Classical Search Algorithms

Dr. Fayyaz ul Amir Afsar Minhas
PIEAS Biomedical Informatics Research Lab
Department of Computer and Information Sciences
Pakistan Institute of Engineering & Applied Sciences
PO Nilore, Islamabad, Pakistan
http://faculty.pieas.edu.pk/fayyaz/

Outline

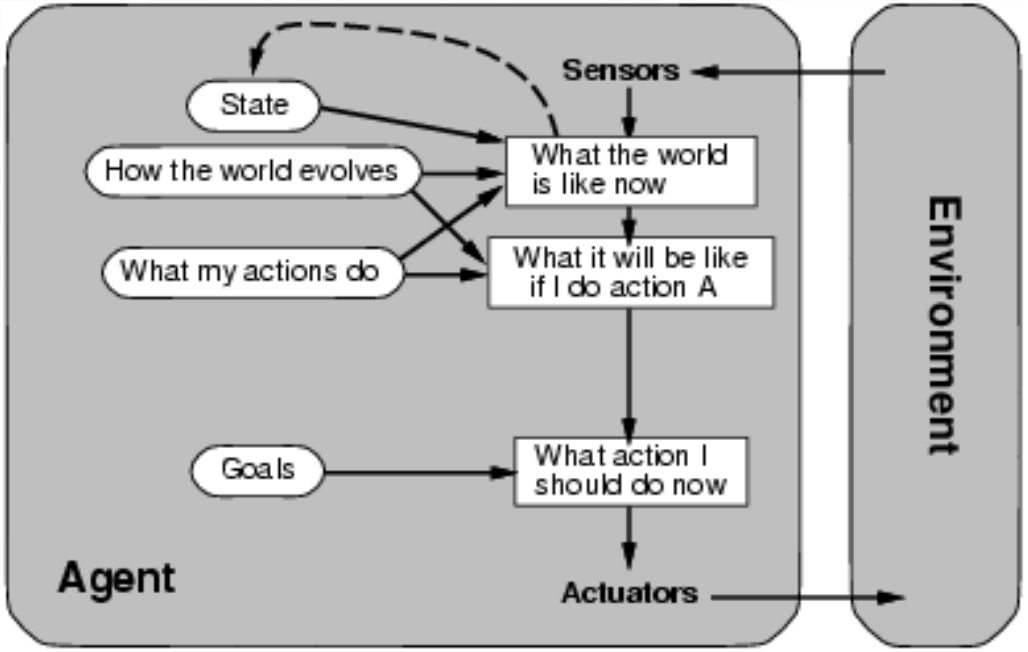
- Role of search in problem solving
- Problem Solving Agents
- Defining a Problem
  - Examples (Toy and Real World)
- Concepts of Search Algorithms (Tree Search)
- Performance Metrics for Search Algorithms
- Uninformed Search Strategies
- Searching with Partial Information
- Contingency Problems

Role of Search in Problem Solving

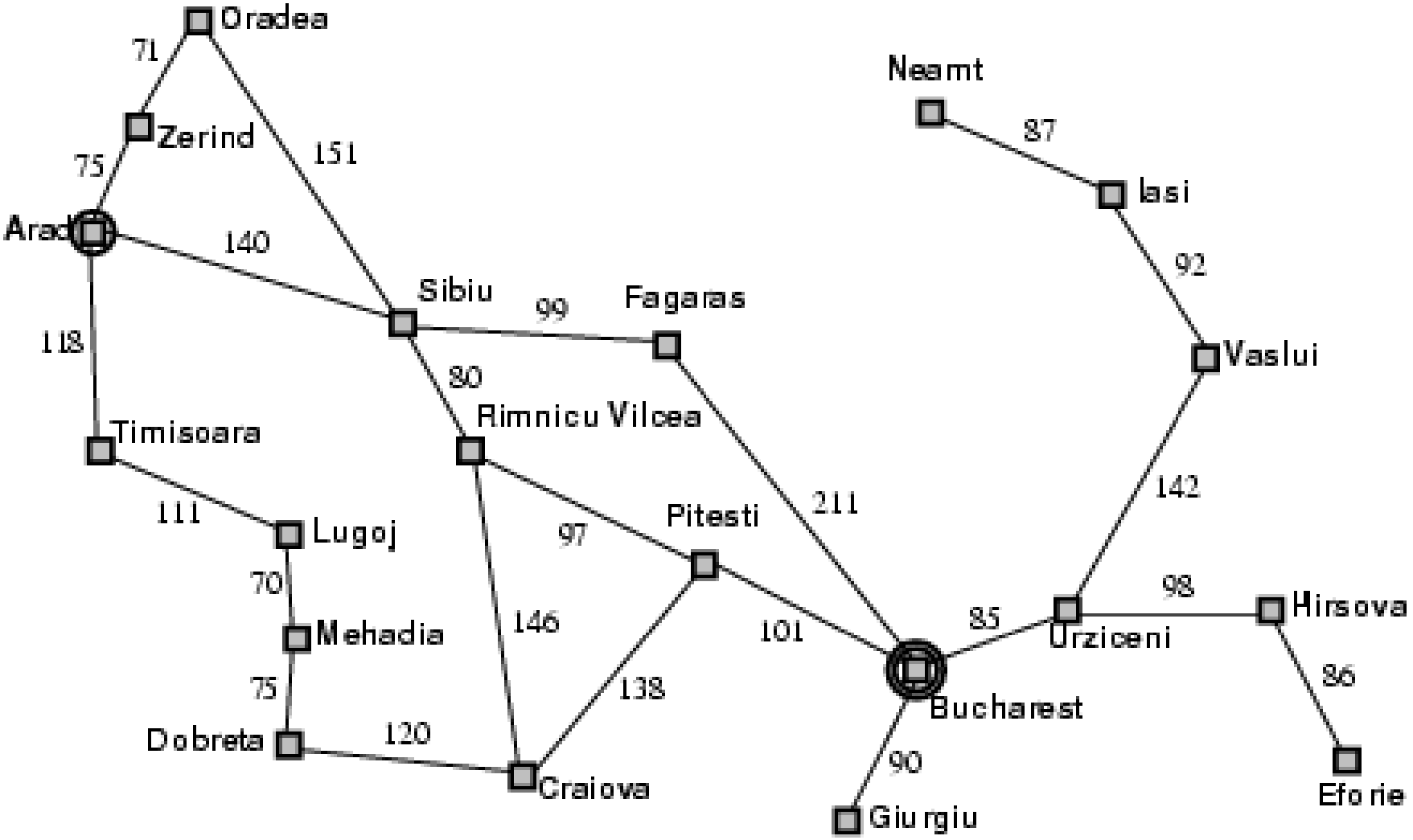
- Search is critical to almost all types of problem solving approaches
- Example
  - Searching for a series of moves which would lead to a win in a chess game
  - Searching for an optimal trajectory to launch a missile or park a truck
  - Searching for a web page with required contents
  - Searching for a state in which both the squares are clean in our vacuum cleaner example

