Chapter 03
Problem Solving

Instructor
LE Thanh Sach, Ph.D.

Instructor's Information

LE Thanh Sach, Ph.D.

Office:

Department of Computer Science,
Faculty of Computer Science and Engineering,
HoChiMinh City University of Technology.

Office Address:

268 LyThuongKiet Str., Dist. 10, HoChiMinh City, Vietnam.

E-mail: LTSACH@hcmut.edu.vn

E-home: http://cse.hcmut.edu.vn/~ltsach/

Tel: (+84) 83-864-7256 (Ext: 5839)

Acknowledgment

The slides in this PPT file are composed using the materials supplied by

- **Prof. Stuart Russell and Peter Norvig**: They are currently from University of California, Berkeley. They are also the author of the book "Artificial Intelligence: A Modern Approach", which is used as the textbook for the course

- **Prof. Tom Lenaerts**, from Université Libre de Bruxelles

Problem solving

Objective

- Study principles to solve any problems that can be formalized as state spaces

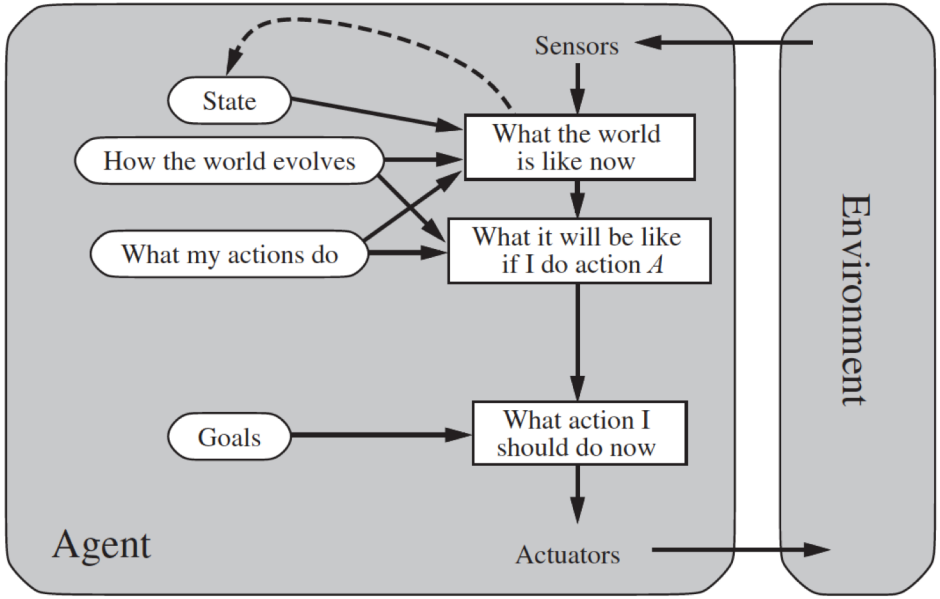
Outline

- Problem-solving agents
  - A kind of goal-based agent
- Problem formulation
  - Example problems
- Problem types
  - Single state (fully observable)
  - Search with partial information
- Basic search algorithms
  - Uninformed

Problem-solving agent

- What?
- How?
- Example

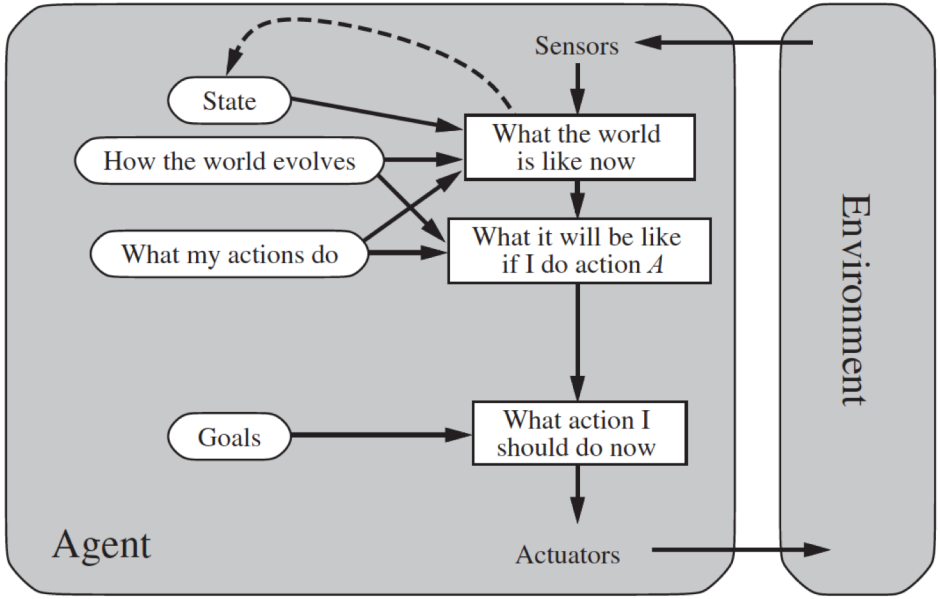
Problem-solving agent
What?



Goal-based agent
(Problem-solving agent)

- AI: is to create "intelligent agents"
- Agent: interact with the environment
  - Sensor: acquires inputs (percepts)
  - Actuator: performs actions, change the environment
  - agent program: (software) selects actions

Problem-solving agent
What?



Goal-based agent
(Problem-solving agent)

- Goal-based agent:
  - Select actions using
    - ✓ goal,
    - ✓ current state,
    - ✓ history of actions,
    - ✓ the information of the world (the problem's world)
  - It also updates the current state using percepts
- Problem-solving agent is a goal-based agent

Problem-solving agent
How?

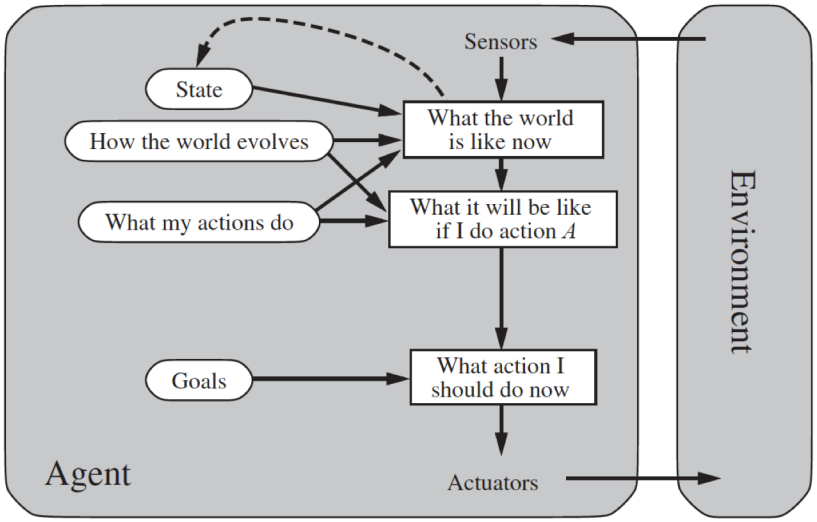
- Four general steps in problem solving:
  - Goal formulation
    - ✓ What are the successful world states
  - Problem formulation
    - ✓ What actions and states to consider to give the goal
  - Search
    - ✓ Determine the possible sequence of actions that lead to the states of known values and then choosing the best sequence.
  - Execute
    - ✓ Give the solution perform the actions.

Problem-solving agent
How?

function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **return** an action

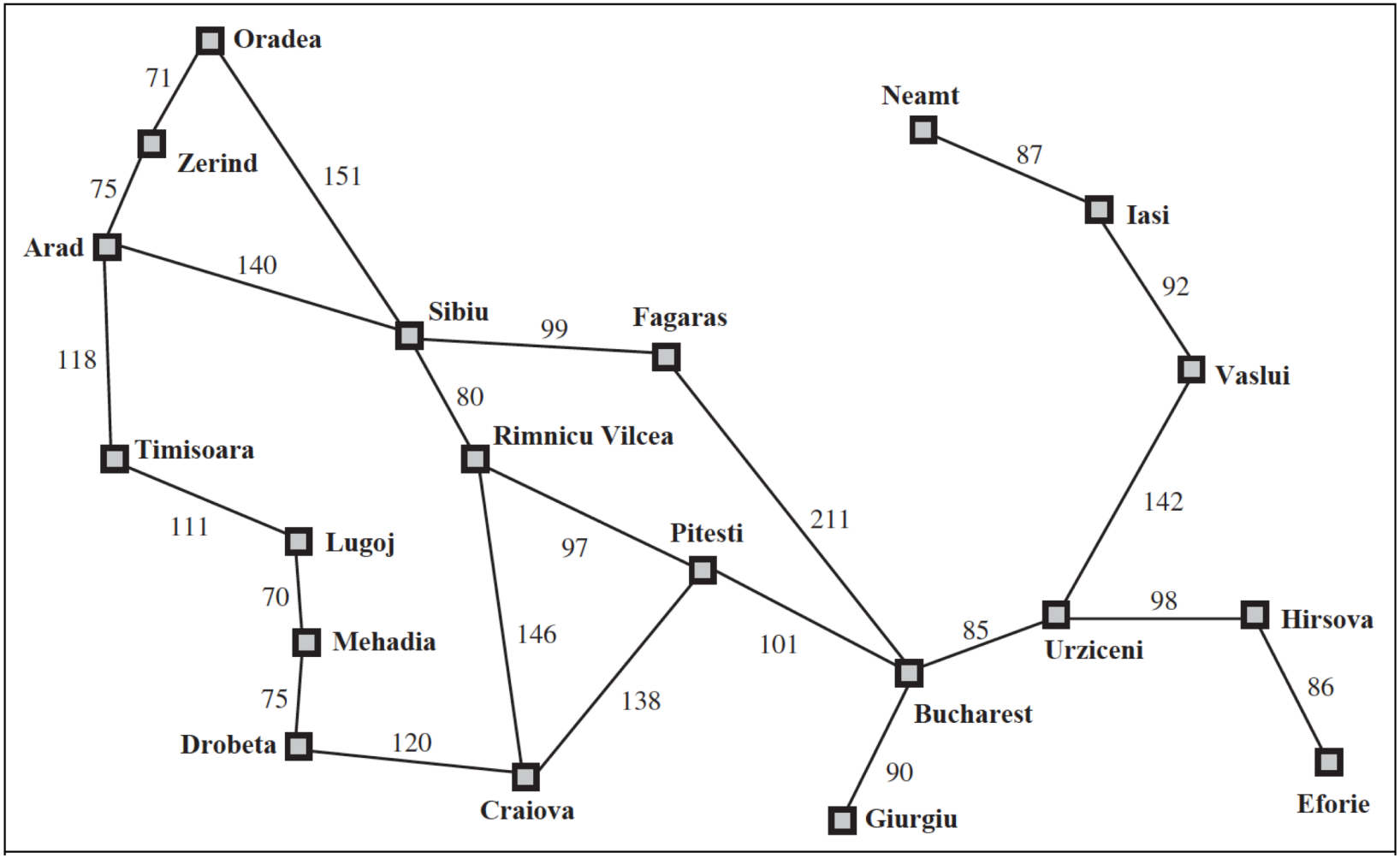
static: *seq*, an action sequence
state, some description of the current world state
goal, a goal
problem, a problem formulation

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-solving agent

Example: Map navigation (Romania)



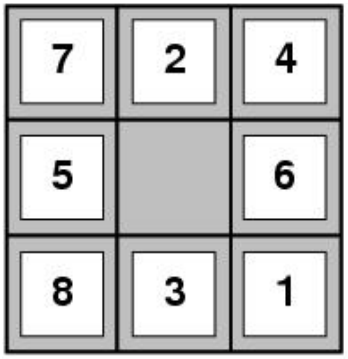
Problem-solving agent
Example: Map navigation (Romania)

- From environment:
  - On holiday in Romania; currently in Arad
  - Want to go to Bucharest
- Formulate goal
  - Be in Bucharest
- Formulate problem
  - States: various cities
  - Actions: drive between cities
- Find a solution
  - Sequence of cities; e.g. Arad, Sibiu, Fagaras, Bucharest, …
- Execute the solution

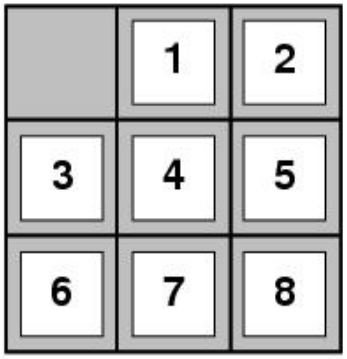
Problem-solving agent

Example: 8-puzzle

- From environment:
  - From the initial game board
  - Want to change the board to the goal
- Formulate goal
  - Game board wanted
- Formulate problem
  - States: board configurations
  - Actions: Left, Right, Up, Down
- Find a solution
  - Sequence of actions



Start State



Goal State

Game's rule:

- Do not lift up titles
- Just shift tiles into the empty to make a move
- Objective: change the initial board to the goal board

End video

Problem formulation

- How?
- Examples

Problem formulation
How?

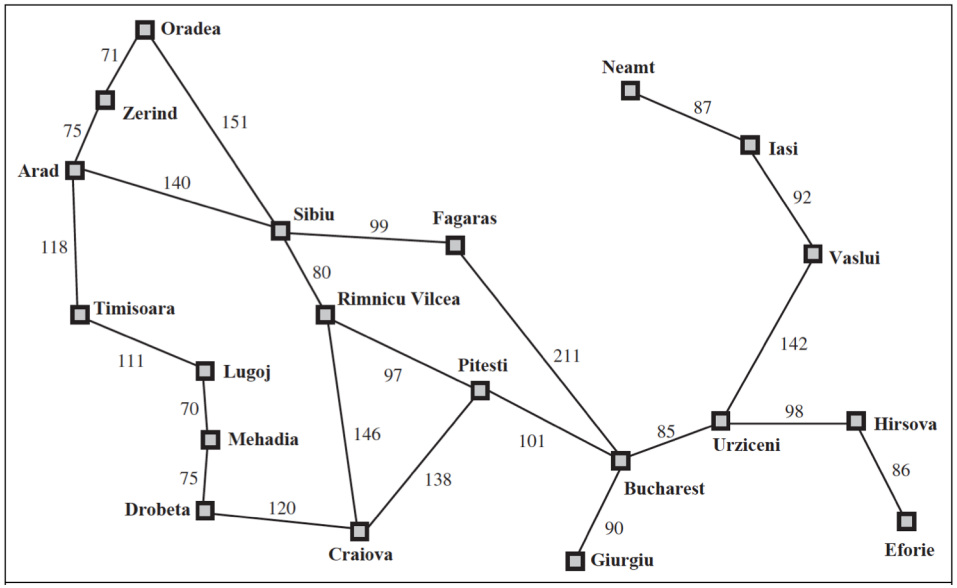
- A problem is defined by:
  - An initial state, e.g. *Arad*
  - Successor function *S(X)*= set of action-state pairs
    - ✓ e.g. *S(Arad)={<Arad → Zerind, Zerind>,…}*

  intial state + successor function = state space
  - Goal test, can be
    - ✓ Explicit, e.g. *x='at bucharest'*
    - ✓ Implicit, e.g. *checkmate(x)*
  - Path cost (additive)
    - ✓ e.g. sum of distances, number of actions executed, …
    - ✓ *c(x,a,y)* is the step cost, assumed to be >= 0

A solution is a sequence of actions from initial to goal state.
Optimal solution has the lowest path cost.

