Solving problems by searching

Chapter 3

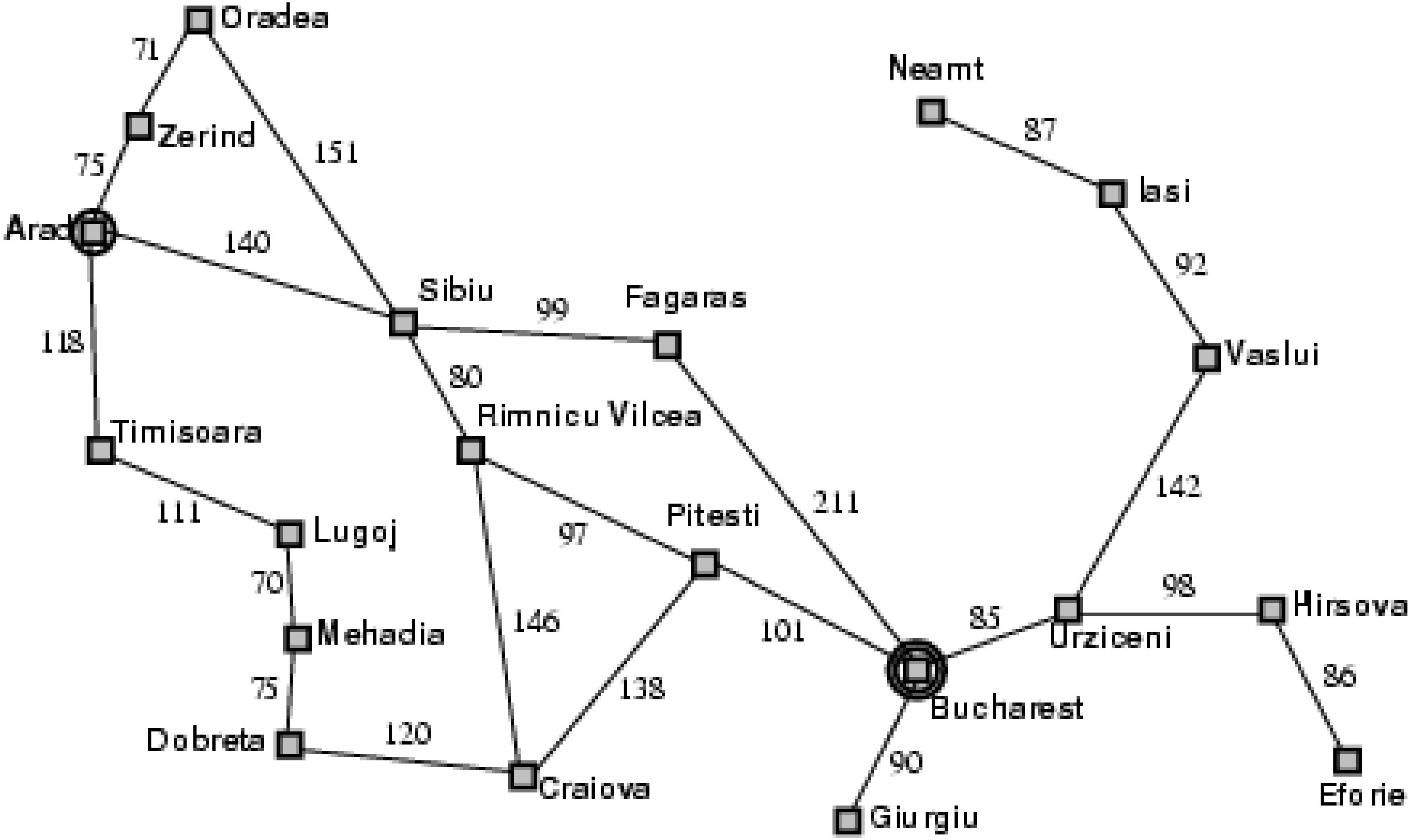
Outline

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

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-solving agent

Restricted form of general agent; solution executed “eyes closed“:

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*)

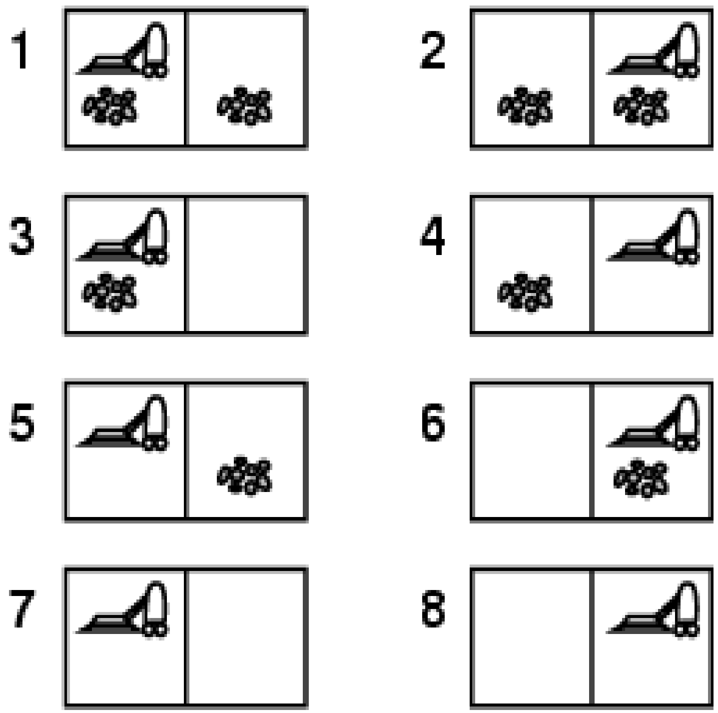
return *action*

Problem types

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable → sensor-less problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence
- Partially observable → contingency problem
  - Perception provides new information about current state
  - Often interleave search, execution
- Unknown state space → exploration problem
  - When states and actions of the environment are unknown

Example: vacuum world

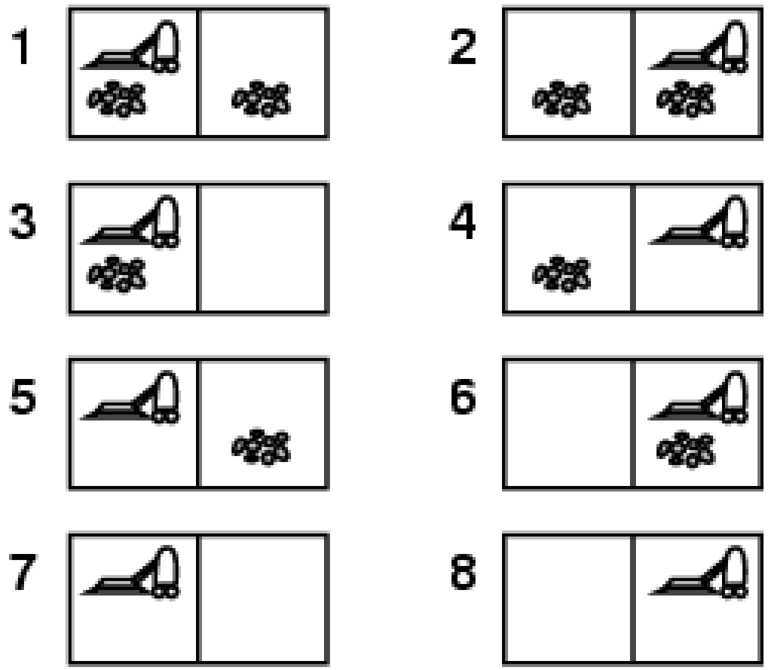
- Single-state, start in #5.
  Solution?



Example: vacuum world

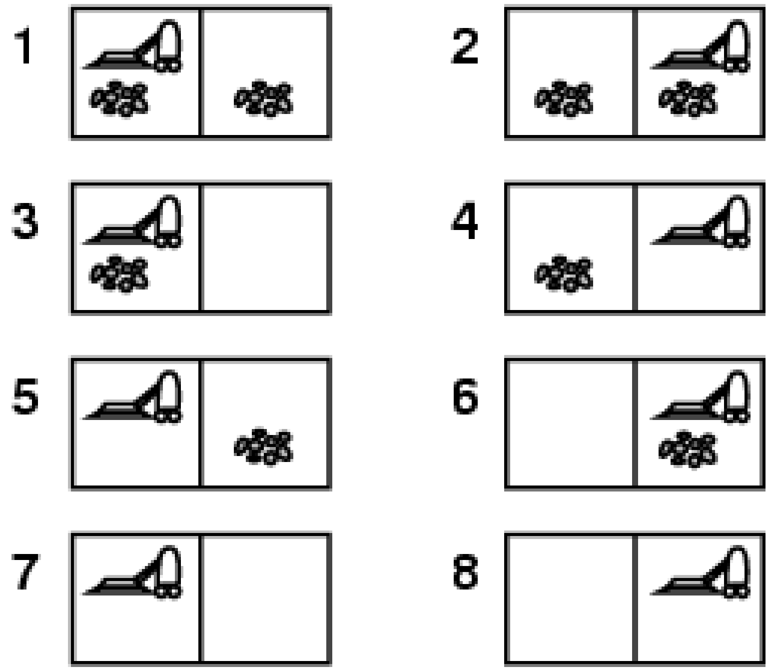
- Single-state, start in #5.
  Solution? *[Right, Suck]*

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



Example: vacuum world

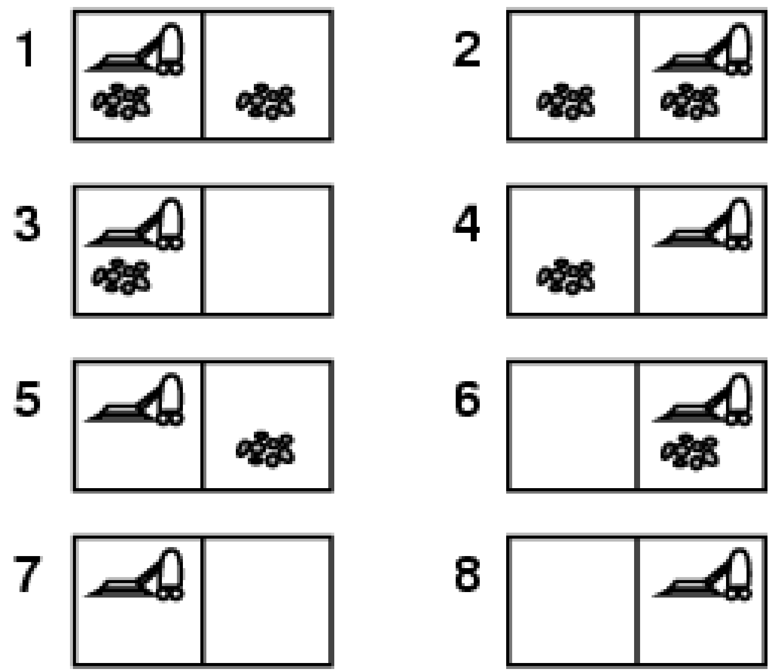
- Sensorless, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*}
  Solution?
  [Right,Suck,Left,Suck]



