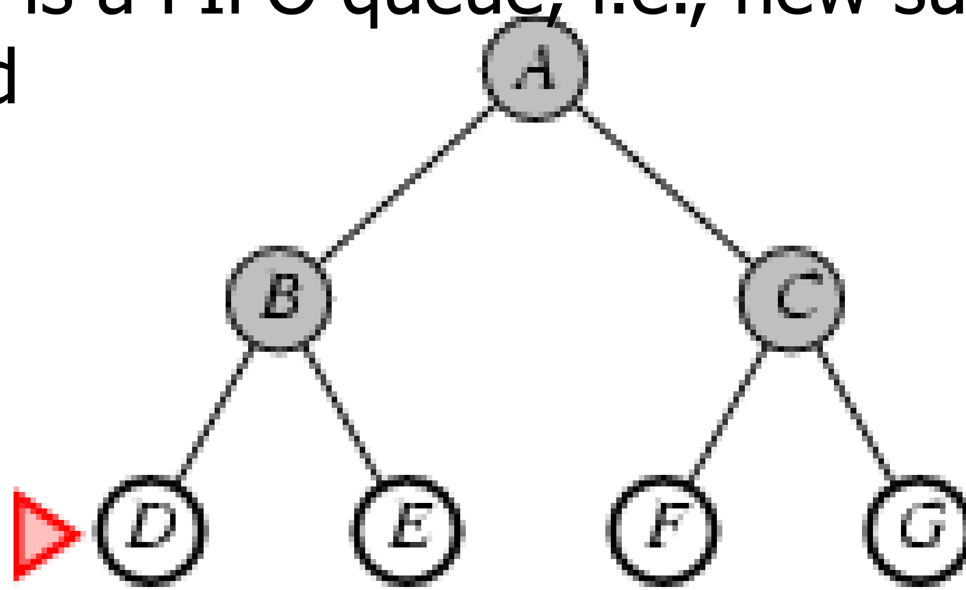
Breadth-first search

- 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

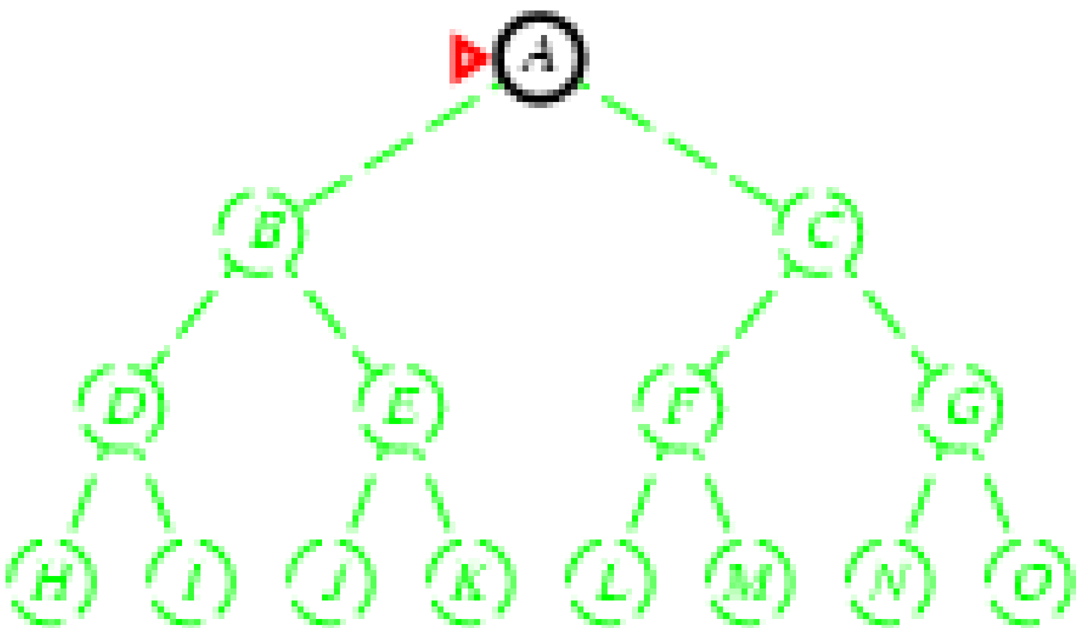
- Complete? Yes (if b is finite)
-
- Time? $1+b+b^2+b^3+\ldots +b^d + b(b^d-1) = O(b^{d+1})$
-
- Space? $O(b^{d+1})$ (keeps every node in memory)
-
- Optimal? Yes (if cost = 1 per step)
-
- Space is the bigger problem (more than time)
-

Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost ≥ ε
- 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 $g(n)$

