Introduction to Artificial Intelligence

Uninformed Search

Jianmin Li

Department of Computer Science and Technology
Tsinghua University

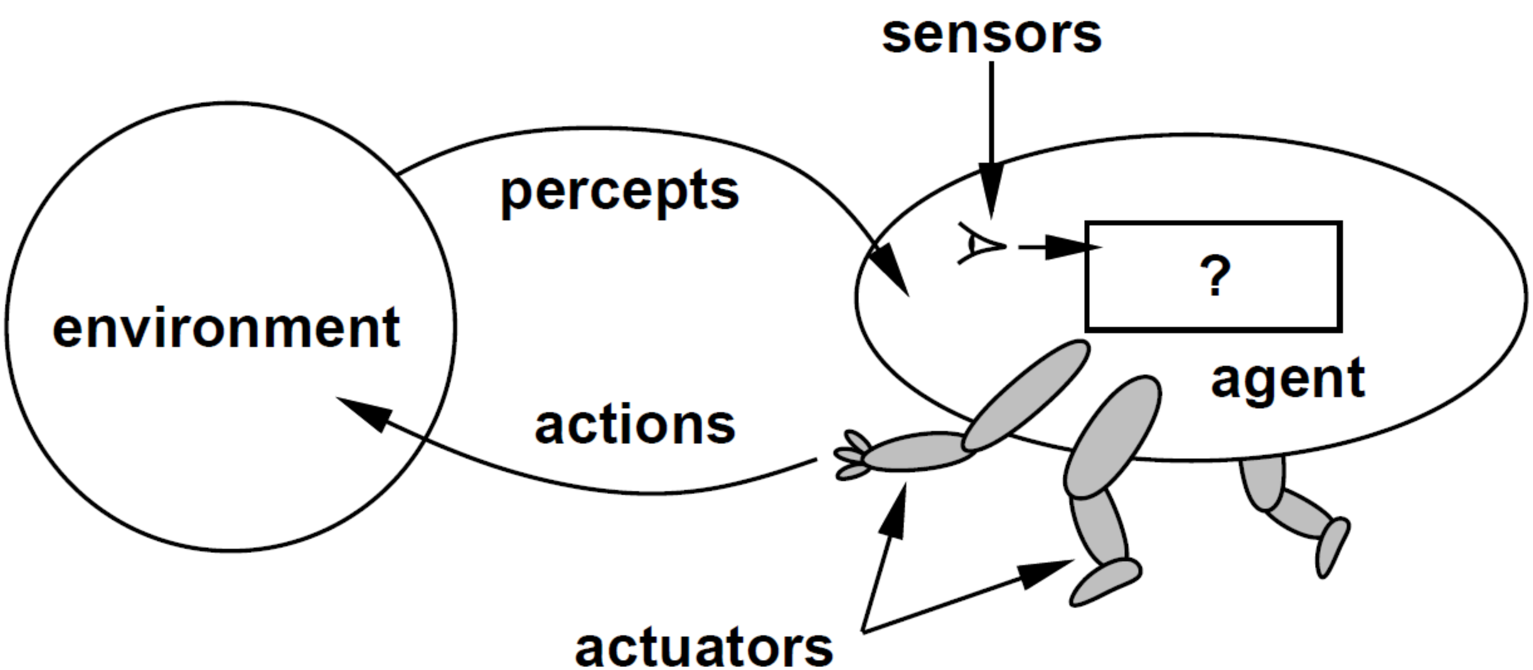
Spring, 2024

Outline

- Problem-solving agents
- Problem formulation
- Basic search algorithms

Problem-solving Agents

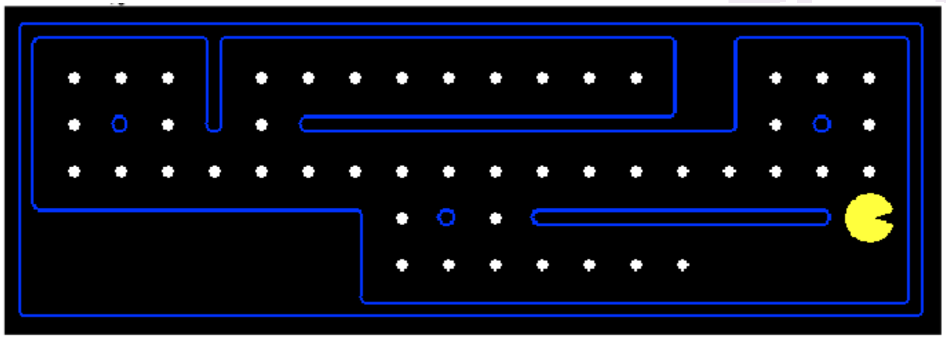
Agent

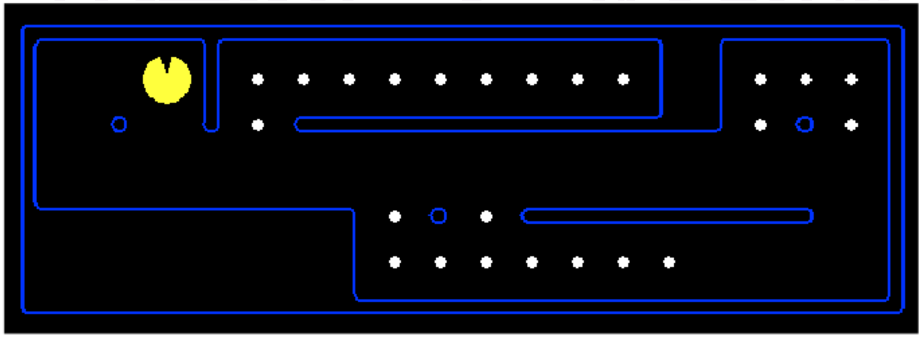


$f : P^* \rightarrow A$

Reflex Agents

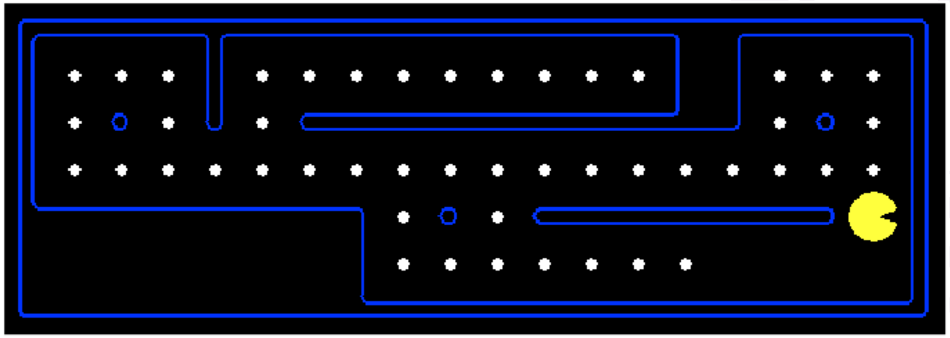
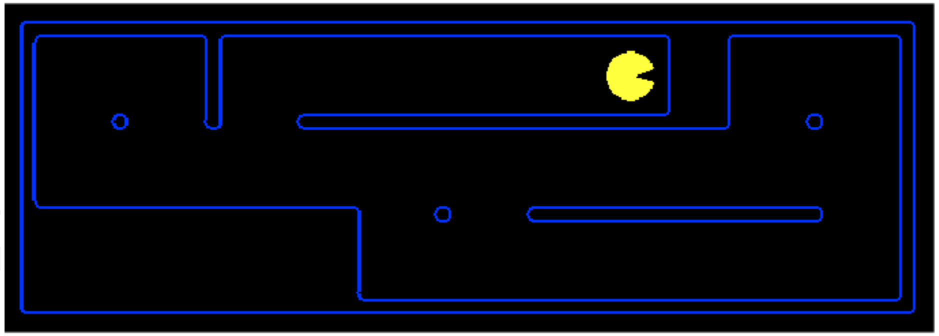
- Choose action based on current percept (and maybe memory)
- May have memory or a model of the world's current state
- Do not consider the future consequences of their actions
- Act on how the world IS





Goal-based agents

- Plan ahead, Ask “what if”
- Decisions based on (hypothesized) consequences of actions
- Must have a model of how the world evolves in response to actions
- Act on how the world WOULD BE