Problem Solving Agents

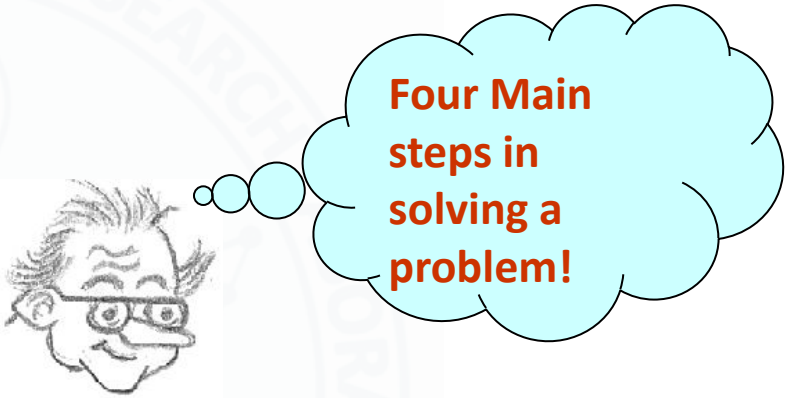
- Problem Solving Agents are goal based agents and they decide what to do by finding a sequence of actions that lead to desirable states



Example Problem



Example Problem…



- Goal Formulation
  - Reach Bucharest
- Problem Formulation
  - *States:*
    - Different Cities
  - *Actions:*
    - Move between cities
- Once the goal and the problem have been formulated we use some search methodology for finding a solution which would achieve our goal for the given problem
  - A solution is a sequence of states (cities) e.g. Arad, Sibiu, Fagaras, Bucharest
- The solution obtained is then executed

Problem Solving Agents

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

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

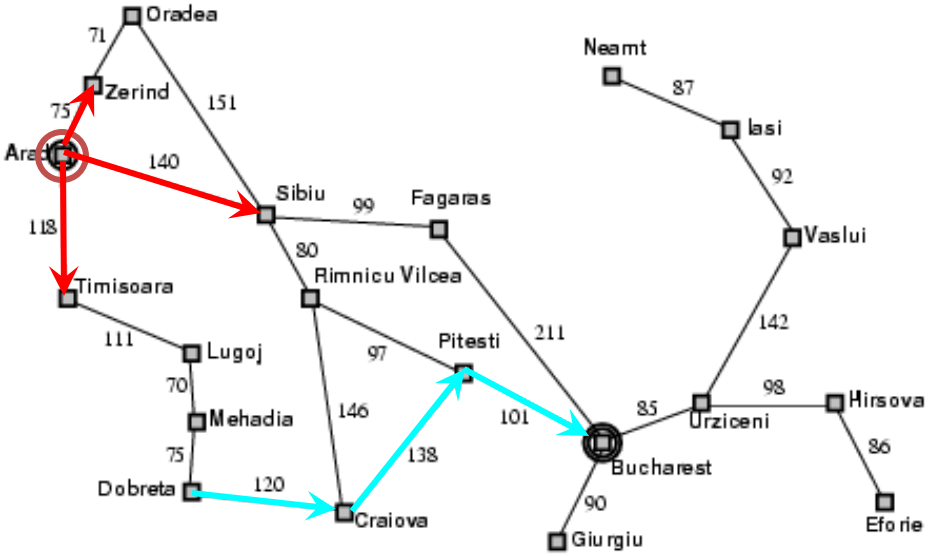
Initial State is considered to be known: *Observability*

Solution is a finite set of states: *Discrete Env.*

Actions executed without considering percepts: *Static and Deterministic Env.*

Defining a Problem

- A problem is defined by 4 components
  - Initial State:
    - Example: In(Arad)
  - Action Description: Using a Successor Function, S(x) which returns an <action, successor> pair
    - Example:

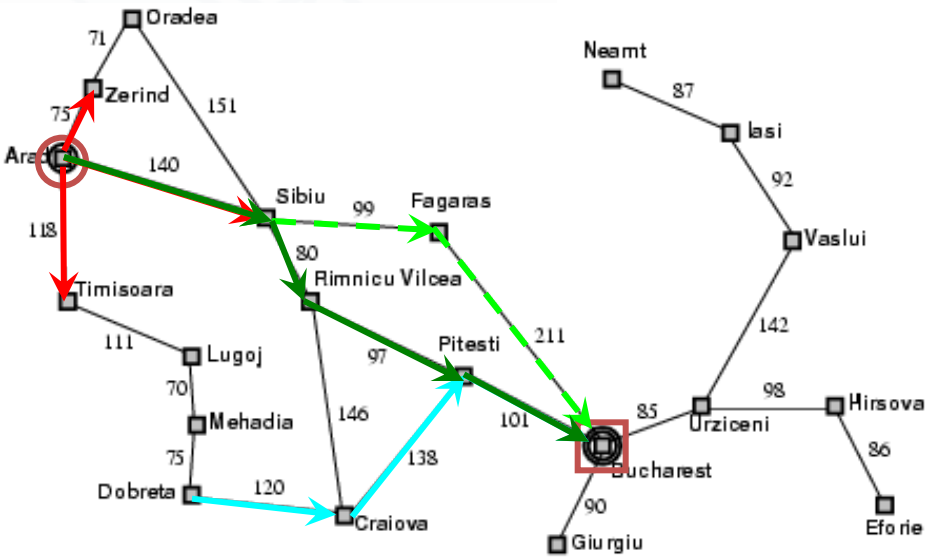


S(In(Arad))={<Go(Sibiu),In(Sibiu)>, <Go(Timisoara),In(Timisoara)>, <Go(Zerind),In(Zerind)>}

- **State Space: Set of All states reachable from the initial state and it forms a graph in which the nodes are states and the arcs are actions**
- **Path: A path in the state space is a sequence of states connected by a sequence of actions**

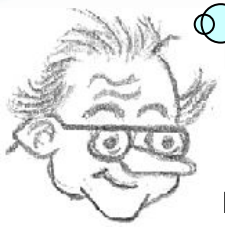
Defining a Problem…

- A problem is defined by 4 components…
  - Goal Test
    - Example: {In(Bucharest)}
  - Path Cost Function
    - Assigns a numeric cost to each path
    - Step Cost: Cost of moving from state to state y via action a, C(x,a,y)



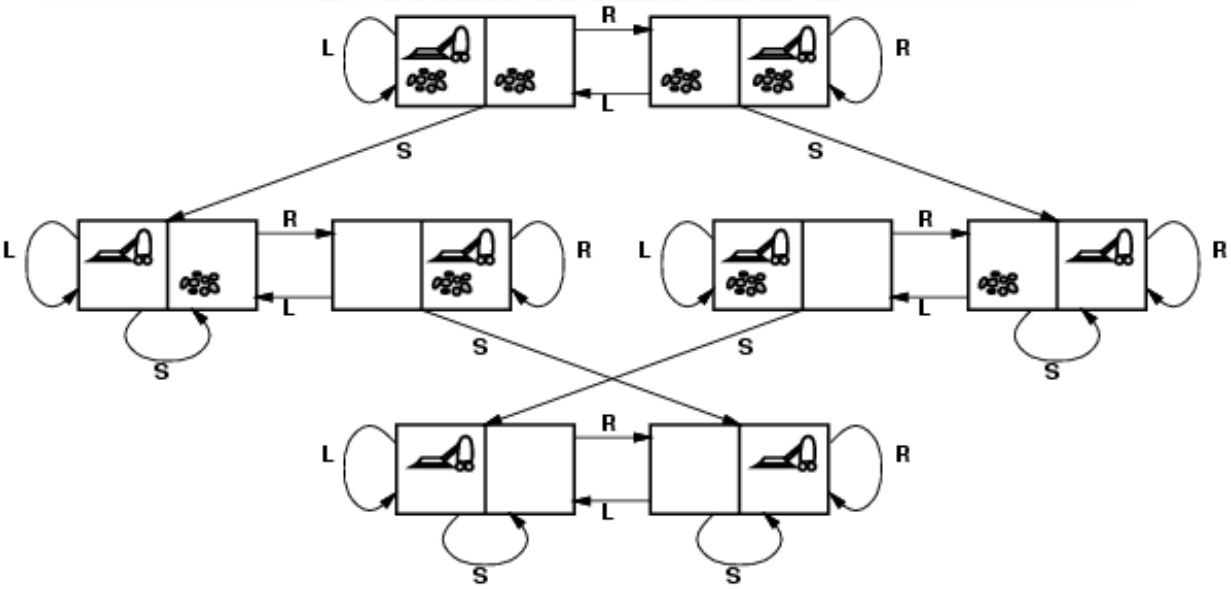
120+138+101 = 359

- **Solution: A path from the initial state to the goal state**
- **Optimal Solution: Has lowest path cost amongst all solutions**



Problems are represented by an abstraction!