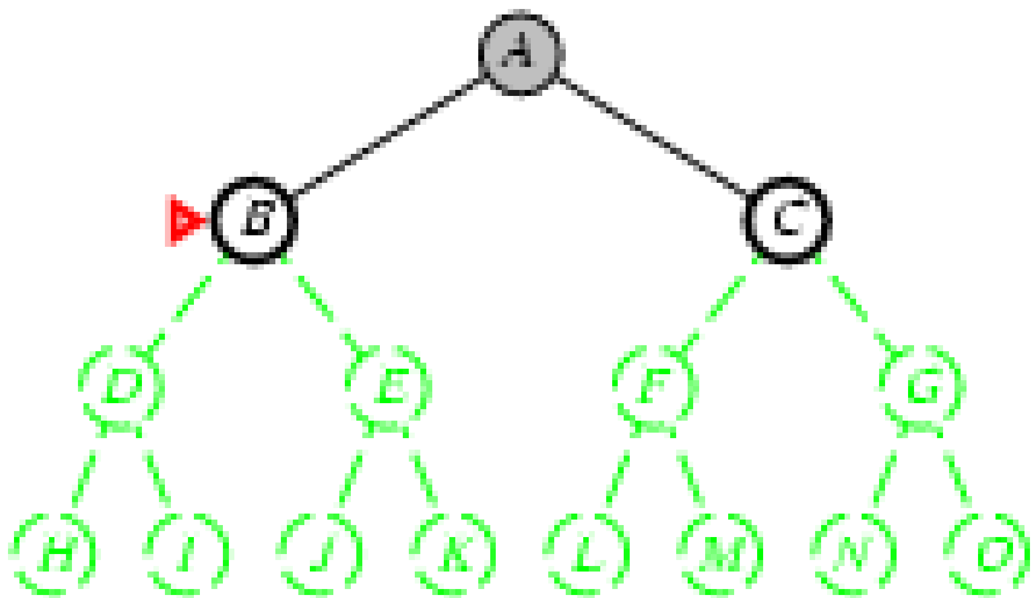
Depth-first search

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



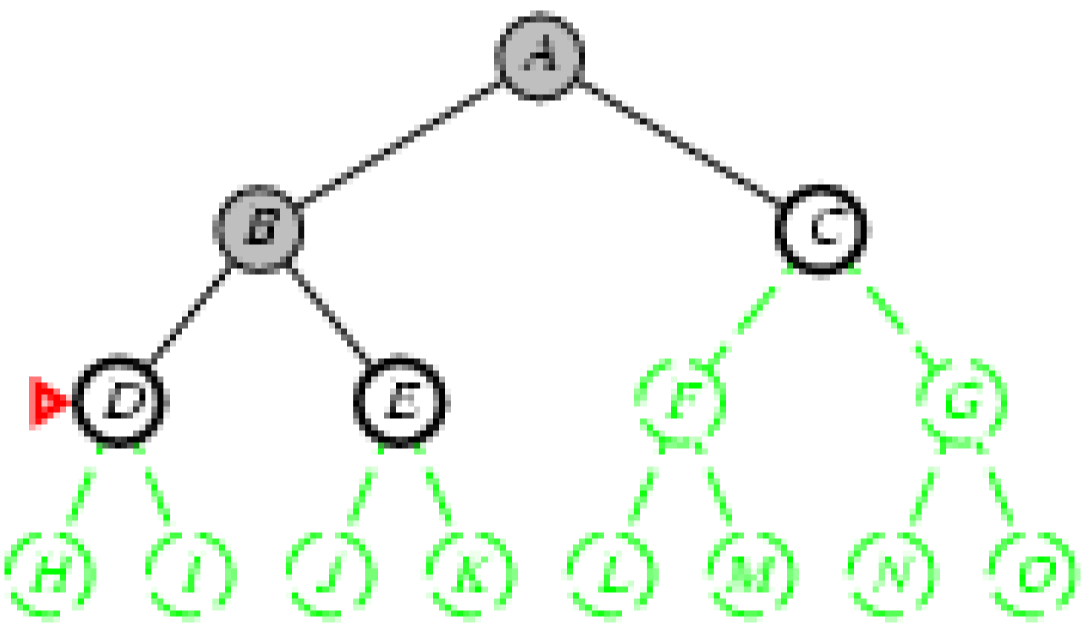
Depth-first search

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



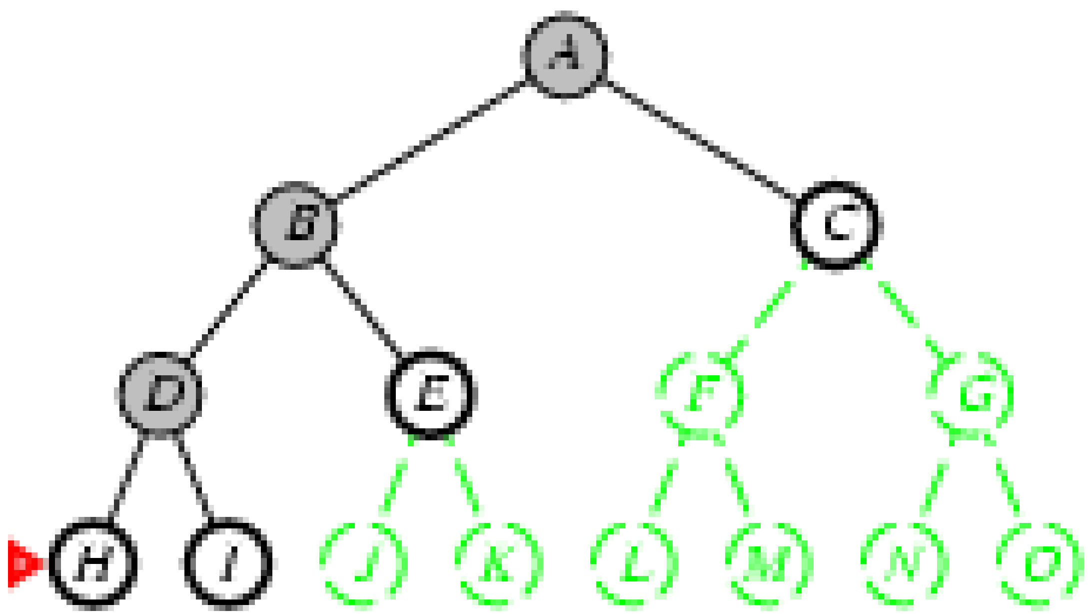
Depth-first search