Problem-Solving Agents

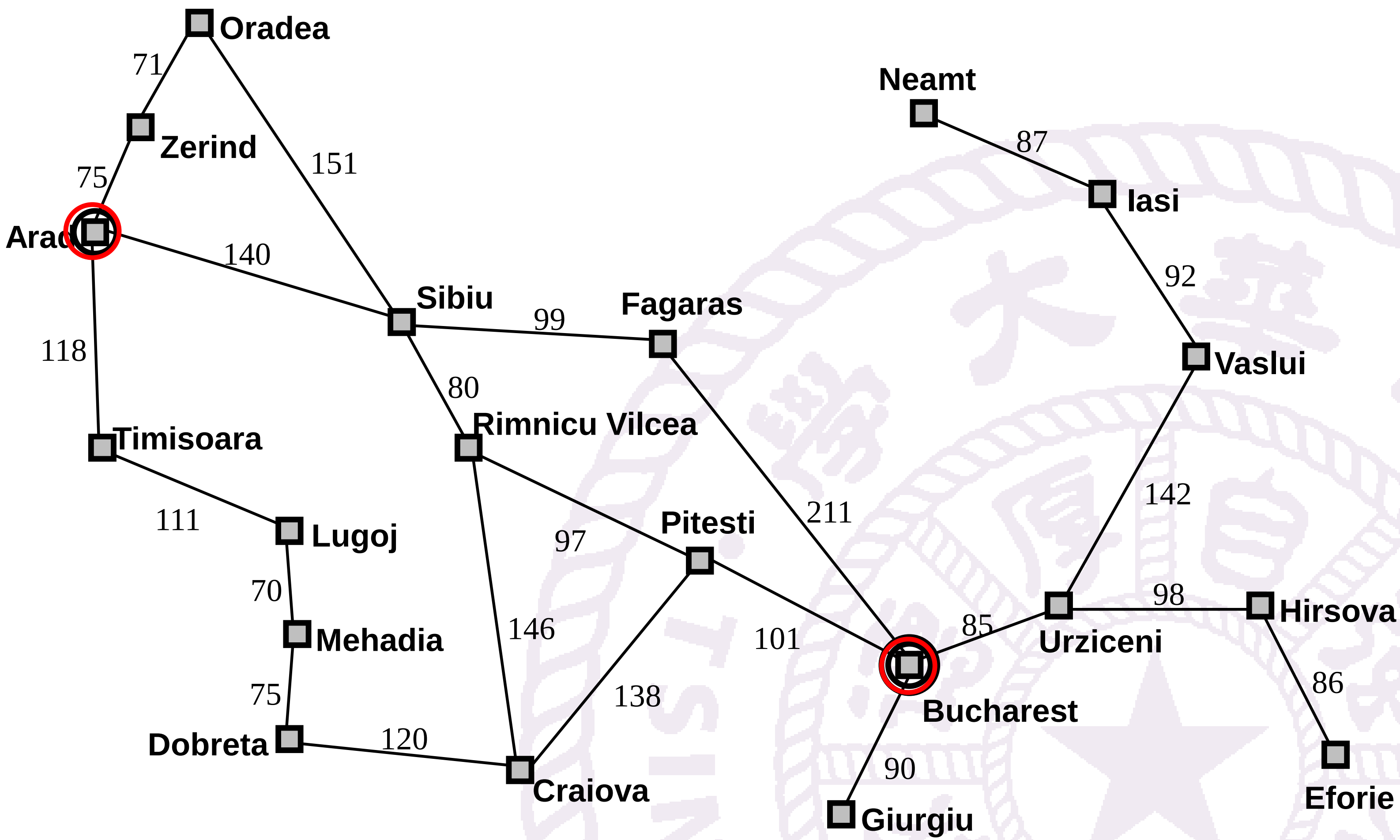
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

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

Problem Formulation

Example: Romania

On holiday in Romania



Currently in Arad; Flight leaves tomorrow from Bucharest

Five items of a problem (1)

- Initial state
  - e.g., “at Arad”
- Possible actions
  - ACTIONS(*s*), the set of actions that can be executed in *s*.
- Transition model
  - RESULT(*s*, *a*), the state that results from doing action *a* in state *s*.
  - e.g., RESULT(In(Arad), Go(Zerind)) = In(Zerind)

Five items of a problem (2)

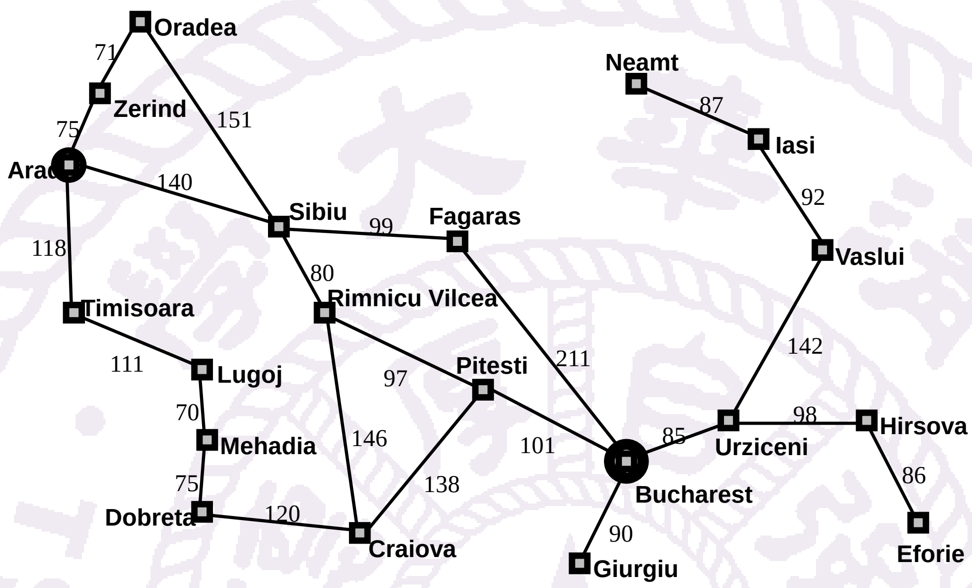
- Goal test
  - explicit, e.g., x = “at Bucharest”
  - implicit, e.g., NoDirt(x) or checkmate(x)
- Path cost (additive)
  - e.g., sum of distances, number of actions executed, etc.
  - c(x, a, y) is the step cost, assumed to be c >= 0

Problem formulation

- State space
  - initial state
  - actions
  - transition model
- A solution is a sequence of actions leading from the initial state to a goal state
- Optimal solution
  - Shortest
  - Fastest

Example: Romania

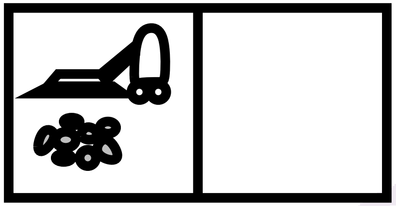
- Formulate goal
  - be in Bucharest
- Formulate problem
  - states: various cities
  - actions: drive between cities
- Find solution
  - sequence of cities
  - Arad, Sibiu, Fagaras, Bucharest



Selecting a state space

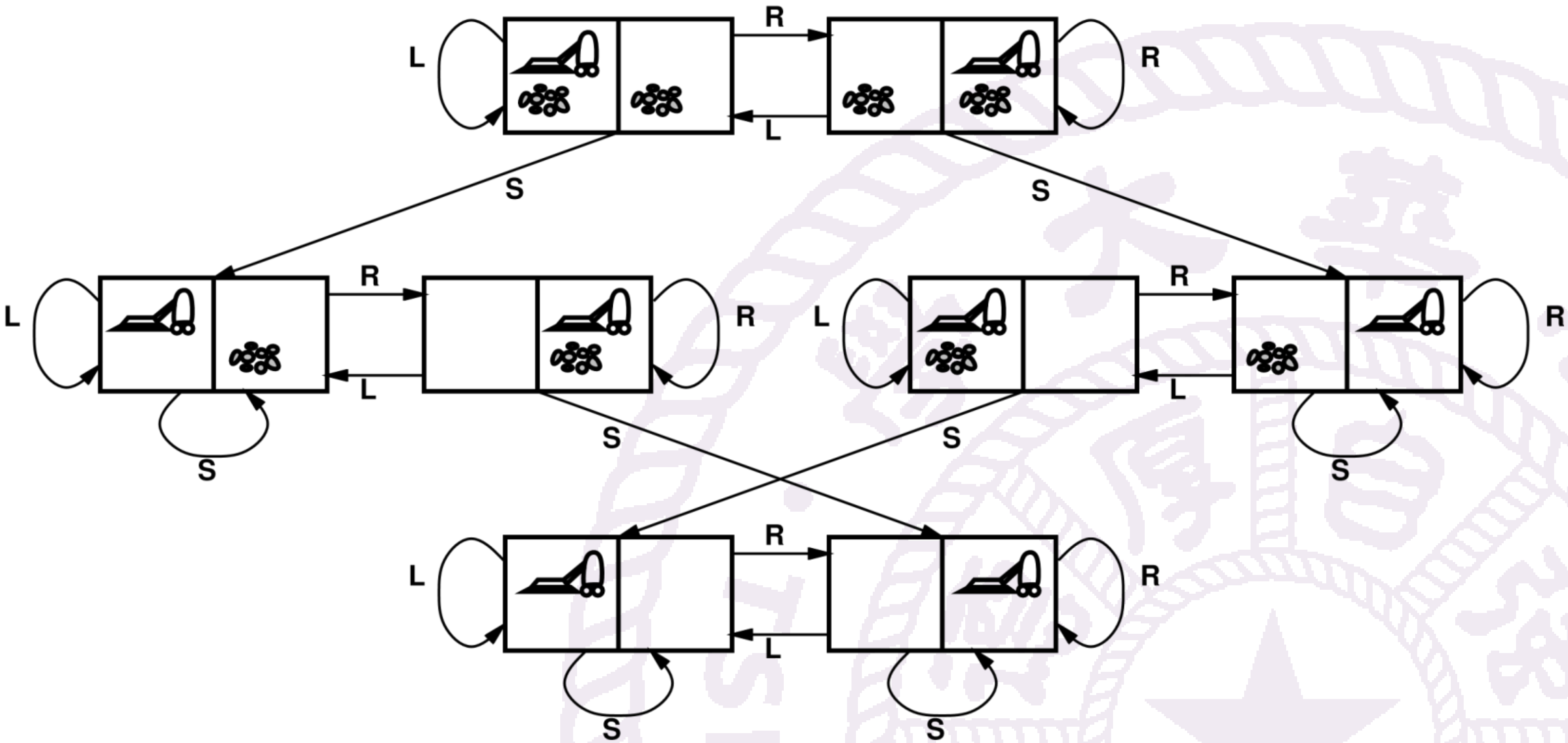
- Real world is absurdly complex
  - state space must be abstracted for problem solving
- (Abstract) state
  - set of real states
- (Abstract) action
  - complex combination of real actions
  - e.g., “Arad → Zerind” represents a complex set of possible routes, detours, rest stops, etc.
- (Abstract) solution
  - set of real paths that are solutions in the real world
- For guaranteed realizability, any real state “in Arad” must get to some real state “in Zerind”
- Each abstract action in the solution should be “easier” than the original problem!