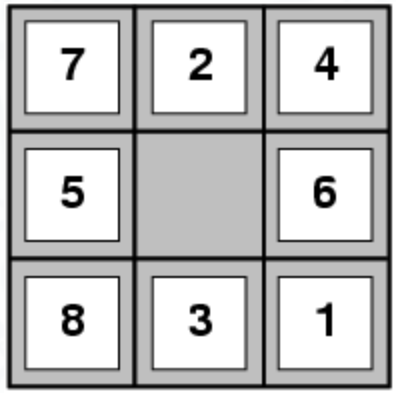
Example Problems



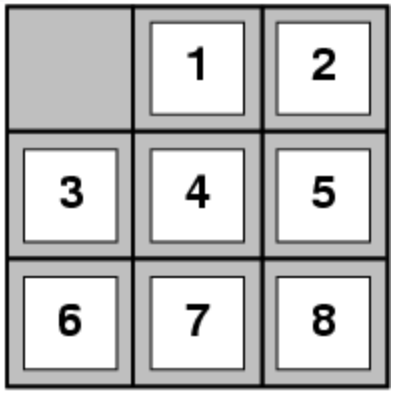
- Vacuum Cleaner
- States
  - two locations with or without dirt: 2 x 2^2=8 states.
- Initial state
  - Any state can be initial
- Actions
  - {*Left*, *Right*, *Suck*}
- Goal test
  - Check whether squares are clean.
- Path cost
  - Number of actions to reach goal.

Example Problems

- 8-Puzzle
  - States
    - locations of tiles
  - Actions
    - move blank left, right, up, down
  - goal test
    - goal state (given)
  - path cost
    - 1 per move
    -



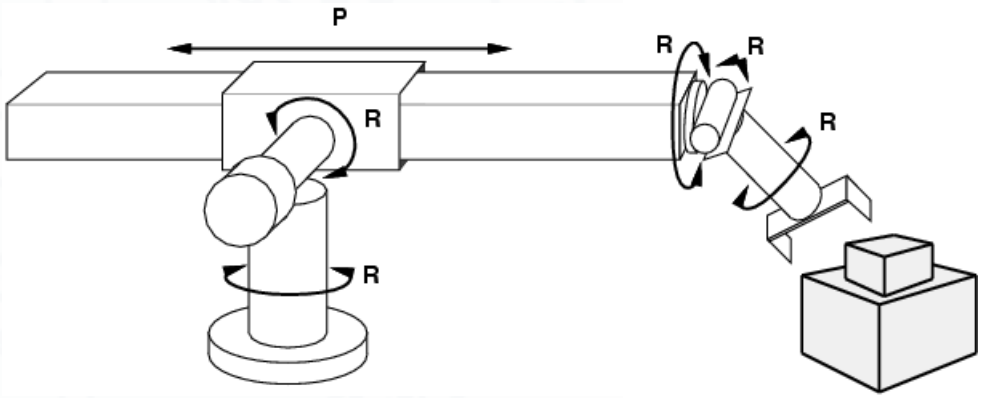
Start State



Goal State

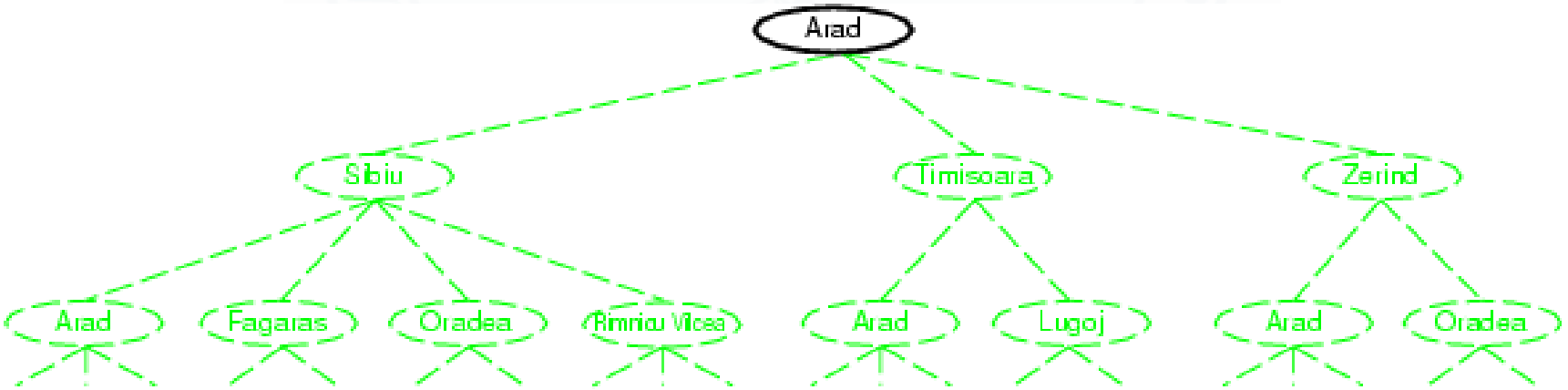
Example Problems

- Robotic Manipulator
  - 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
    -



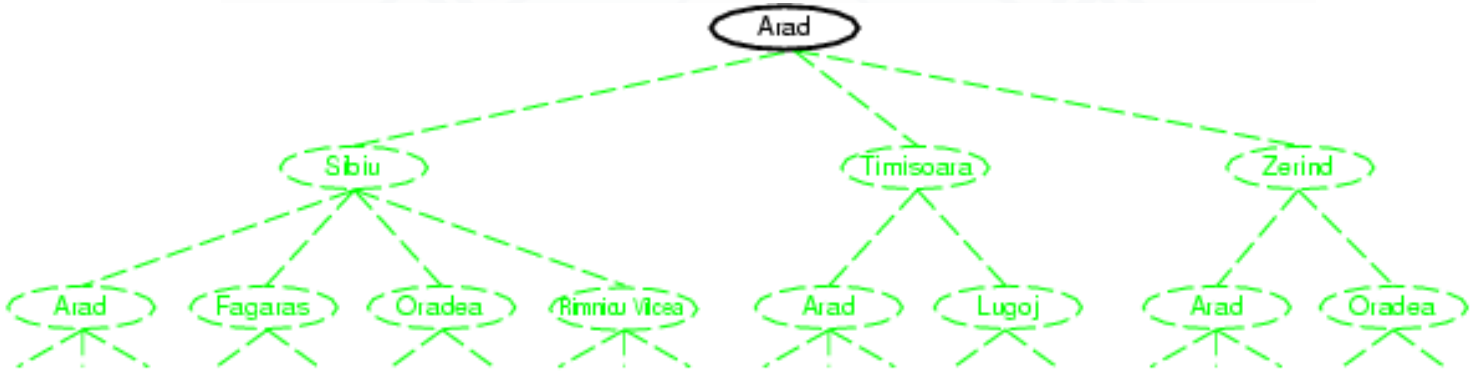
Searching for Solutions

- How do we find solutions to these problems?
  - Search the state space
- Basic Search Method
  - Tree Search
    - 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) and requires detection of repeated states and avoiding them