Problem formulation

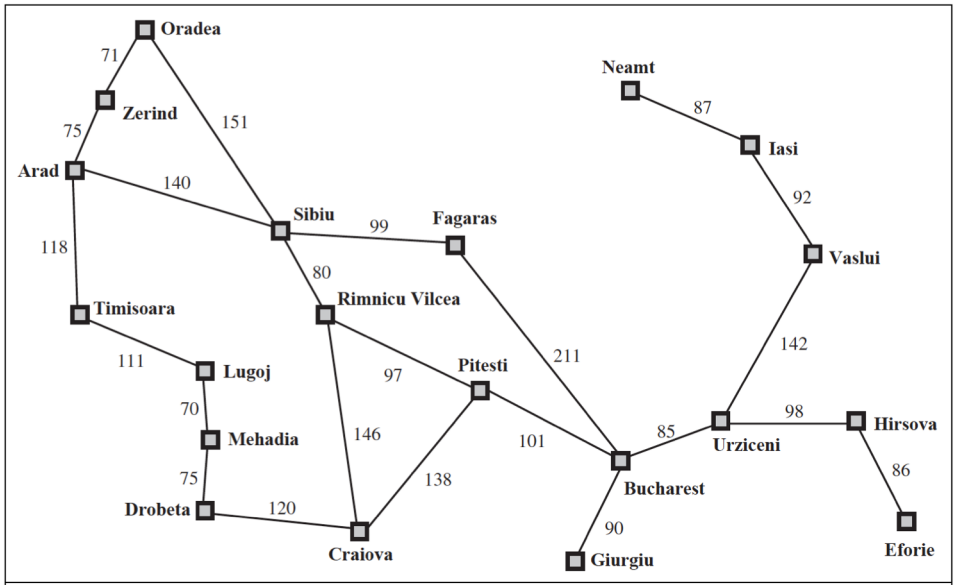
Example: Map navigation (Romania)



- States:
  - Arad, Oradea, Sibiu, etc
- Intitial State:
  - e.g.: Arad
- Successor:
  - S(Arad) =
    - ✓ <go(Arad,Zerind), Zerind>,
    - ✓ <go(Arad,Sibiu), Sibiu>,
    - ✓ <go(Arad,Timisoara), Timisoara >,

Problem formulation

Example: Map navigation (Romania)



- Goal:
  - e.g.: Bucharest
- Step cost:
  - cost of taking an action
    - ✓ go(Arad, Zerind): 75
    - ✓ go(Arad, Sibiu): 140
    - ✓ go(Arad, Timisoara): 118
    - ✓ …
- Path cost:
  - Cost of taking a sequence of actions:
    - ✓ Path(Arad,Sibiu, Fagaras, Bucharest): 140 + 99 + 211 = 450

Problem formulation

Concepts

- State:
  - Abstraction representation of the environment or the problem's world
  - **(Abstract) state = set of real states.**
- Action:
  - Abstraction representation of a complex combination of real actions.
  - The abstraction is valid if the path between two states is reflected in the real world.
  - Example:
    - ✓ **go(Arad, Zerind)**: represents a complex set of possible routes, detours, rest stops, etc.

Problem formulation
Concepts

- State space:
  - Abstraction representation of the real world or the problem's world.
  - **State space = Initial state + successor**
  - **State space is formalized as a directed graph**

- (Abstract) solution = set of real paths that are solutions in the real world.
  - Searching on state space ➔ solution

End Video

Problem type

- Concepts
- Examples

Problem type
Concepts

- Type of problems in this chapter:
  - Fully observable:
  - Deterministic
  - Discrete
  - Static

Problem type
Concepts

- Fully observable:
  - Using sensors
  - Agents have enough information for selecting actions
    - ✓ Know where it is
    - ✓ Know on which state it stays
    - ✓ Know which actions can be performed with a given state?
- Partially observable:
  - Using sensors, maybe with noises or inaccurate
  - Lack of information for selecting actions
- Unobservable:
  - No sensors

Problem type
Concepts

- Deterministic
  - Perform an action at a state => result only state
  - Fully observable + deterministic
    - ✓ The solution: a sequence of actions instead of a tree
- Stochastic
  - Perform an action at a state => result more than one states
    - ✓ Each state is associated with a probability taken
- Non-deterministic
  - Perform an action at a state => result more than one states
    - ✓ Resultant states need not attached with a probability taken

Problem type
Concepts

- Discrete
  - Space of states, of actions: discrete
  - So, we can say "set of actions", "set of states"

- Continuous:
  - Space of states, of actions: discrete
  - For examples
    - ✓ Autonomous car:
      - States defined in continuous spaces: location of the agent and of other moving objects/agents; time of navigation, etc
      - Action: turn left/right with a continuous angle (45.5 degree, example)

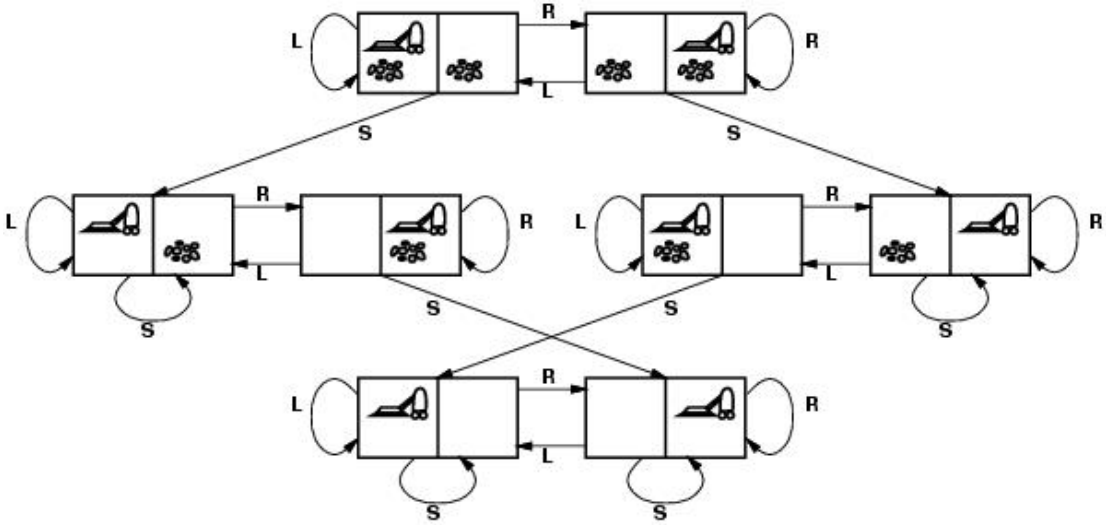
Problem type
Concepts

- Static
  - The environment does not change while an agent is deliberating
- Dynamic
  - The environment changes while an agent is deliberating
  - Examples
    - ✓ Autonomous car:
      - Other agents or moving objects continuous changes their location.
- Semi-Dynamic:
  - The environment does not change while an agent is deliberating, but the performance score does

End video

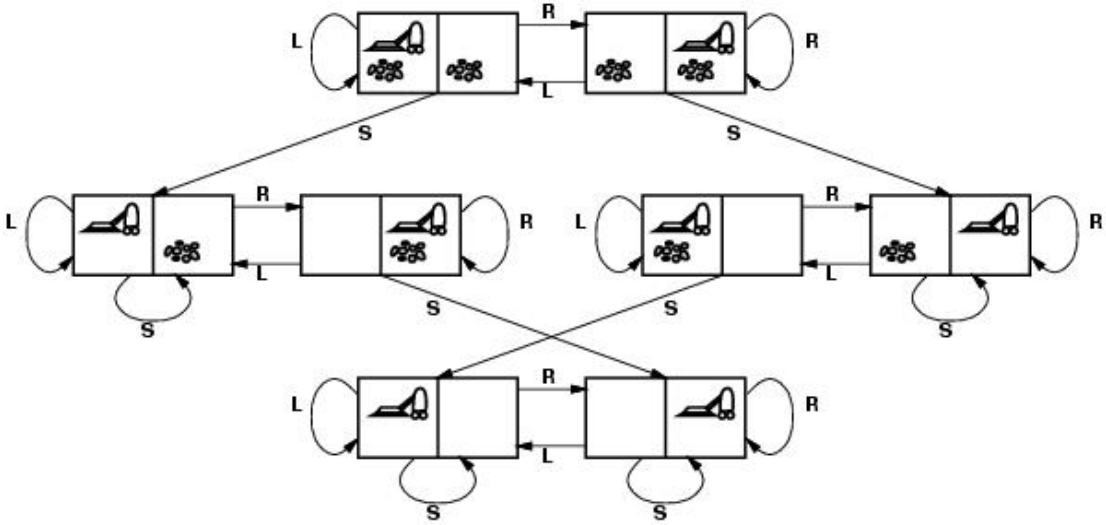
Typical problems

Typical problems
Example: Vacuum world



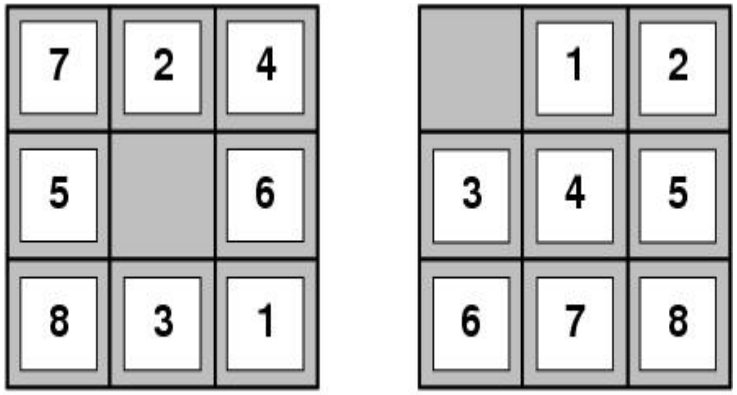
- States??
- Initial state??
- Actions??
- Goal test??
- Path cost??

Typical problems
Example: Vacuum world



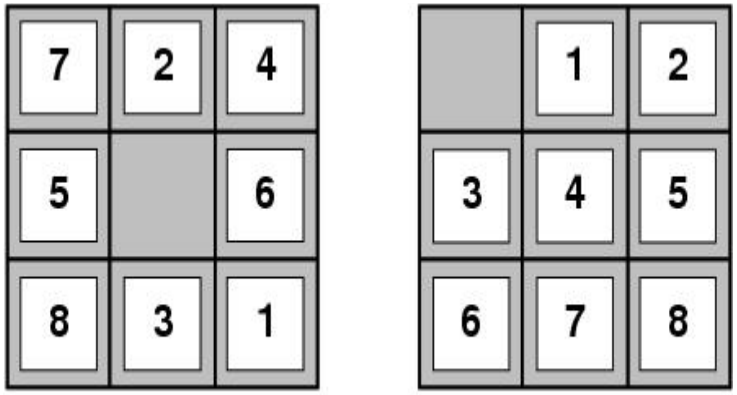
- States?? two locations with or without dirt: 2 x 2^2=8 states.
- Initial state?? Any state can be initial
- Actions?? {*Left*, *Right*, *Suck*}
- Goal test?? Check whether squares are clean.
- Path cost?? Number of actions to reach goal (step cost = 1)

Typical problems
Example: 8-puzzle



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

Typical problems
Example: 8-puzzle

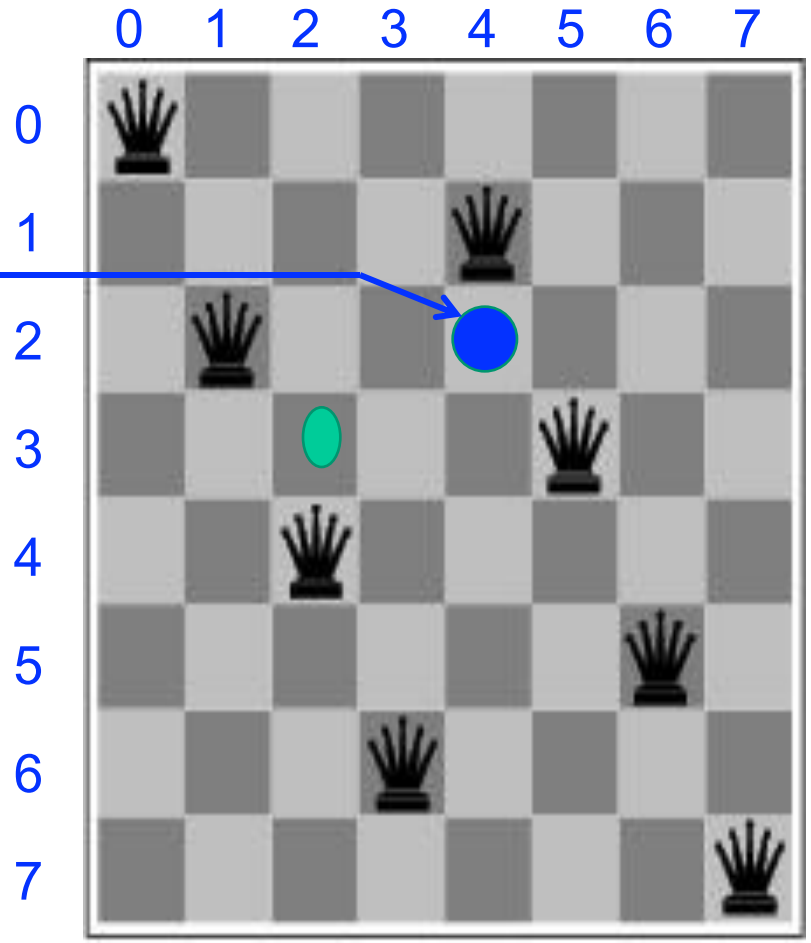


Start State Goal State

- States?? Integer location of each tile
- Initial state?? Any state can be initial
- Actions?? {*Left*, *Right*, *Up*, *Down*}
- Goal test?? Check whether goal configuration is reached
- Path cost?? Number of actions to reach goal

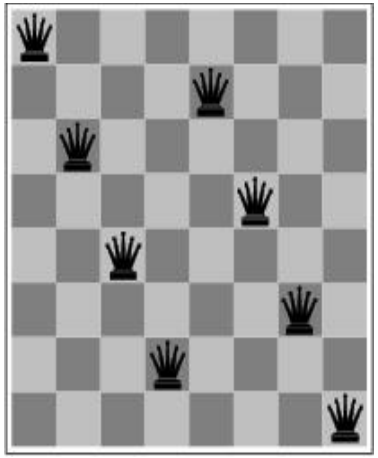
Typical problems
Example: 8-queens

Coodinate: (x, y) = (4, 2)