Example: Vacuum world



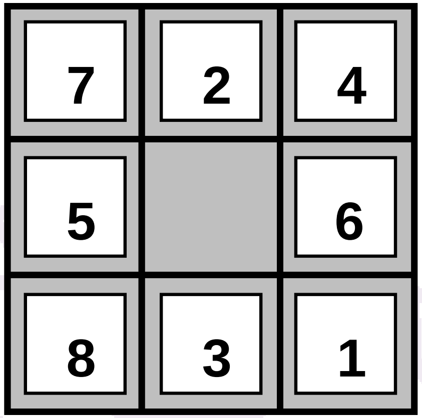
- states??
  - both dirt locations and robot location
- initial states??
  - any state
- actions??
  - Left, Right, Suck
- goal test??
  - no dirt
- path cost??
  - 1 per action
  - the number of steps in the path

Example: Vacuum world state space graph

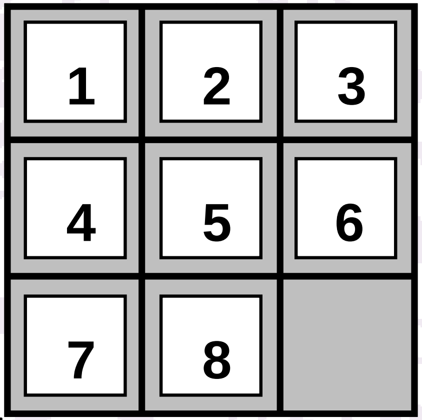


Example: The 8-puzzle

- states??
  - integer locations of tiles
- initial states??
  - any state
- actions??
  - move blank left, right, up, down
- goal test??
  - = goal state given
- path cost??
  - 1 per move
  - the number of steps in the path



Start State

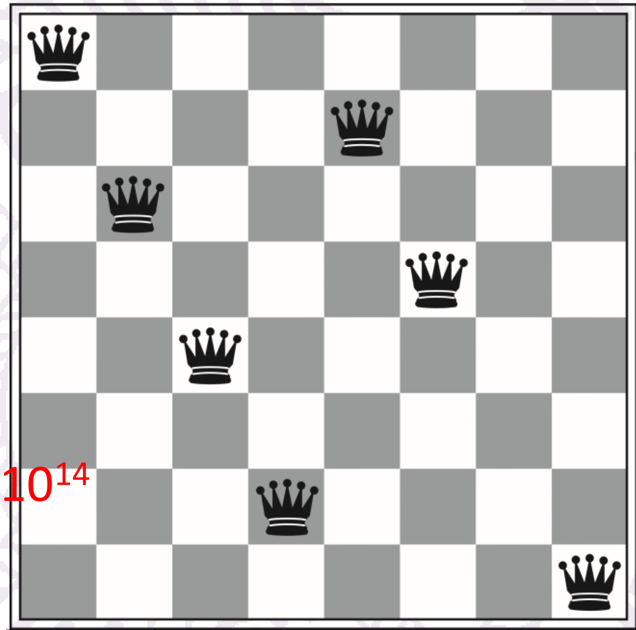


Goal State

Note: optimal solution of n-Puzzle family is NP-Complete

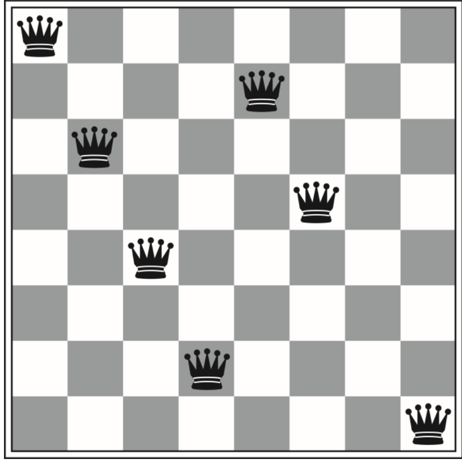
Example: 8 queens problem

- states??
  - Any arrangement of 0 to 8 queens on the board
- initial states??
  - No queens on the board
- actions??
  - Add a queen to any empty square
- goal test??
  - 8 queens are on the board, none attacked
- path cost??
  - number of actions



Note: $64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 = 3 \times 10^{14}$ possible states!

Example: 8 queens problem



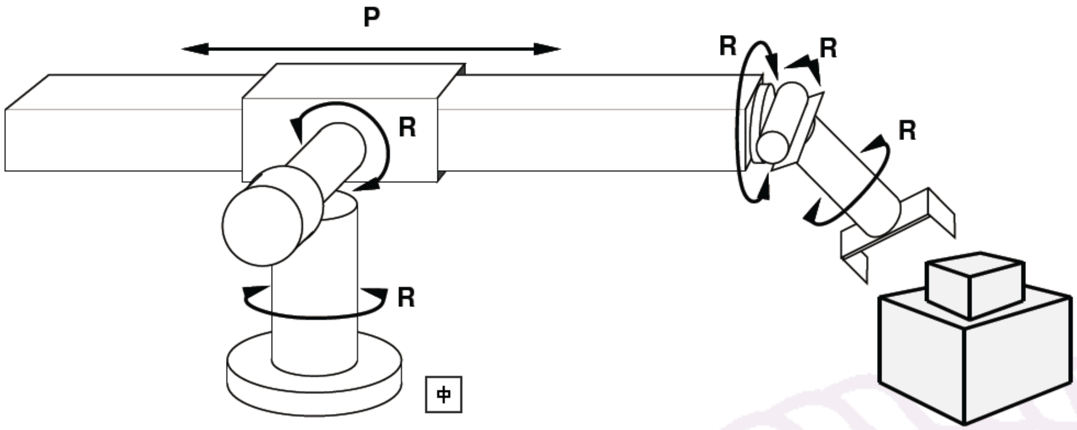
- states??
  - All possible arrangements of *n* queens (0 < *n* < 8), one per column in the leftmost *n* columns, with no queen attacking another
- initial states??
  - No queens on the board
- actions??
  - Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen
- path cost??
  - number of actions

Note: 2057 possible states to investigate!

But, for 100 queens, the reduction is from roughly 10^{400} states to about 10^{52} states