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



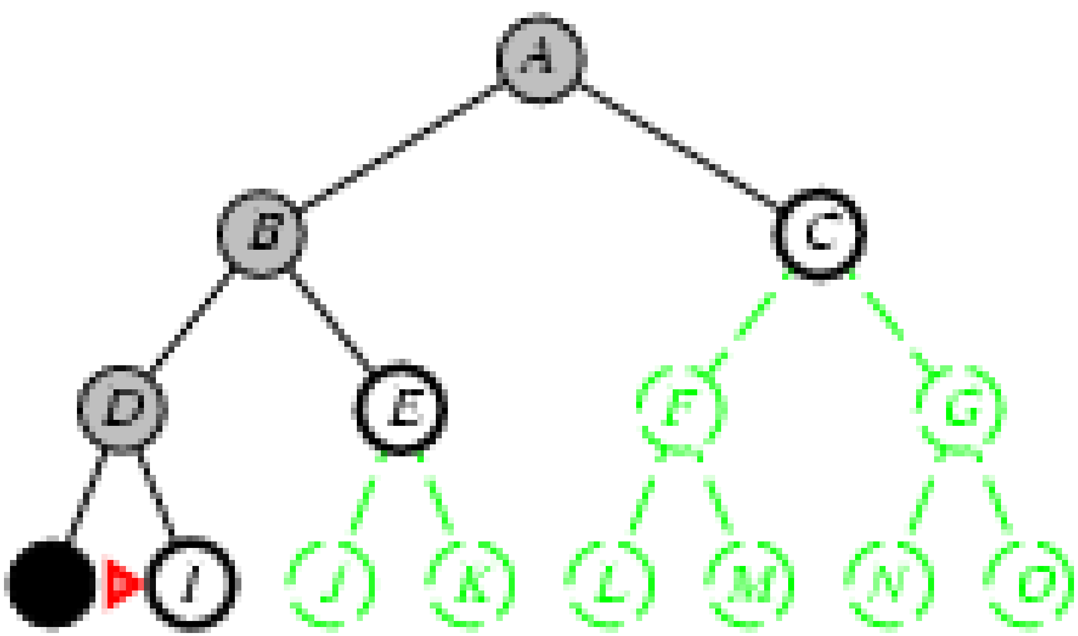
Depth-first search

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



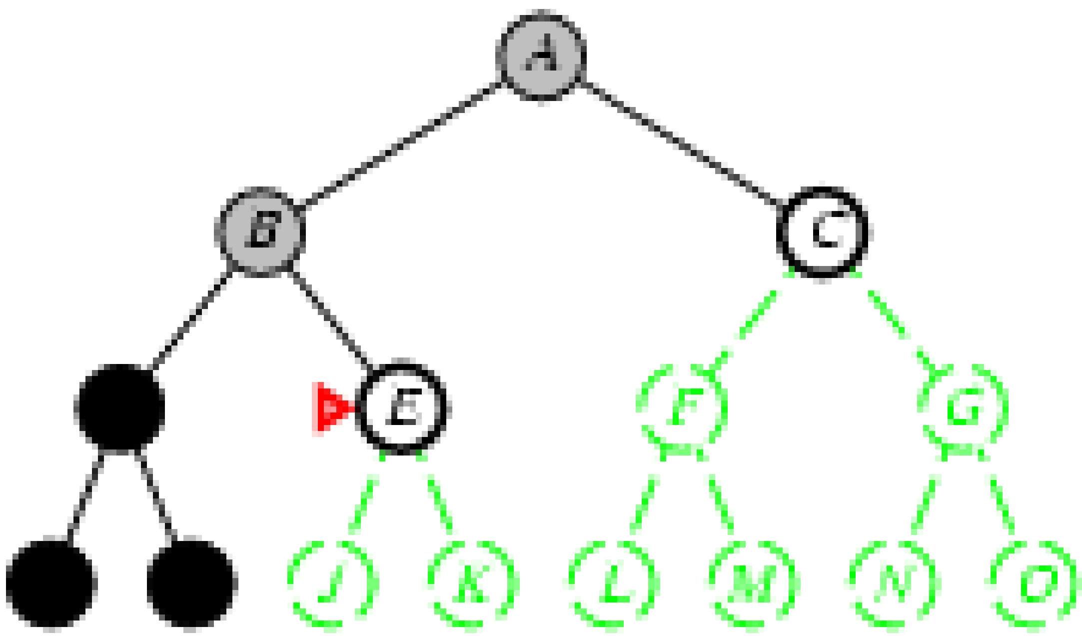
Depth-first search

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