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

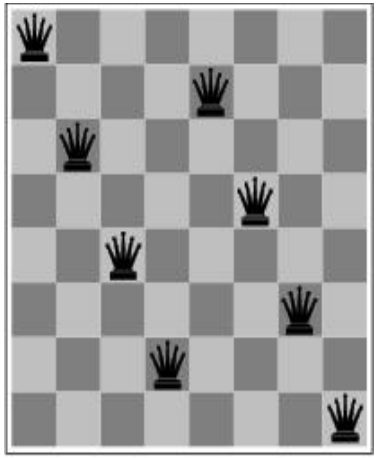
Typical problems
Example: 8-queens



Incremental formulation vs. complete-state formulation

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

Typical problems
Example: 8-queens

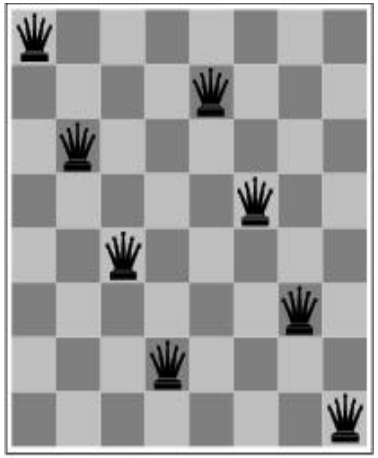


Incremental formulation

- 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*62* .. * 57 ~ 3 x 10^{14} possible sequences to investigate

Typical problems
Example: 8-queens

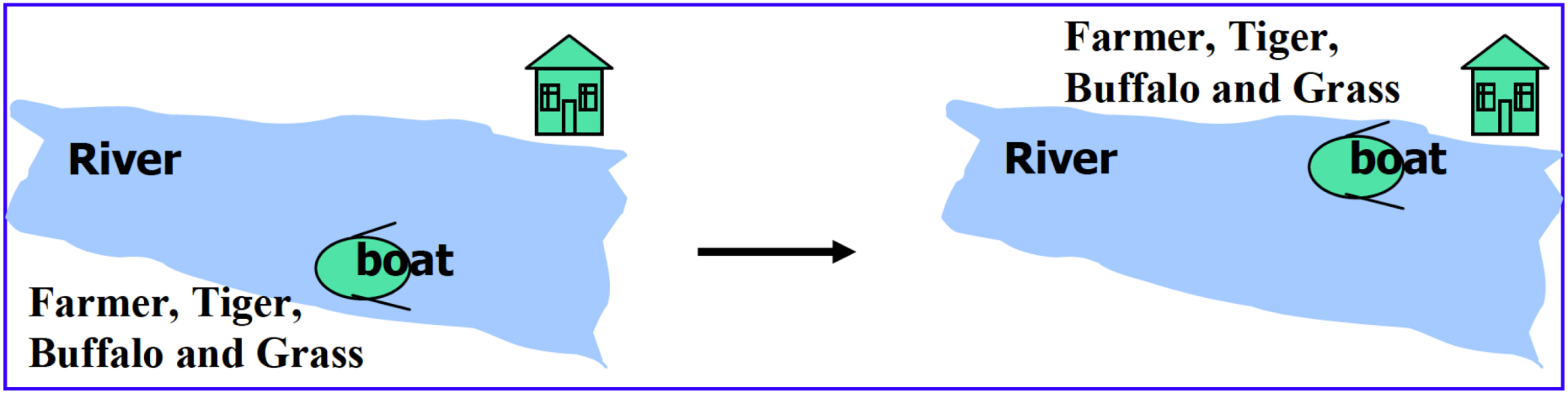


Incremental formulation (alternative)

- States?? n ($0 \leq n \leq 8$) 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

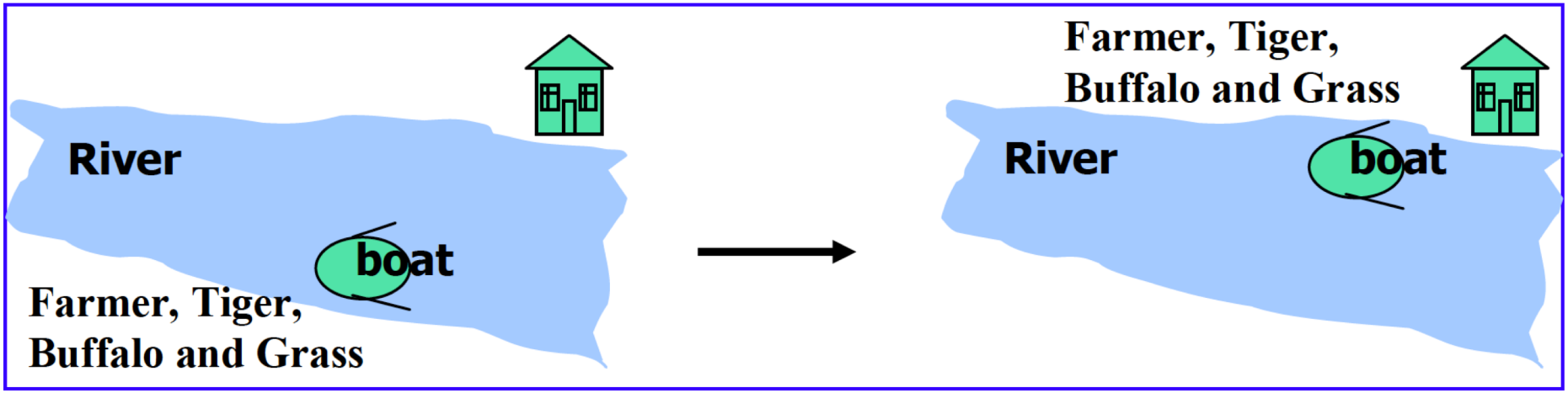
  2057 possible sequences to investigate; Yet makes no difference when n=100

Typical problems
Water jugs



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

Typical problems
Water jugs



- States?? [Farmer:W/E, Tiger:W/E, Buffalo:W/E, Grass:W/E]
  - W: West; E: East
- Initial state?? [W, W, W, W]
- Actions?? Possible moves that satisfies the game
- Goal test?? [E, E, E, E]
- Path cost?? Step-cost = 1

Typical problems

Missionaries and cannibals

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

Typical problems
Missionaries and cannibals

- States?? [W: Mis-coun, W: Can-count, E: Mis-coun, E: Can-count]
  - W: West; E: East; Miss-count = Count of missionaries; Can-count = count of cannibals
- Initial state?? [0, 0, 3, 3]
- Actions?? Possible moves that satisfies the game
- Goal test?? [3, 3, 0, 0]
- Path cost?? Step-cost = 1

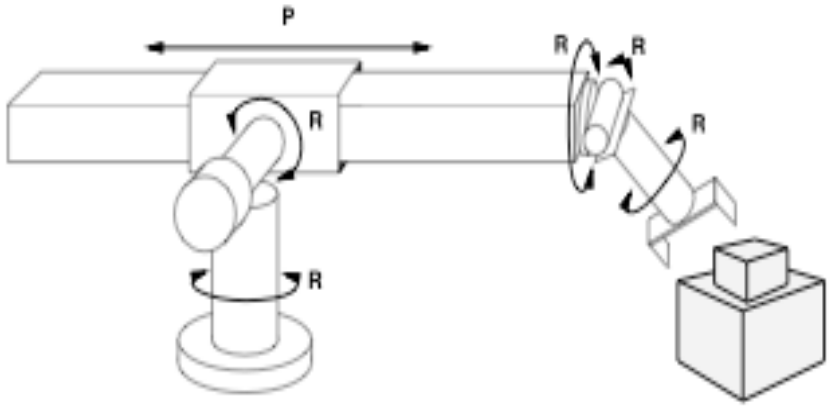
Typical problems
Water jugs

Given 4-liter and 3-liter pitchers, how do you get exactly 2 liters into the 4-liter pitcher?

State: (x, y) **for # liters in 4-liter and 3-liter pitchers, respectively**
Actions: empty, fill, pour water between pitchers
Initial state: (0, 0)
Goal state: (2, *)

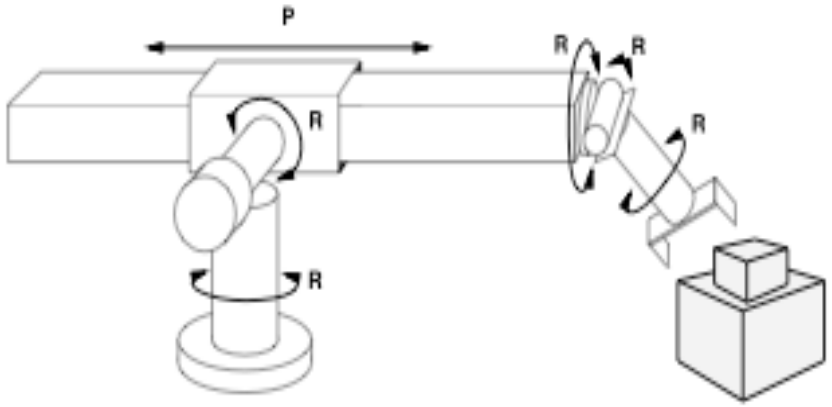
(source: https://www.chegg.com/)

Typical problems
Example: robot assembly



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

Typical problems
Example: robot assembly



- States?? Real-valued coordinates of robot joint angles; parts of the object to be assembled.
- Initial state?? Any arm position and object configuration.
- Actions?? Continuous motion of robot joints
- Goal test?? Complete assembly (without robot)
- Path cost?? Time to execute

End video

Tree search

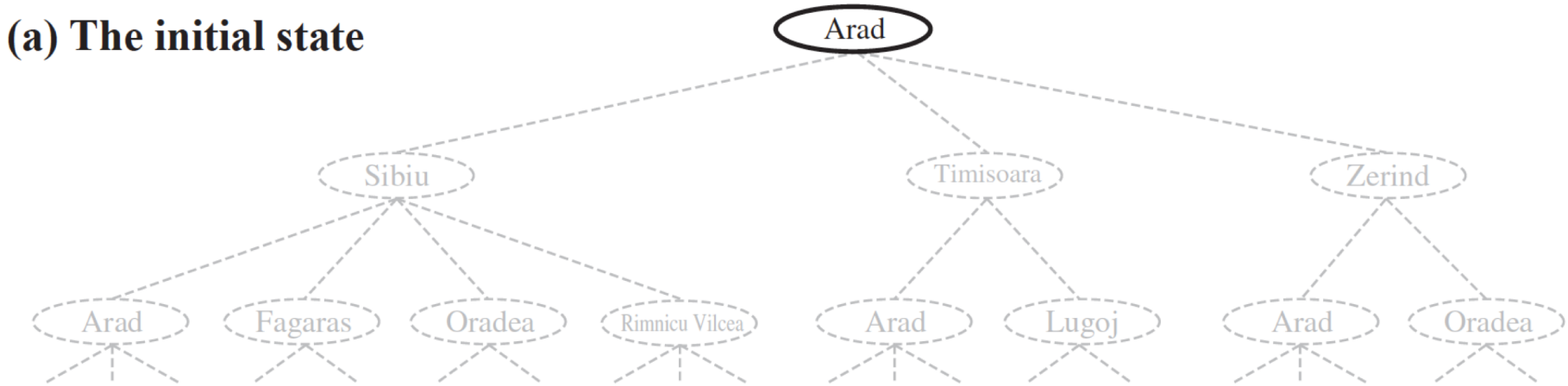
(State space is formulated as a tree instead of a graph)

- Why?
- Simple tree search example
- State space vs. search tree
- Tree search algorithm
- Criteria for evaluation

Search: why?

- 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*

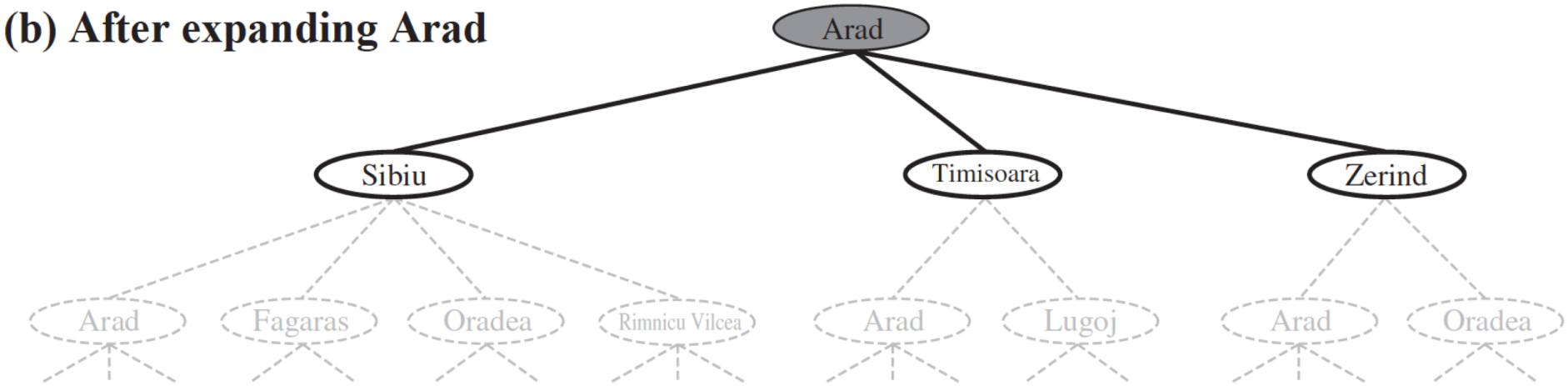
choose leaf node for expansion according to *strategy*

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*

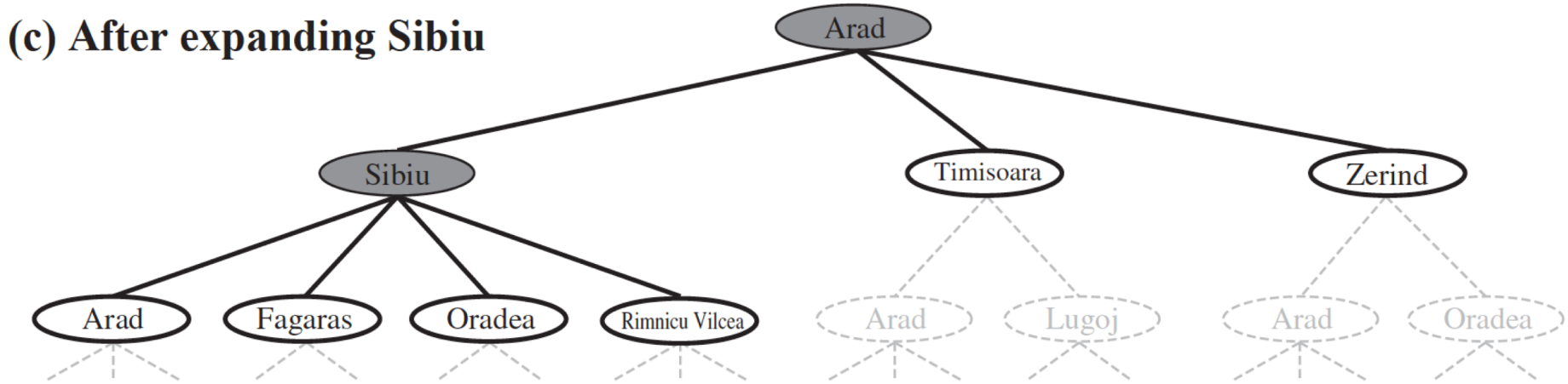
choose leaf node for expansion according to *strategy*