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

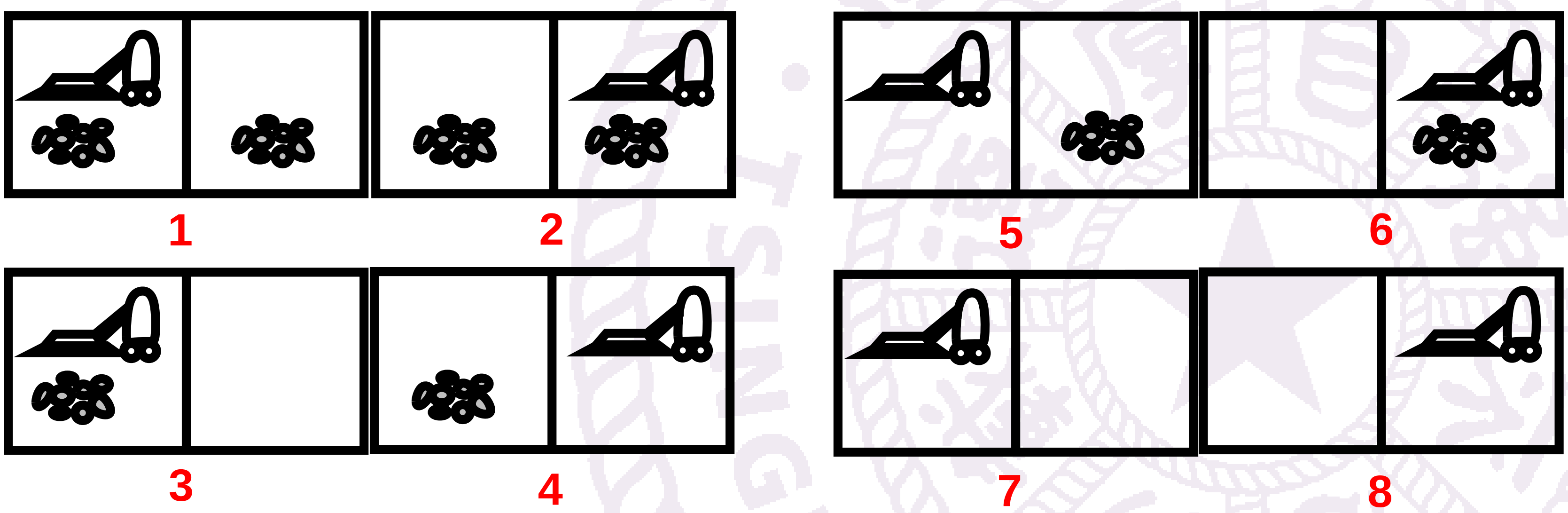


Problem types

- Deterministic, fully observable => single-state problem
  - Agent knows exactly which state it will be in
  - solution is a sequence
- Non-observable => conformant problem
  - Agent may have no idea where it is
  - solution (if any) is a sequence
- Nondeterministic and/or partially observable => contingency problem
  - percepts provide new information about current state
  - solution is a contingent plan or a policy
- Unknown state space => exploration problem ("online")

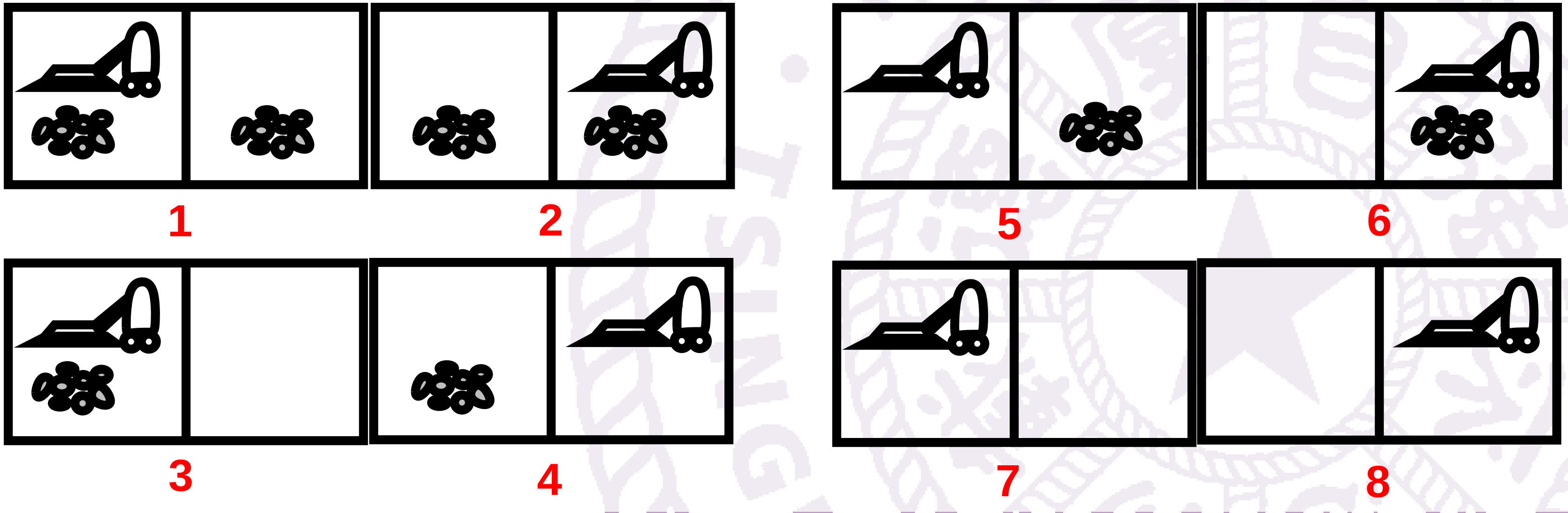
Example: Vacuum World

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



Example: Vacuum World

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

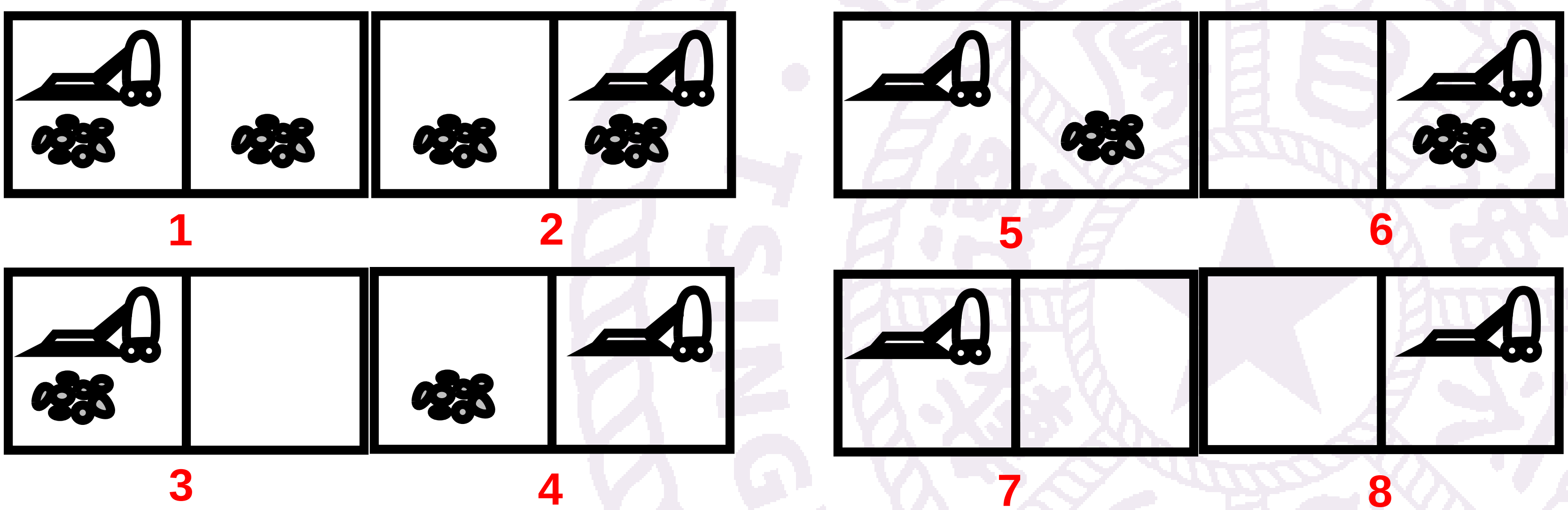


Question 1

- 下面哪个方案可以把房间打扫干净？
    A. [Suck, Left]
    B. [Right, Suck, Left, Suck]
    C. [Left, Suck]
    D. [Suck, Left, Suck, Right]

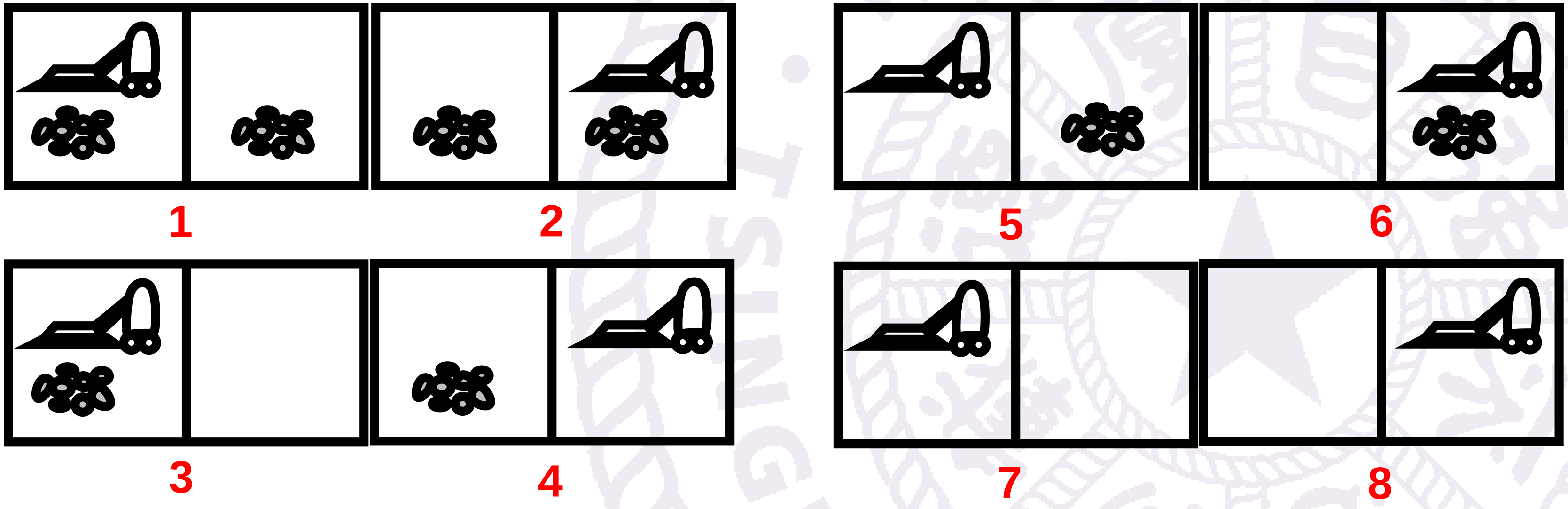
Example: Vacuum World

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



Example: Vacuum World

- Contingency, start in #5
  - Murphy's Law: Suck can dirty a clean carpet
  - Local sensing: dirt, location only
- Solution??
  - [Right, if dirt then Suck]