- 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:

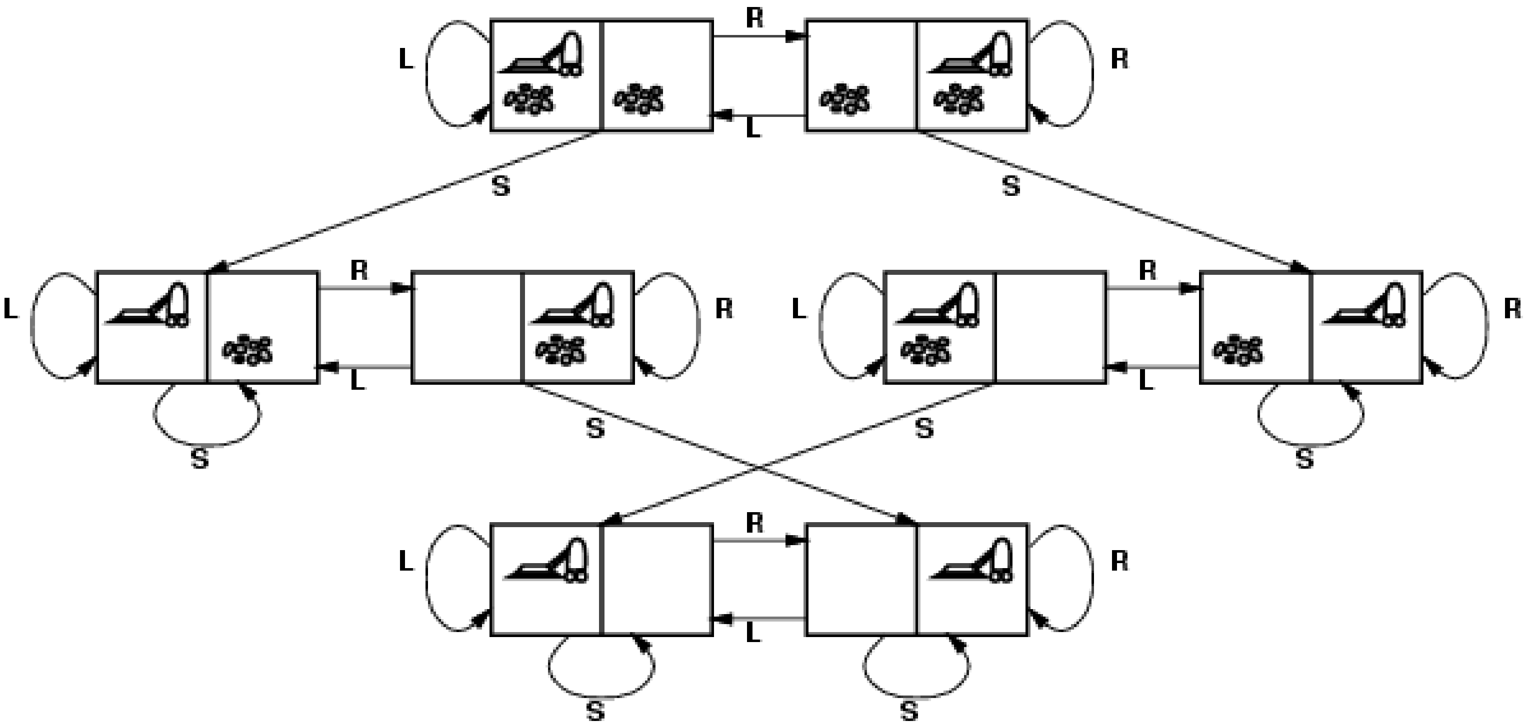
1. initial state, e.g. "at Arad"
2. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad)* = {*<Arad → Zerind, Zerind>, …* }
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   - e.g., sum of distances, 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

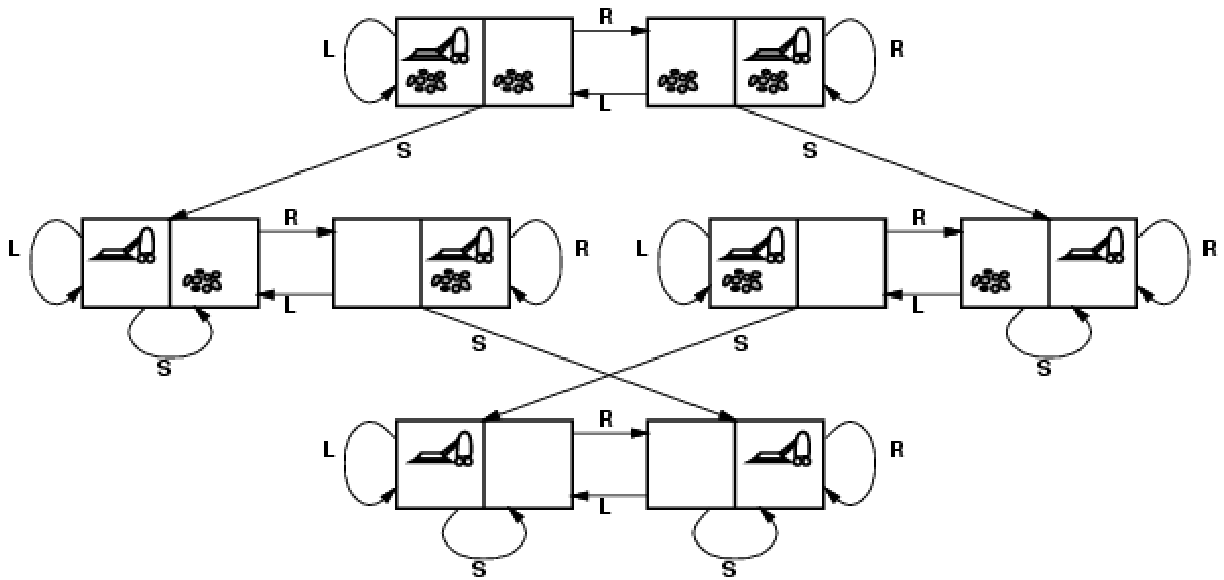
- Real world is absurdly complex
  → State space must be abstracted for problem solving
- (Abstract) state corresponds to set of real states
- (Abstract) action corr. to complex combination of real actions
  - E.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad“ must get to some real state "in Zerind"
- (Abstract) solution corresponds to
  - 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



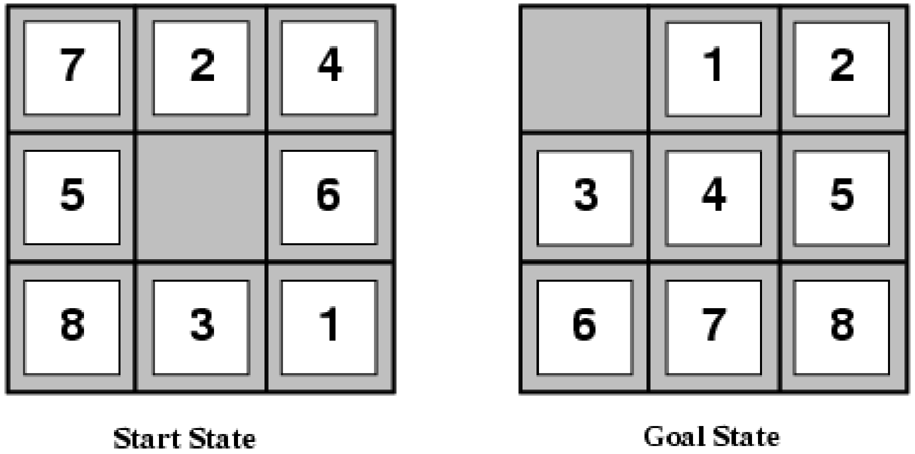
- States?
- Actions?
- Goal test?
- Path cost?

Vacuum world state space graph



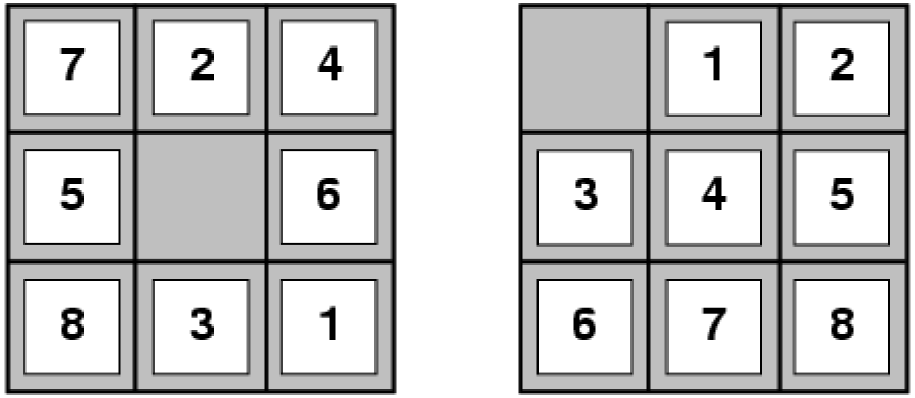
- <u>States?</u> two locations, dirt, and robot location
- <u>Actions?</u> *Left*, *Right*, *Suck*
- <u>Goal test?</u> no dirt at all locations
- <u>Path cost?</u> 1 per action

Example: The 8-puzzle



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

Example: The 8-puzzle

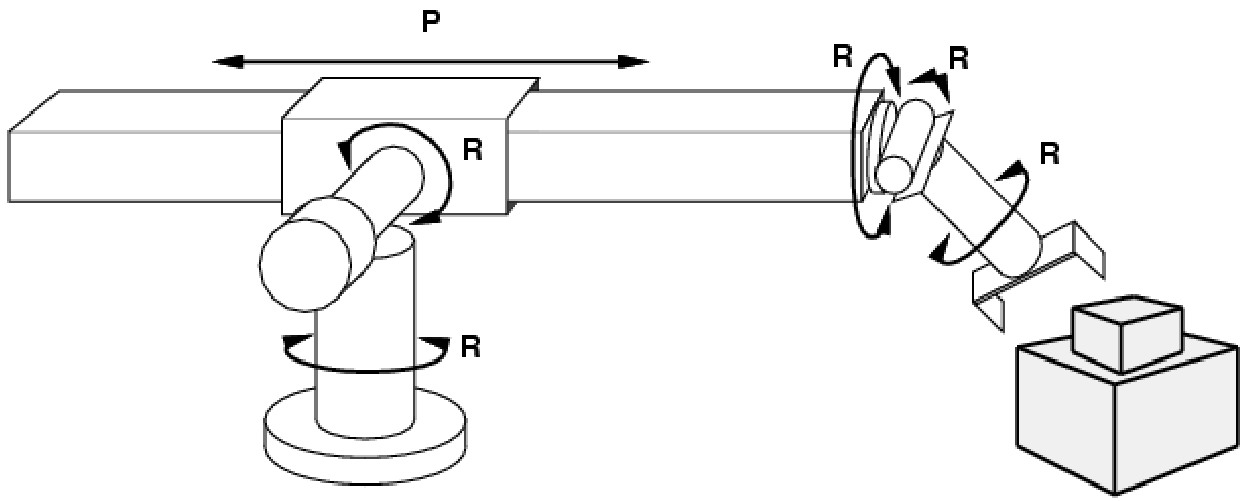


Start State Goal State

- States? locations of tiles
- Actions? move blank left, right, up, down
- Goal test? = goal state (given)
- Path cost? 1 per move

