Search in the state space[1]

Lecture 3

[1]The slides have been prepared using the textbook material available on the web, and the slides of the previous editions of the course by Prof. Luigia Carlucci Aiello

Problem solving

Summary(Russell&Norvig Chap. 3)

♢ Problem solving agents

♢ Types of problems

♢ Problem formulation

♢ Example problems

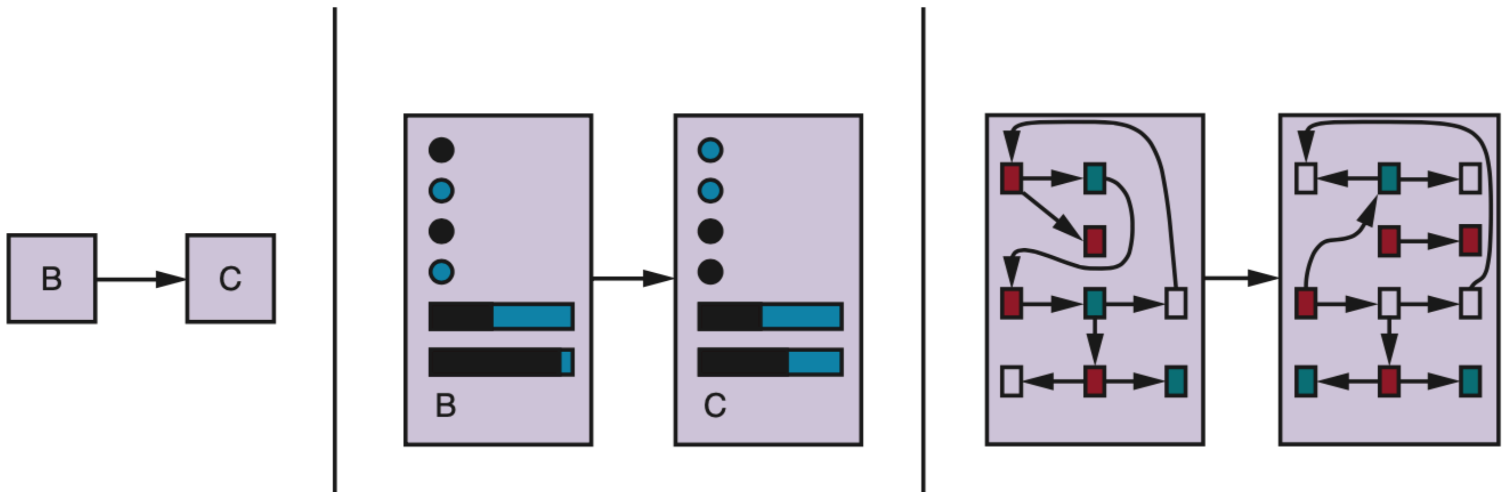
♢ Basic search algorithms

Agent Architectures

- **simple-reflex agents** (**stimoulus–response**, **tropistic**): react to stimuli
- **model-based agents** build a representation of the world
- **goal-based agents** have a goal and try to achieve it
- **utility-based agents** evaluate the utility of (goal) states according to a performance measure and trying to maximize it

Design of goal-based agents

- **search**: translating knowledge into a specialized representation of **states** and a set of **operators**
- **planning**: explicit representation of the state and specialized techniques for reasoning.
- **inference**: general state representation and reasoning.



(a) Atomic (b) Factored (c) Structured

The problem solving agent

- **problem formulation**:
  - state definition, the set of states is called **state space**
  - **initial state**
  - **actions** (operators) characterizing state transitions; **path** in the state space is a sequence of actions
- **goal formulation**: set of states or goal test
- **search**: process to find a **solution**, i.e. a path from the initial state to one of the goal states.
- **execution**: the solution is performed by the agent

```
function SIMPLE-PS-AGENT( percept) returns an action
    static: seq, an action sequence, initially empty
            state, 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 ← RECOMMENDATION(seq, state)
    seq ← REMAINDER(seq, state)
    return action
```

Offline vs Online Problem Solving

The agent implements *offline* problem solving;
solution executed "eyes closed."

Online problem solving involves acting without complete knowledge.

Example: Romania

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

Formulate goal:

be in Bucharest

Formulate problem:

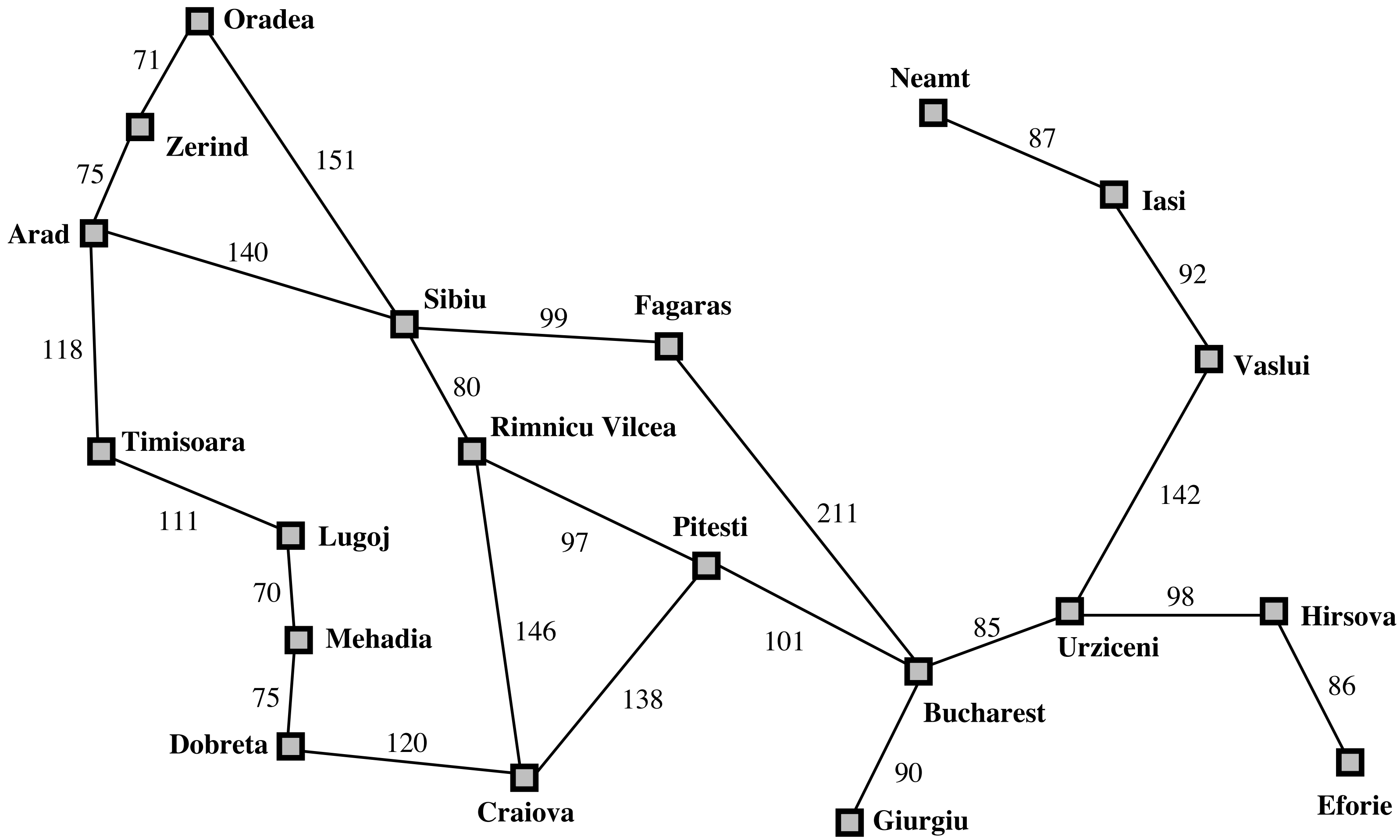
states: various cities

actions: drive between cities

Find solution:

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

Example: Romania



Single-state problem formulation

initial state e.g., "at Arad"

actions e.g., "$Arad \rightarrow Zerind$" and a function $Action(s)$ returning a (finite) set of actions *applicable* in a state s

transition model (successor function) $S(x) =$ set of action–state pairs

e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind\rangle, \ldots\}$

action cost $ACost(s, a, s') =$ the cost of applying action a in state s to reach state s'

e.g., $ACost(Arad, Arad \rightarrow Zerind, Zerind) = 75$

Single-state goal formulation and solution

goal test, can be

explicit, e.g., $x =$ "at Bucharest"

implicit, e.g., $NoDirt(x)$

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

solution $=$ sequence of actions from init state to goal state

Selecting a state space

Real world is absurdly complex

$\Rightarrow$ state space must be *abstracted* for problem solving

(Abstract) state = set of real states

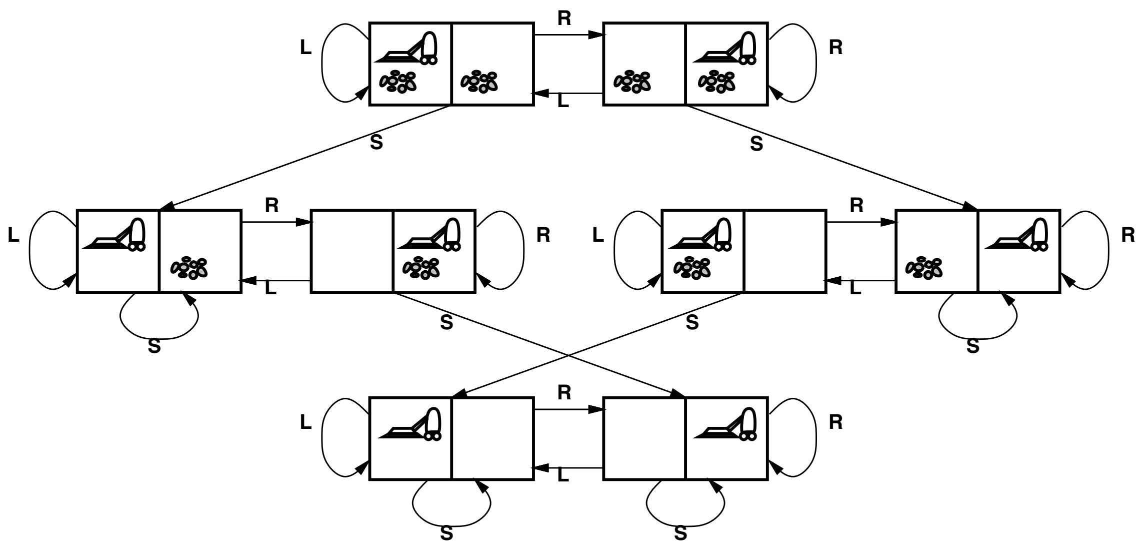
(Abstract) action = complex combination of real actions

e.g., "Arad $\rightarrow$ Zerind" represents a complex set
of possible routes, detours, rest stops, etc.

(Abstract) solution =

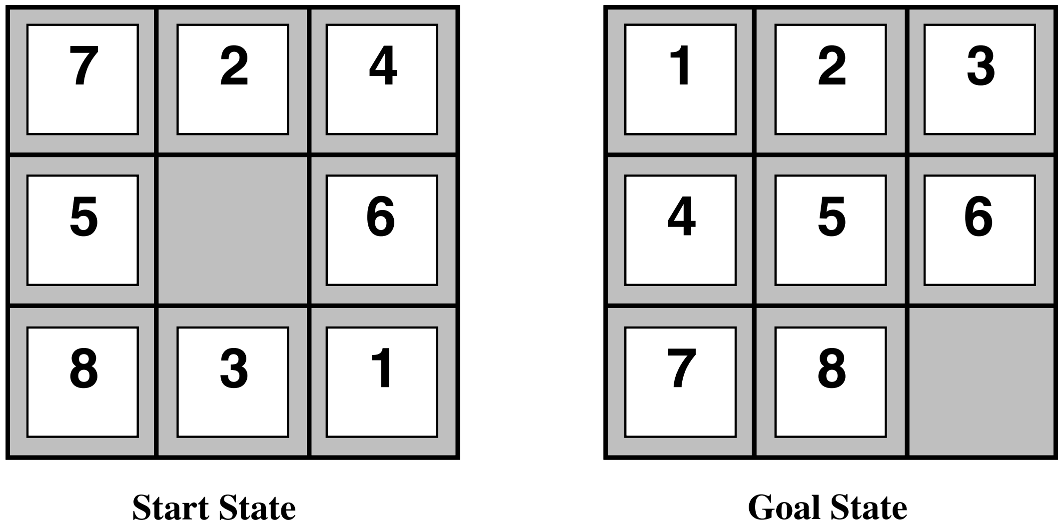
set of real paths that are solutions in the real world

Example: vacuum world state space graph



states: integer dirt and robot locations (ignore dirt *amounts*)
actions: $Left$, $Right$, $Suck$, $NoOp$
goal test: no dirt
path cost: 1 per action (0 for $NoOp$)

Example: The 8-puzzle



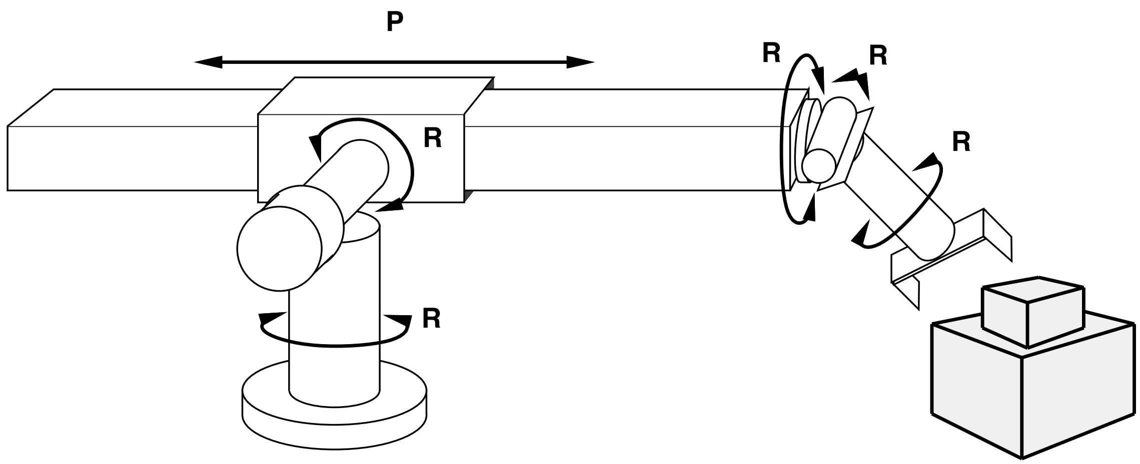
states: locations of tiles (ignore intermediate positions)
actions: move blank left, right, up, down (ignore unjam etc.)
goal test: = goal state (given)
path cost: 1 per move

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

Real world problems

- Find a flight itinerary
- VLSI design
- Mobile robot motion
- Assembly sequences
- Internet search

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 to the initial state of *problem*

loop do

    if there are no candidates for expansion

    then return failure

    choose a leaf node for expansion based on *strategy*

    if the node contains a goal state

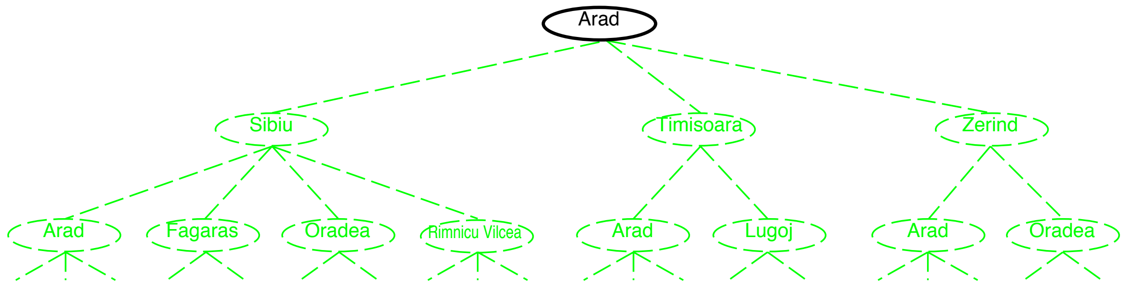
    then return the corresponding solution

    else expand the node and

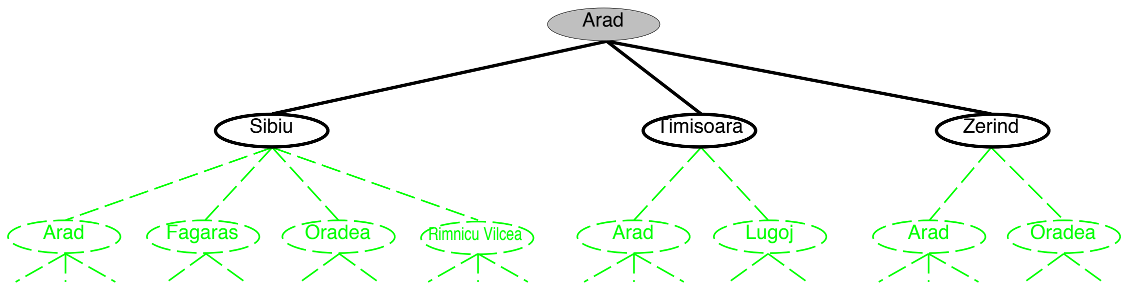
        add the resulting nodes to the search tree