Basic Search Algorithms

General Search Problem

- Given:
  - Problem space (or state space)
    - A set of nodes *N* (each representing a problem state)
    - A function *Next*(*n*) defining the next states
  - Start node
  - Goal
    - a subset of *N*
- To find
  - One path from the start node to a goal node
  - All paths from the start node to any goal node
  - The best path from the start to the best goal

Tree search algorithms

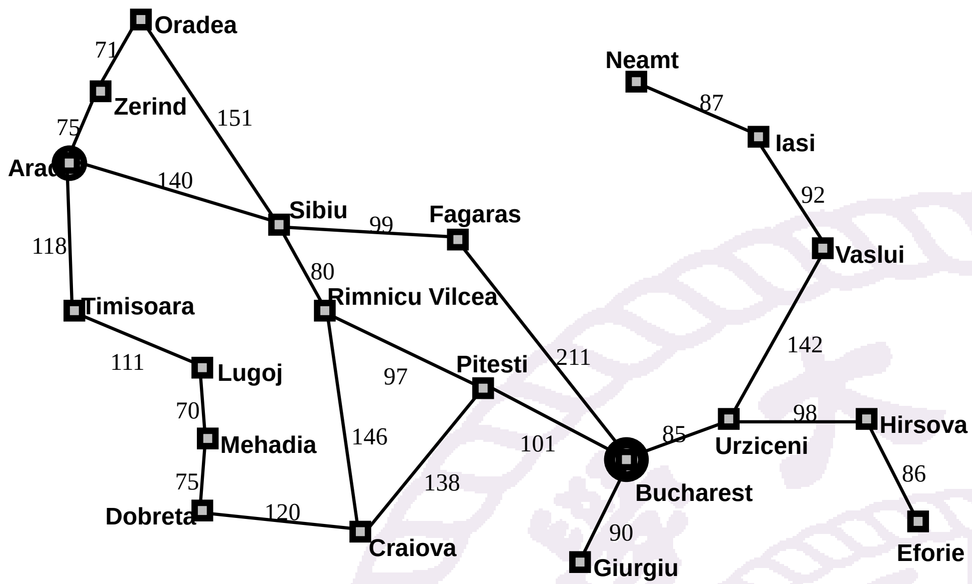
function TREE-SEARCH(*problem, strategy*) **returns** a solution, or failure
  initialize the search tree using the initial state of *problem*
  loop do
    if th
    choo
    if th … solution
    else expand the node and add the resulting nodes to the search tree
  end

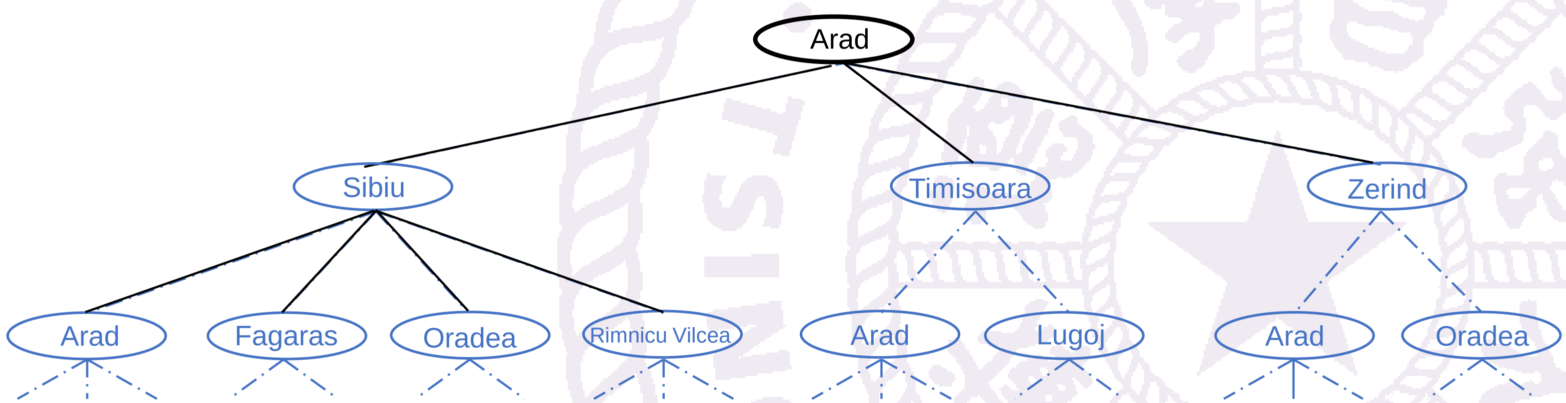
which fringe nodes to explore?

Exploration strategy

- Basic idea
  - The fringe (frontier) of the tree, the nodes yet to be explored
  - Expanding states, generating successors of already-explored states