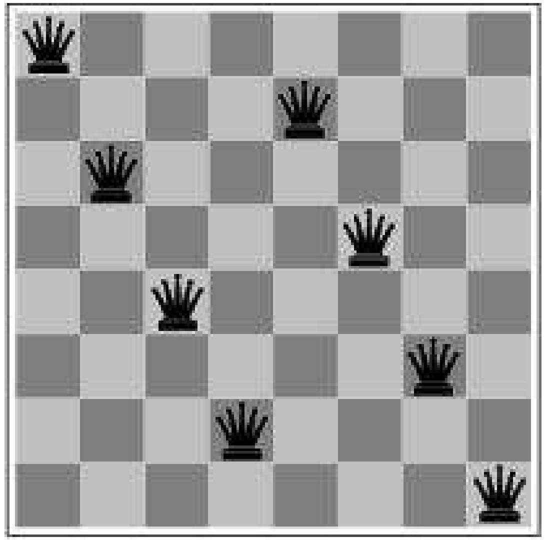
[Note: optimal solution of *n*-Puzzle family is NP-hard]

Example: robotic assembly



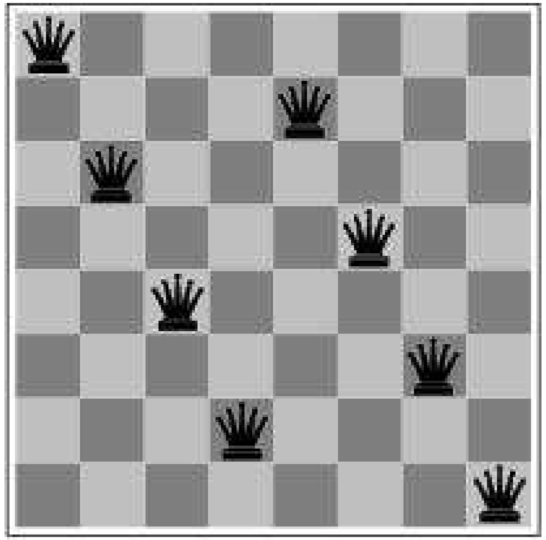
- States? real-valued coordinates of robot joint angles and parts of the object to be assembled
- Actions? continuous motions of robot joints
- Goal test? complete assembly
- Path cost? time to execute

Example: 8-queens problem



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

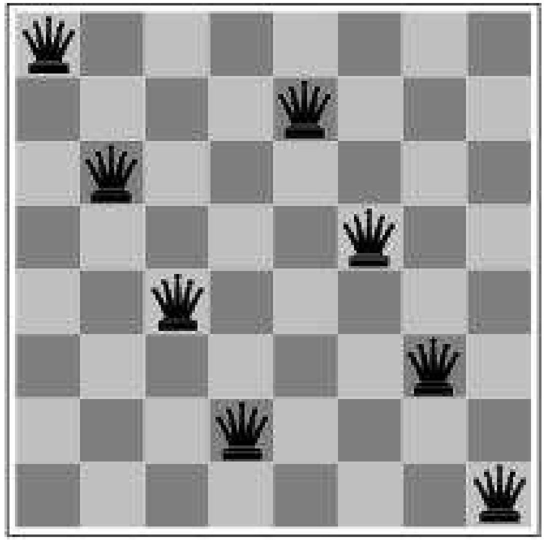
Example: 8-queens problem



Incremental formulation vs. complete-state formulation

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

Example: 8-queens problem

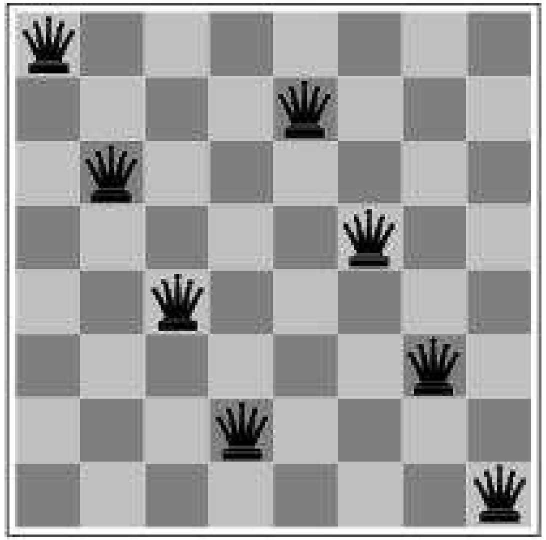


Incremental formulation

- 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

64*63*…*57 approx. 1.8 x 10^{14} possible sequences to investigate

Example: 8-queens problem



Incremental formulation (alternative)

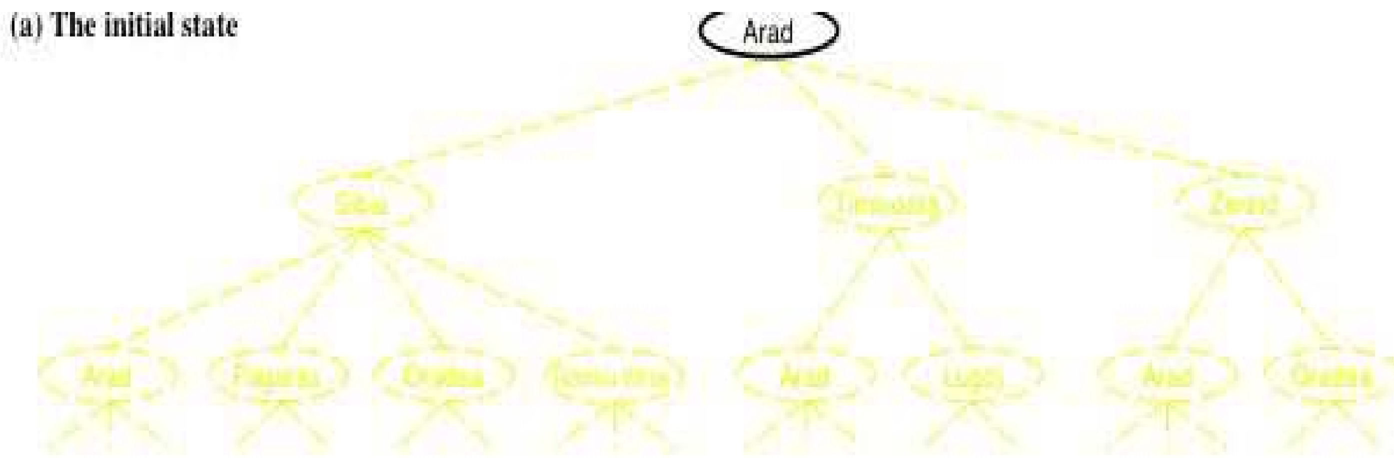
- States? *n* (0≤ *n*≤ 8) queens on the board, one per column in the *n* leftmost columns with no queen attacking another.
- Actions? Add queen in leftmost empty column such that is not attacking other queens

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)

Simple tree search example



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

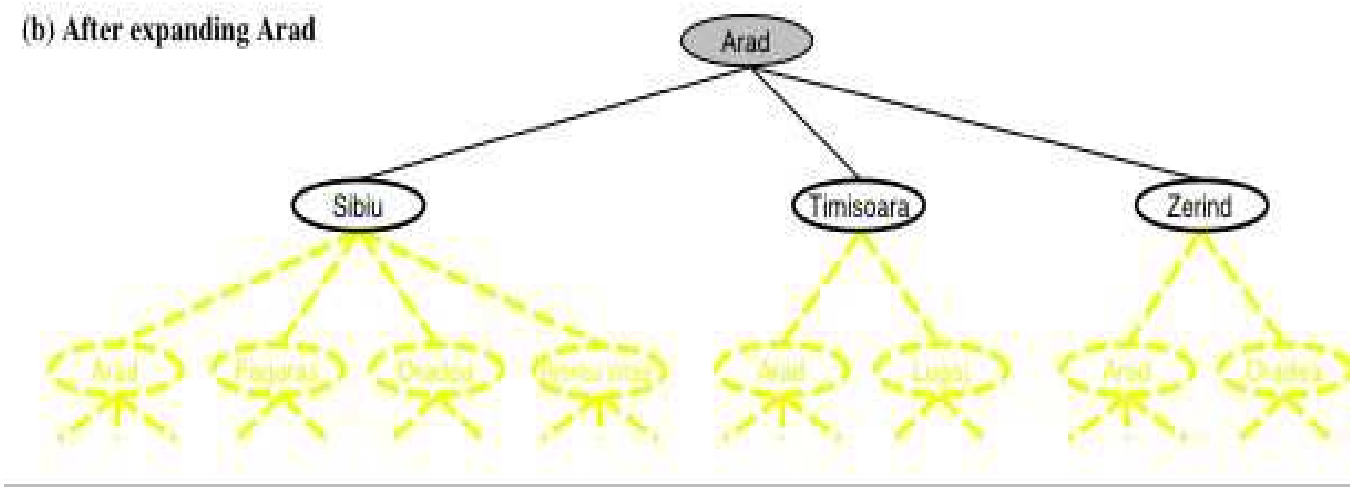
Initialize search tree to the *initial state* of the *problem*

do

if no candidates for expansion **then return** *failure*
choose leaf node for expansion according to *strategy*
if node contains goal state **then return** *solution*
else expand the node and add resulting nodes to the search tree

enddo

Simple tree search example



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

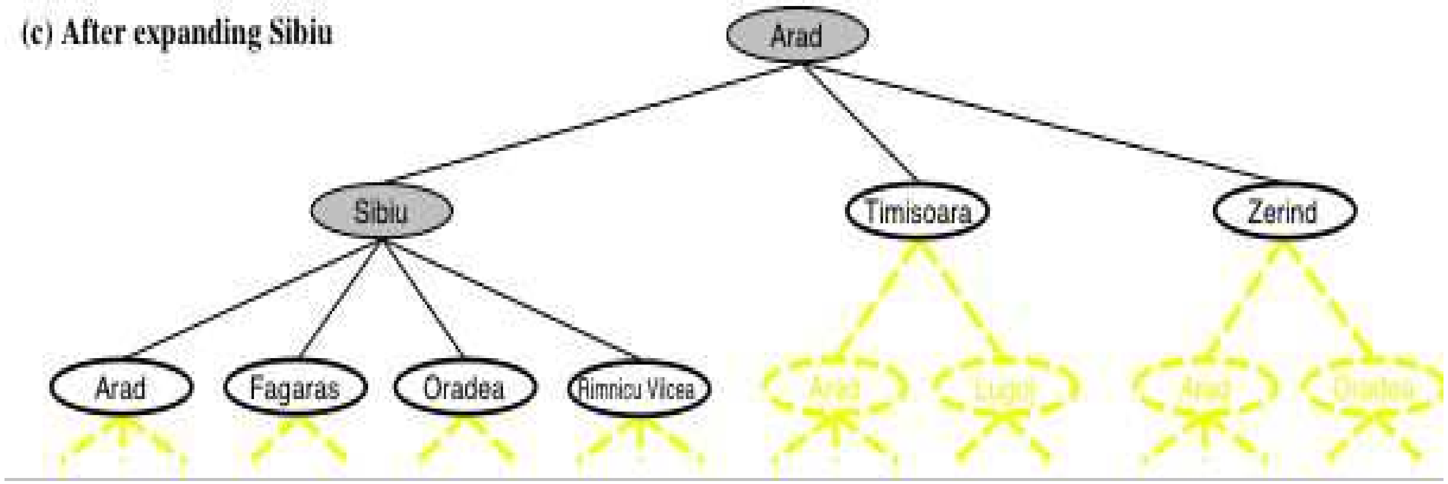
Initialize search tree to the *initial state* of the *problem*