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*

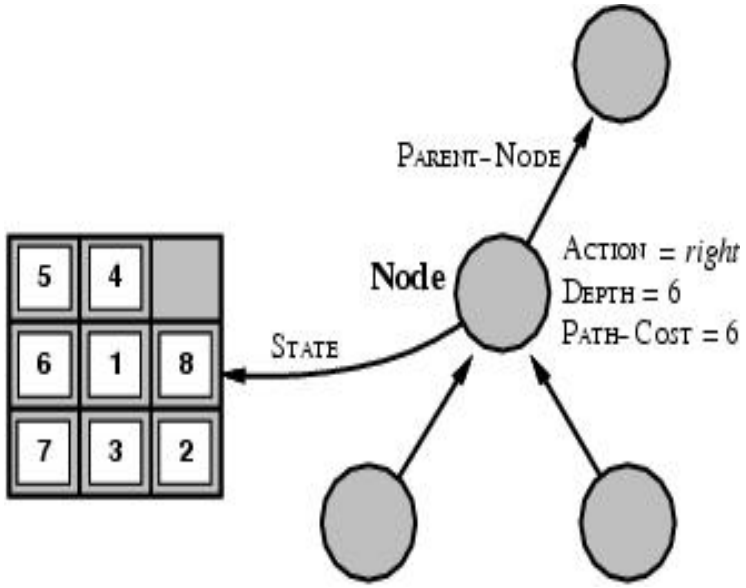
choose leaf node for expansion according to *strategy* ← **Determines search process!!**

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 ncludes 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

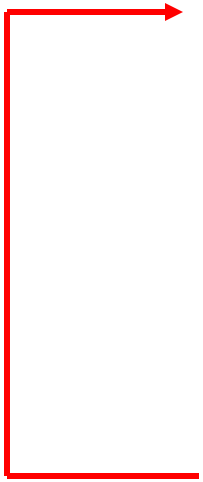
if EMPTY?(*fringe*) **then return** failure

node ← REMOVE-FIRST(*fringe*)

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

then return SOLUTION(*node*)

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



Node being explored (xem xét/xử lý)

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 strategy is defined by picking the order of node expansion.
- Problem-solving performance is measured in four ways:
  - Completeness (trọn vẹn)
    - ✓ *Does it always find a solution if one exists?*
  - Optimality (tối ưu)
    - ✓ *Does it always find the least-cost solution?*
  - Time Complexity (thời gian)
    - ✓ *Number of nodes generated/expanded?*
  - Space Complexity (bộ nhớ)
    - ✓ *Number of nodes stored in memory during search?*

Search strategies

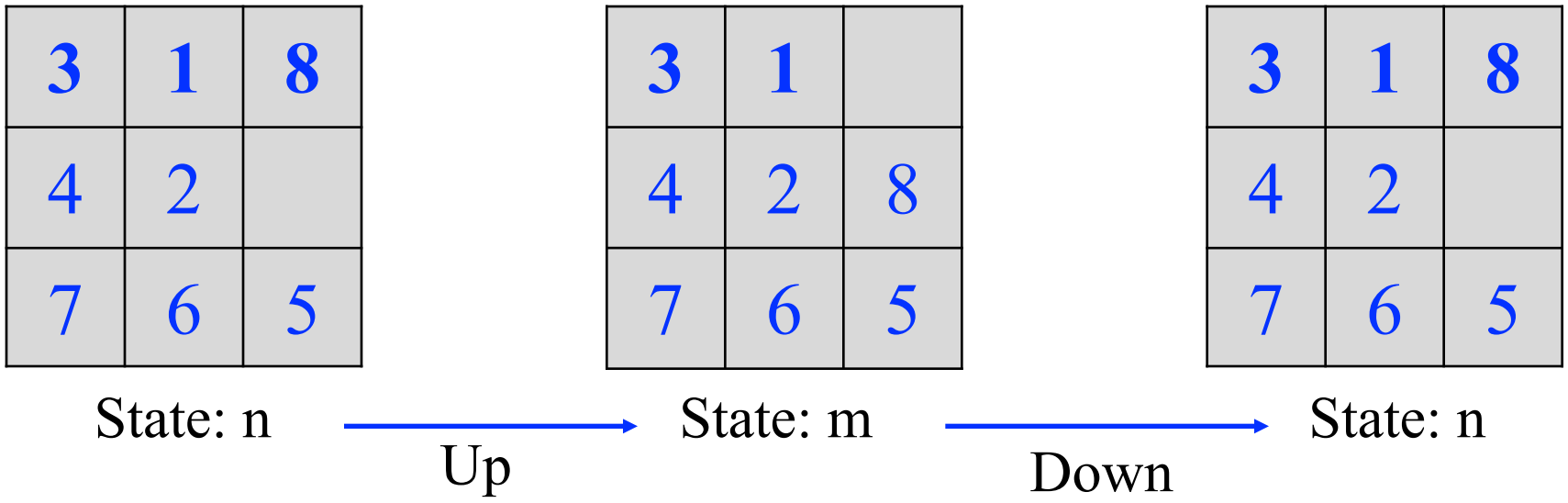
- Time and space complexity are measured in terms of problem difficulty defined by:
  - *b :maximum branching factor of the search tree*
  - *d :depth of the least-cost solution (the goal state that has the shallowest depth)*
  - *m :maximum depth of the state space (may be ∞)*
    - ✓ *m != ∞: **finite space***
    - ✓ *m = ∞: **infinite space***

Graph search

(State space is formulated as a directed graph instead of a tree model)

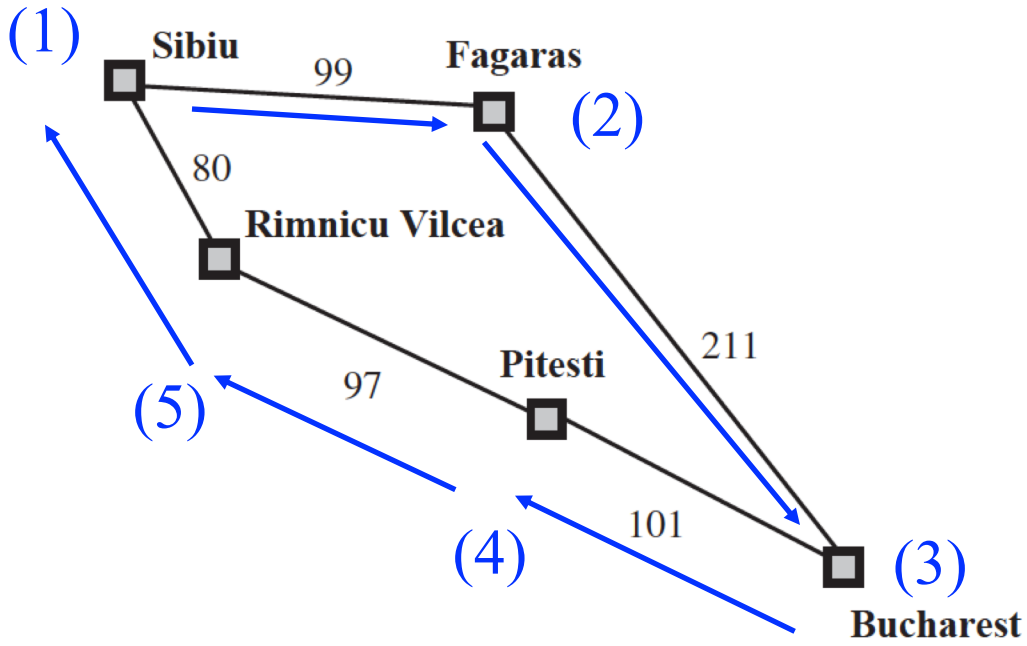
- Repeated states, loopy paths, redundant paths
- Graph search algorithm

Repeated state, loopy path

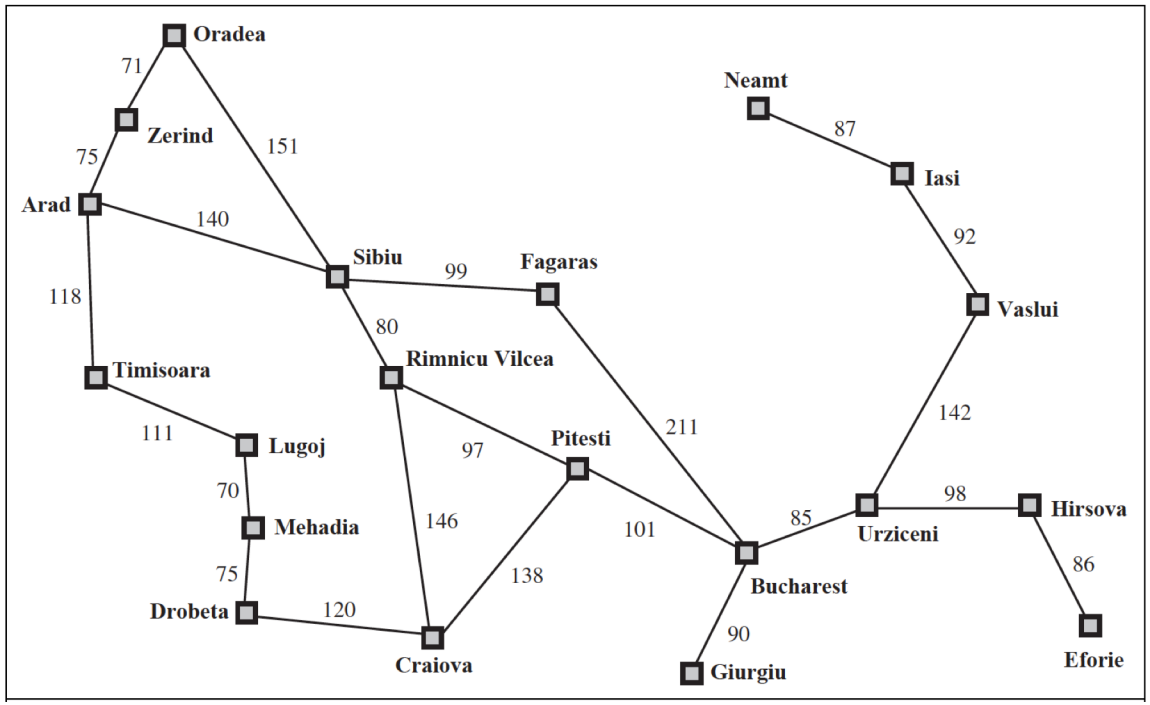


Loopy path: n → m → n

Repeated state, loopy path



Redundant path



Arad → Sibiu:

- Path 1: Arad → Sibiu (140)
- Path 2 (redundant): Arad → Zerind → Oradea → Sibiu (75 + 71 + 151)

Frontier and closed-list

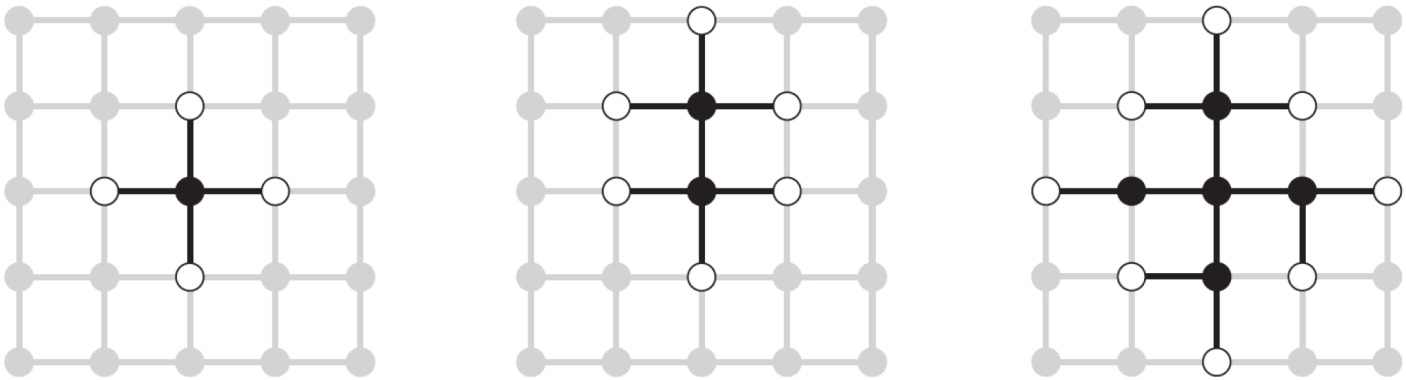


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

- Fringe: stores all nodes being explored (white nodes, called frontier)
- Closed list stores all expanded nodes (black nodes)