Tree search example



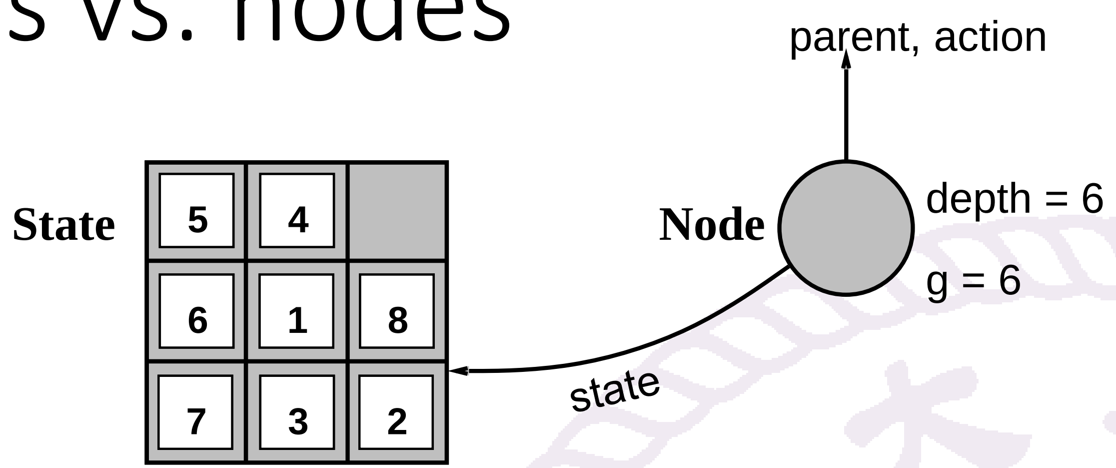


Implementation: general tree search

```
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 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



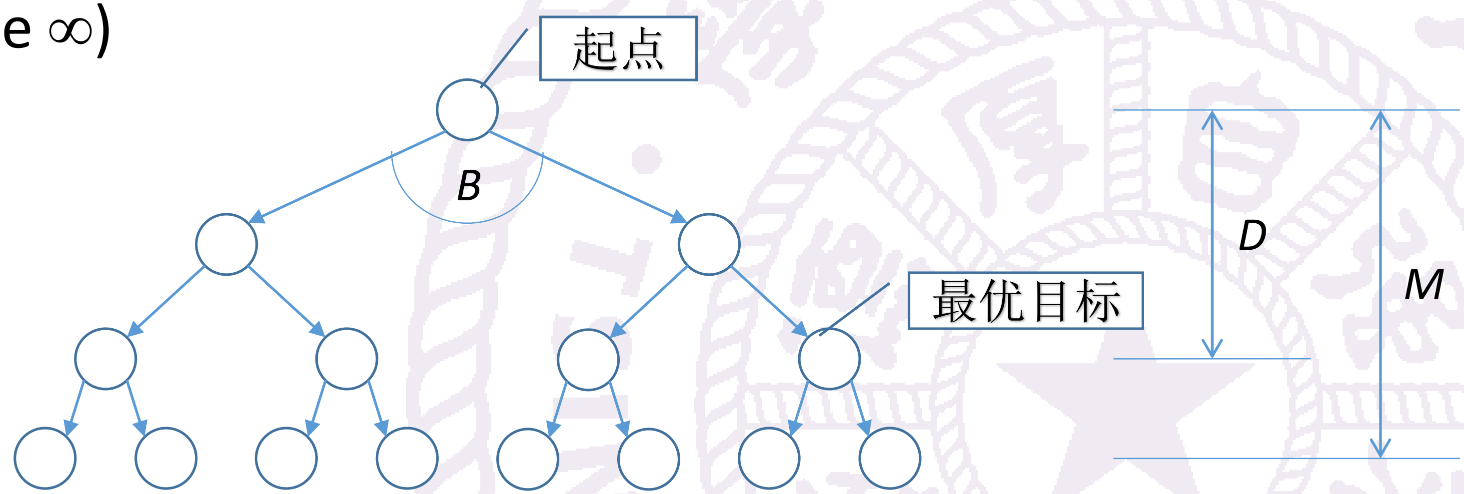
- A node is a data structure constituting part of a search tree
  - includes parent, children, depth, path cost g(x)
- A state is a representation of a physical configuration
  - States do not have parents, children, depth, or path cost!
- Two different nodes can contain the same world state if that state is generated via two different search paths

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 - How long does it take to find a solution?
    - e.g. number of nodes generated/expanded
  - Space complexity - How much memory is needed to perform the search?
    - e.g. maximum number of nodes in memory
  - Optimality - Does it always find a least-cost solution?

Search strategies

- 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 path in the state space (may be ∞)

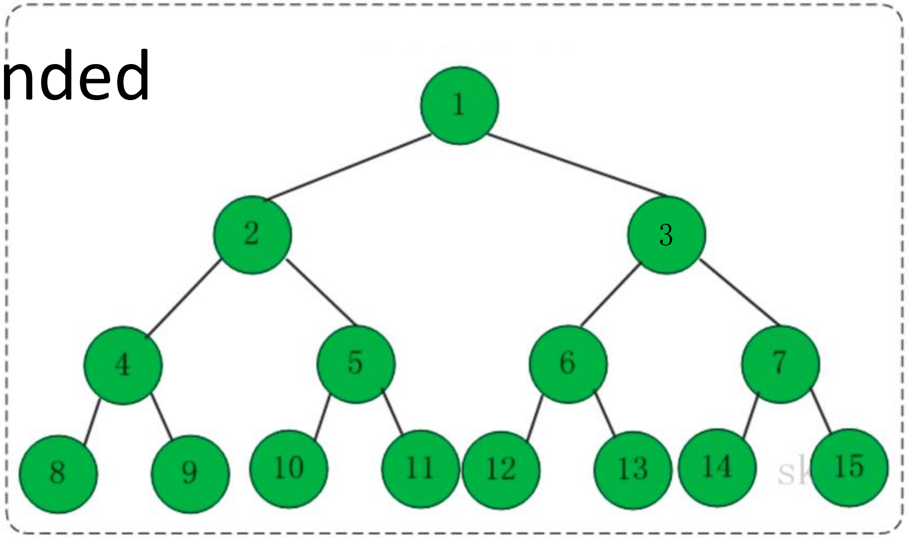


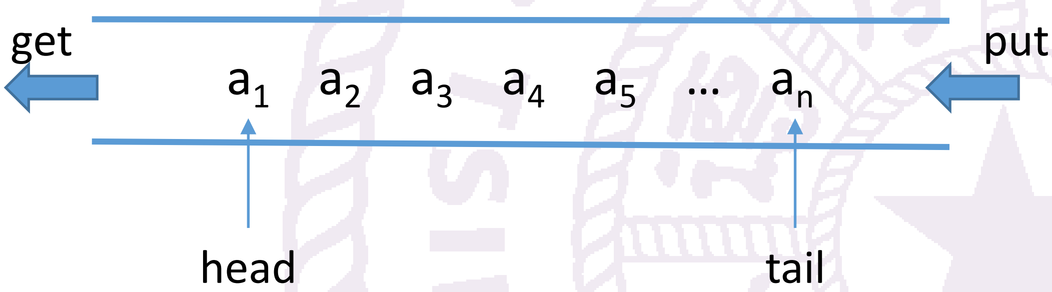
Uninformed search strategies

- 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

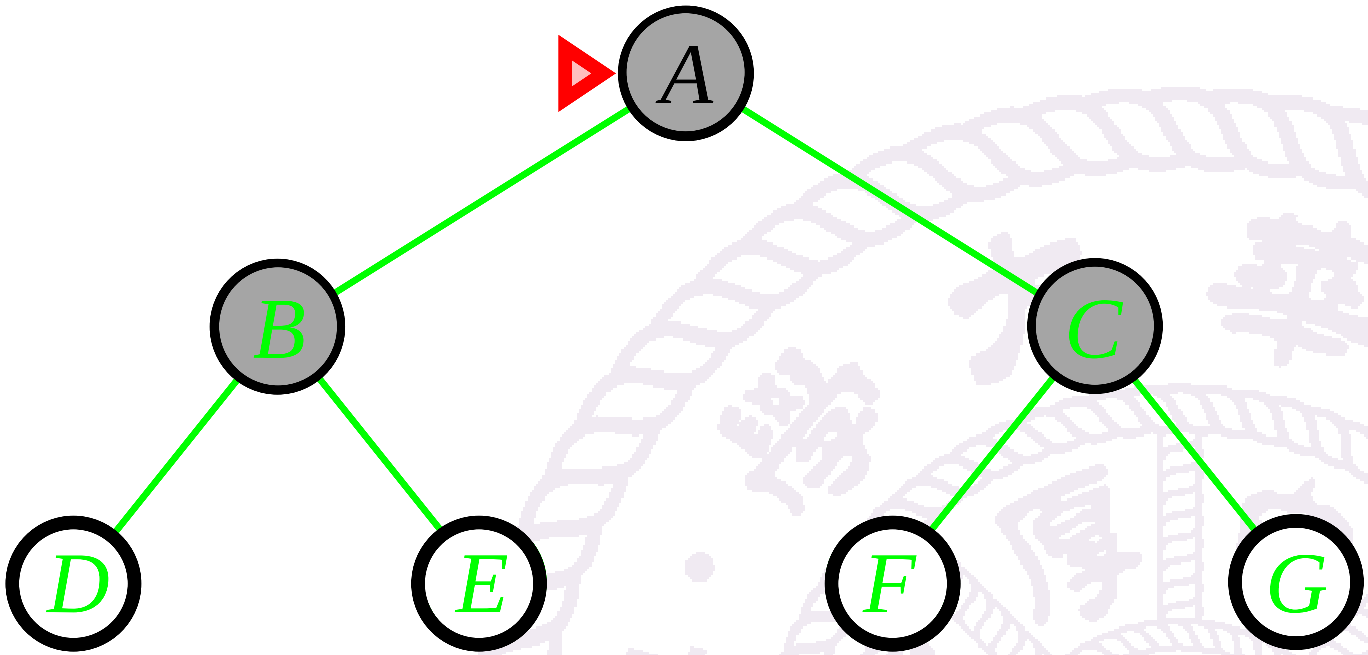
Breadth-first search

- Expand shallowest unexpanded node
- Implementation
  - fringe is a FIFO queue



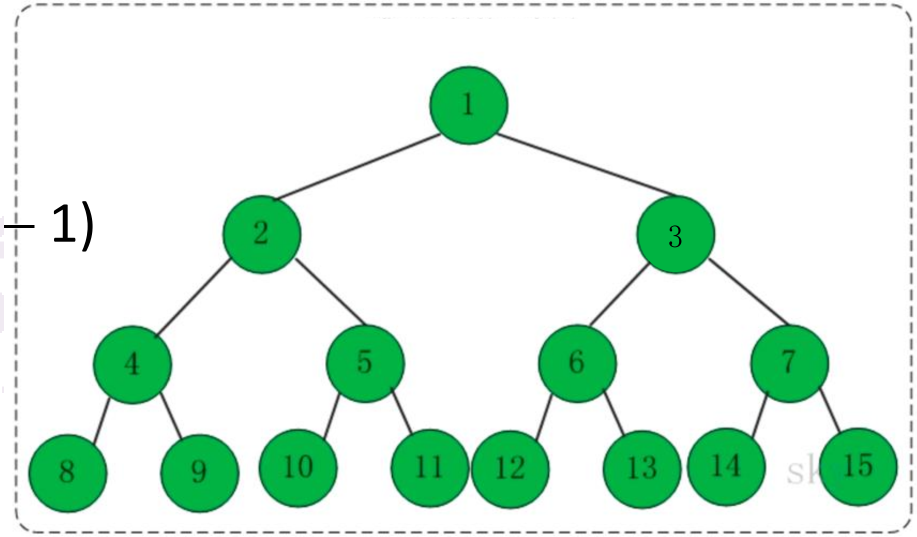


Breadth-first search



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??
  - $O(b^{d+1})$
- Optimal??
  - Yes (if cost = 1 per step)
  - Yes (if the path cost is a nondecreasing function of depth of the node)
  - not optimal in general



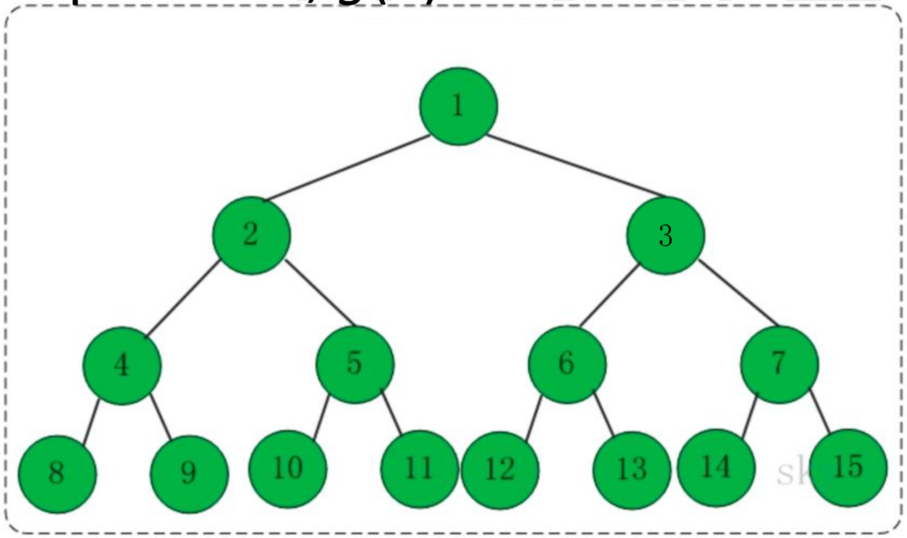
Space is the big problem

- b=10, 10,000 nodes/sec, 1KB/node

Depth	Nodes	Time	Memory (Byte)
2	1101	.11 seconds	1 M
4	111101	11 seconds	106 M
6	10^7	19 minutes	10 G
8	10^9	31 hours	1 T
10	10^{11}	129 days	101 T
12	10^{13}	35 years	10 P
14	10^{15}	3523 years	1 E

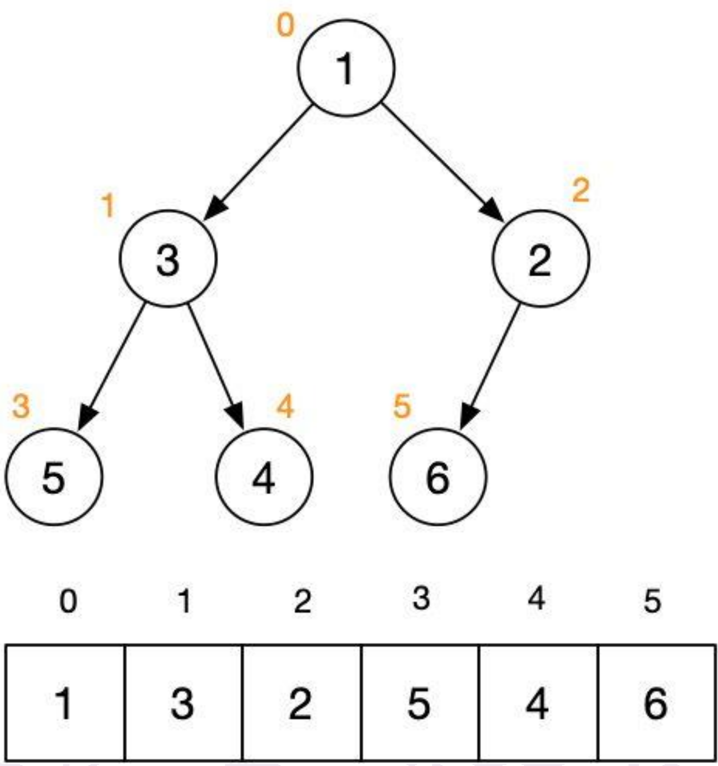
Uniform-cost search

- Breadth-first search
  - finds the shallowest goal state
  - is not guaranteed to find the best solution
- Uniform cost search