Tree-Search



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***

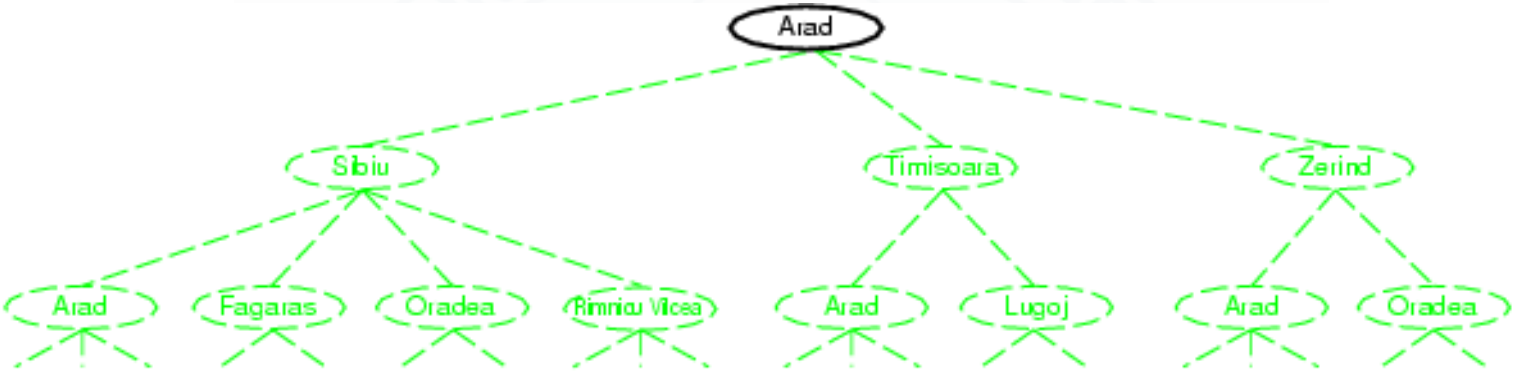
choose leaf node for expansion according to *strategy*

if **node contains goal state** then return ***solution***

else **expand the node and add resulting nodes to the search tree**

enddo

Tree-Search…



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***

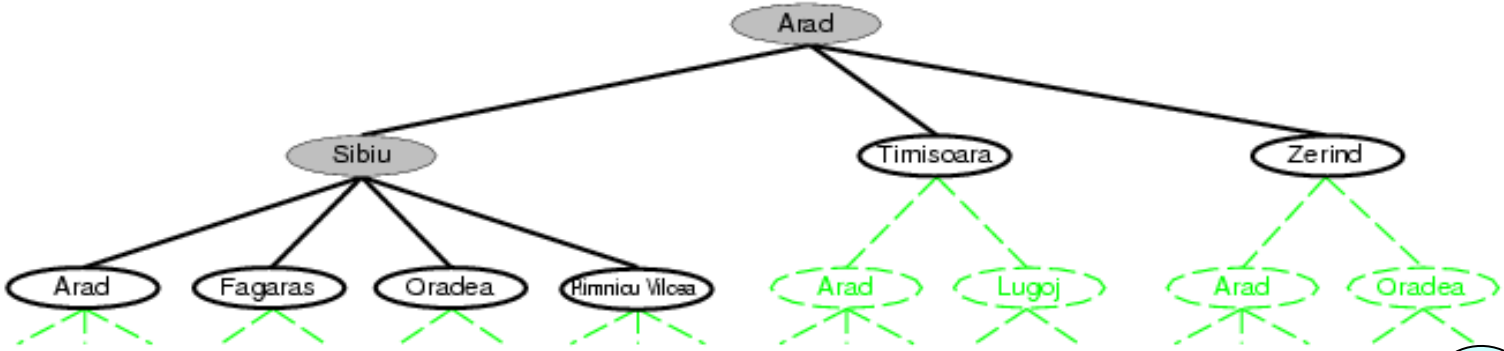
choose leaf node for expansion according to *strategy*

if **node contains goal state** then return ***solution***

else **expand the node and add resulting nodes to the search tree**

enddo

Tree-Search…



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***

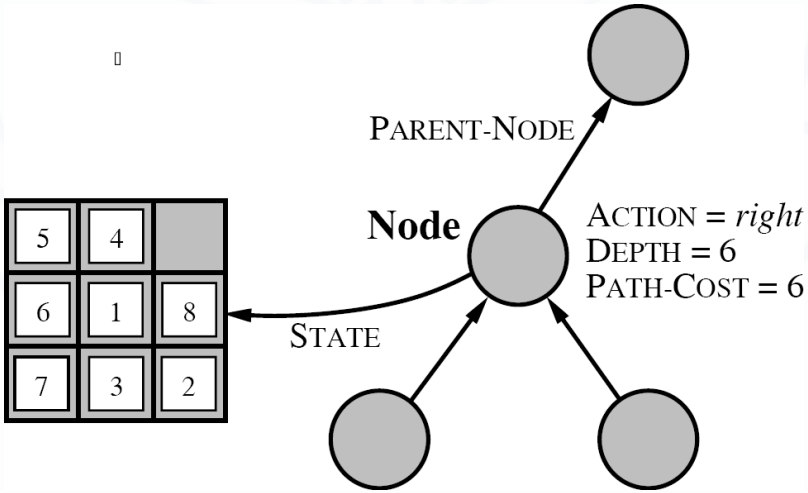
else **expand the node and add resulting nodes to the search tree**

enddo

Strategy Determines Search Process!!

State vs. Node

- A *state* is a (representation of) a physical configuration
- A *node* is a data structure belonging to a search tree
  - *node= <state, parent-node, action, path-cost, depth>*



Tree Search Algorithm

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)

function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

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 shallowest goal node
  - *m*: maximum depth of the state space (may be ∞)
  -