end

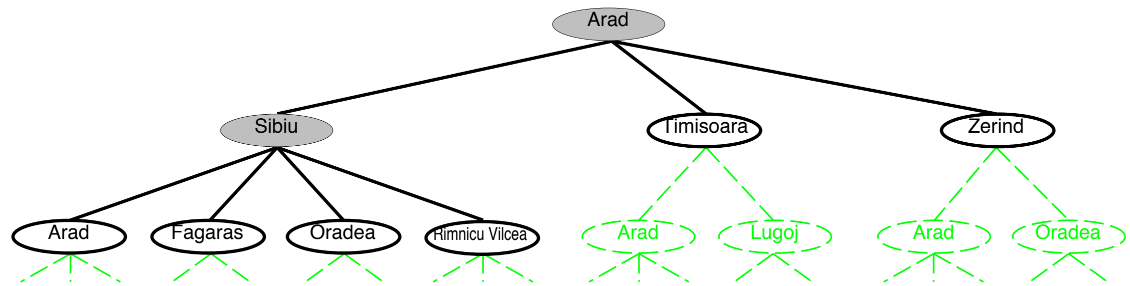
Tree search algorithms



Tree search algorithms



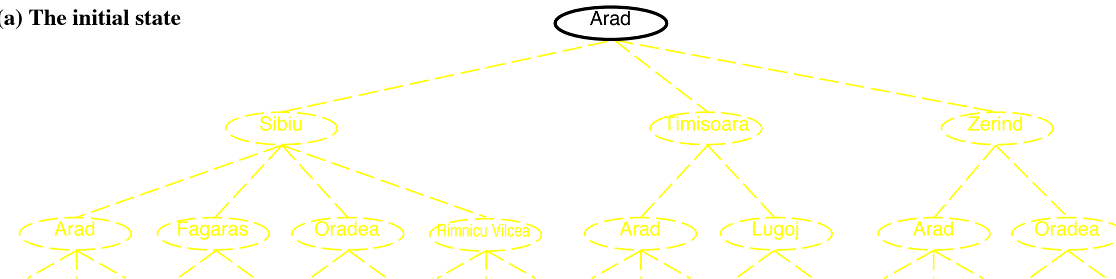
Tree search algorithms



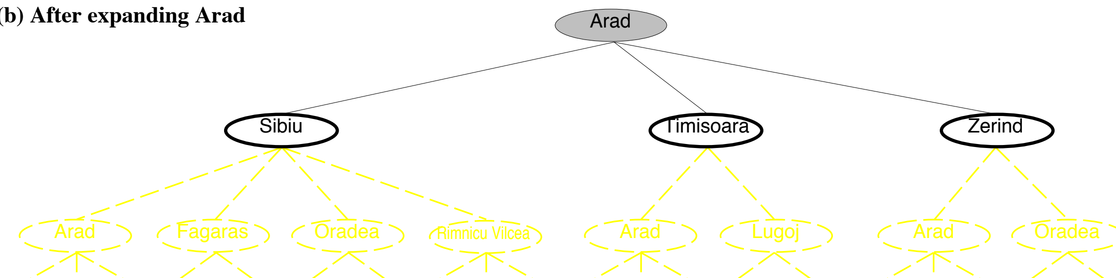
The search tree

- The **search tree** is composed by the search nodes and models the search process.
- **Expansion** is the process that builds successor nodes by applying the OPERATORS.
- The choice of the node to be expanded represents the **search strategy**.

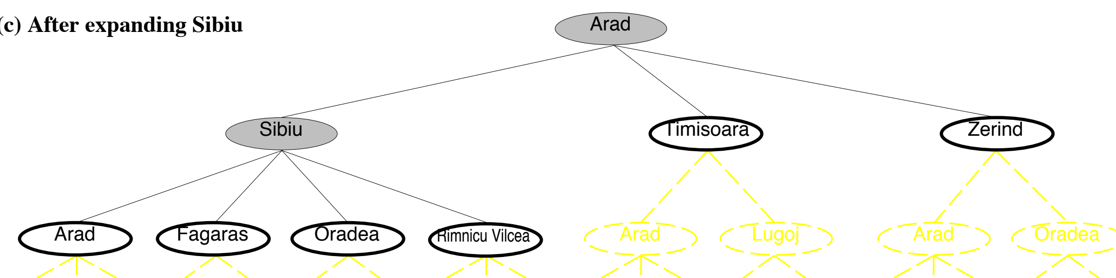
(a) The initial state



(b) After expanding Arad



(c) After expanding Sibiu

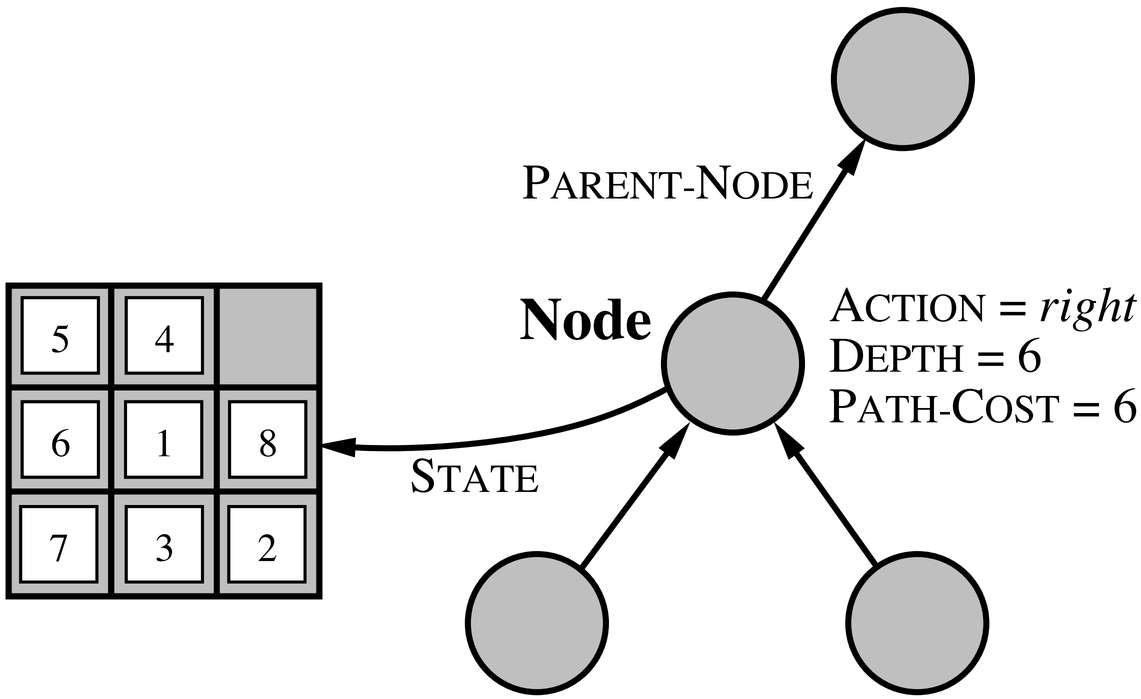


Implementation: states vs. nodes

A *state* is a (representation of) a physical configuration

A *node* is a data structure constituting part of a search tree
includes *parent*, *children*, *depth*, *path cost* $g(x)$

States do not have parents, children, depth, or path cost!



Implementation: general tree search

function TREE-SEARCH(*problem*, *fringe*) **returns** a solution, or failure

fringe ← INSERT(MK-NODE(INIT-STATE[*problem*]), *fringe*)

loop do

  if *fringe* is empty **then return** failure

  node ← REMOVE-FRONT(*fringe*)

  if GOAL-TEST[*problem*] on STATE(*node*) succeeds

   then return *node*

  fringe ← INSALL(EXPAND(*node*, *problem*), *fringe*)

Fringe → Frontier

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*

Strategies are evaluated along the following dimensions:
completeness—does it always find a solution if one exists?
time complexity—number of nodes generated/expanded
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 strategies use only the information available in the problem definition