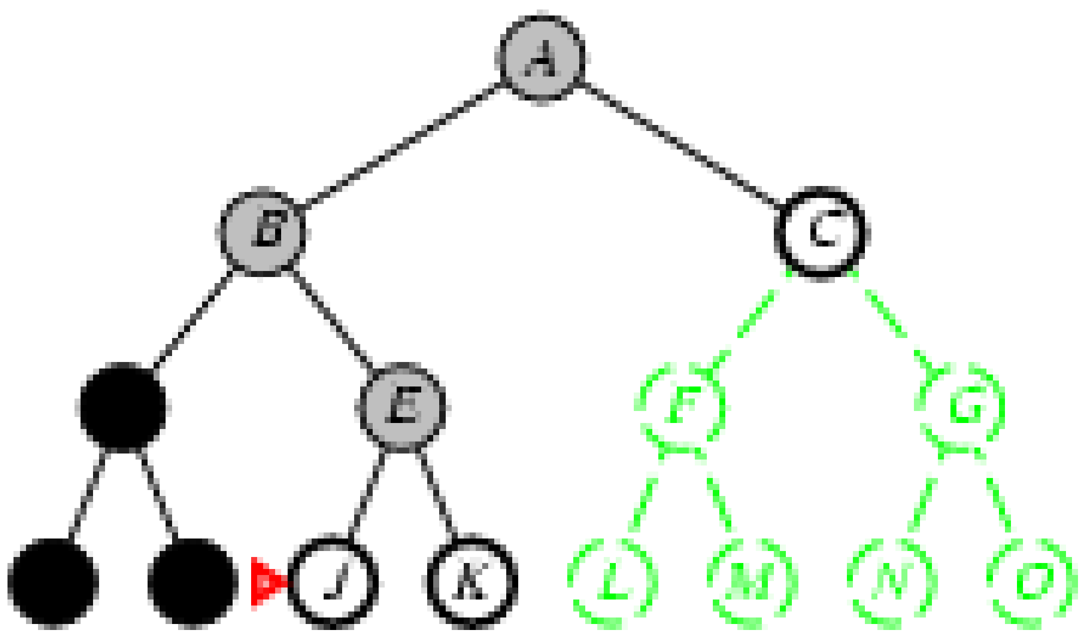
Depth-first search

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



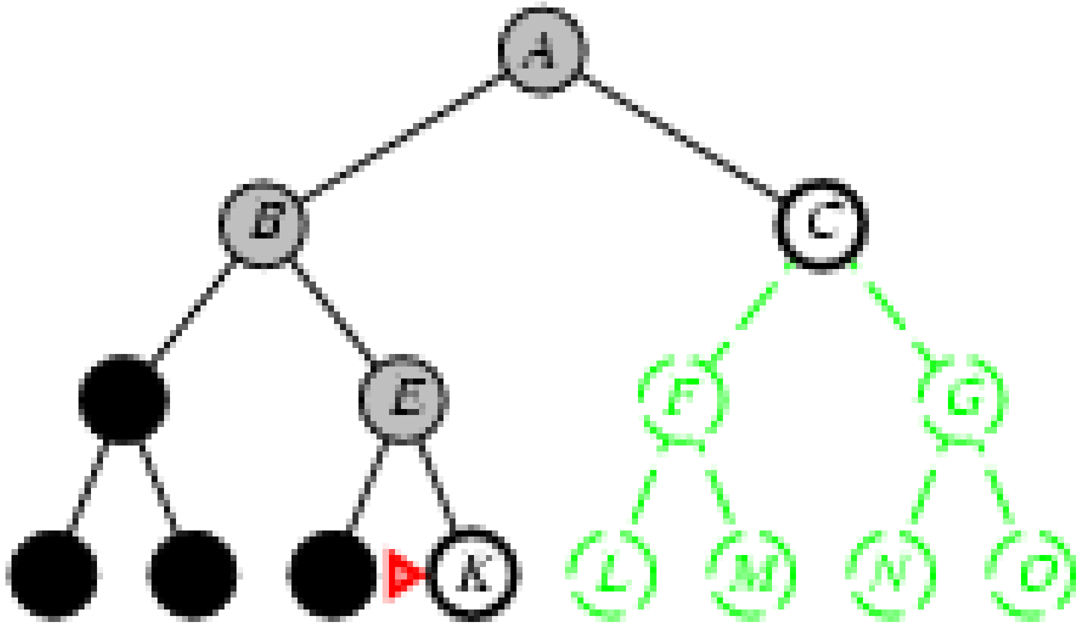
Depth-first search

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



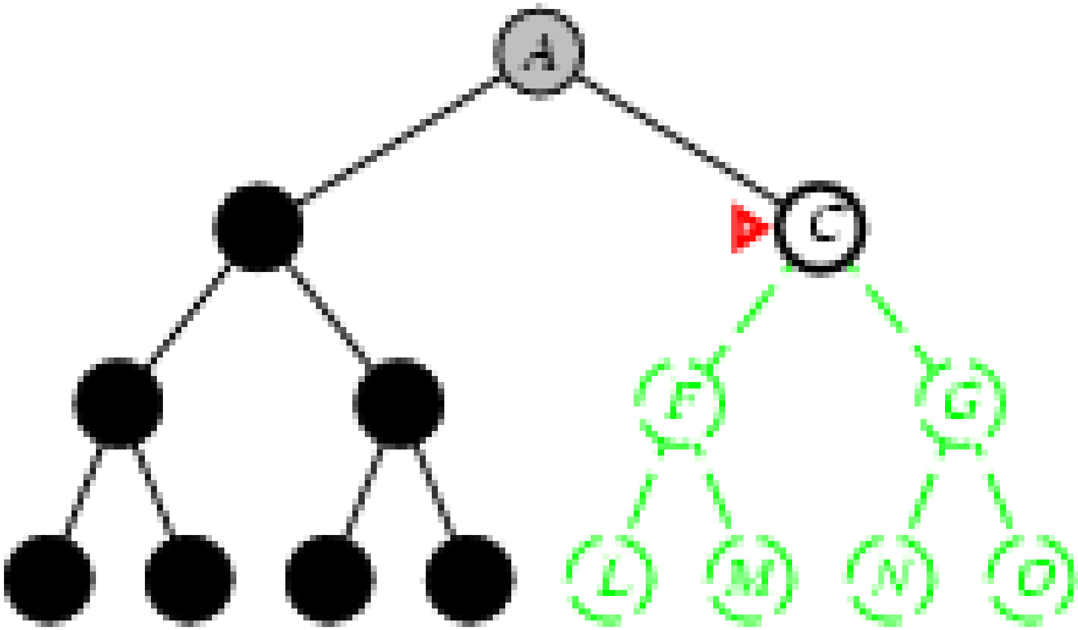