Graph search algorithm

```
function GRAPH-SEARCH(problem,fringe) return a solution or failure
    closed ← an empty set
    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)
        if STATE[node] is not in closed then
                add STATE[node] to closed
                fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

Graph 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*

Modify for graph search

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!!

End video

Basic search strategies (I)

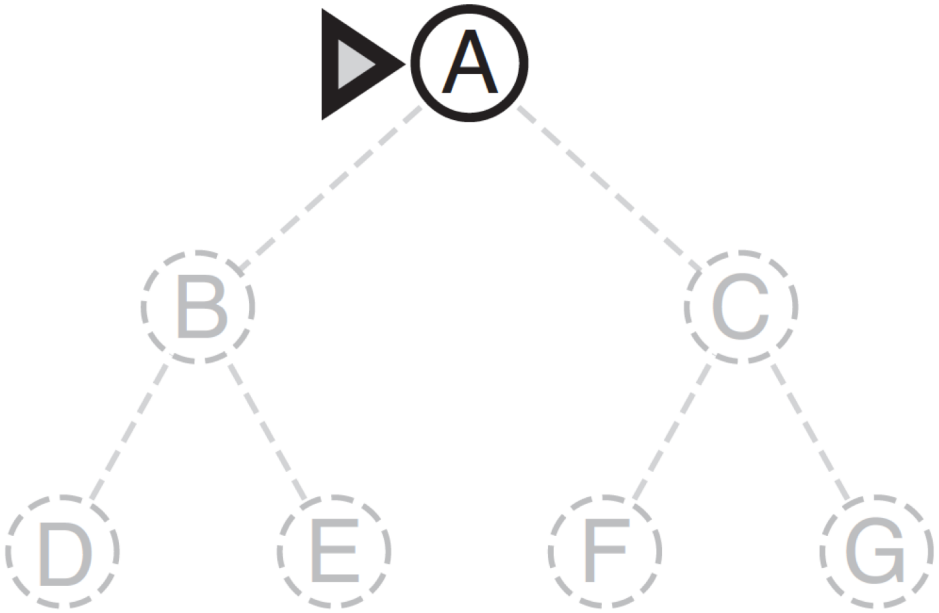
- Breadth First Search (BFS, BF-Search)
- Uniform Cost Search (UCS, UC-Search)

Uninformed search strategies

- (a.k.a. blind search, exhaustive search) = use only information available in problem definition.
  - When strategies can determine whether one non-goal state is better than another → informed search.
- Model of state space
  - Tree model (tree search)
  - Graph model (graph search)
- Search strategies
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search.
  - Bidirectional search
- **State space search = Model of state space + Search strategies**

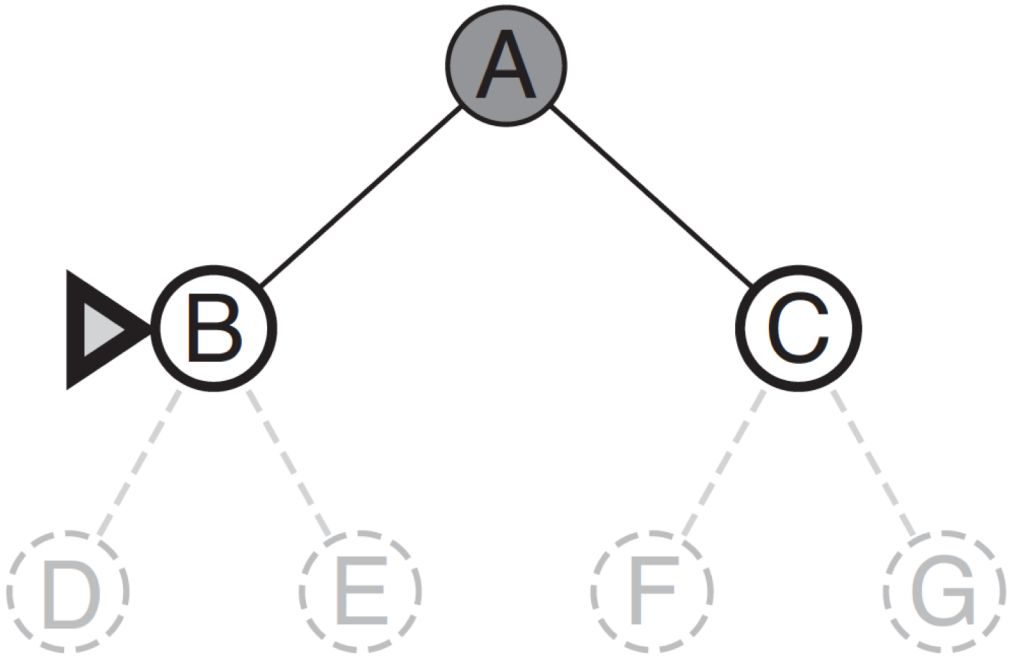
Breadth First Search, an example

- Breadth-First-Search: BF-Search
- Expand *shallowest* unexpanded node
- Implementation: *fringe* is a FIFO queue



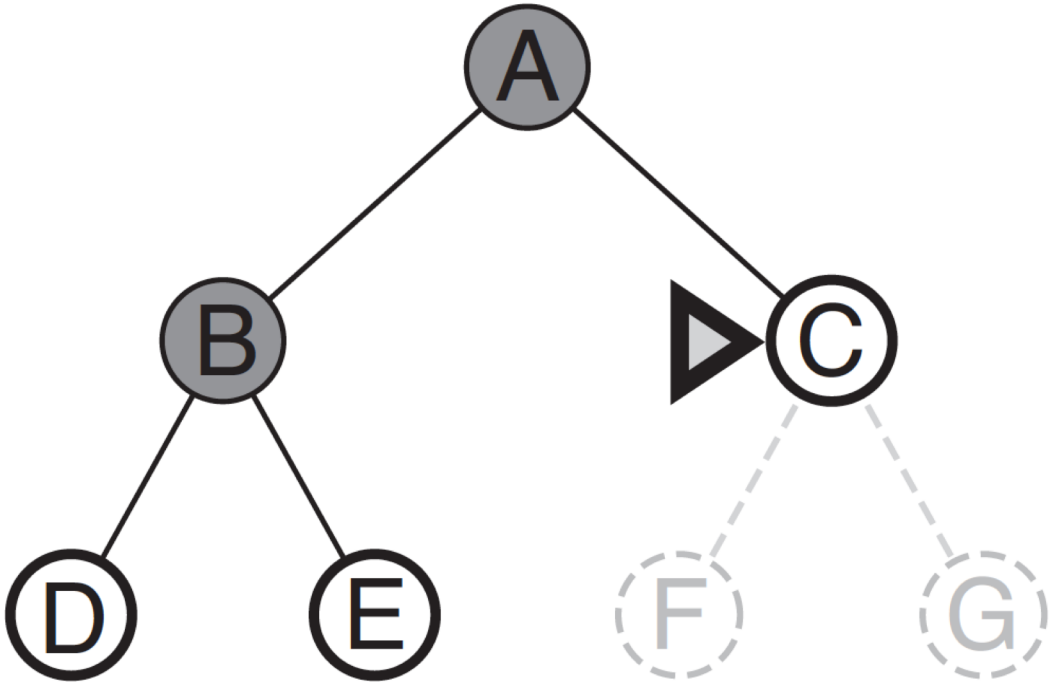
Breadth First Search, an example

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



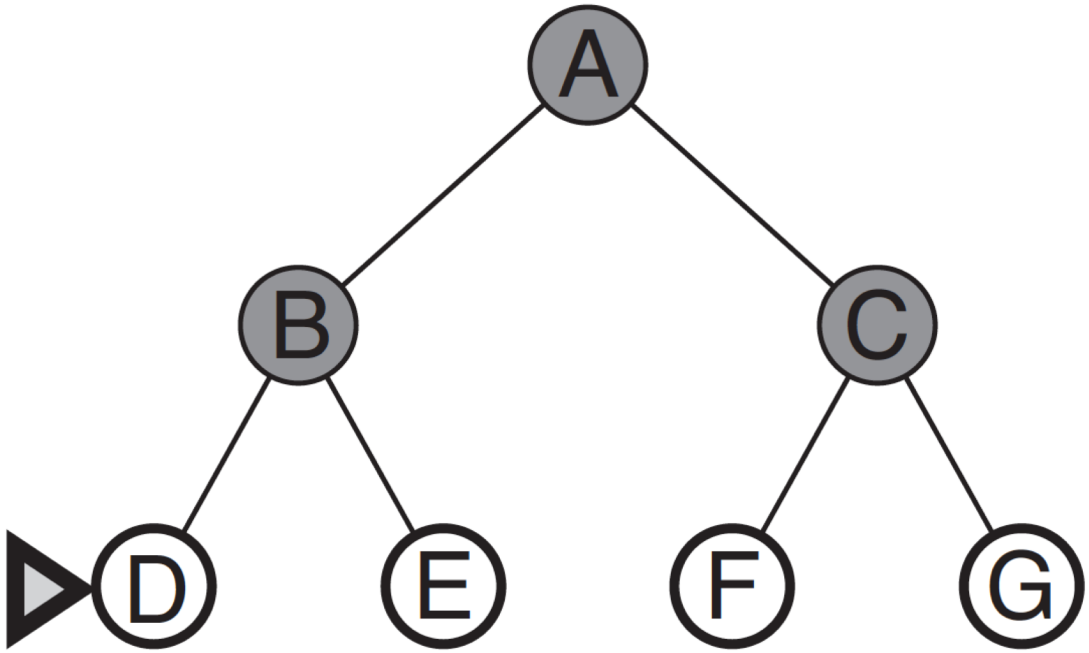
Breadth First Search, an example

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



Breadth First Search, an example

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



Breadth First Search; evaluation

- Completeness:
  - *Does it always find a solution if one exists?*
  - YES
    - ✓ If shallowest goal node is at some finite depth d
    - ✓ Condition: If *b* is finite
      - (maximum num. Of succ. nodes is finite)

BF-search; evaluation

- Completeness:
  - YES (if *b* is finite)
- Time complexity:
  - Assume a state space where every state has *b* successors.
    - ✓ root has *b* successors, each node at the next level has again b successors (total b^2), …
    - ✓ Assume solution is at depth *d*
    - ✓ Worst case; expand all but the last node at depth *d*
    - ✓ Total numb. of nodes generated:

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

Meet goal => not generate b children for the goal

BF-search; evaluation

- Completeness:
  - YES (if *b* is finite)
- Time complexity:
  - Total numb. of nodes generated:

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

- Space complexity:
  - Idem if each node is retained in memory

BF-search; evaluation

- Completeness:
  - YES (if *b* is finite)
- Time complexity:
  - Total numb. of nodes generated:

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

- Space complexity:
  - Idem if each node is retained in memory
- Optimality:
  - *Does it always find the least-cost solution?*
  - In general YES
    - ✓ unless actions have different cost.

BF-search; evaluation

- Two lessons:
  - Memory requirements are a bigger problem than its execution time.
  - Exponential complexity search problems cannot be solved by uninformed search methods for any but the smallest instances.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Uniform-cost search

- Extension of BF-search:
  - Expand node with *lowest path cost*
- Implementation: *fringe* = queue ordered by path cost.
  - *Fringe : priority queue (heap)*
- UC-search is the same as BF-search when all step-costs are equal.

```
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
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Uniform-cost search

- Completeness:
  - YES, if step-cost > ε (small positive constant)
- Time complexity:
  - Assume C* the cost of the optimal solution.
  - Assume that every action costs at least ε
  - Worst-case: $O(b^{C^*/\varepsilon})$
- Space complexity:
  - Idem to time complexity
- Optimality:
  - nodes expanded in order of increasing path cost.
  - YES, if complete.

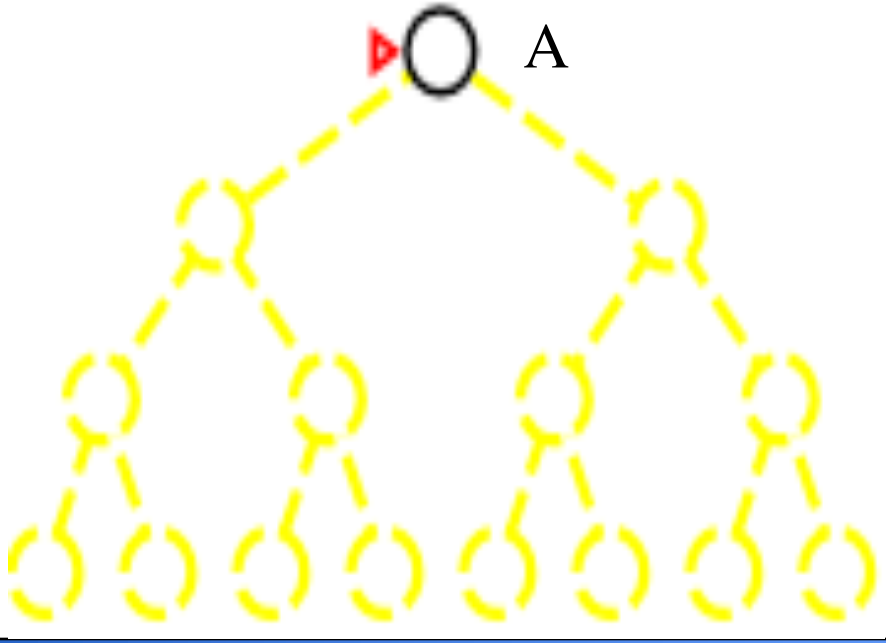
End video

Basic search strategies (II)

- DF-Search
- DLS
- IDS
- Bidirectional Search
- Graph Search algorithm

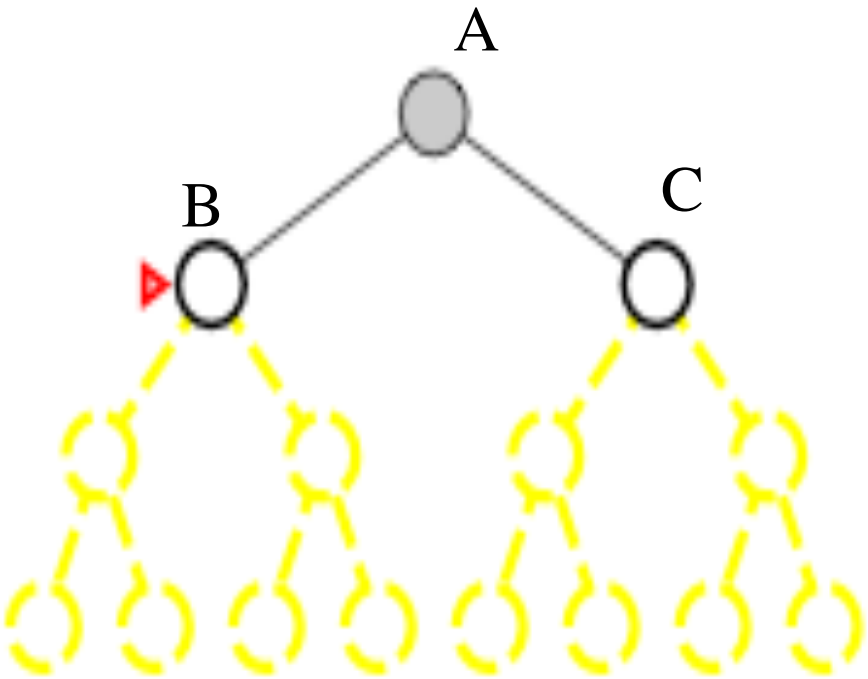
Depth-First-Search, an example

- Depth-First-Search: DF-Search
- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



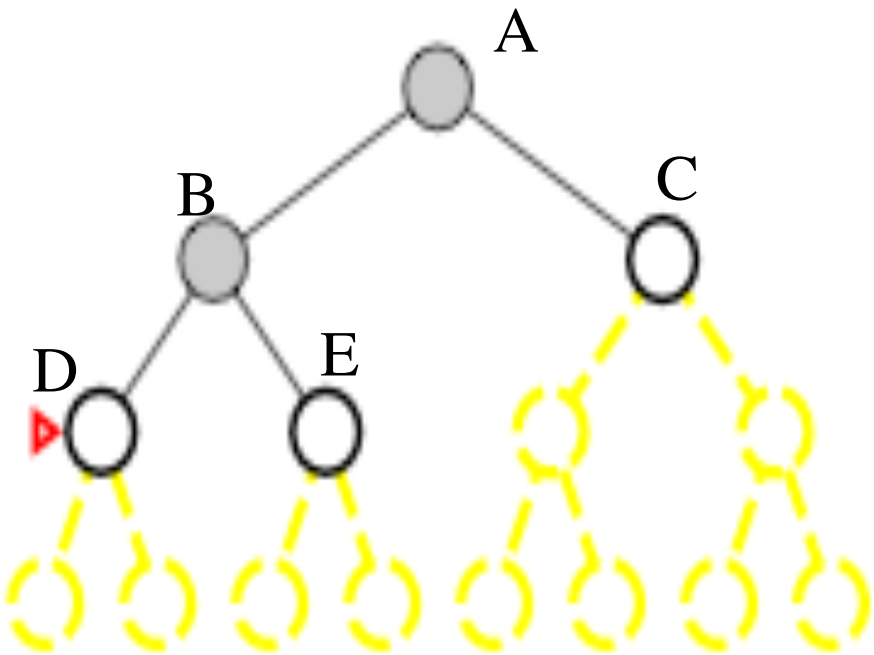
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