do

if no candidates for expansion **then return** *failure*
choose leaf node for expansion according to *strategy*
if node contains goal state **then return** *solution*
else expand the node and add resulting nodes to the search tree

enddo

Simple tree search example



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

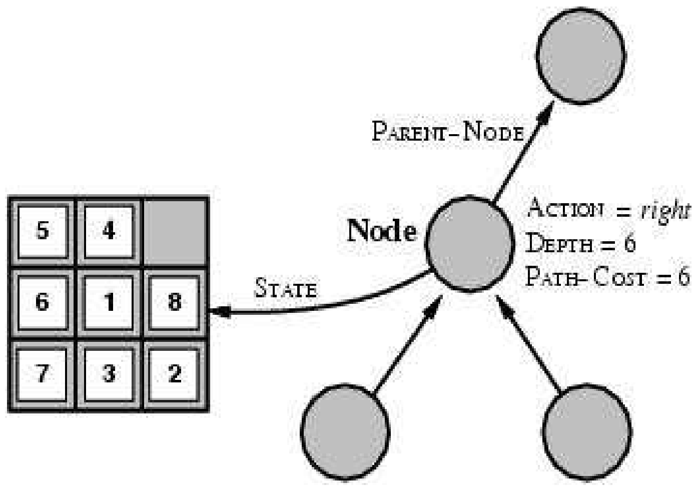
Initialize search tree to the *initial state* of the *problem*

do

if no candidates for expansion **then return** *failure*
choose leaf node for expansion according to *strategy* ←**Determines search process!!**
if node contains goal state **then return** *solution*
else expand the node and add resulting nodes to the search tree

enddo

State space vs. search tree



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

A *node* is a data structure belong to a search tree

- A node has a parent, children, … and includes path cost, depth, …
- Here *node*= *<state, parent-node, action, path-cost, depth>*
- *FRINGE*= contains generated nodes which are not yet expanded
  - White nodes with black outline

Tree search algorithm

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

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

loop do

if EMPTY?(*fringe*) **then return** failure

node ← REMOVE-FIRST(*fringe*)

if GOAL-TEST[*problem*] applied to STATE[*node*] succeeds

then return SOLUTION(*node*)

fringe ← INSERT-ALL(EXPAND(*node*, *problem*), *fringe*)

Tree search algorithm (2)

function EXPAND(*node,problem*) **return** a set of nodes

successors ← the empty set

for each <*action*, *result***> in** SUCCESSOR-FN**[***problem***]**(STATE[*node*]) **do**

s ← a new NODE

STATE[*s*] ← *result*

PARENT-NODE[*s*] ← *node*

ACTION[*s*] ← *action*

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 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

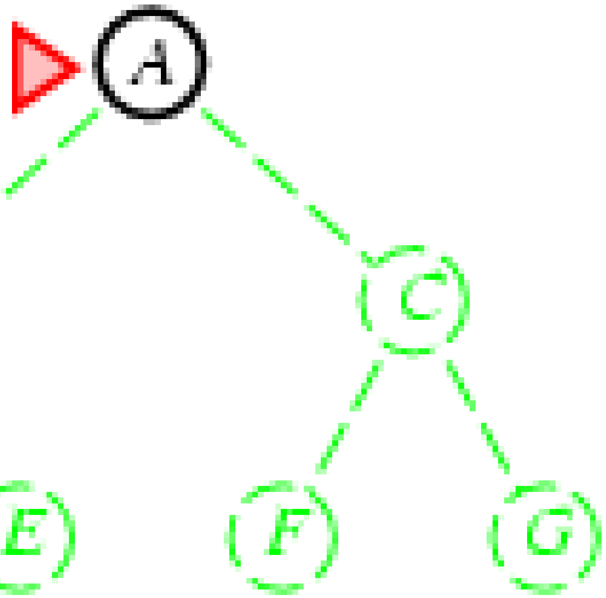
Uninformed search strategies use only the information available in the problem definition

When strategies can determine whether one non-goal state is better than another → informed search

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

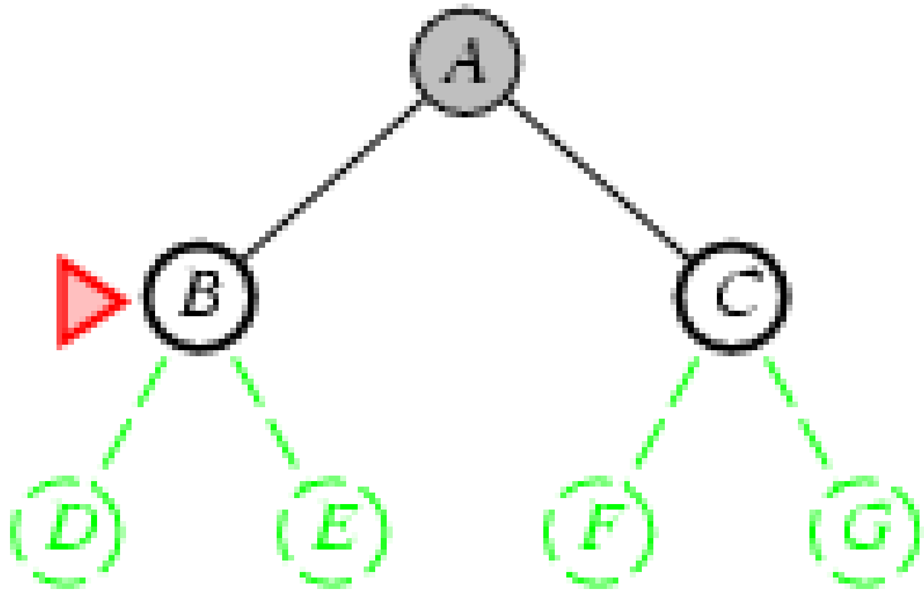
Breadth-first search

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



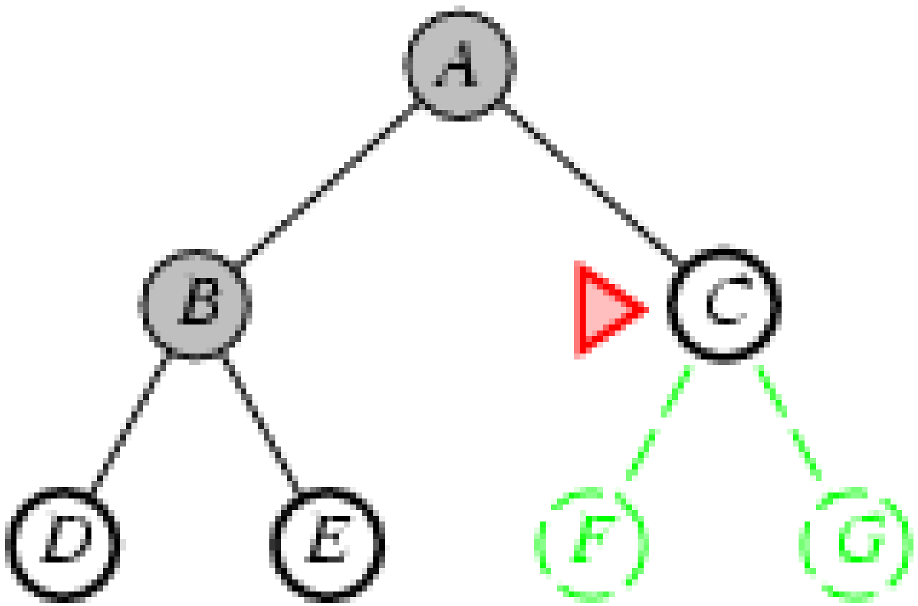
Breadth-first search

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



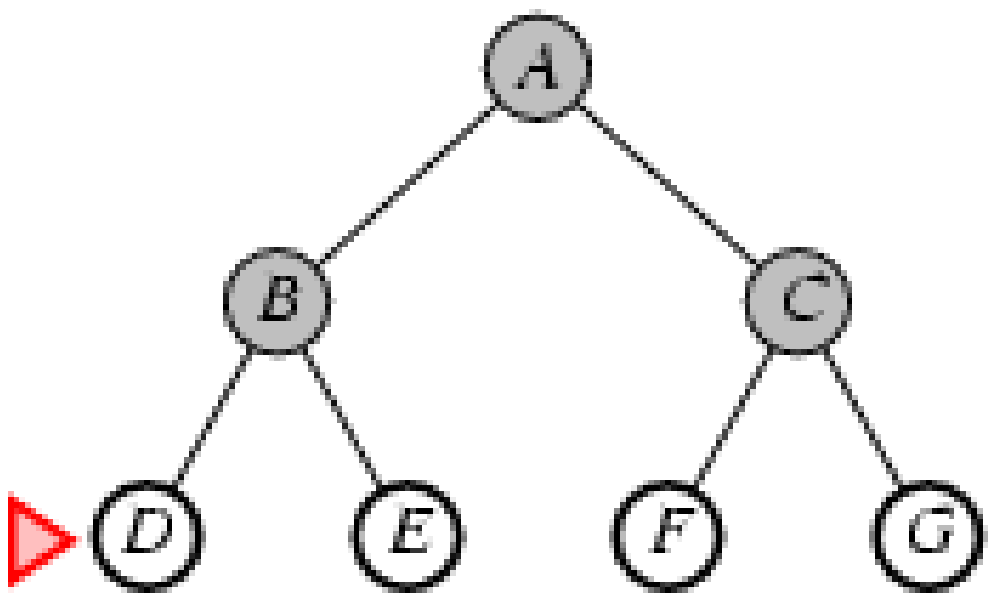
Breadth-first search

- 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)