  - expanding the lowest cost node on the fringe
  - cost is the path cost, $g(n)$.



Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - fringe = queue ordered by path cost
  - priority queue, lowest first
- Equivalent to breadth-first if step costs all equal

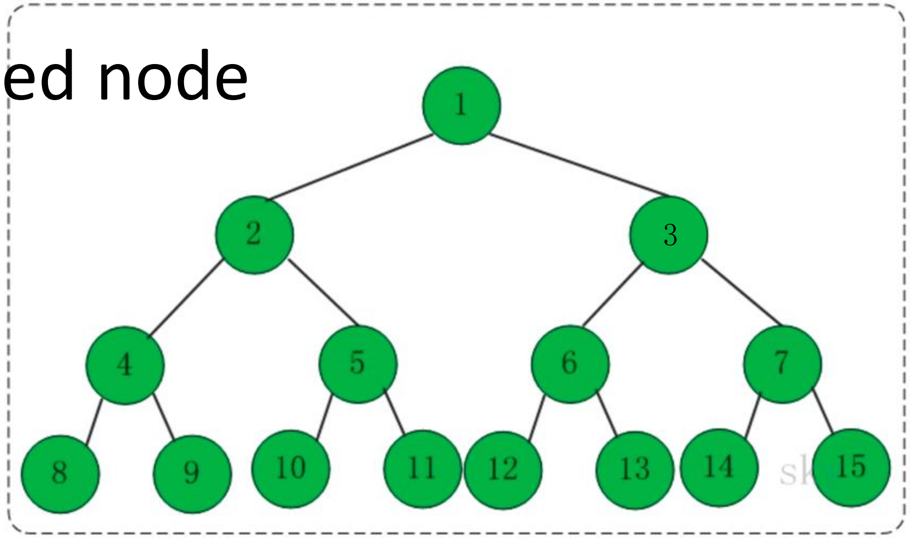


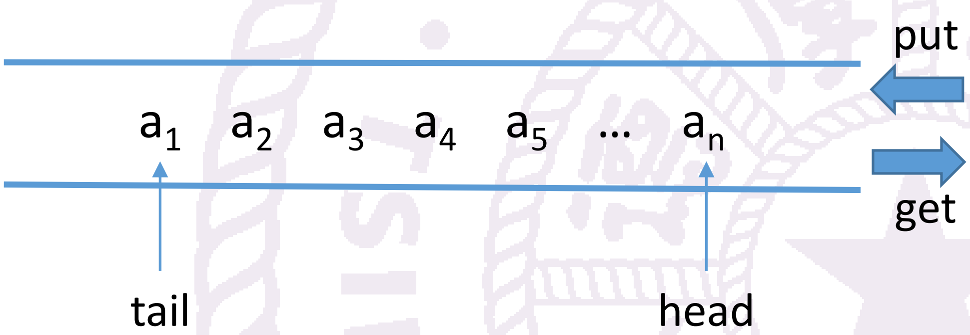
Properties of uniform-cost search

- Complete??
  - Yes, if step *cost* >= ε
- Time??
  - # of nodes with *g* <= cost of optimal solution, $O(b^{\lceil C^*/\varepsilon \rceil})$
  - where C^* is the cost of the optimal solution
- Space??
  - # of nodes with *g* <= cost of optimal solution, $O(b^{\lceil C^*/\varepsilon \rceil})$
- Optimal??
  - Yes - nodes expanded in increasing order of $g(n)$

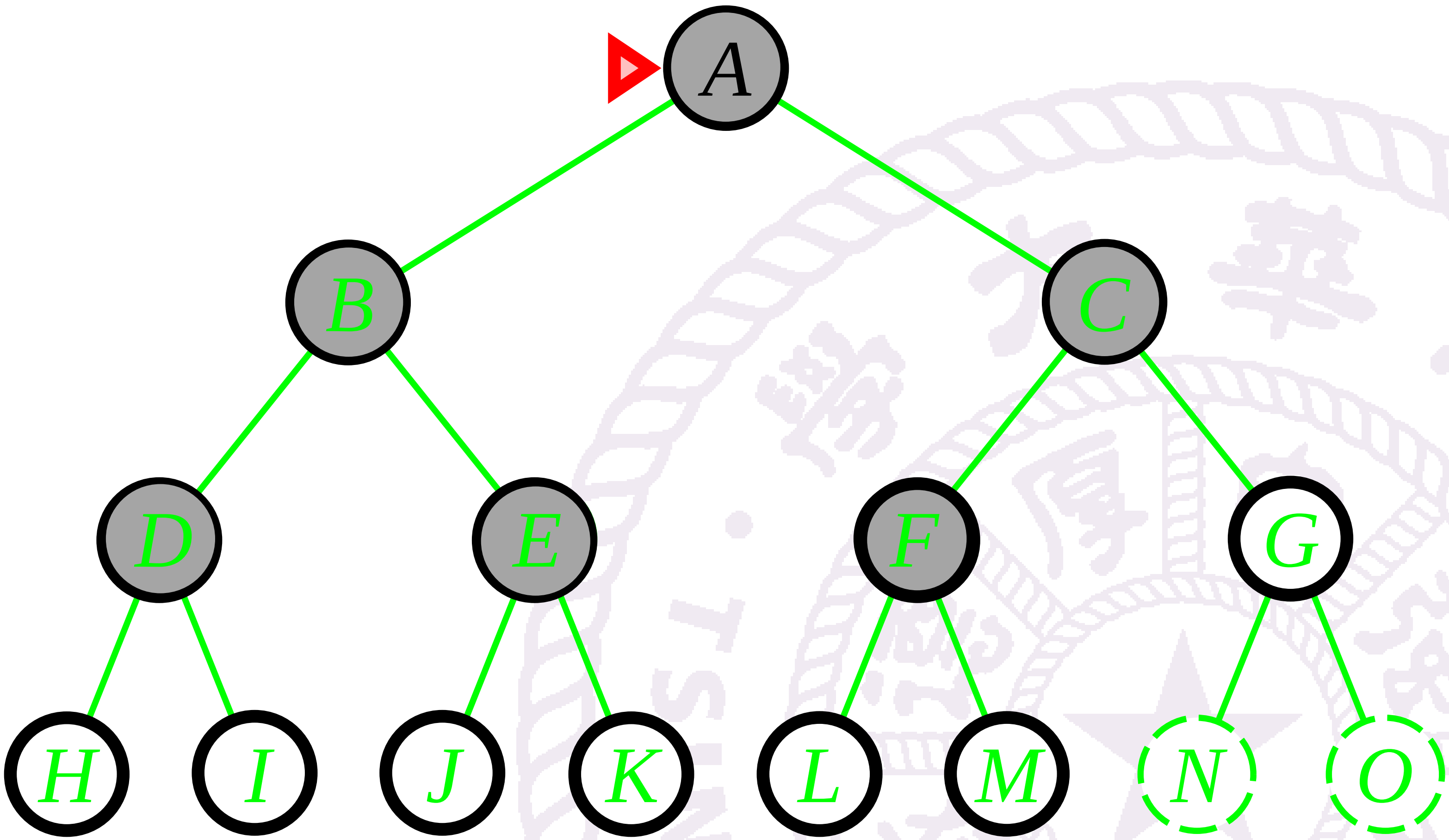
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - fringe = LIFO queue





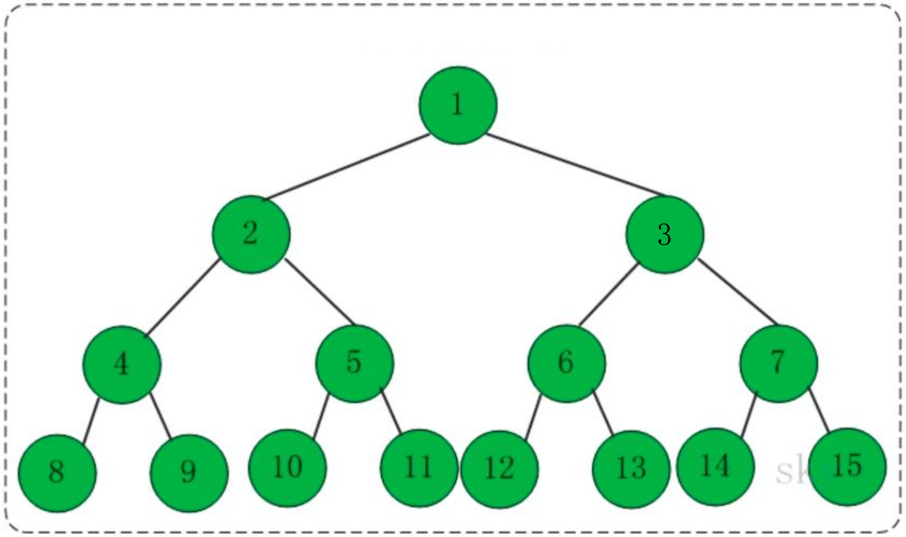
Depth-first search



fringe:

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 a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
     cutoff_occurred? ← false
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        result ← RECURSIVE-DLS(child, problem, limit − 1)
        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 failure
    for $depth = 0$ **to** ∞ **do**
        $result \leftarrow$ DEPTH-LIMITED-SEARCH($problem$, $depth$)
        if $result \neq$ cutoff **then return** $result$

Properties of iterative deepening search

- Complete??
  - Yes
- Time??
  - $(d + 1)\, b^0 + db + (d - 1)\, b^2 + (d - 2)\, b^3 + \ldots + b^d = O(b^d)$
- Space??
  - $O\,(bd)$
- Optimal??
  - Yes, if step cost = 1
  - is optimal if the path cost is a nondecreasing function of depth of the node

Properties of iterative deepening search

- Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:
  - $N(\text{IDS}) = 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}450$
  - $N(\text{BFS}) = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 + 999{,}990 = 1{,}111{,}100$
- IDS does better because other nodes at depth d are not expanded
- BFS can be modified to apply goal test when a node is generated

Summary of algorithms (1)

Tree Search Algorithms