Depth-first search

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



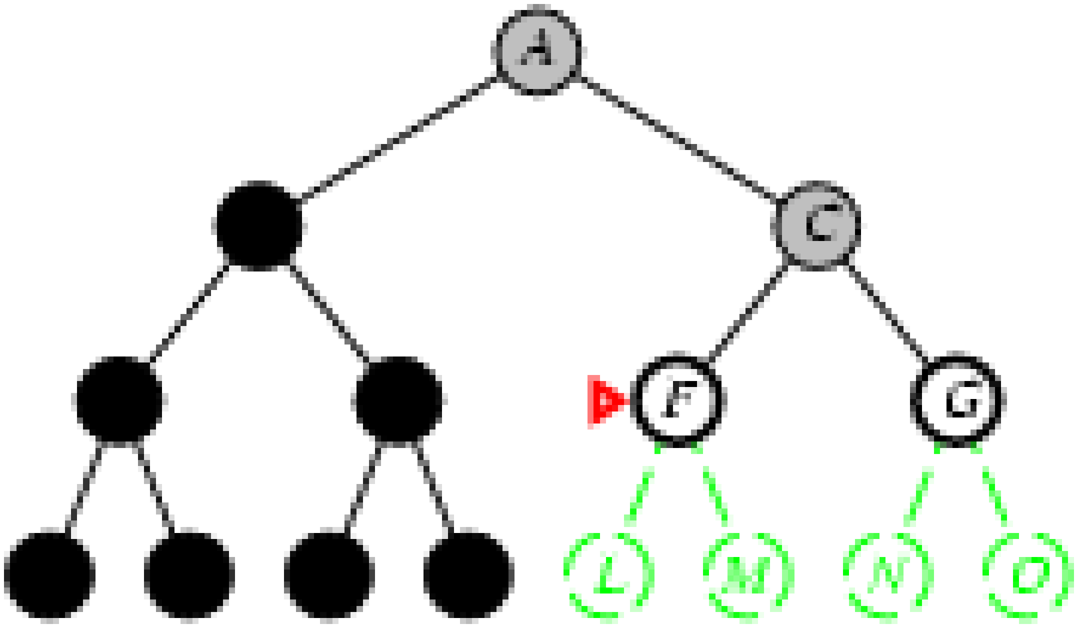
Depth-first search

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



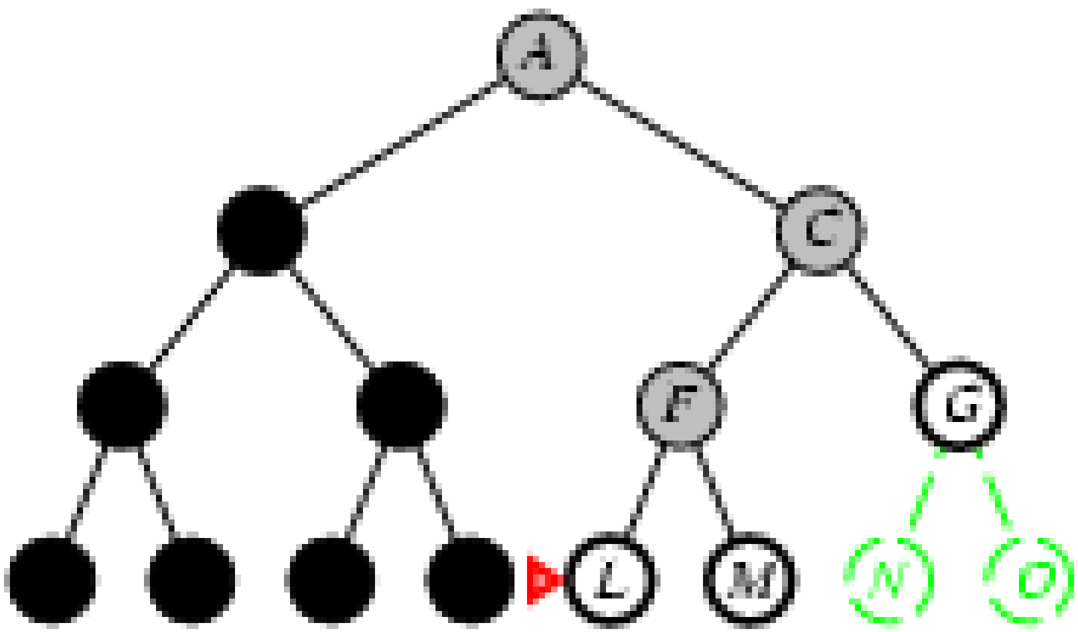
Depth-first search

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