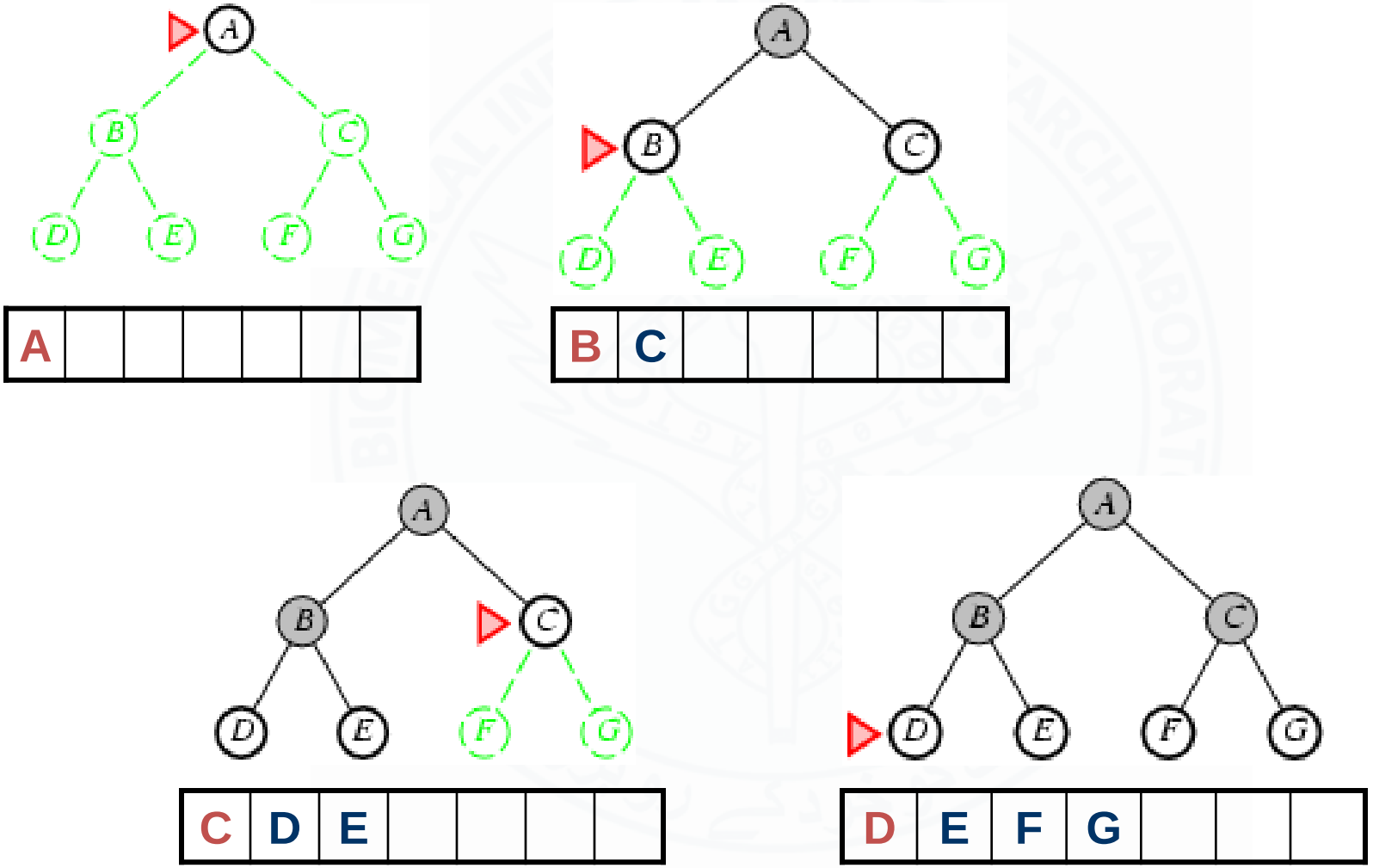
Uninformed Search Strategies

- These techniques utilize no more information than that provided in the problem definition
  - When strategies can determine whether one non-goal state is better than another in reaching a goal → *informed or heuristic search*.
- Uninformed or blind search techniques can only generate successors and distinguish a goal state from a non goal state
- Breadth First Search
- Uniform Cost Search
- Depth First Search
- Depth Limited Search
- Iterative Deepening Search

Breadth First Search

- Expand Shallowest Unexpanded Node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end
  -

BFS in Action



Breadth First Search…

- Completeness
  - If the branching factor *'b'* is finite then BFS will find the solution
- Optimality
  - BFS always finds the shallowest goal node first which may or may not be the optimal one
  - BFS is optimal when the path cost is a non-decreasing function of the depth of the node
    - When all actions have the same cost

Breadth First Search…

- 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 number of nodes generated is:

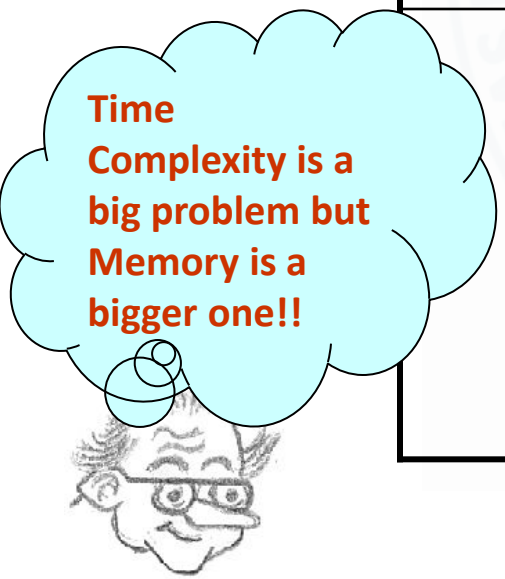
$$b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

- Space Complexity
  - Each node generated must remain in memory because it is either part of the fringe or is an ancestor of a fringe node

Breadth First Search…

- Consider a problem with a branching factor of b=10 being solved using BFS on a computer with a capacity to solve 10K nodes per second with each node requiring 1KB of main memory

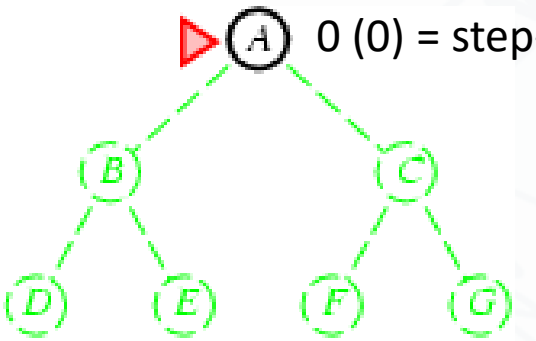
DEPTH	NODES	TIME	MEMORY
2	1100	0.11 seconds	1 megabyte
4	111100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabytes
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3523 years	1 exabyte



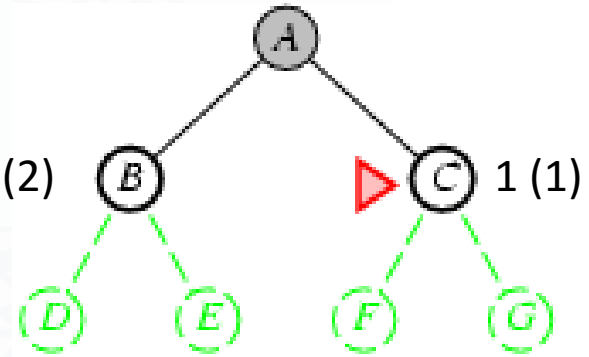
Uniform Cost Search

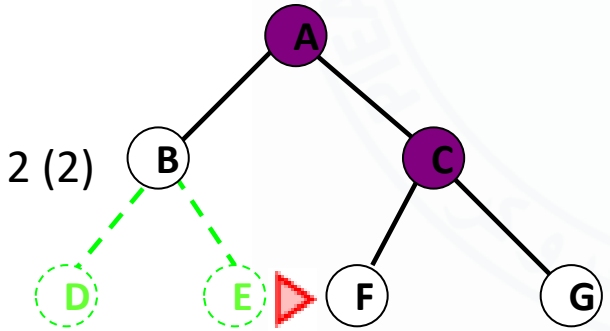
- BFS is optimal only when the path costs is a non decreasing function of depth
- UCS is a derivative of BFS and it expands the node with the lowest path cost first
- Implemented by having the fringe stored as a priority queue ordered by path cost
- If the step cost is same for all steps then BFS and UCS are equivalent

UCS in Action

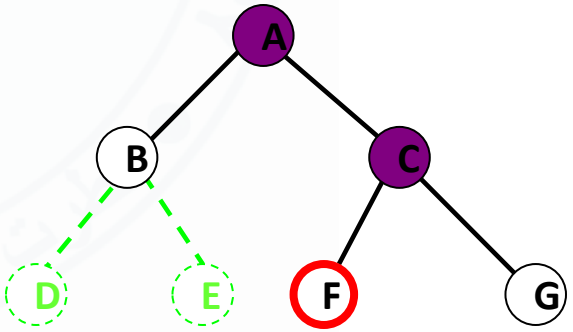


0 (0) = step-cost (Path-Cost)