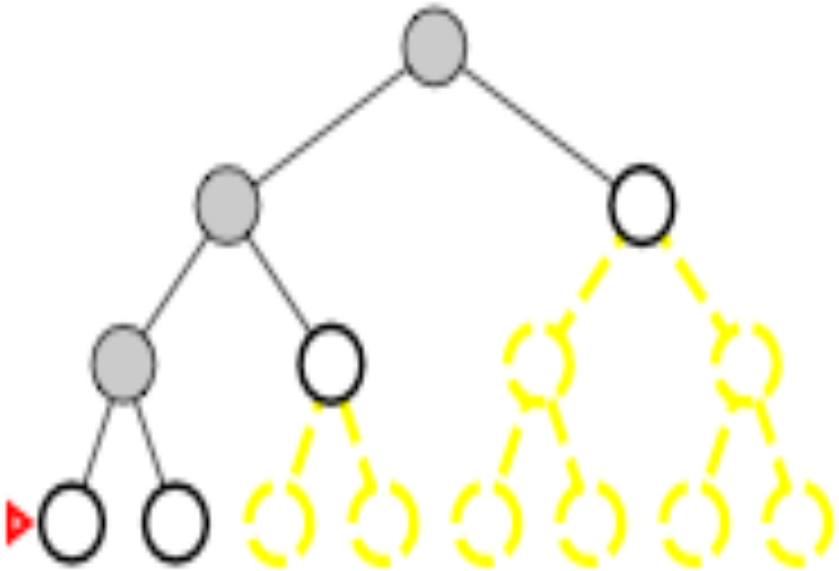
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



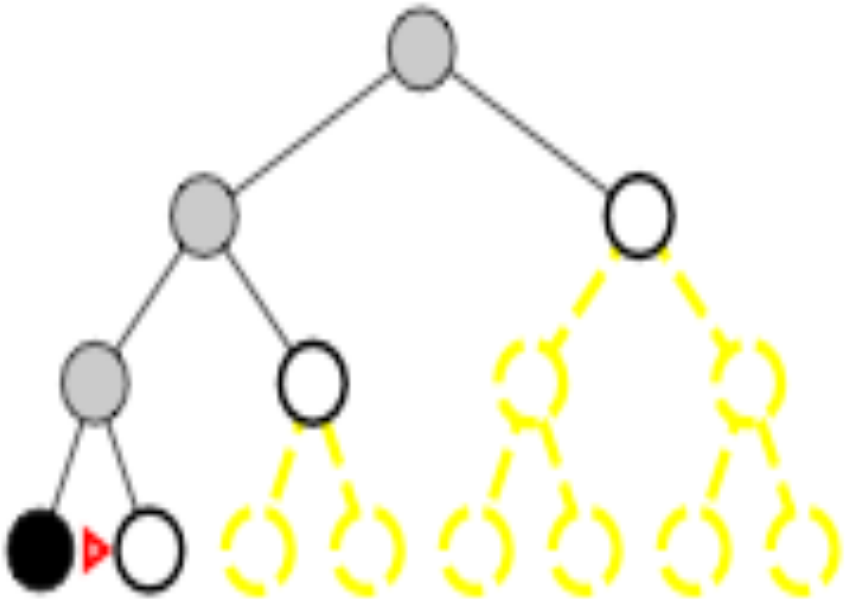
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



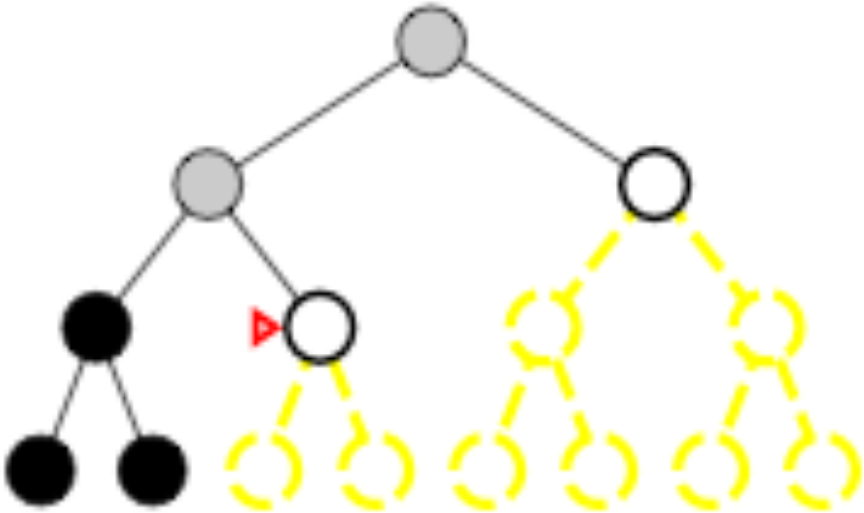
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



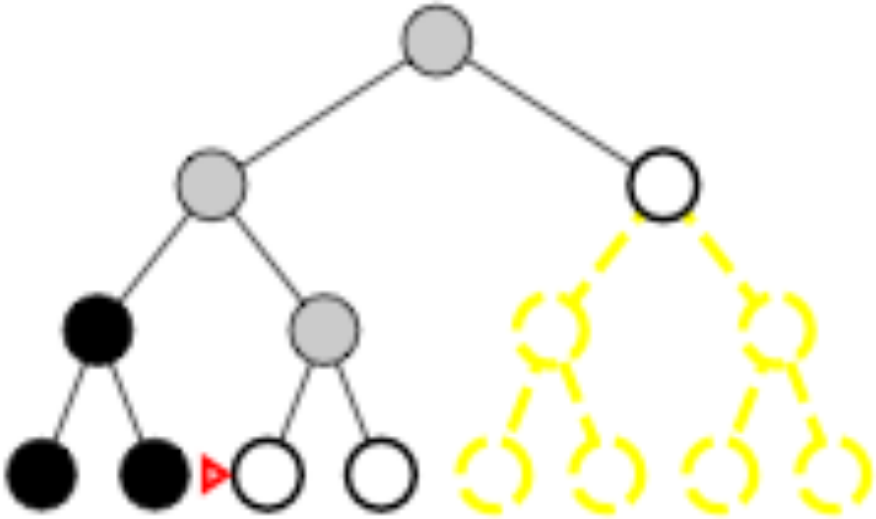
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



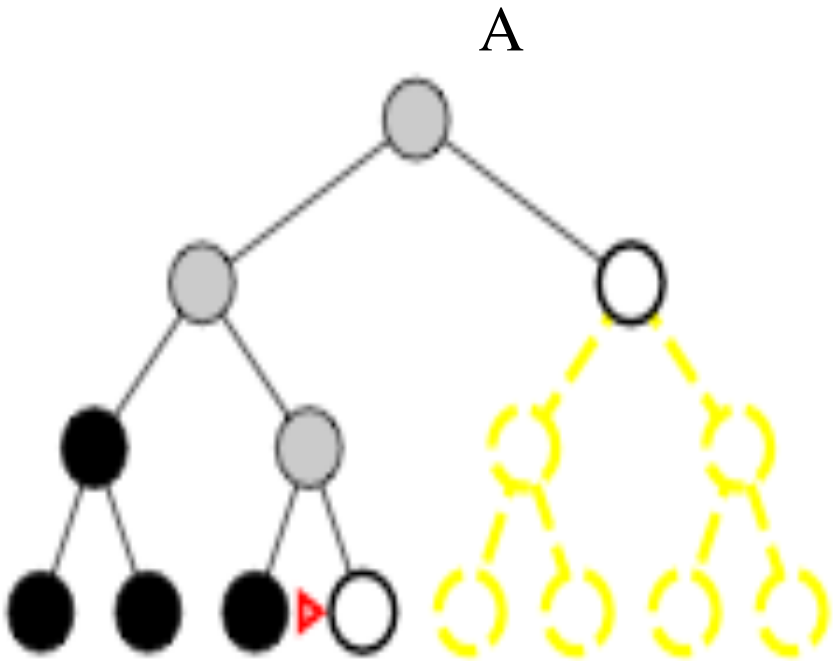
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



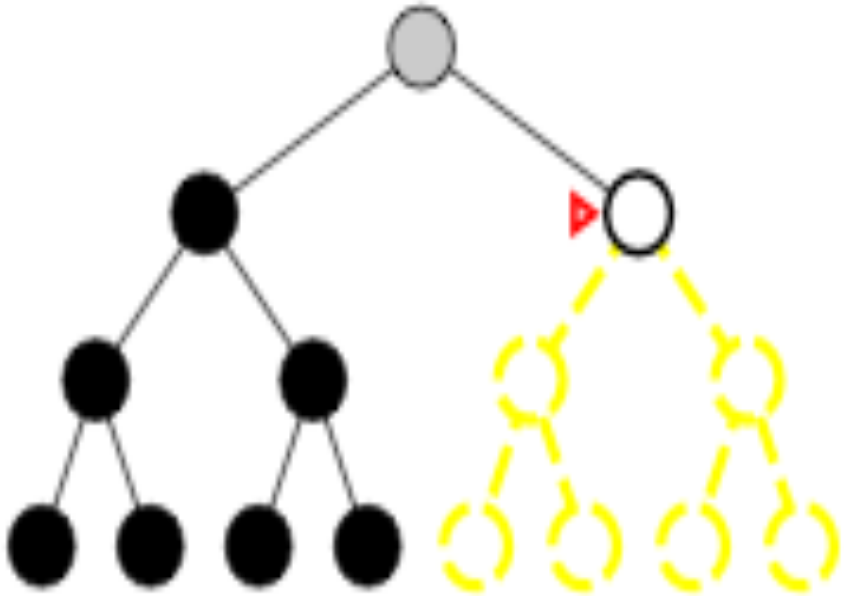
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



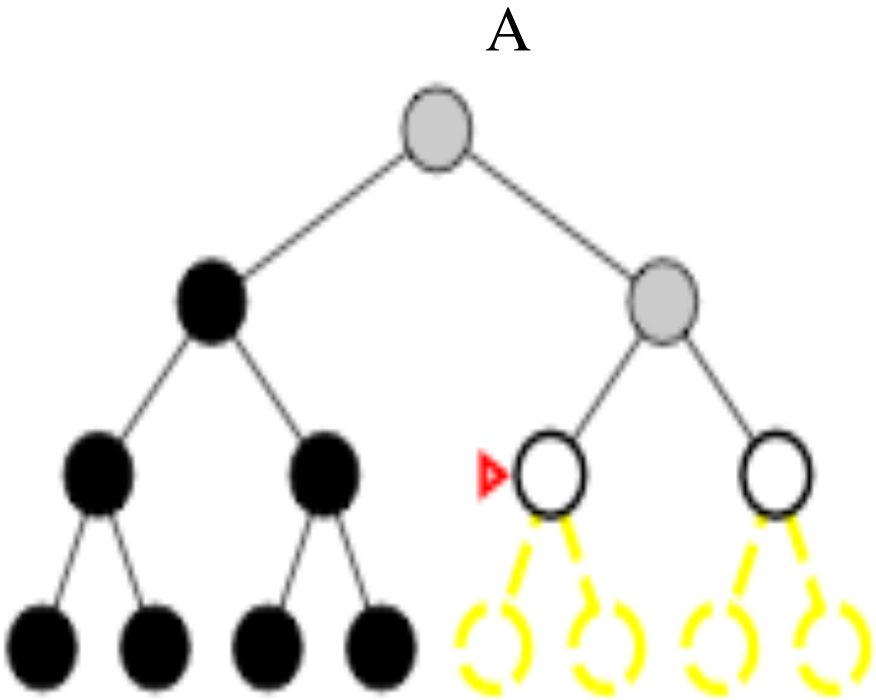
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



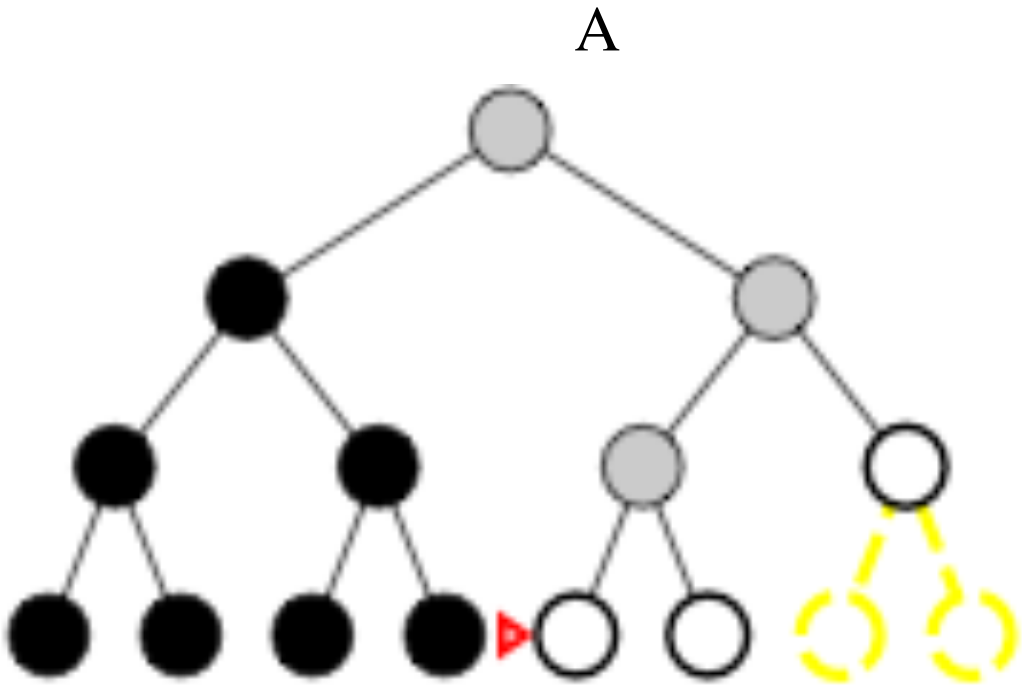
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



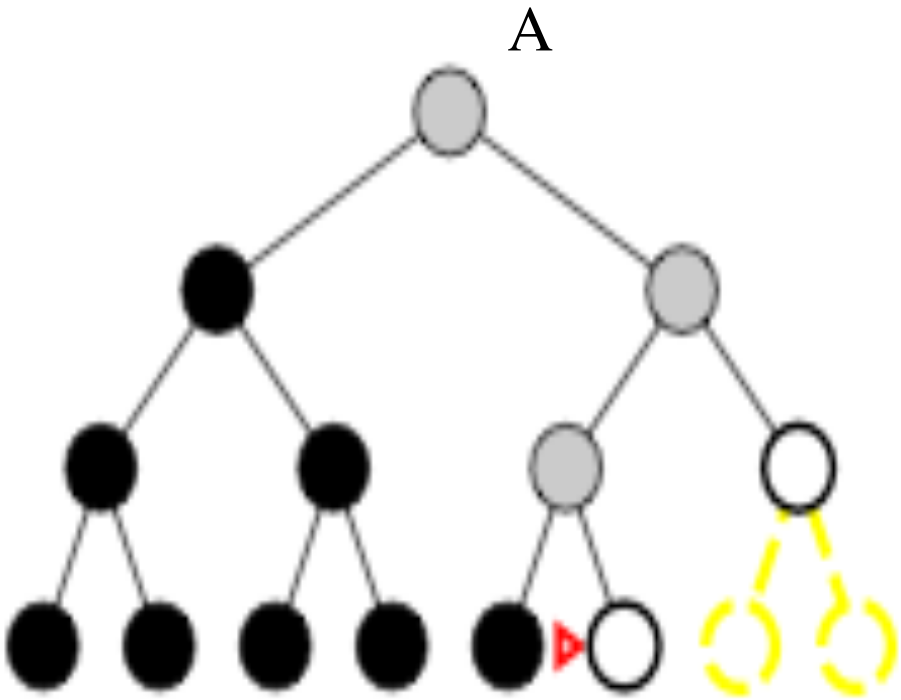
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