BF-search; evaluation

b=10; 10.000 nodes/sec; 1000 bytes/node

DEPTH	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

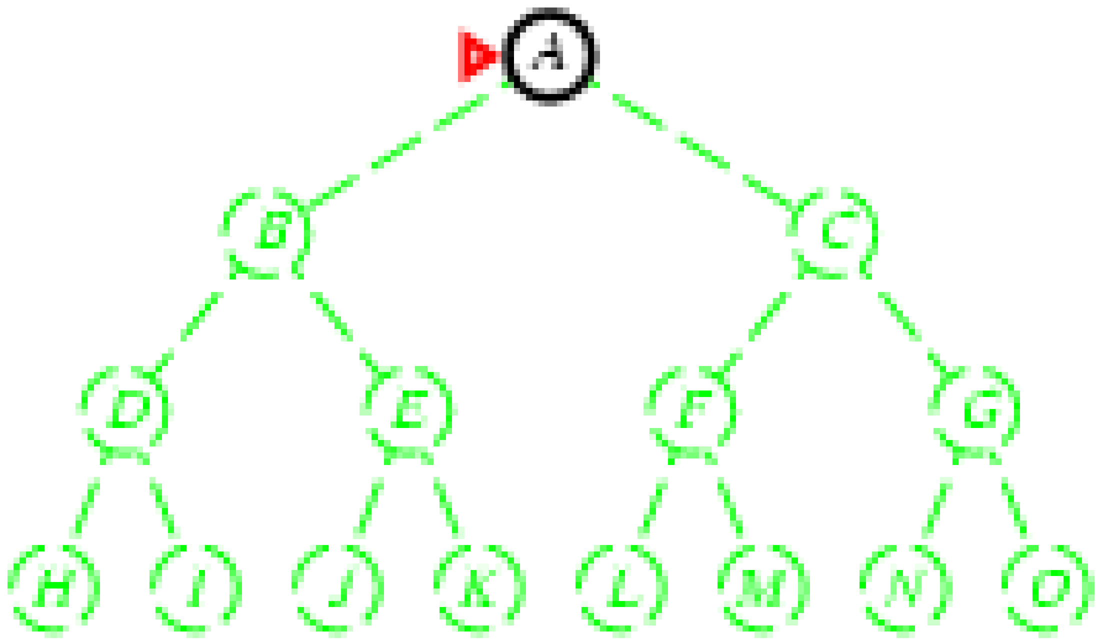
- Two lessons:
  - Memory requirements are a bigger problem than its execution time
  - Uniformed search only applicable for small instances
    -> Exploit knowledge about the problem

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 ≤ cost of optimal solution, $O(b^{1+floor(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- Space? # of nodes with g ≤ cost of optimal solution, $O(b^{1+floor(C^*/\varepsilon)})$
- Optimal? Yes – nodes expanded in increasing order of *path costs*