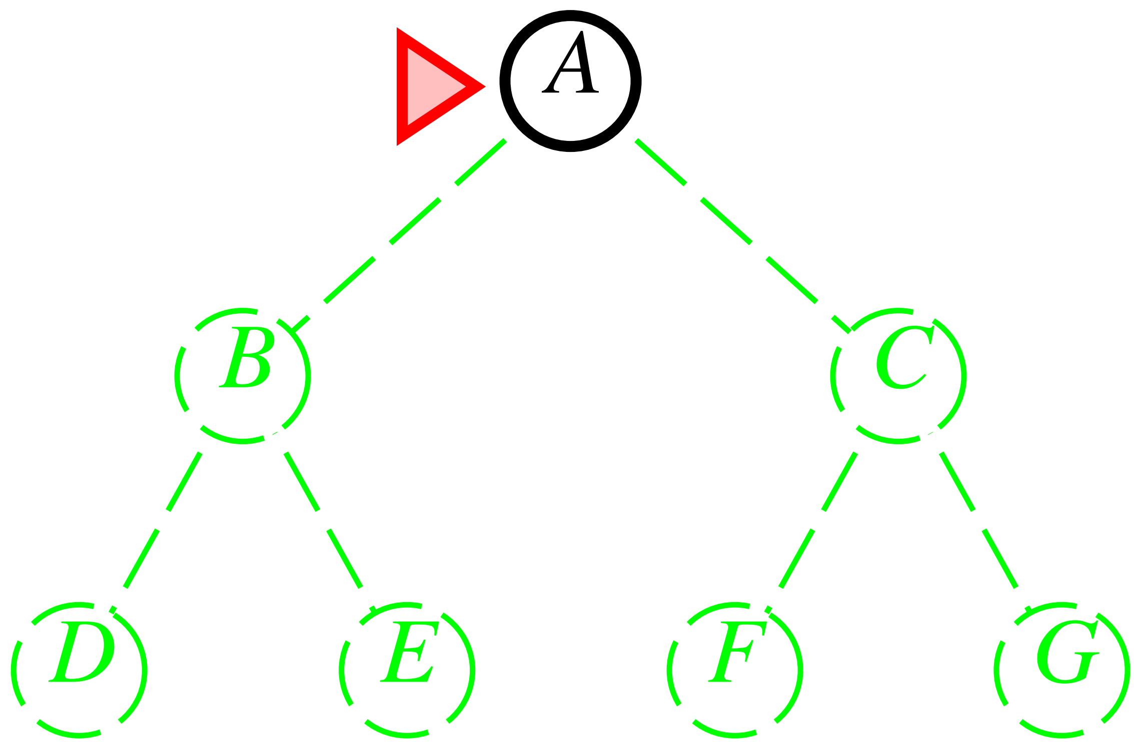
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end