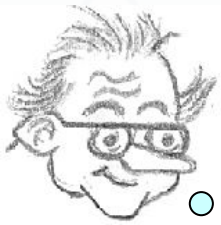


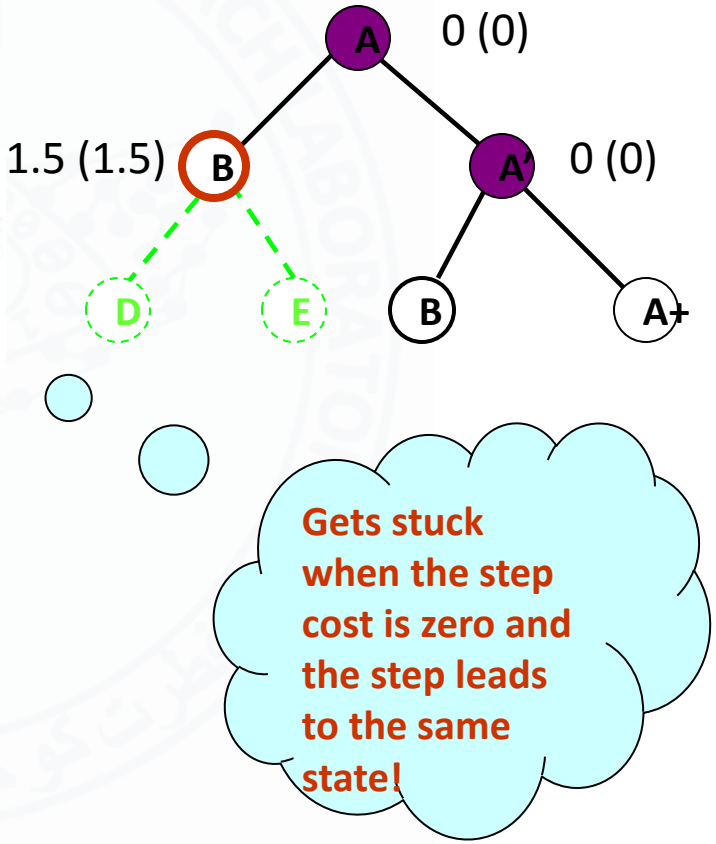
0.5 (1.5) 2 (3)



Uniform Cost Search

- Completeness
  - Every step must have a positive cost $\epsilon > 0$
- Optimality
  - When UCS is complete it is opti
    too
- Space & Time Complexity
  - As UCS is guided by path costs instead of depth therefore its time & space complexity cannot be characterized in terms of b and d
  - If C* is the optimal solution path cost and ϵ is the minimum path cost then the worst case complexity is $O(b^{C^*/\epsilon})$ which can be much larger than $O(b^d)$

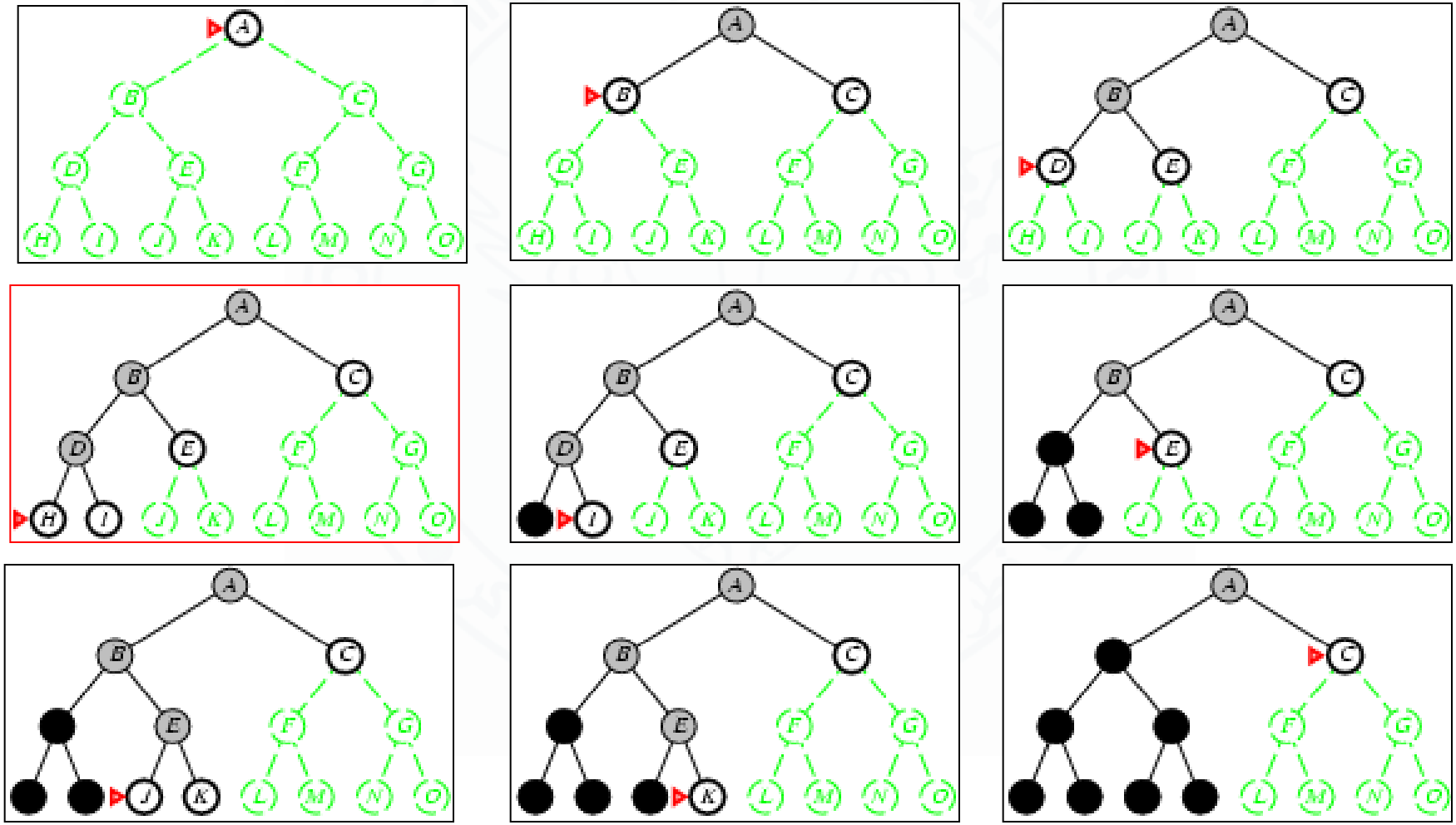




Depth First Search

- Expand the deepest node first
- Implemented by representing the Fringe as a LIFO Queue (Stack)