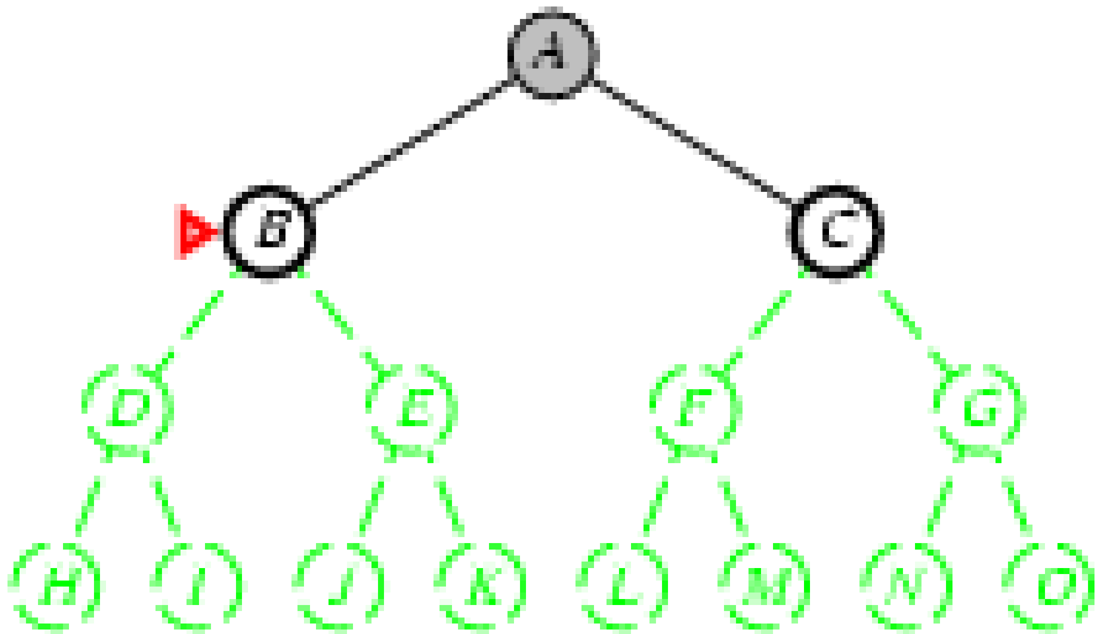
Depth-first search

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



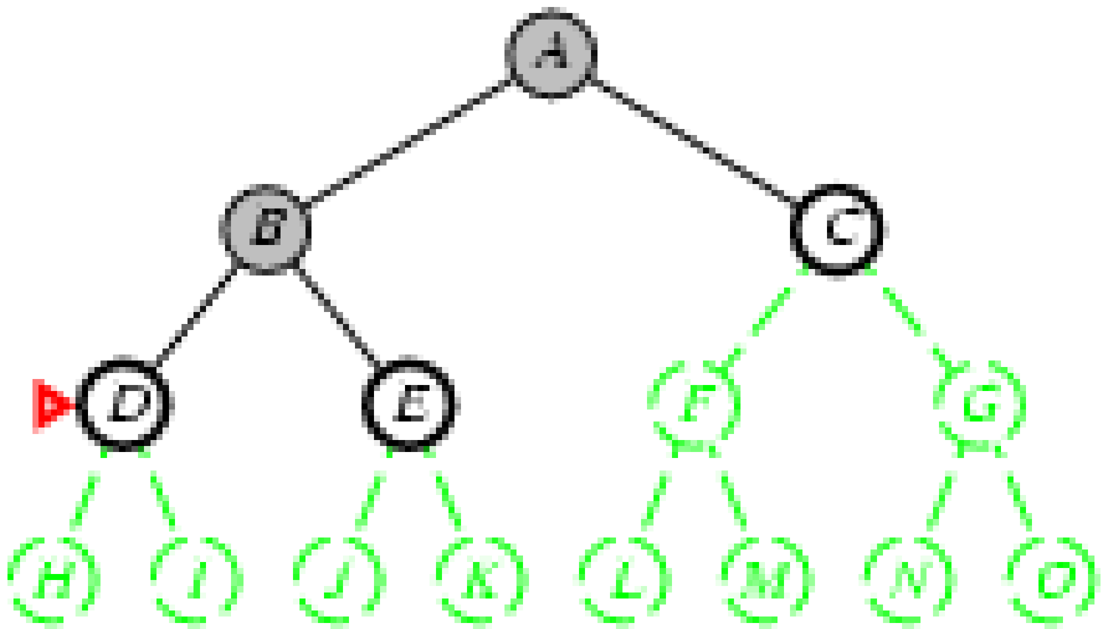
Depth-first search

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



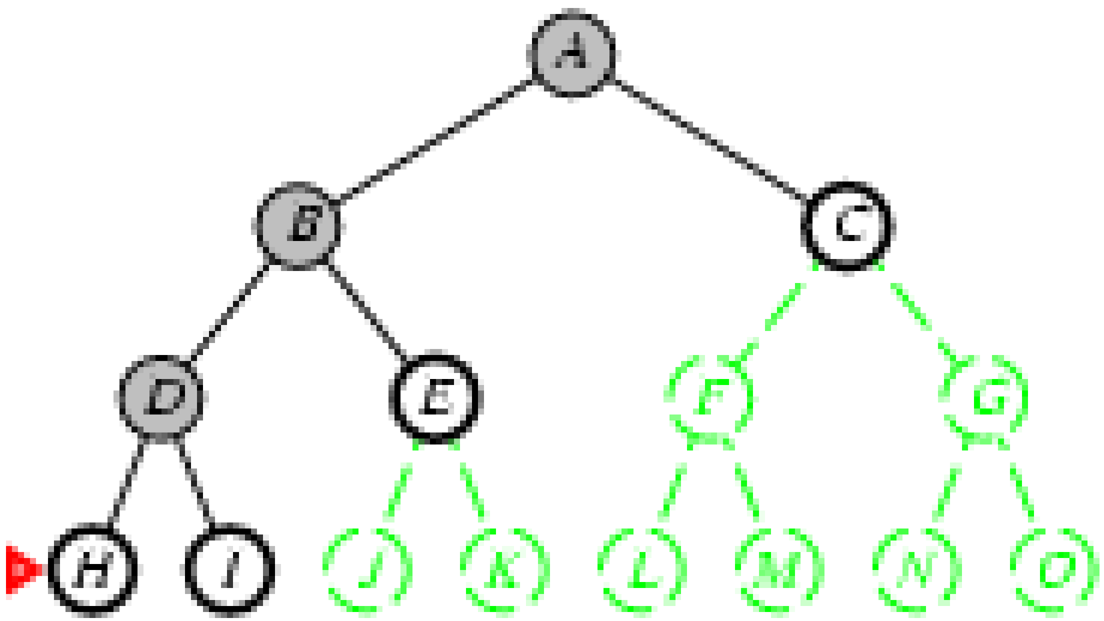
Depth-first search

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



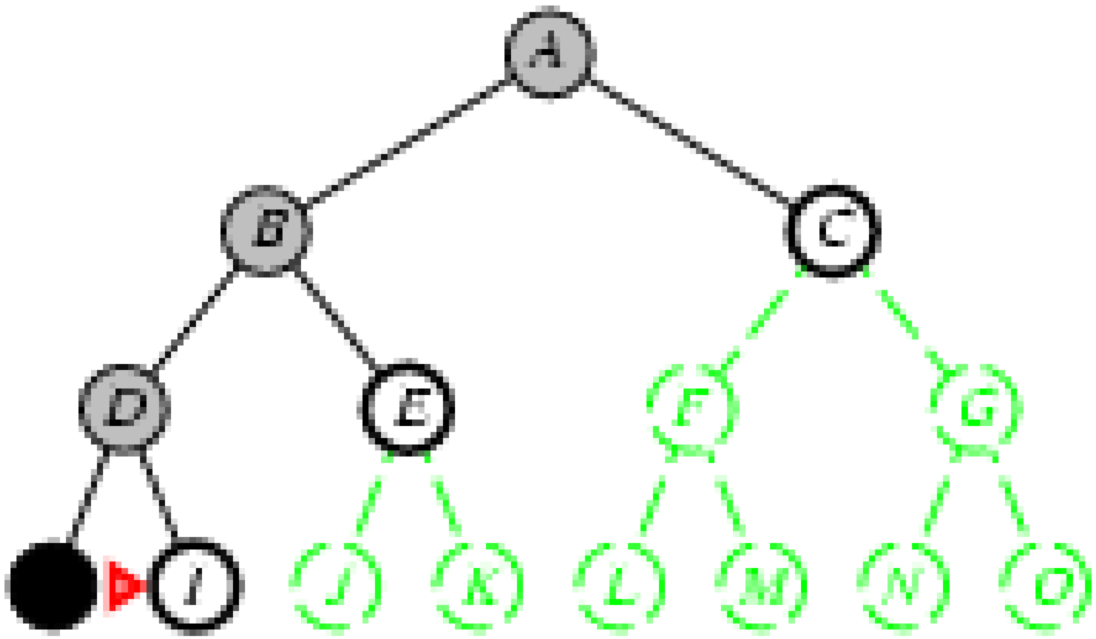
Depth-first search

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



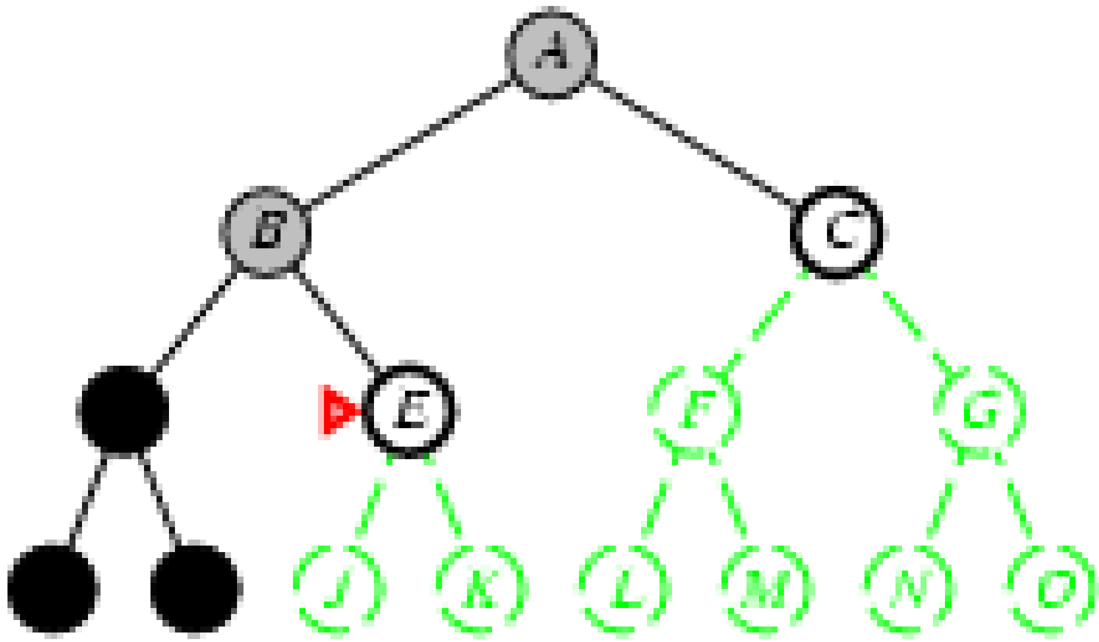
Depth-first search

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



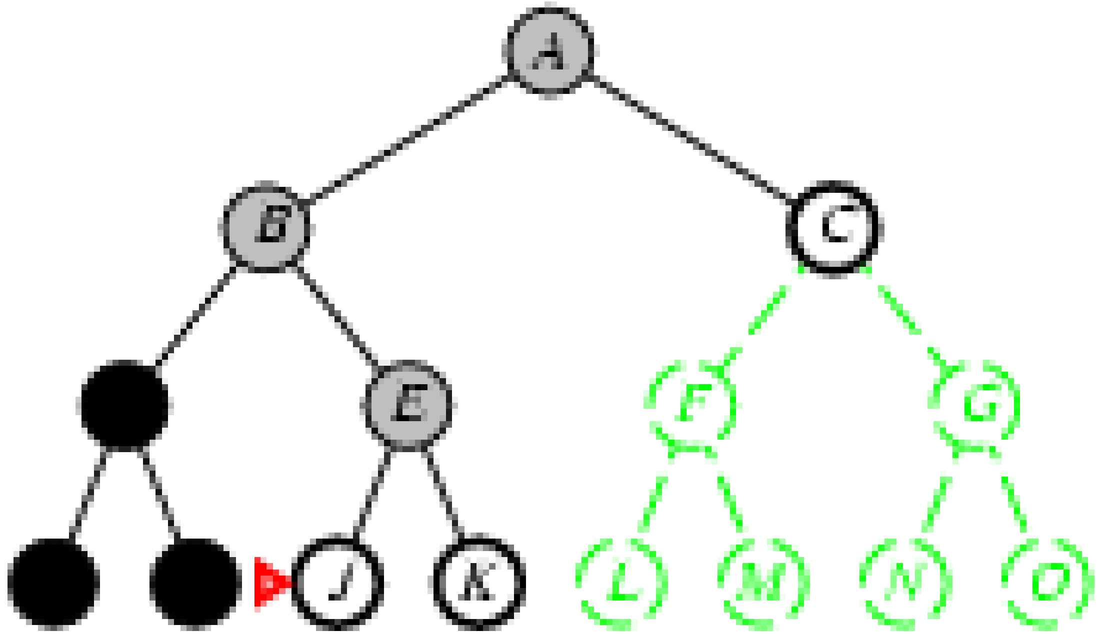
Depth-first search

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



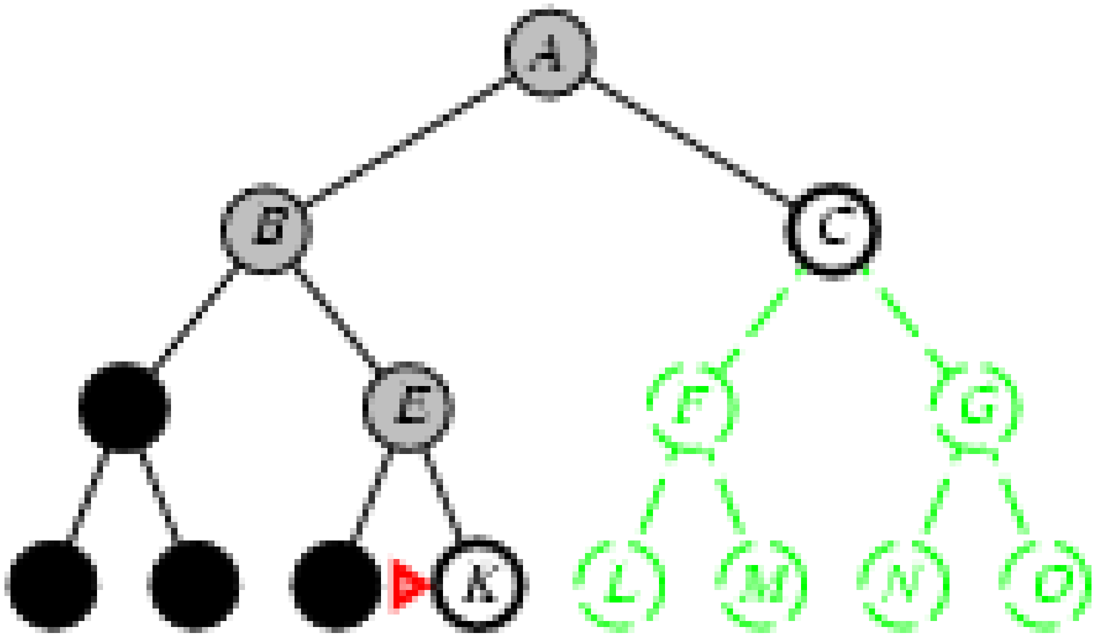
Depth-first search

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



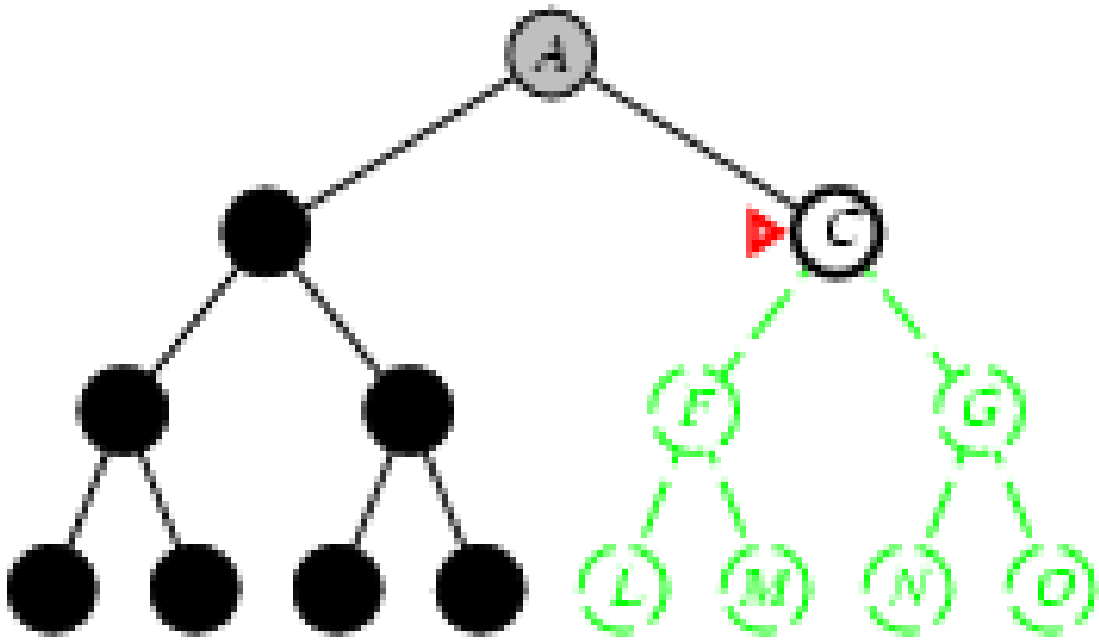
Depth-first search

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



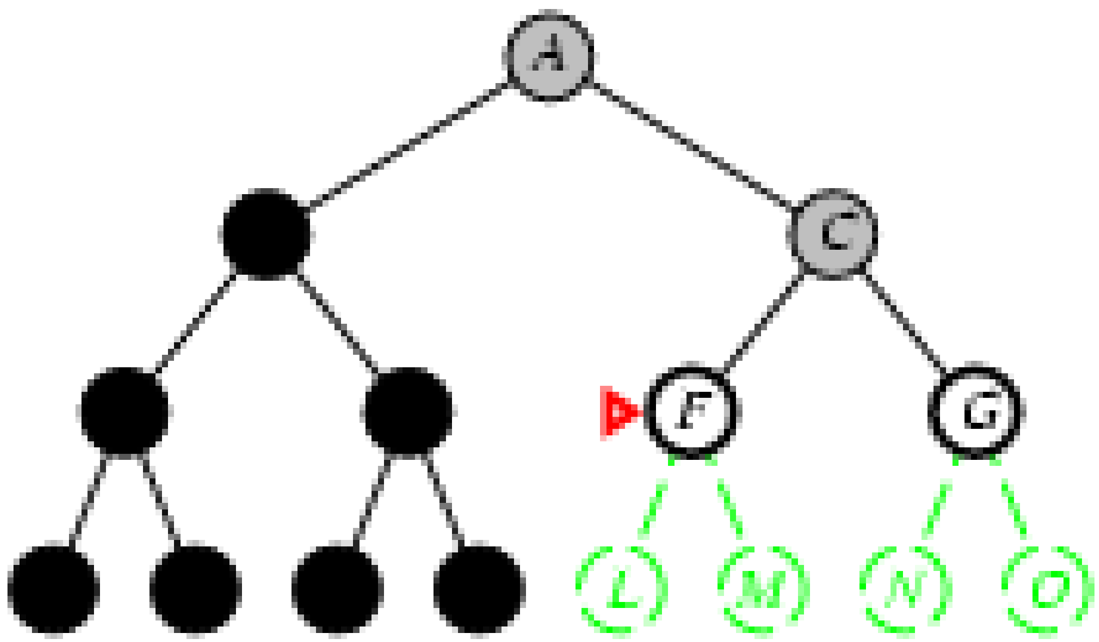
Depth-first search

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



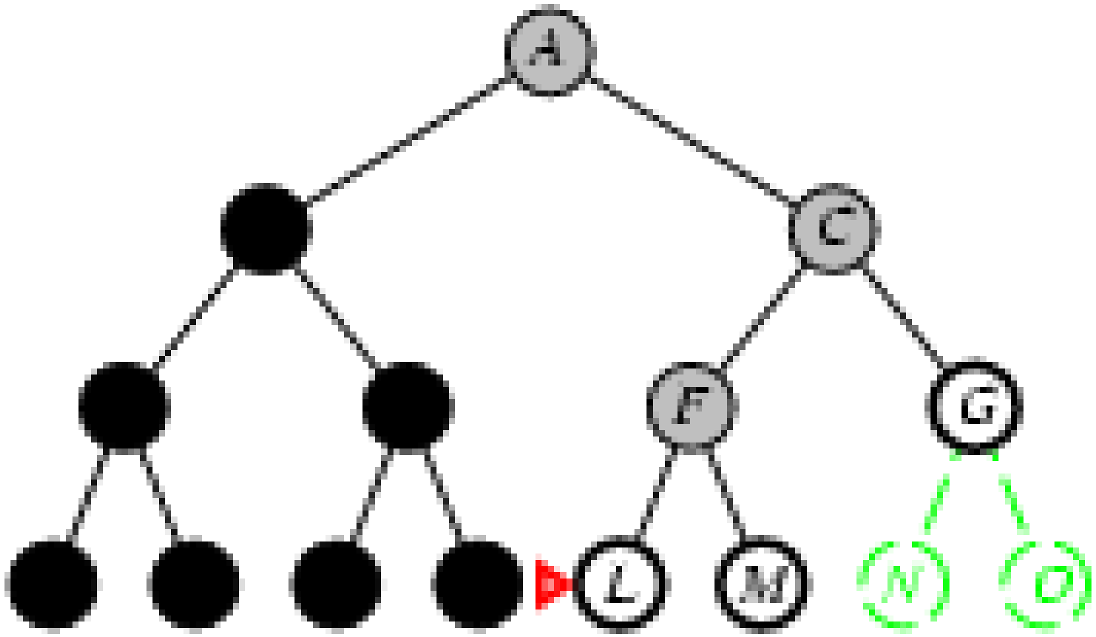
Depth-first search

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



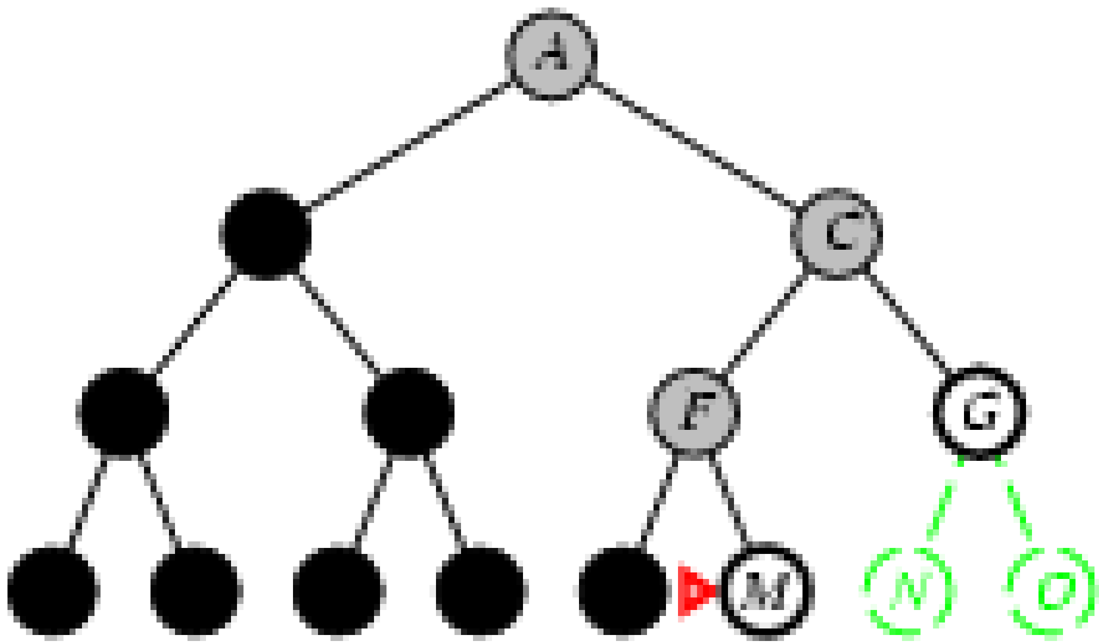
Depth-first search

- 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