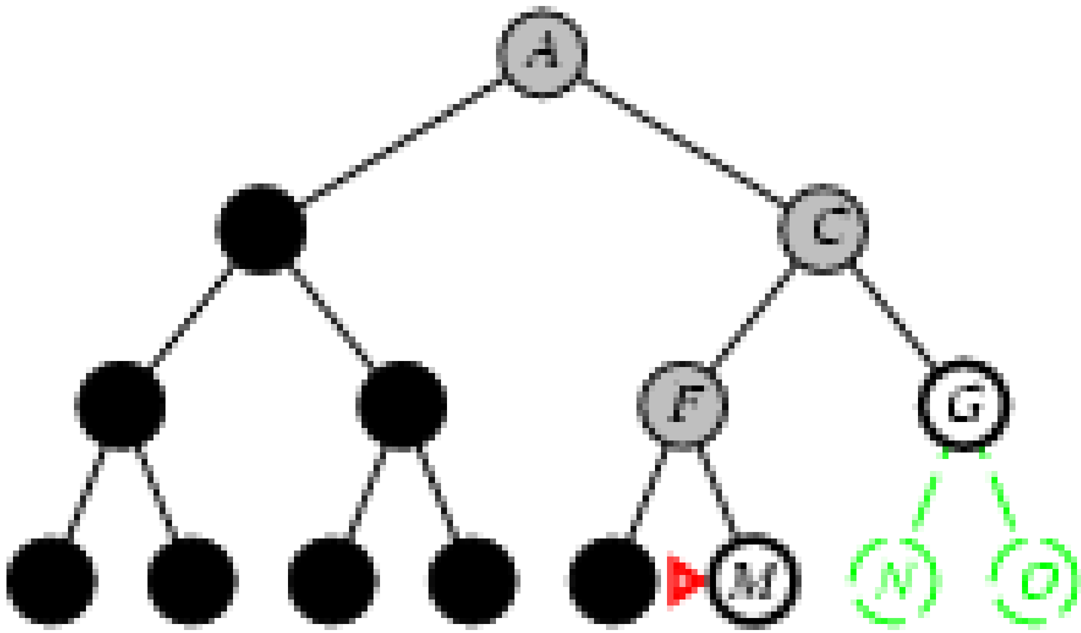
Depth-first search

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



Depth-first search

- 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
```

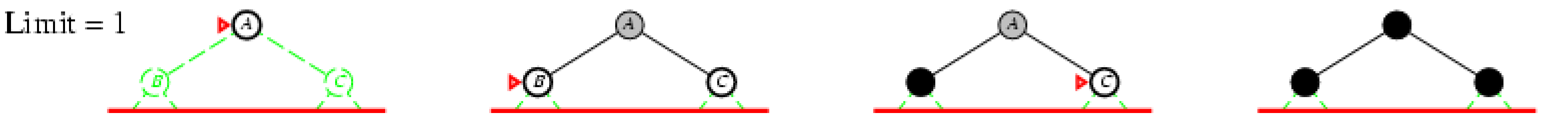
Iterative deepening search