DFS in Action

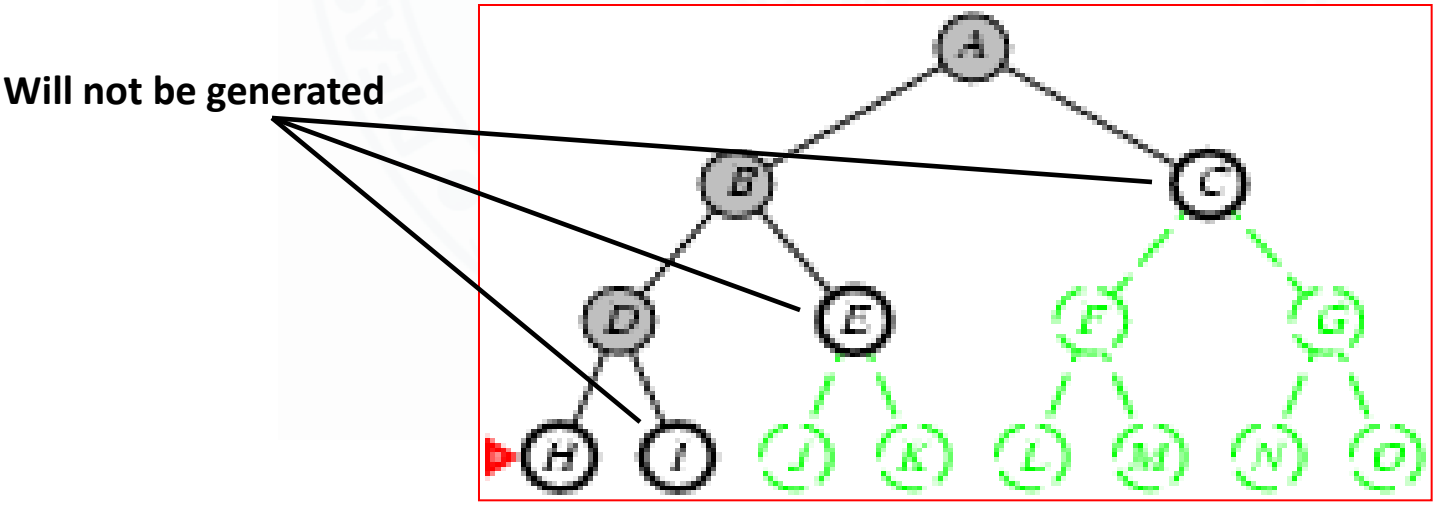


Depth First Search

- Completeness
  - Only if the search space is finite and no loops are possible
- Optimality
  - The first solution found may not be optimal therefore DFS is NOT optimal
- Memory
  - It needs to store only a single path from root to leaf, along with the remaining
  - Once all the descendants of a node have been explored then that node can be removed
  - If the branching factor is b and the maximum depth is m then the number of nodes required to be stored is bm+1, thus making the worst case space complexity O(bm+1)
- Time
  - In the worst case DFS can generate all the nodes so its time complexity is $O(b^m)$
  - m (max. depth of the search space) can be much larger than d so its time complexity can be greater than BFS

Depth First Search

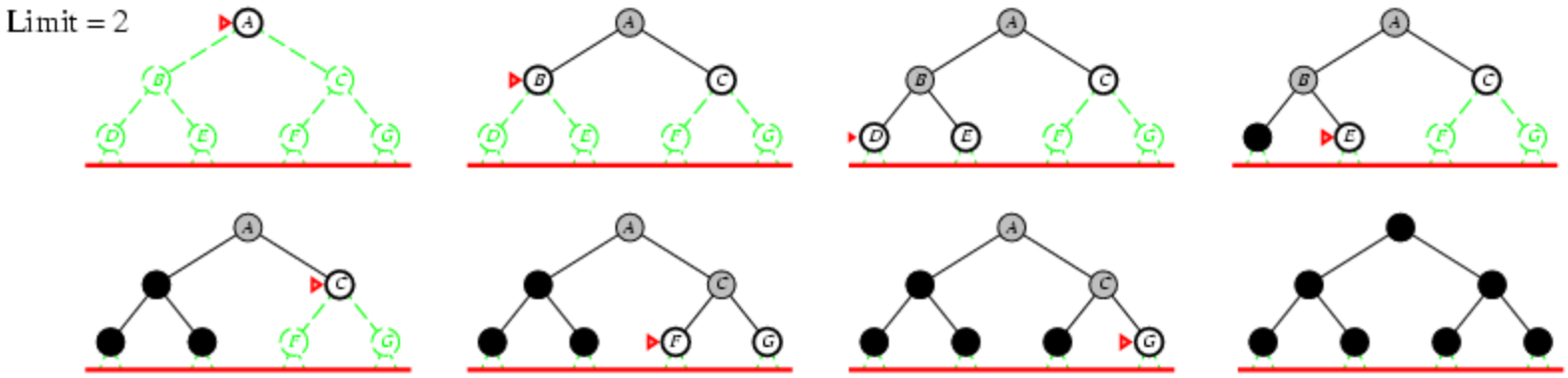
- Backtracking DFS
  - Generate only one successor and remember which successors have been generated
  - Space Complexity reduced to O(m)



Depth Limited Search

- Depth First Search with a depth Limit at l
- Solves the infinite-path problem.
- Completeness
  - If $l < d$ then incompleteness results.
- Optimality
  - If $l > d$ then not optimal.
- Time complexity
  - $O(b^l)$
- Space complexity
  - $O(bl)$

DLS in Action



Depth Limited Search

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Iterative Deepening Search

- Keep on increasing the depth limit for Depth Limited Search until the solution is found
- IDS is the preferred search method when there is a large search space and the depth of the solution is not known

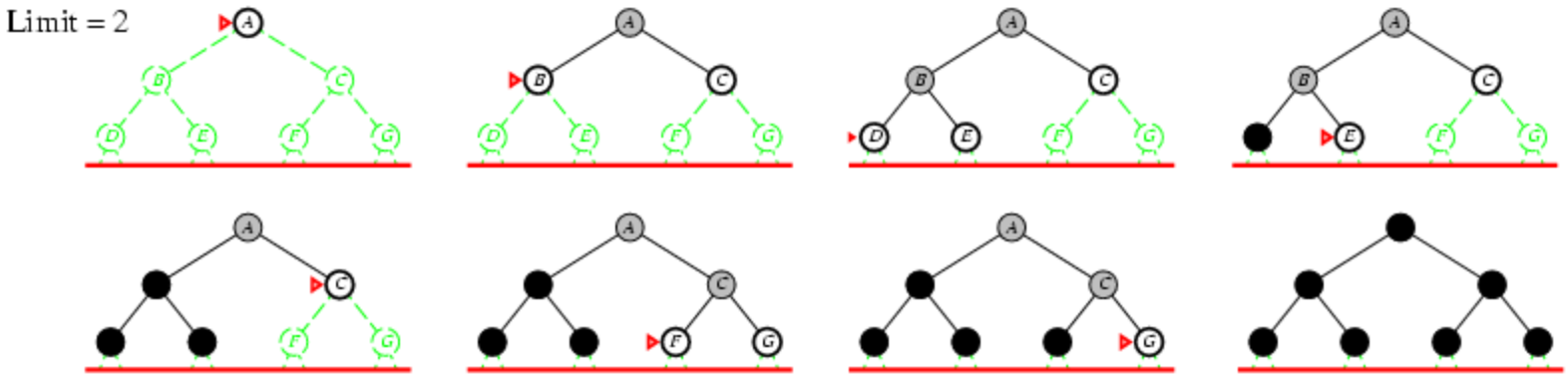
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