  (remember: m … maximum depth of search space)
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space!
- Optimal? No

Depth-limited search

Is DF-search with depth limit l.

- i.e. nodes at depth l have no successors
- Problem knowledge can be used

Solves the infinite-path problem, but

If l < d then incompleteness results

If l > d then not optimal

Time complexity: $O(b^l)$

Space complexity: $O(bl)$

Depth-limited algorithm

function DEPTH-LIMITED-SEARCH(*problem,limit*) **return** a solution or failure/cutoff

    return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[*problem*]),*problem*,*limit*)

function RECURSIVE-DLS(*node*, *problem, limit*) **return** a solution or failure/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

What?

- A general strategy to find best depth limit *l*
  - Solution is found at depth *d*, the depth of the shallowest solution-node
- Often used in combination with DF-search

Combines benefits of DF- and BF-search

Iterative deepening search

function ITERATIVE_DEEPENING_SEARCH(*problem*)
return a solution or failure

inputs: *problem*

for *depth* $\leftarrow$ 0 to ∞ **do**

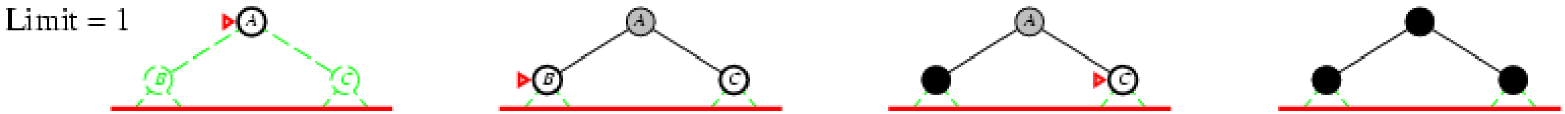
result $\leftarrow$ DEPTH-LIMITED_SEARCH(*problem*, *depth*)