```
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
    end
```

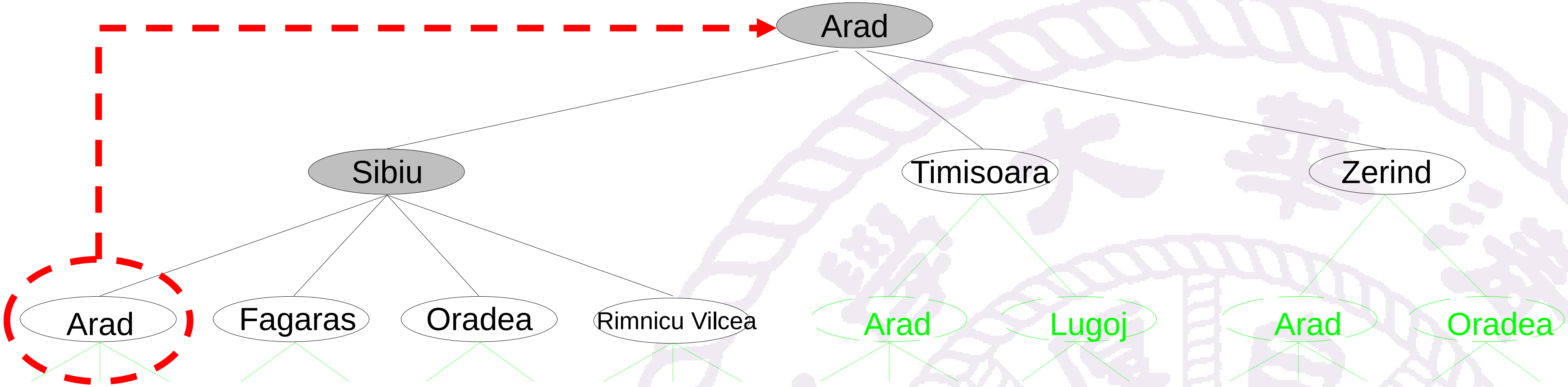
Strategy is defined by picking the
order of node expansion

Summary of algorithms (2)

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete	Yes	Yes	No	Yes, if $l >= d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\varepsilon \rceil}$	b^m	b^l	B^d
Space	b^{d+1}	$b^{\lceil C^*/\varepsilon \rceil}$	bm	bl	bd
Optimal	Yes	Yes	No	No	Yes

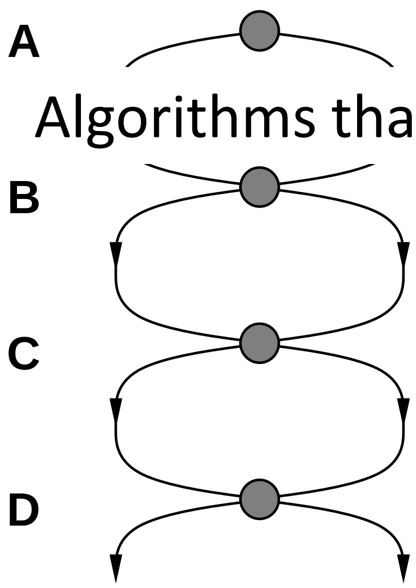
Repeated states (1)

- Infinite search tree

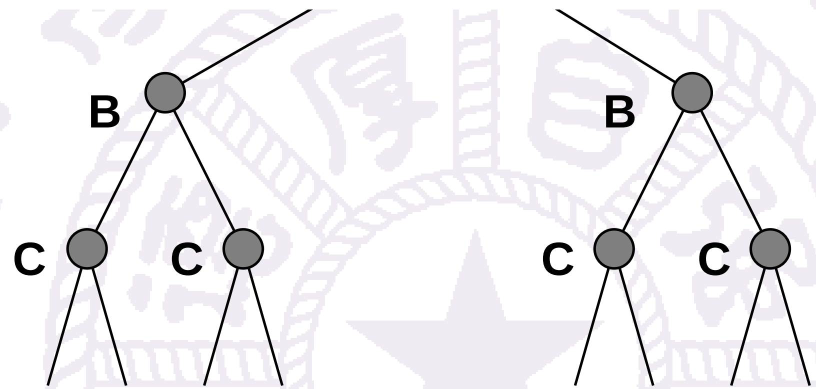


Repeated states (2)

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



Algorithms that forget their history are doomed to repeat it!



Graph search

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

---

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

explored set
(closed table)

BFS on Graph

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

UCS on Graph

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

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, BFS, DFS, …
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms
- Graph search can be more efficient than tree search

谢谢！