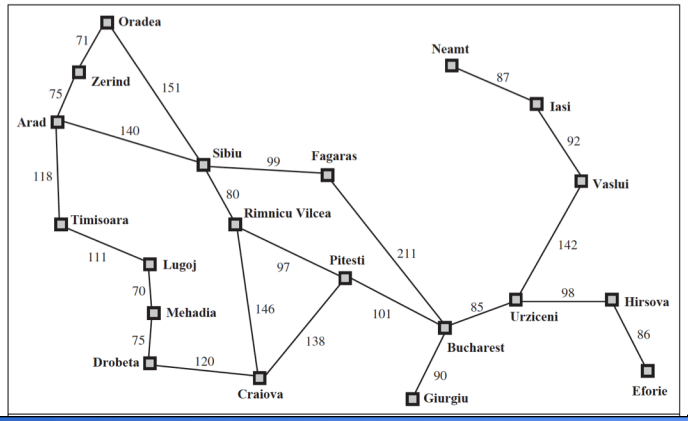
DF-search, an example

- Expand *deepest* unexpanded node
- Implementation: *fringe* is a LIFO queue (=stack)



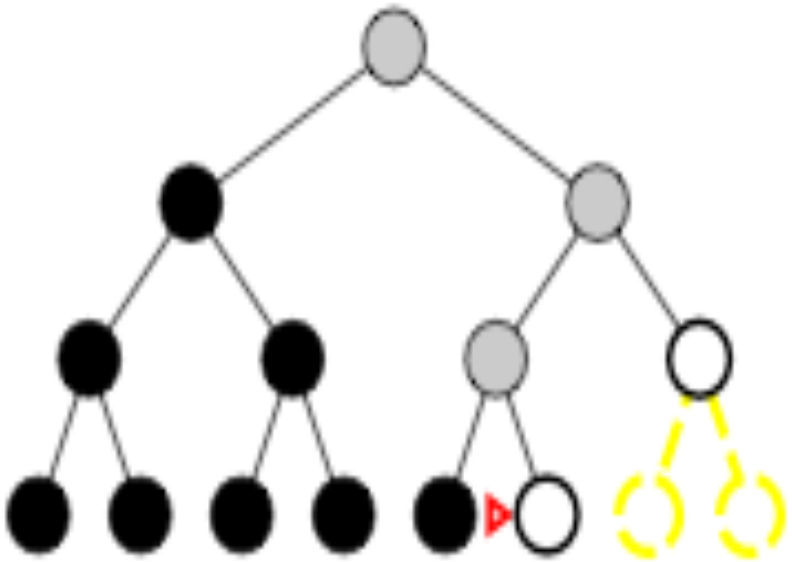
DF-search; evaluation

- Completeness;
  - *Does it always find a solution if one exists?*
  - NO
    - ✓ *unless* search space is finite and no loops are possible.
    - ✓ Loopy path for map navigation, examples:
      - Arad → Sibiu → Arad
      - Arad → Zerind → Oradea → Sibiu → Arad



DF-search; evaluation

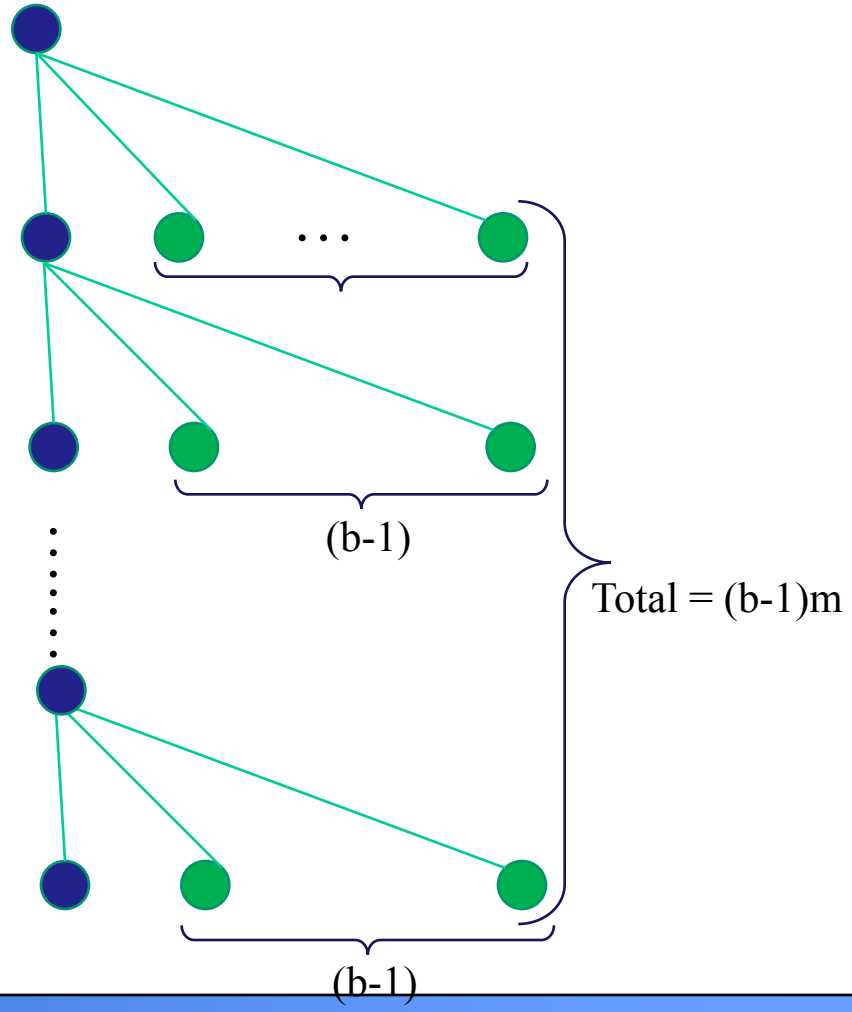
- Completeness;
  - NO unless search space is finite.
- Time complexity: $O(b^{m+1})$
  - Terrible if *m* is much larger than *d* (depth of optimal solution)
  - But if many solutions, then faster than BF-search



$$\text{Total} = b^{m+1}$$

DF-search; evaluation

- Completeness;
  - NO unless search space is finite.
- Time complexity: $O(b^{m+1})$
- Space complexity: O(bm)
  - Backtracking search uses even less memory
    - ✓ One successor instead of all *b*.



DF-search; evaluation

- Completeness;
  - NO unless search space is finite.
- Time complexity: $O(b^m)$
- Space complexity: O(bm)
- Optimality: No
  - Same issues as completeness
  - Assume node J and C contain goal states

BF-search, UC-Search, and DF-search

Criteria	BF-Search	DF-Search	UC-Search
Completeness	Yes	No	Yes
Time complexity	$O(b^{d+1})$	$O(b^{m})$	$O(b^{C^*/\varepsilon})$
Space complexity	$O(b^{d+1})$	O(bm)	$O(b^{C^*/\varepsilon})$
Optimality	Yes	No	Yes

b: branch factor; d: depth of solution; m: current depth explored
C*: cost of the solution; $\boldsymbol{\varepsilon}$: min step cost

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.
- 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*.
    - ✓ Goals is found at depth *d*, the depth of the shallowest goal-node.
  - Often used in combination with DF-search
- Combines benefits of DF- en 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*

ID-search, example

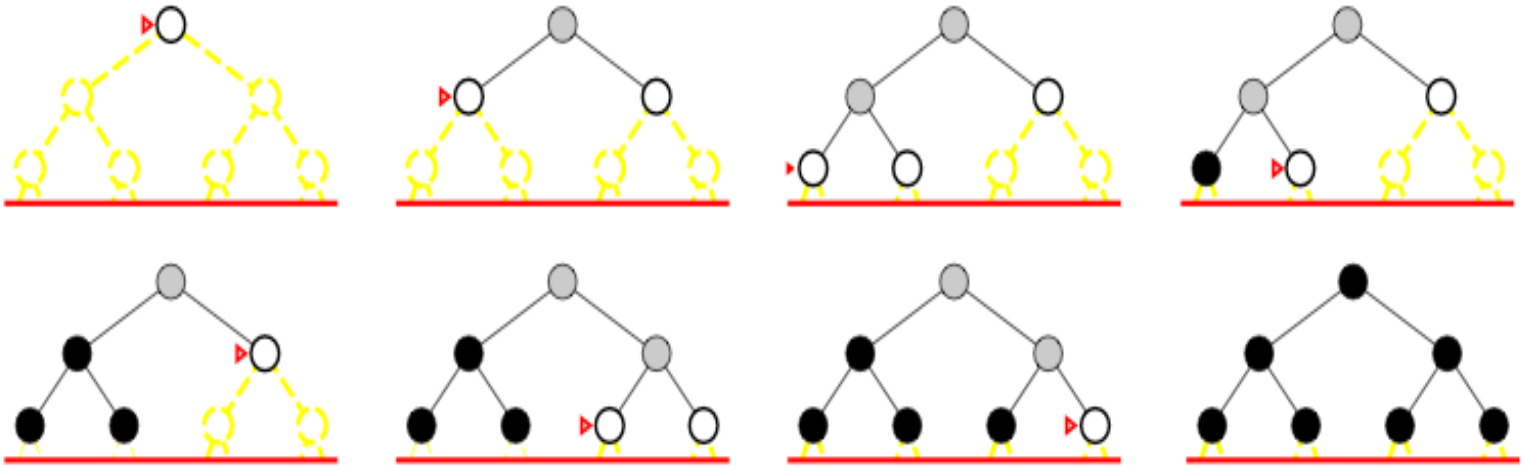
- Limit=0

ID-search, example

- Limit=1

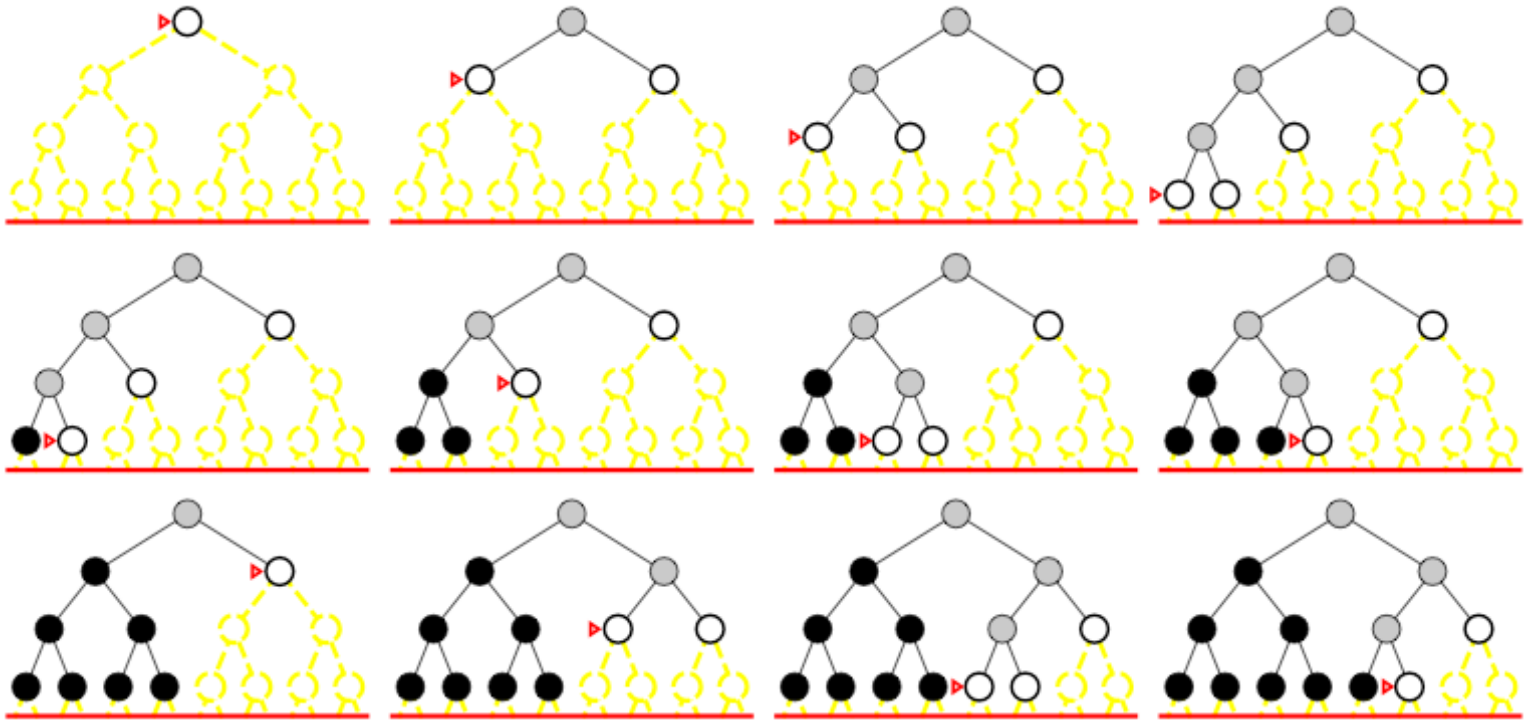
ID-search, example

- Limit=2

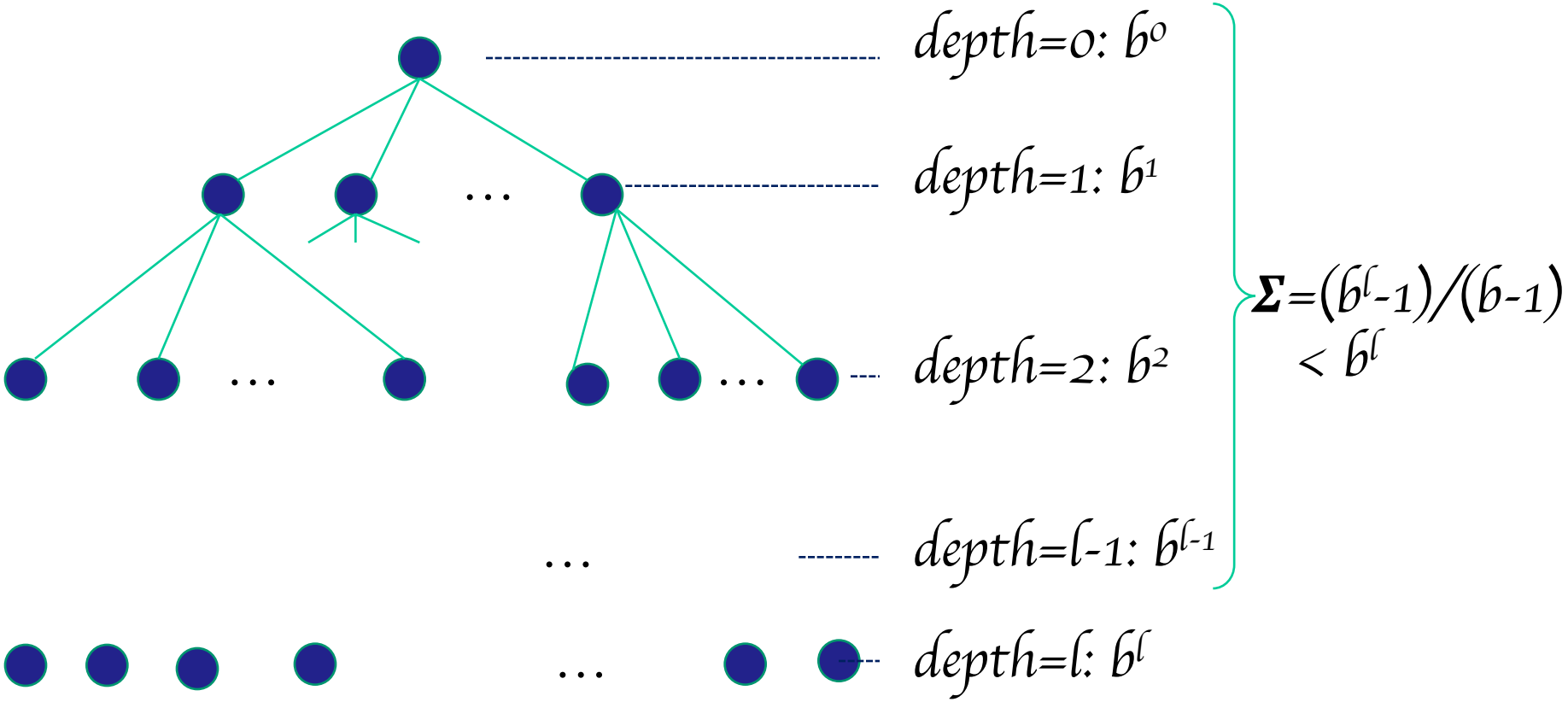


ID-search, example

- Limit=3



ID-search, property



#states explored by previous limit $(b^l-1)/(b-1)$ < #new states added (b^l)