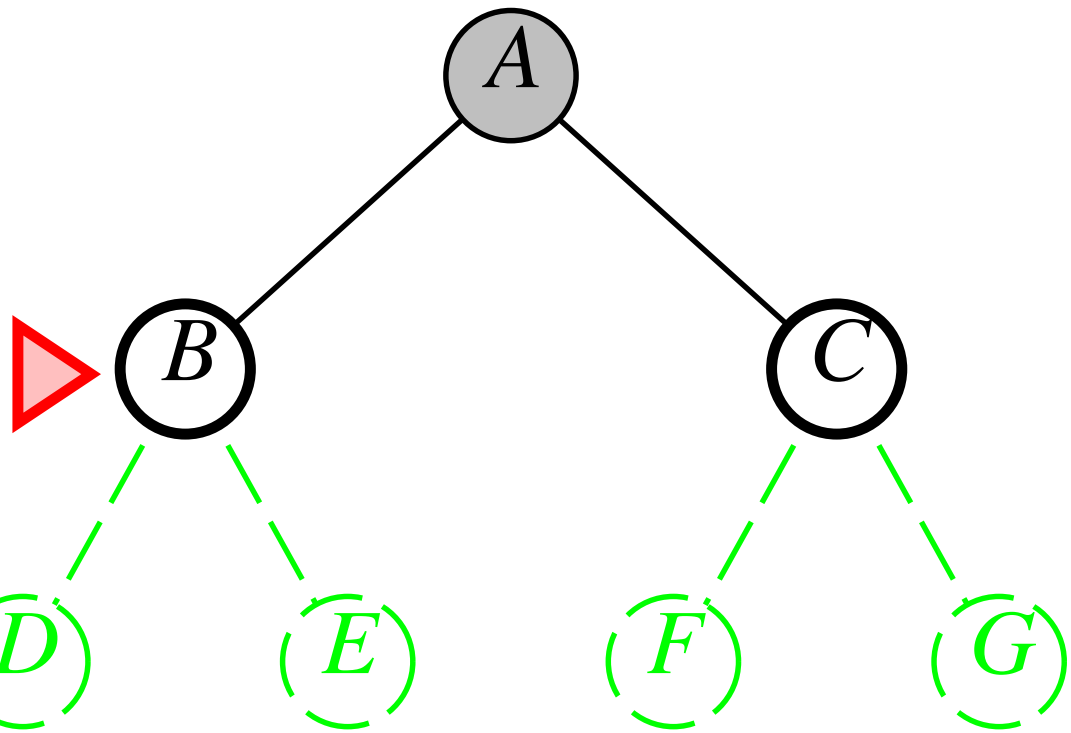


Breadth-first search

Expand shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end

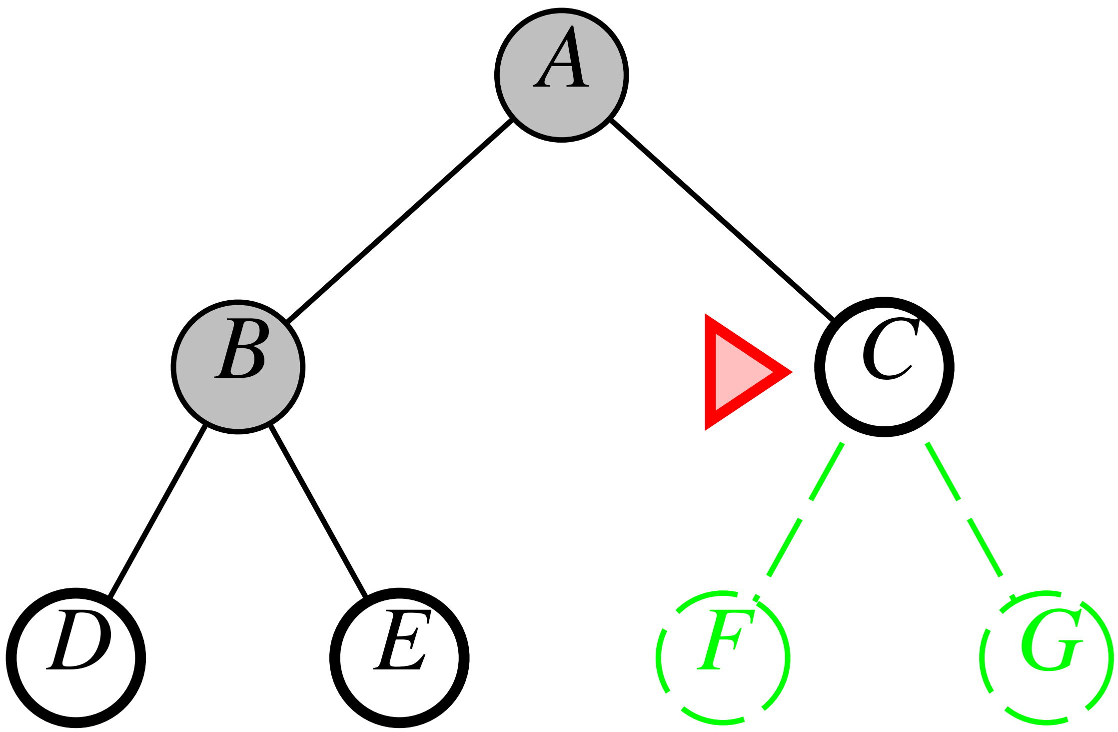


Breadth-first search

Expand shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end

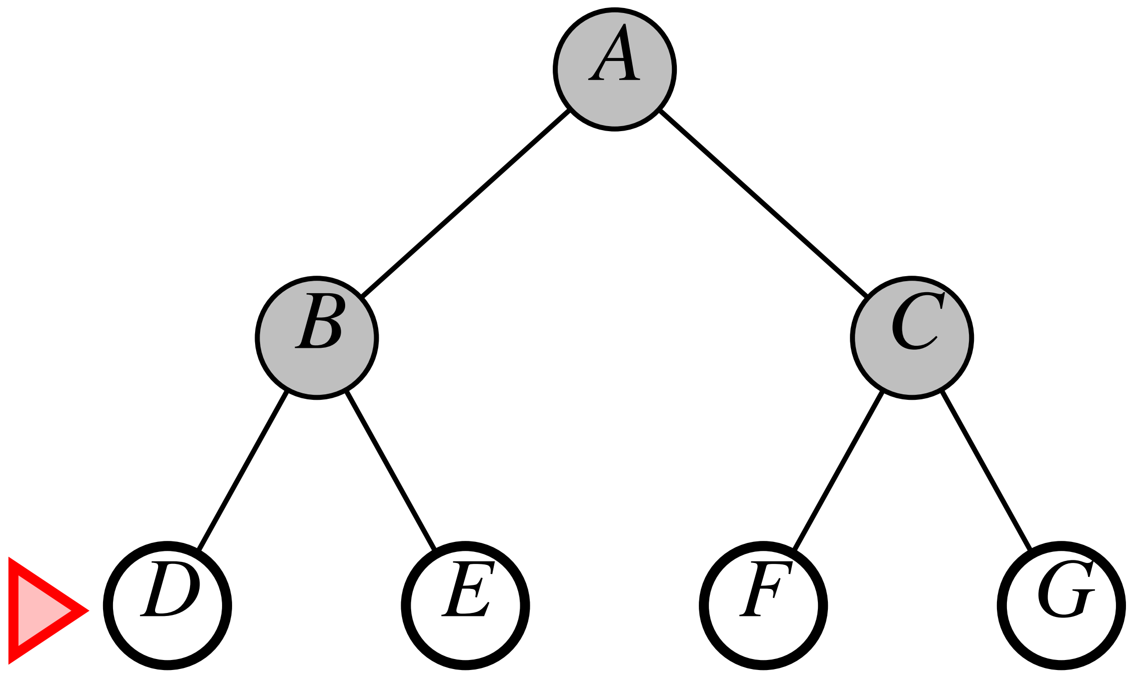


Breadth-first search

Expand shallowest unexpanded node

Implementation:

frontier is a FIFO queue, i.e., new successors go at end



Properties of breadth-first search

Complete Yes (if b is finite)

Time $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$, i.e., exp. in d

Space $O(b^d)$ (keeps every node in memory)

Optimal Yes (if cost $= 1$ per step); not optimal in general

Space is the big problem: with $b = 10$ and $D = 10$ would require 10 Terabytes (in about 3 hours time).

Uniform-cost search

Expand least-cost unexpanded node (Dijkstra)

Implementation:

$frontier =$ queue ordered by path cost

Equivalent to breadth-first if step costs all equal

Complete Yes, if step cost $\geq \epsilon$

Time # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil (1+C^*/\epsilon) \rceil})$, where C^* is the cost of the optimal solution

Space # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

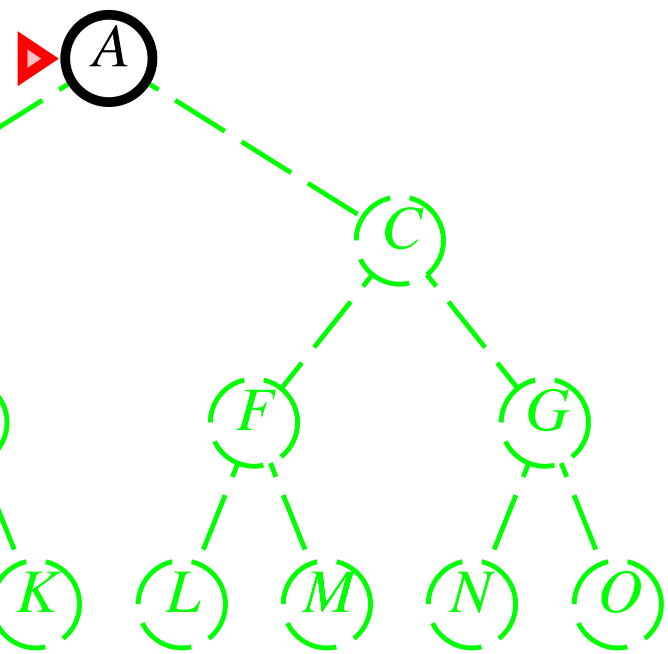
Optimal Yes—nodes expanded in increasing order of $g(n)$

Depth-first search

Expand deepest unexpanded node

Implementation:

$frontier =$ LIFO queue, i.e., put successors at front

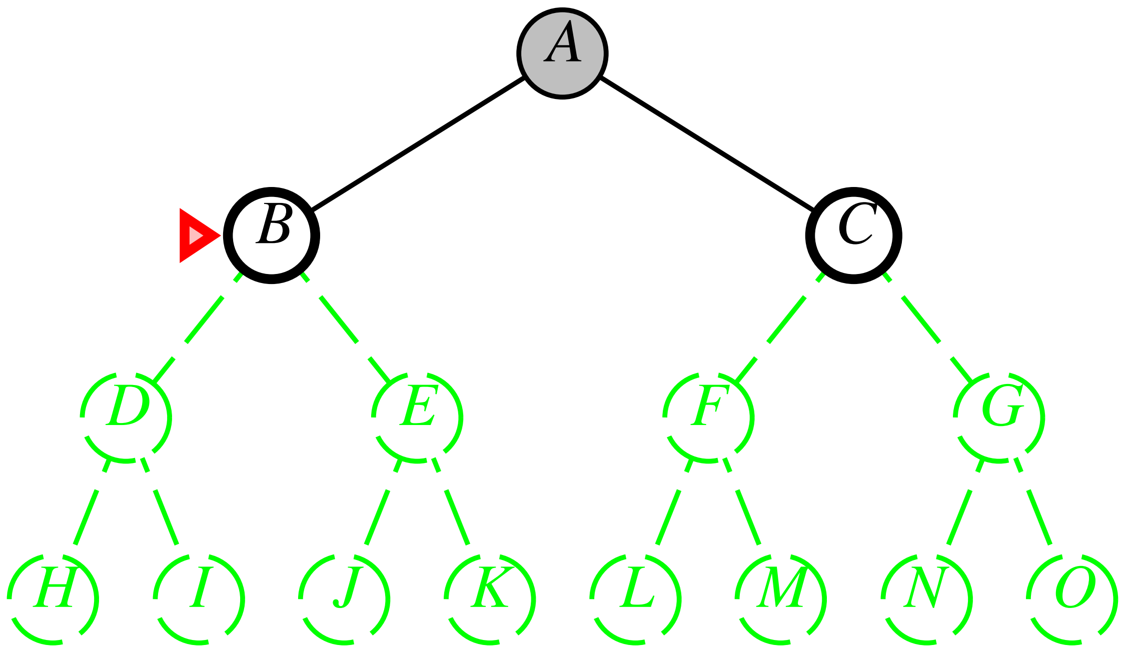


Depth-first search

Expand deepest unexpanded node

Implementation:

$frontier =$ LIFO queue, i.e., put successors at front

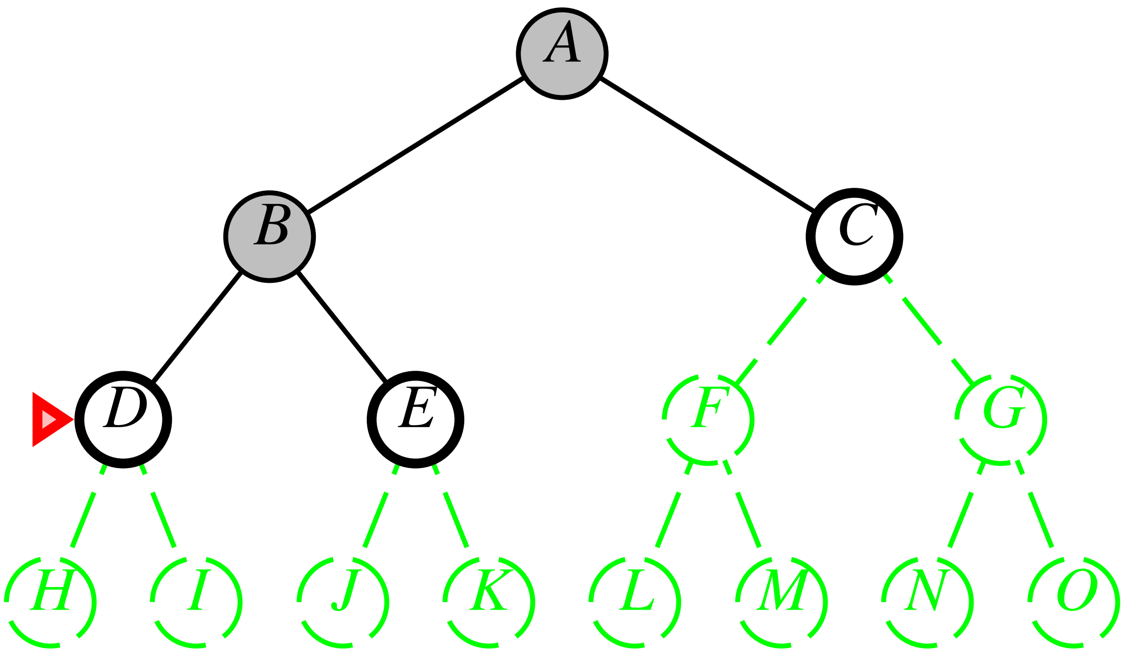


Depth-first search

Expand deepest unexpanded node

Implementation:

$frontier =$ LIFO queue, i.e., put successors at front

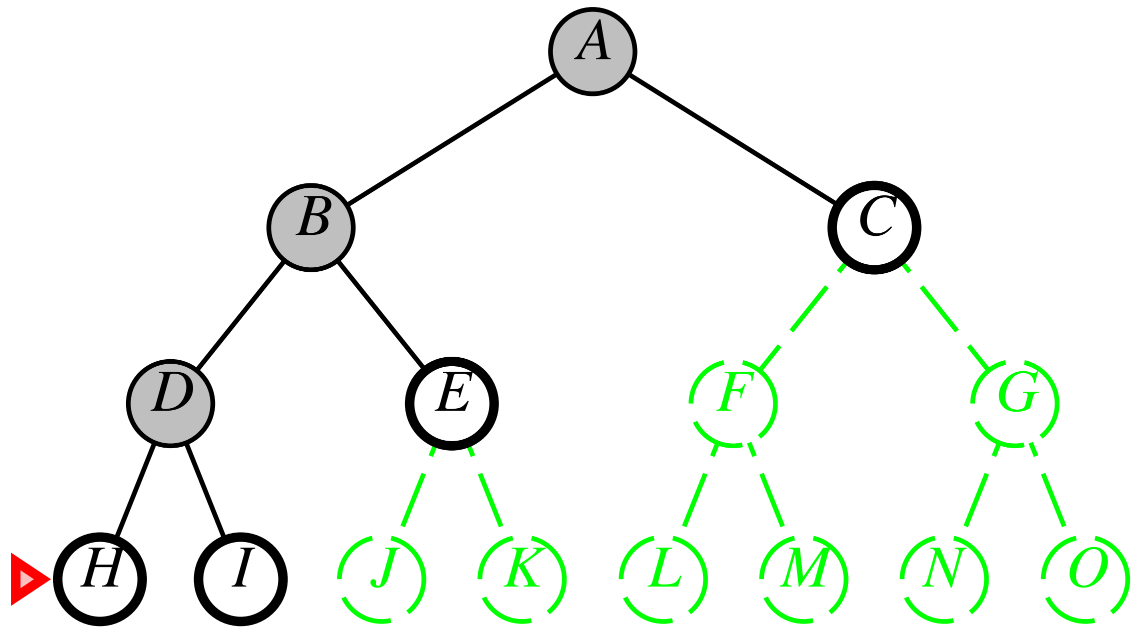


Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe$ $frontier$ = LIFO queue, i.e., put successors at front

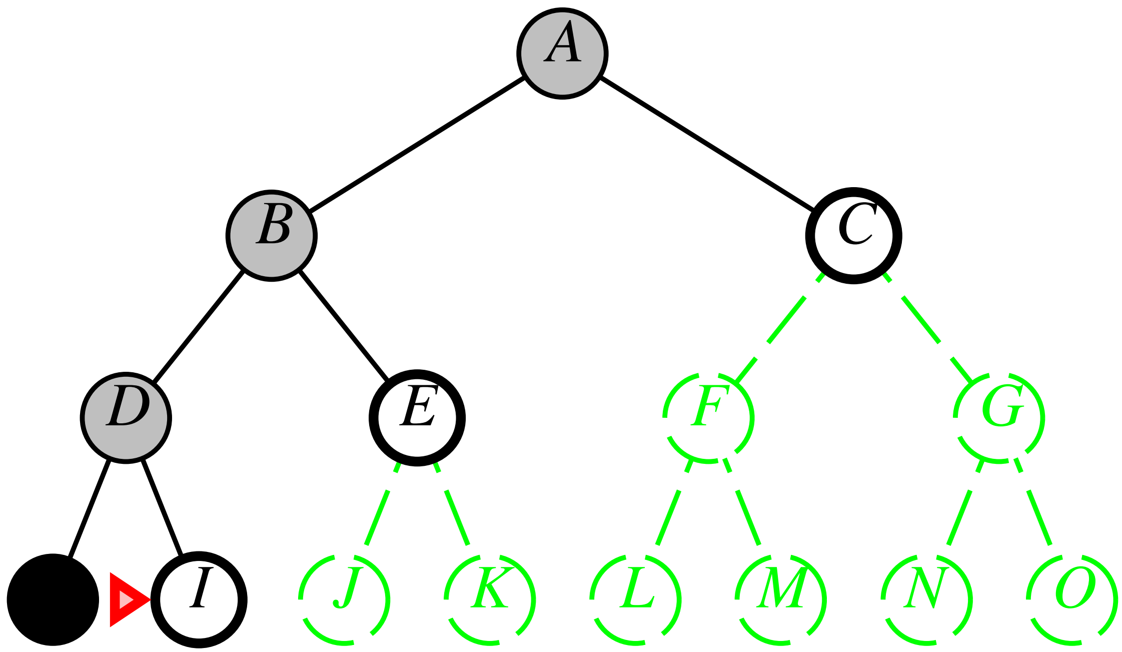


Depth-first search

Expand deepest unexpanded node

Implementation:

$frontier =$ LIFO queue, i.e., put successors at front

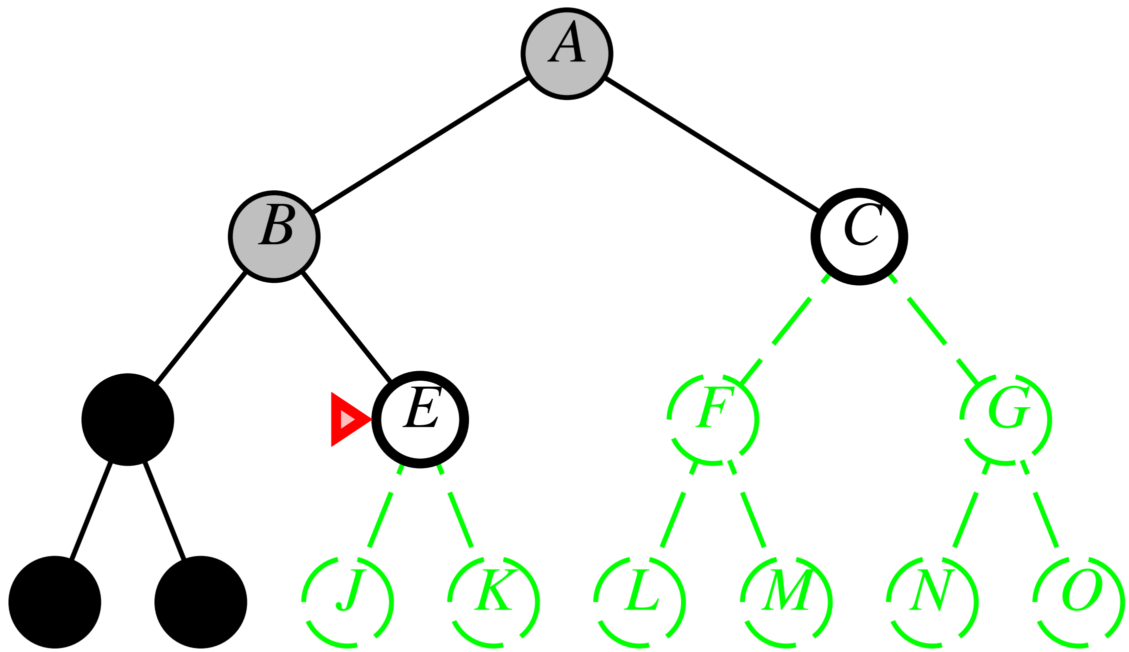


Depth-first search

Expand deepest unexpanded node

Implementation:

$frontier =$ LIFO queue, i.e., put successors at front

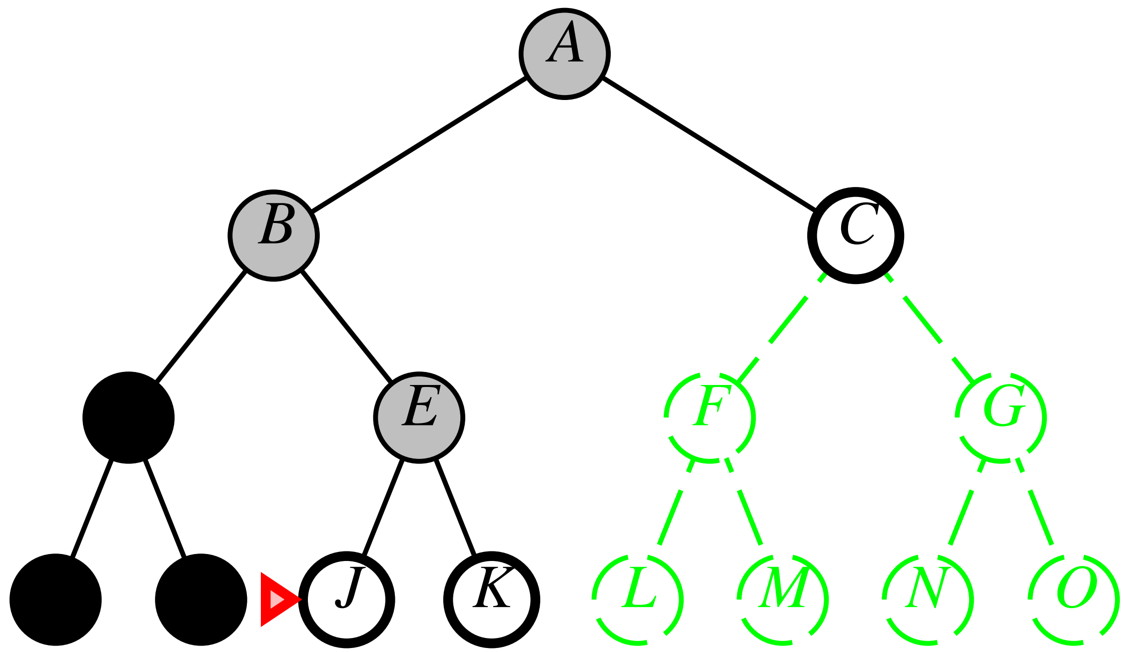


Depth-first search

Expand deepest unexpanded node

Implementation:

$frontier =$ LIFO queue, i.e., put successors at front

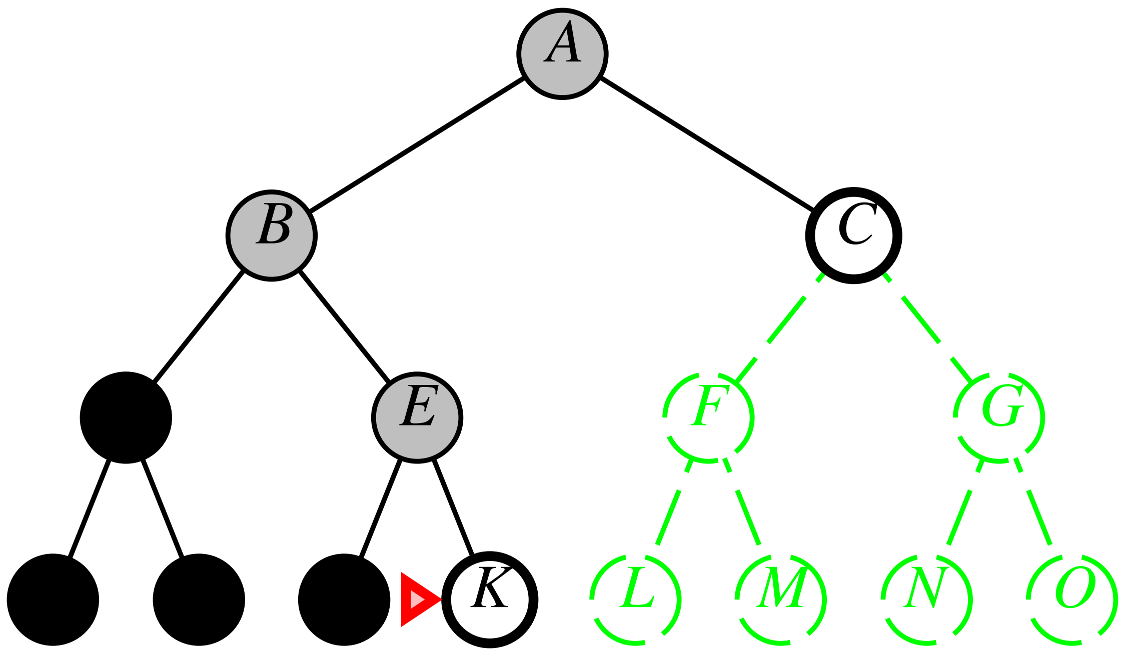


Depth-first search

Expand deepest unexpanded node

Implementation:

$frontier =$ LIFO queue, i.e., put successors at front

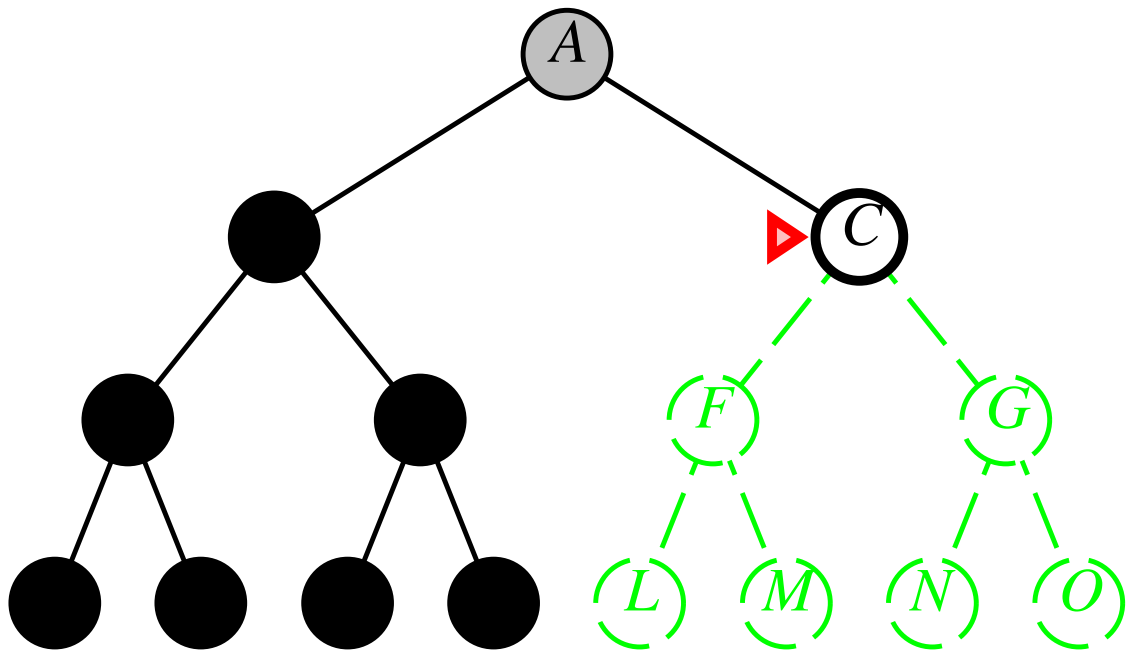


Depth-first search

Expand deepest unexpanded node

Implementation:

$fringe$ = 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
$\Rightarrow$ 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
REC-DLS(MAKE-NODE(INI-STATE[*problem*]), *problem*, *limit*)

function REC-DLS(*node*, *problem*, *limit*) **returns** soln/fail/cutoff
  cutoff-occurred? $\leftarrow$ false
  if GOAL-TEST[*problem*](STATE[*node*]) **then return** *node*
  else if DEPTH[*node*] $=$ *limit* **then return** *cutoff*
  else for each *successor* **in** EXPAND(*node*, *problem*) **do**
    result $\leftarrow$ RECURSIVE-DLS(*successor*, *problem*, *limit*)
    if *result* $=$ *cutoff* **then** *cutoff-occurred?* $\leftarrow$ true
    else if *result* $\neq$ *failure* **then return** *result*
  if *cutoff-occurred?* **then return** *cutoff* **else return** *failure*

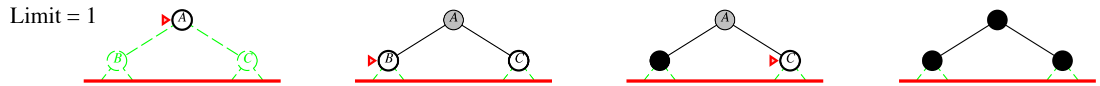
Iterative deepening search

function ITERATIVE-DEEPENING-SEARCH(*problem*)
returns a solution
  inputs: *problem*, a problem
  for *depth* $\leftarrow$ 0 **to** ∞ **do**
    result $\leftarrow$ DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* $\neq$ cutoff **then return** *result*
  end