if *result* $\neq$ *cuttoff* **then return** *result*

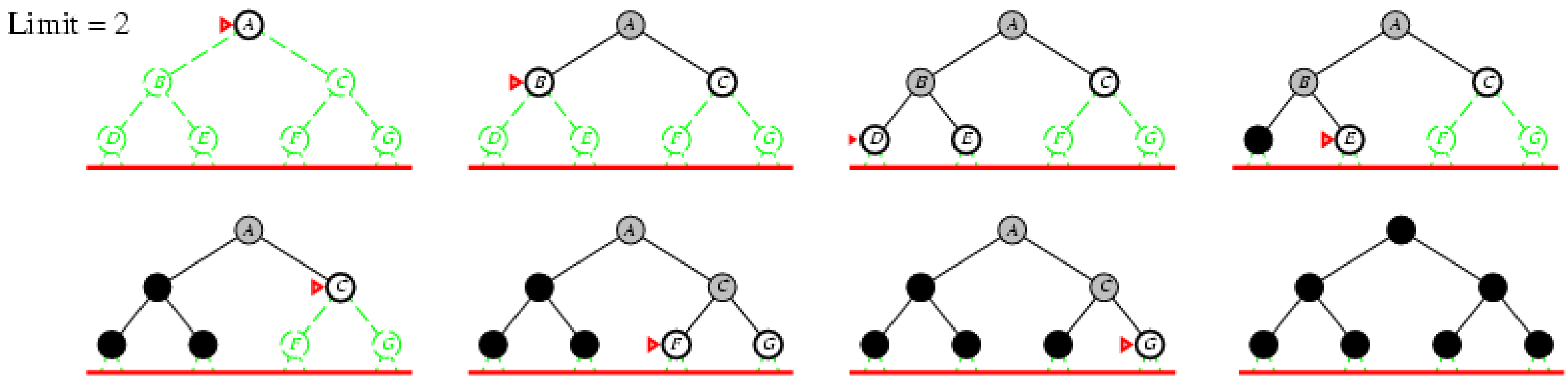
Iterative deepening search *l* =0

Limit = 0

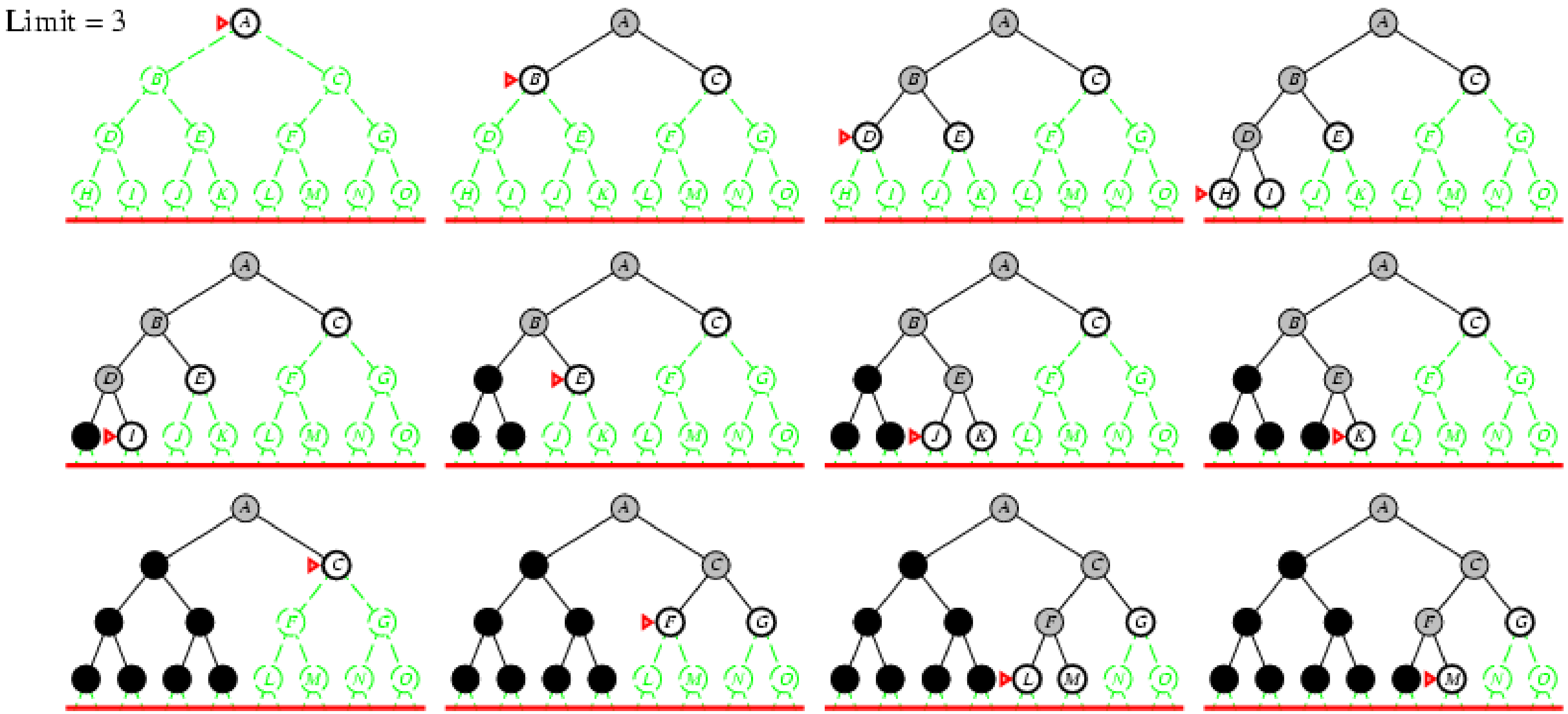
Iterative deepening search *l* =1



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*:

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

- 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%

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

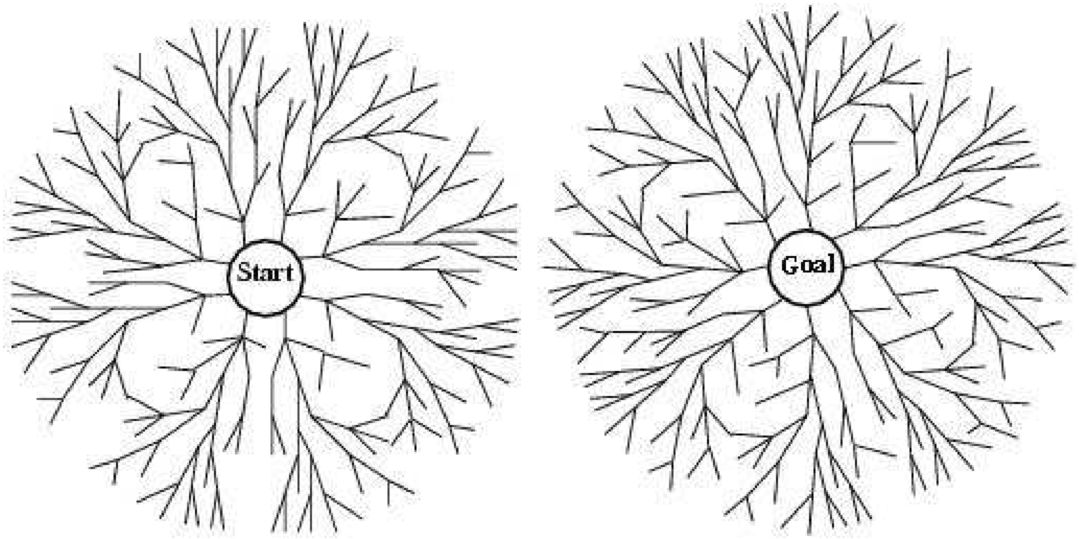
Num. comparison for b=10 and d=5 solution at far right

N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456

N_{BFS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990= 1,111,101

- IDS does better because nodes at depth d are not further expanded
- BFS can be modified to apply goal test when a node is generated

Bidirectional search



Two simultaneous searches from start an goal

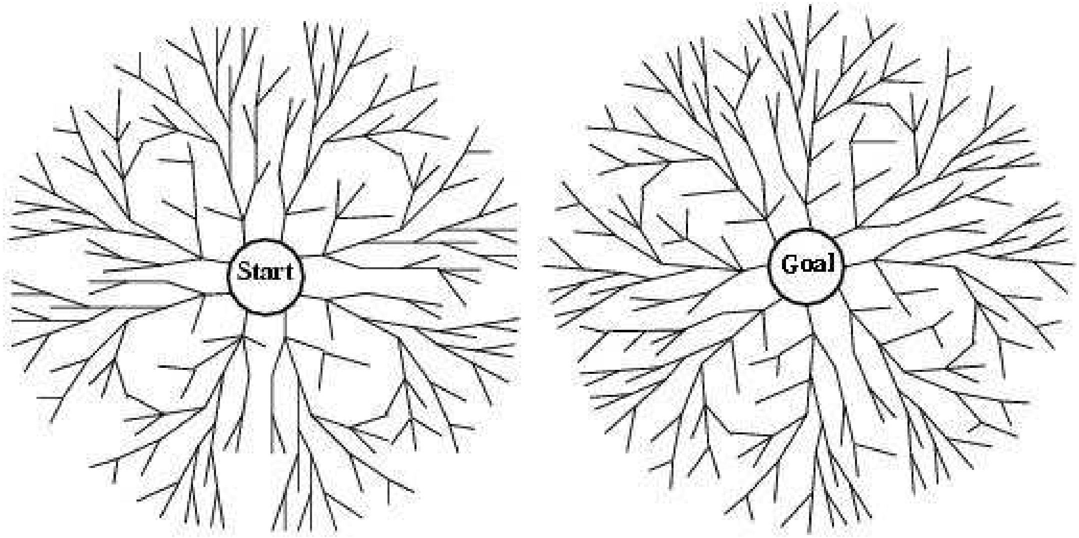
- Motivation: $$b^{d/2} + b^{d/2} \neq b^d$$

Check whether the node belongs to the other fringe before expansion

Complete and optimal if both searches are BF

Space complexity is the most significant weakness

How to search backwards?



The predecessor of each node should be efficiently computable

- When actions are easily reversible

Number of goal states does not explode

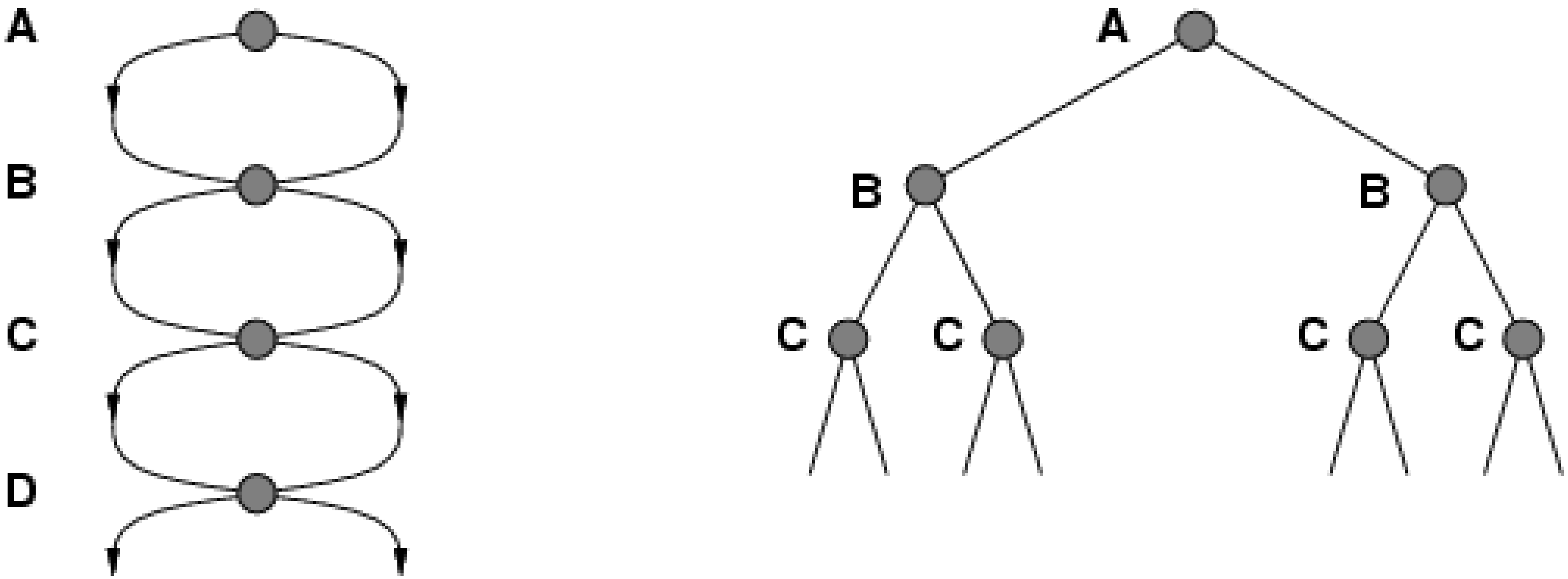
Summary of algorithms

Criterion	Breadth-First	Uniform-cost	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
Complete?	YES[a]	YES[a,b]	NO	YES, if $l \geq d$	YES[a]	YES[a,d]
Time	b^{d+1}	$b^{1+floor(C^*/e)}$	b^m	b^l	b^d	$b^{d/2}$
Space	b^{d+1}	$b^{1+floor(C^*/e)}$	bm	bl	bd	$b^{d/2}$
Optimal?	YES[c]	YES	NO	NO	YES[c]	YES[c,d]

a … if d is finite
b … if step costs >= e
c … if step costs are equal
d … if both directions use BFS

Repeated states

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



Graph search algorithm

"Closed"-list stores all expanded nodes

function GRAPH-SEARCH(*problem,fringe*) **return** a solution or failure
  closed $\leftarrow$ an empty set
  fringe $\leftarrow$ INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
  loop do
    if EMPTY?(*fringe*) **then return** failure
    node $\leftarrow$ REMOVE-FIRST(*fringe*)
    if GOAL-TEST[*problem*] applied to STATE[*node*] succeeds
      then return SOLUTION(*node*)
    if STATE[*node*] is not in *closed* **then**
      add STATE[*node*] to *closed*
      fringe $\leftarrow$ INSERT-ALL(EXPAND(*node*, *problem*), *fringe*)

Graph search, evaluation

Optimality:

- GRAPH-SEARCH discard newly discovered paths
  - This may result in a sub-optimal solution
  - YET: when uniform-cost search or BF-search with constant step cost

Time and space complexity,

- proportional to the size of the state space

  (may be much smaller than $O(b^d)$)
- DF- and ID-search with closed list no longer has linear space requirements since all nodes are stored in closed list!!

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