ID search, evaluation

- Completeness:
  - YES (no infinite paths)

ID search, evaluation

- Completeness:
  - YES (no infinite paths)
- Time complexity:
  - Algorithm seems costly due to repeated generation of certain states.
  - Node generation: $O(b^d)$
    - ✓ level d: once
    - ✓ level d-1: 2
    - ✓ level d-2: 3
    - ✓ …
    - ✓ level 2: d-1
    - ✓ level 1: d

$$N(IDS) = (d)b + (d-1)b^2 + \ldots + (1)b^d$$

$$N(BFS) = b + b^2 + \ldots + b^d + (b^{d+1} - b)$$

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

$$N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100$$

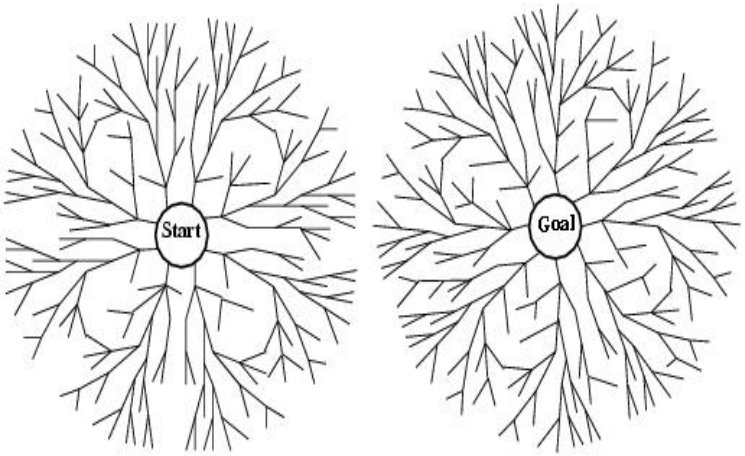
ID search, evaluation

- Completeness:
  - YES (no infinite paths)
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$
  - Cfr. depth-first search

ID search, evaluation

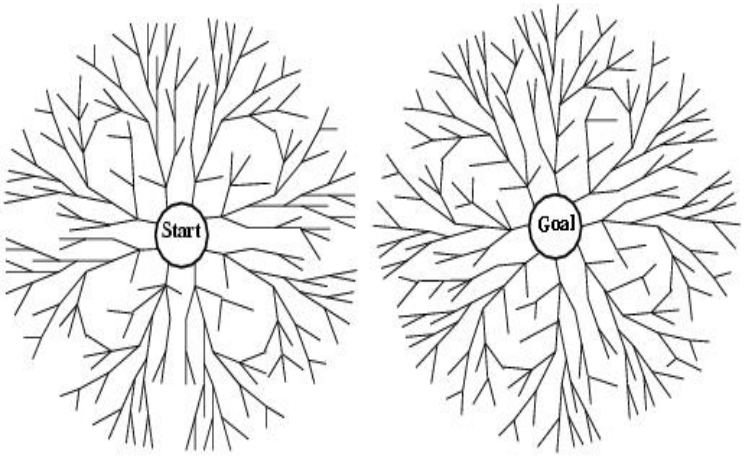
- Completeness:
  - YES (no infinite paths)
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$
- Optimality:
  - YES if step cost is 1.
  - Can be extended to iterative lengthening search
    - ✓ Same idea as uniform-cost search
    - ✓ Increases overhead.

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.
- Space complexity is the most significant weakness.
- Complete and optimal if both searches are BF.

How to search backwards?



- The predecessor of each node should be efficiently computable.
  - When actions are easily reversible.

Summary of algorithms

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

End video

Search with partial information

- Previous assumption:
  - Environment is fully observable
  - Environment is deterministic
  - Agent knows the effects of its actions

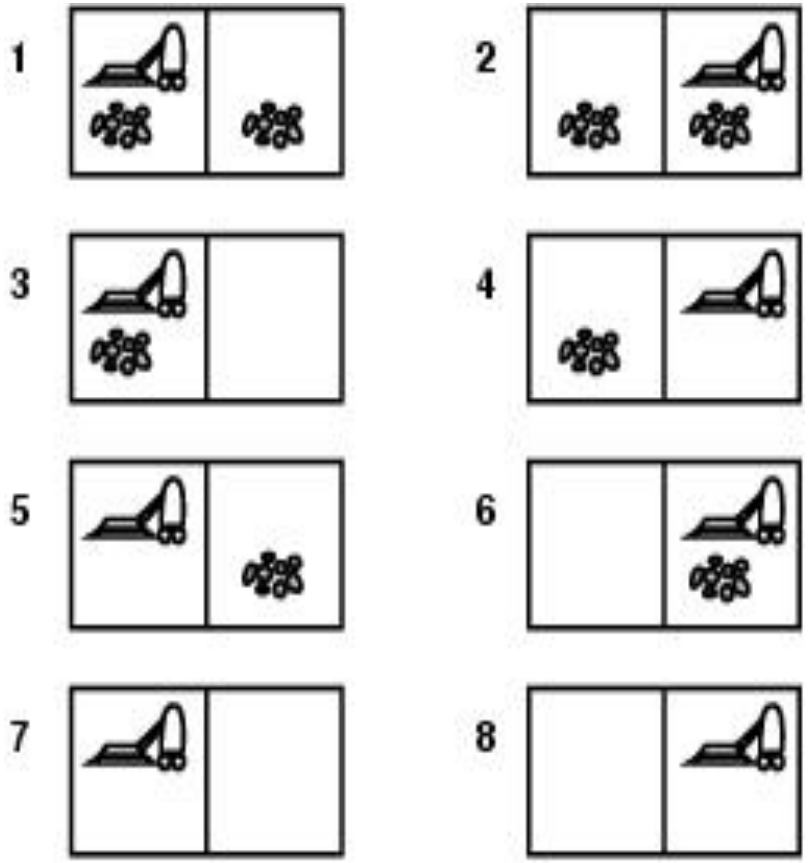
What if knowledge of states or actions is incomplete?

Search with partial information

- Partial knowledge of states and actions:
  - *sensorless or conformant problem*
    - ✓ Agent may have no idea where it is; solution (if any) is a sequence.
  - *contingency problem*
    - ✓ Percepts provide *new* information about current state; solution is a tree or policy; often interleave search and execution.
    - ✓ If uncertainty is caused by actions of another agent: *adversarial problem*
  - *exploration problem*
    - ✓ When states and actions of the environment are unknown.

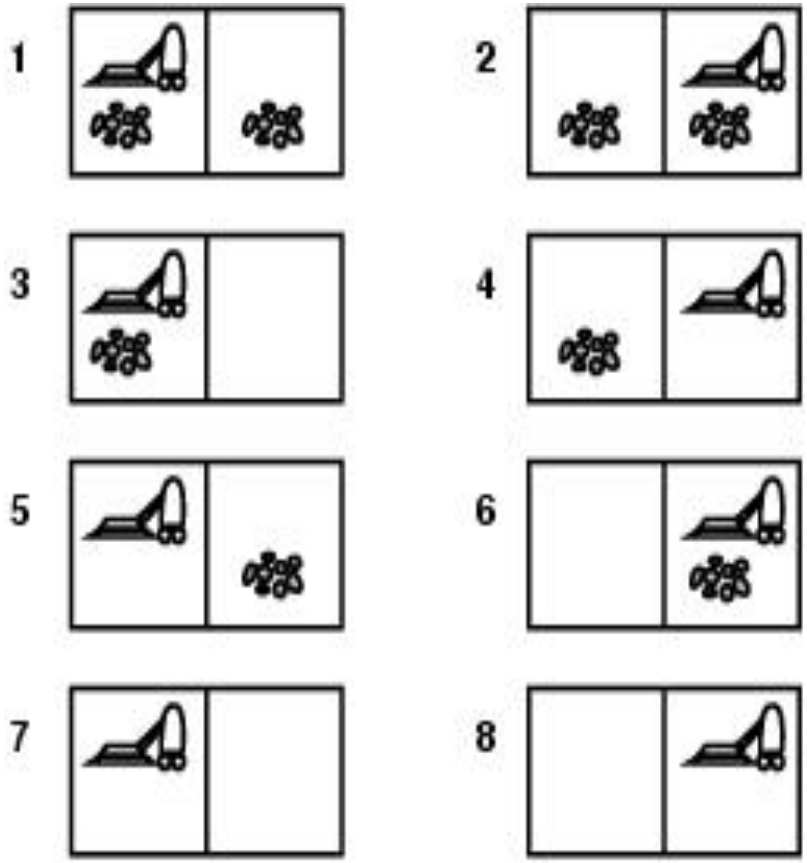
Conformant problems

- start in {1,2,3,4,5,6,7,8}
  e.g Right goes to {2,4,6,8}. Solution??
  - *[Right, Suck, Left,Suck]*
- *When the world is not fully observable: reason about a set of states that migth be reached*
  =belief state

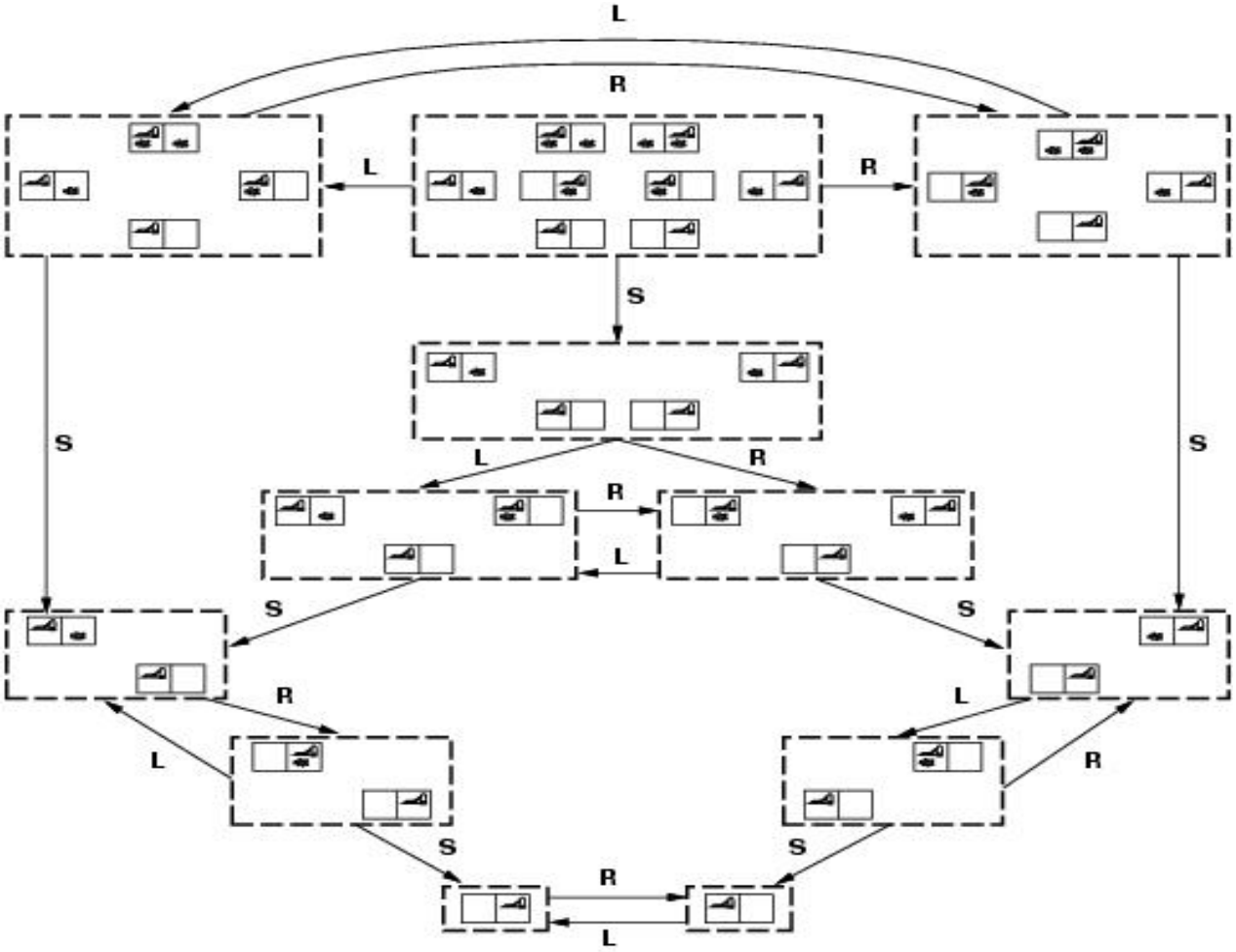


Conformant problems

- Search space of belief states
- Solution = belief state with all members goal states.
- If S states then 2^S belief states.
- Murphy's law:
  - *Suck can dirty a clear square.*



Belief state of vacuum-world



Contingency problems

- Contingency, start in {1,3}.
- Murphy's law, Suck *can* dirty a clean carpet.
- Local sensing: dirt, location only.
  - Percept = [L,Dirty] ={1,3}
  - [Suck] = {5,7}
  - [Right] ={6,8}
  - [Suck] in {6}={8} (Success)
  - BUT [Suck] in {8} = failure
- Solution??
  - Belief-state: no fixed action sequence guarantees solution
- Relax requirement:
  - [*Suck*, *Right*, if *[R,dirty]* then *Suck*]
  - Select actions based on contingencies arising during execution.