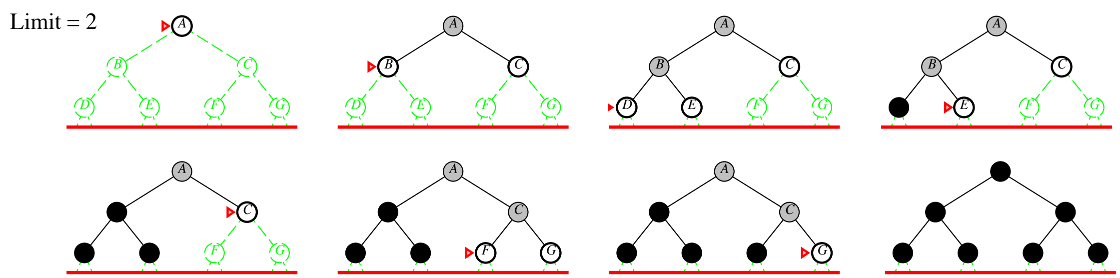
Iterative deepening search $l = 0$

Limit = 0

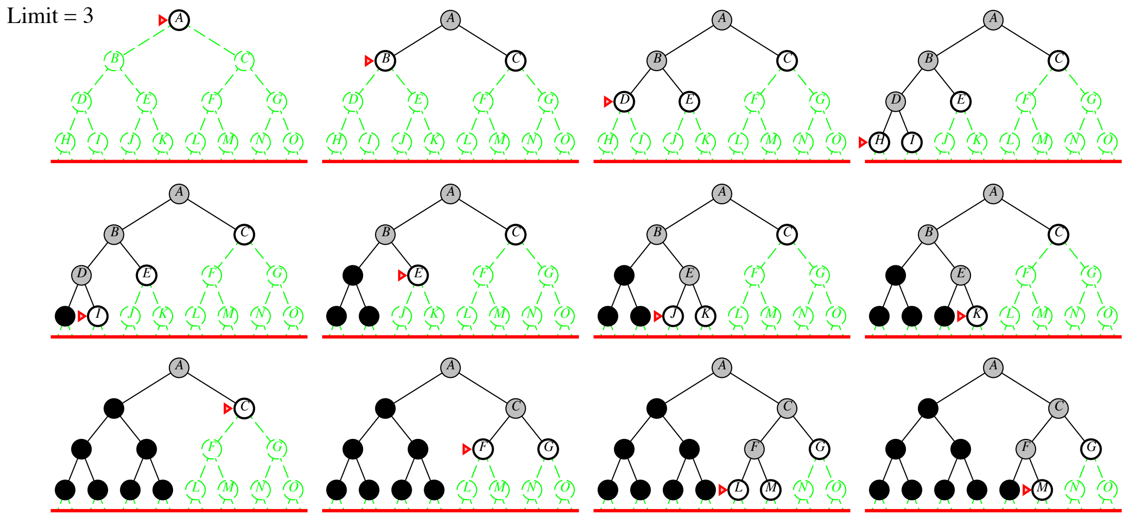
Iterative deepening search $l = 1$



Iterative deepening search $l = 2$



Iterative deepening search $l = 3$



Properties of iterative deepening search

Complete Yes

Time $db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space $O(bd)$

Optimal Yes, if step cost = 1

Can be modified to explore uniform-cost tree

Comparison for $b = 10$ and $d = 5$, solution at far right:

$$N(\mathsf{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\mathsf{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$

Search direction

Forward, or data driven
Backward, or goal driven
Bidirectional, or mixed

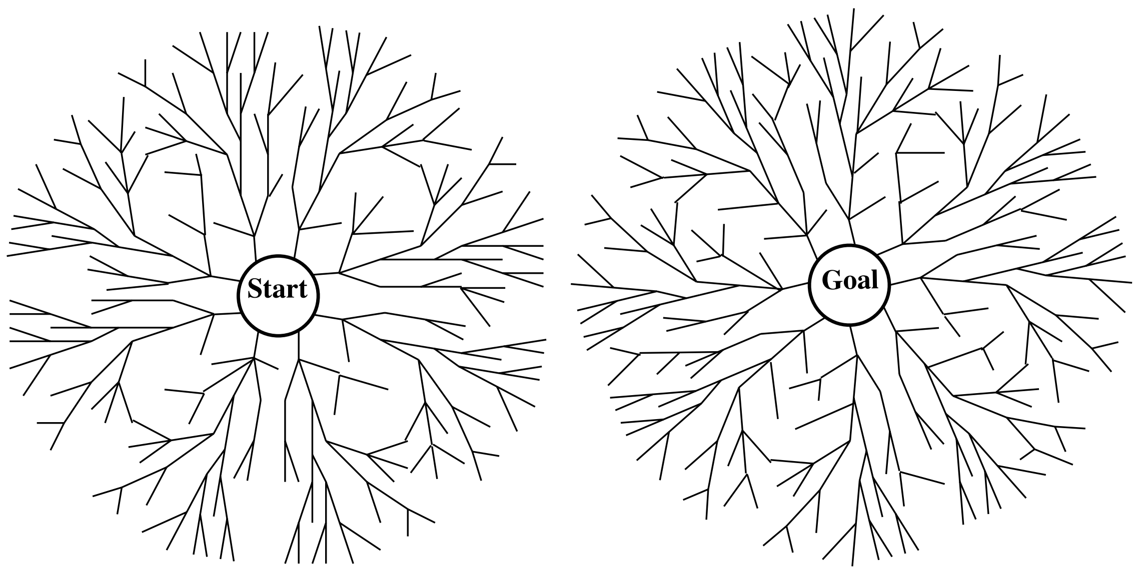
how do we choose?
1. Invertible operators?
2. "Structure" of the state space: branching factor, number of goal states, ecc.

Example of invertible operator:
move one step right $\rightarrow$ move one step left

Non invertible operators: chess mate

Bidirectional search

If a problem allows for both forward and backward search:
We start from the initial state in forward and form the goal state in backward, thying to meet "in the middle"



Properties of bidirectional search

Time $O(b^{d/2})$
Space $O(b^{d/2})$
Completeness ed **Optimality** depend on the techniques chosen for the search

Problems:
- check the meet nodes $O(b^{d/2})$
- goal states implicitely defined

Summary of algorithms

Crit	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iter Deep	Bid (if app)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optim	Yes	Yes	No	No	Yes	Yes
Compl	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

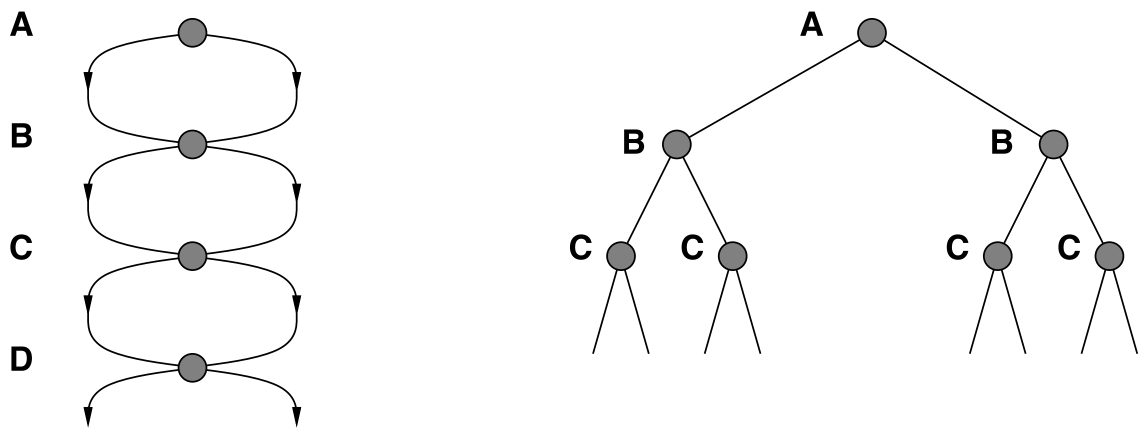
b is the branching factor (finite);
d is the depth pf the solution;
m is the maximum depth of the search tree;
l is the depth limit.

State repetition

- Avoid cyclic paths. The expansion step (or the set of operators) must avoid to generate successors that coincide with any ancestor.
- Avoid the generation of already generated states. Every state must be recorded in memory: space complexity $O(b^d)$ ($O(s)$, where s is the number of states). Hash tables.

Repeated states

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



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

closed ← an empty set

fringe ← INSERT(MAKE-NODE(INIT-STATE[*problem*]), *fringe*)

loop do

if *fringe* is empty **then return** failure

node ← REMOVE-FRONT(*fringe*)

if GOAL-TEST[*problem*](STATE[*node*])

then return *node*

if STATE[*node*] is not in *closed* **then**

add STATE[*node*] to *closed*

fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)

end

Summarizing

Problem formulation must lead to a search space that can be systematically explored by applying suitable operators

Abstraction is essential in problem formulation

There are several uninformed search strategies

Iterative deepening search uses linear space
and not much more time than other blind methods

Uninformed search allows to solve only small size problems

Missionaries and Cannibals

"Suppose you have a raw boat"   Robot ?

3 cannibals and 3 missionaries must cross a river with a small boat that can hold 2 passengers at most. The number of missionaries must always be more or equal wrt the number of cannibals, otherwise they are eaten by the cannibals.

How can the missionaries cross the river without being eaten?