```
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
```

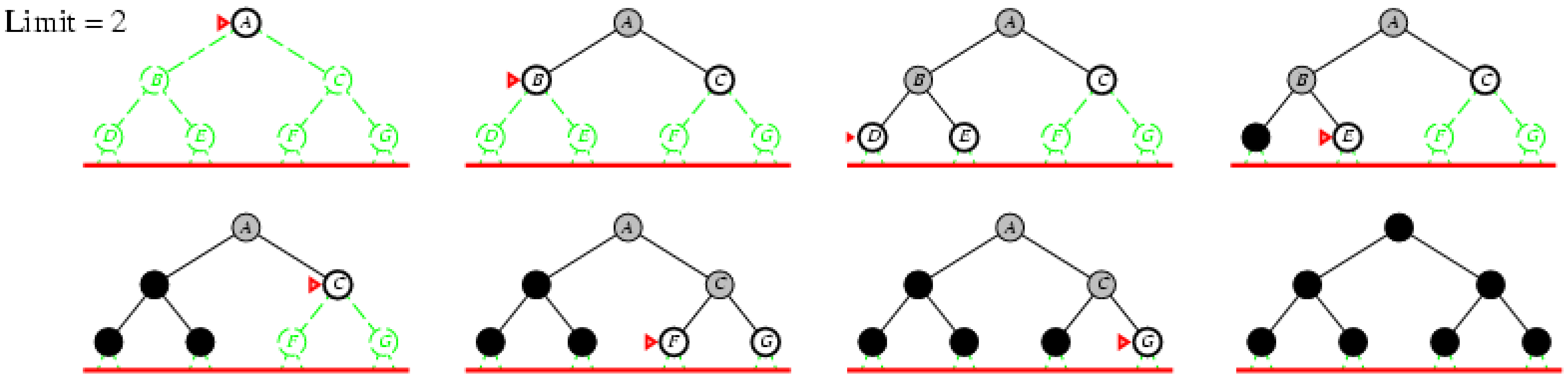
Iterative deepening search $l = 0$

Limit = 0

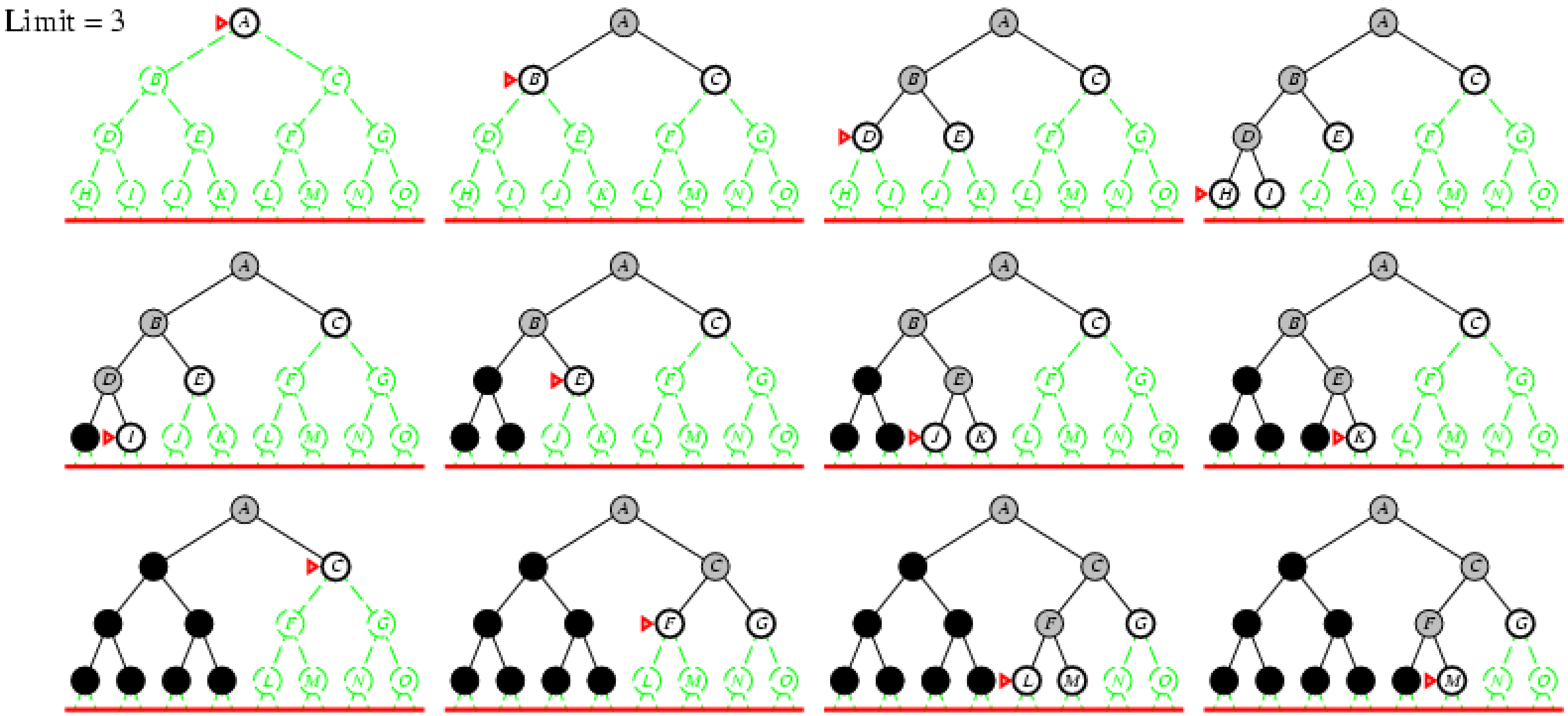
Iterative deepening search *l* =1



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*:

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

- 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

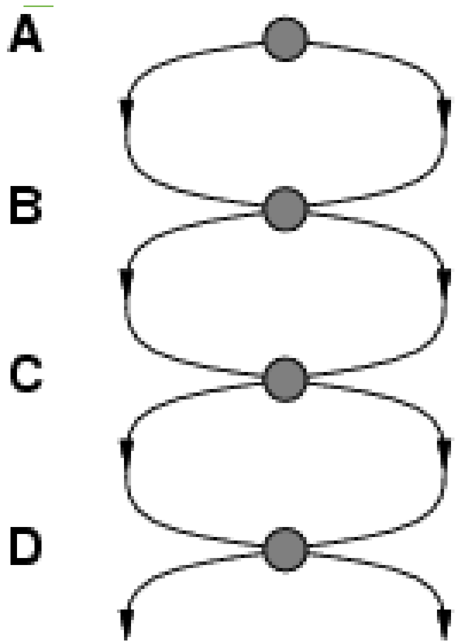
- Complete? Yes
-
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
-
- Space? $O(bd)$
-
- Optimal? Yes, if step cost = 1

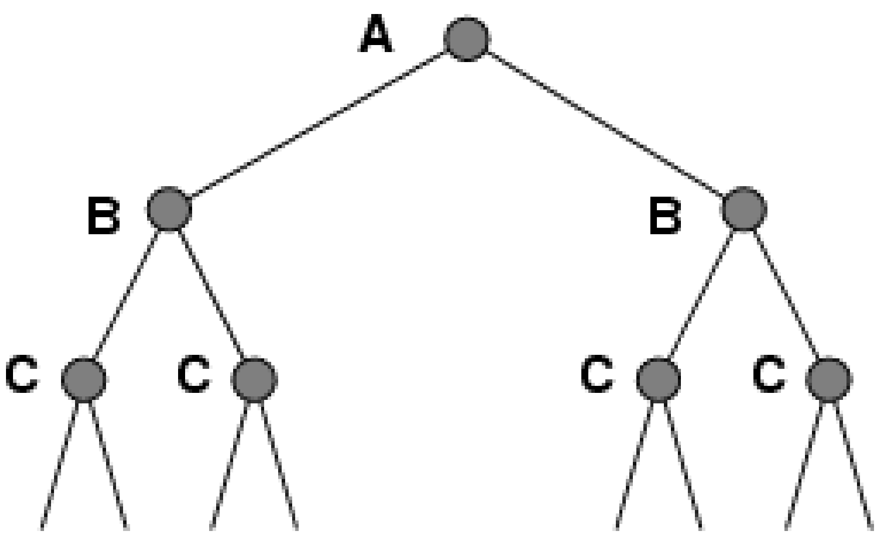
Summary of algorithms

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

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

- 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