Iterative Deepening Search in Action

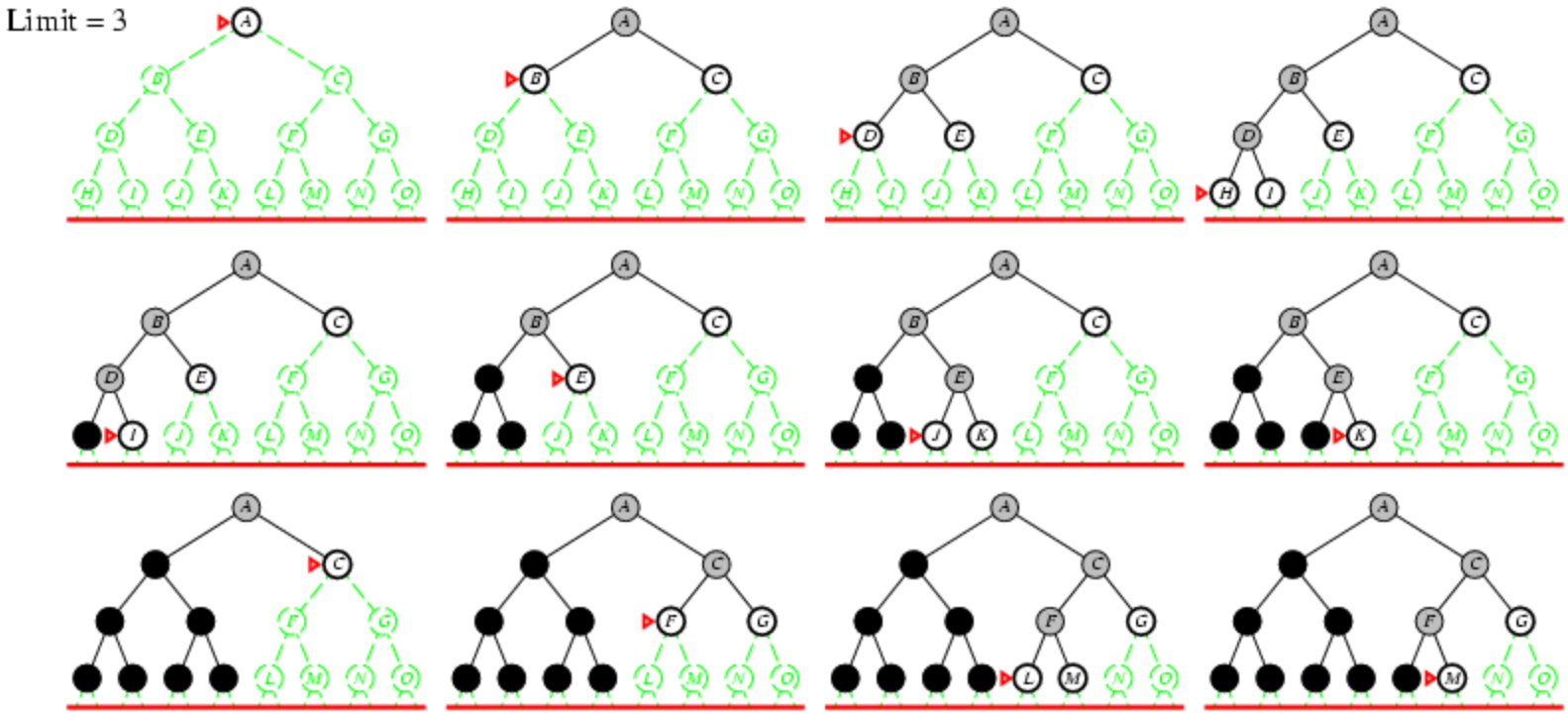
Limit = 0

Iterative Deepening Search in Action…

Iterative Deepening Search in Action…



Iterative Deepening Search in Action…



Iterative Deepening Search…

- Complete
- Optimal
  - Only when all step costs are the same
- Memory Requirements
  - O(bd)
- Time complexity:
  - Algorithm seems costly due to repeated generation of certain states.
  - Node generation:
    - level d: once
    - level d-1: 2
    - level d-2: 3
    - …
    - level 2: d-1
    - level 1: d
  - Num. Comparison for b=10 and d=5 solution at far right

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

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

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

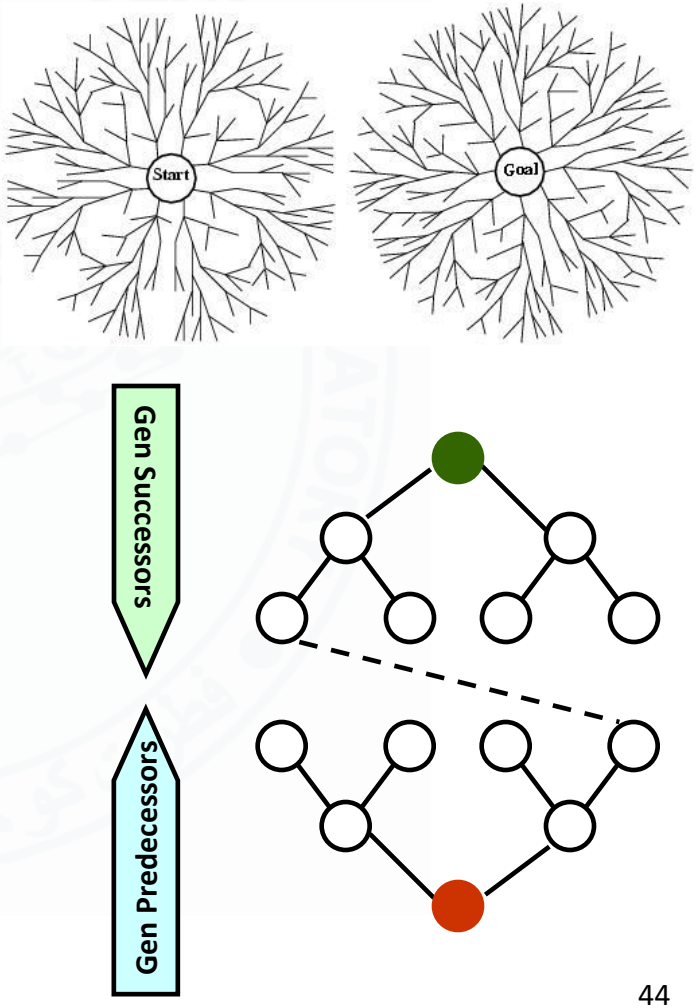
$$N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100$$

Iterative Lengthening Search

- Use increasing path-cost limits instead of increasing depth limits
- Analogous to UCS but has low memory requirements because of its DF nature

Bi-directional Search

- Two simultaneous searches from start an goal.
  - Motivation:
    - $b^{d/2} + b^{d/2} < b^d$
  - Check whether the node belongs to the other fringe before expansion.
- Space complexity $O(b^{d/2})$ is the most significant weakness because one of the two trees must be kept in memory
- Complete and optimal if both searches are BF.

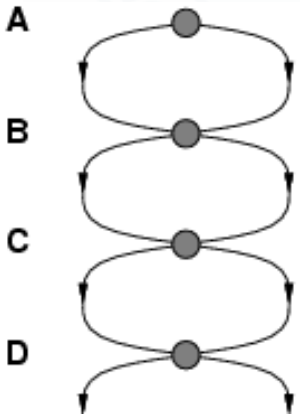


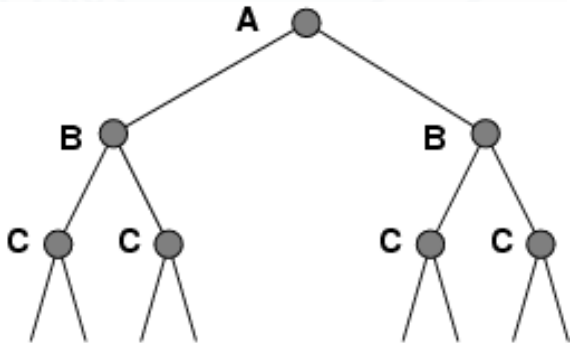
Summary

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Avoiding Repeated States

- For some problems repeated states are unavoidable
  - 8-puzzle
  - Route Finding
- Repeated States can make a solvable problem, unsolvable and can make exponential problems out of linear ones





Avoiding Repeated States: Graph Search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set                    (Remembers Expanded States)
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

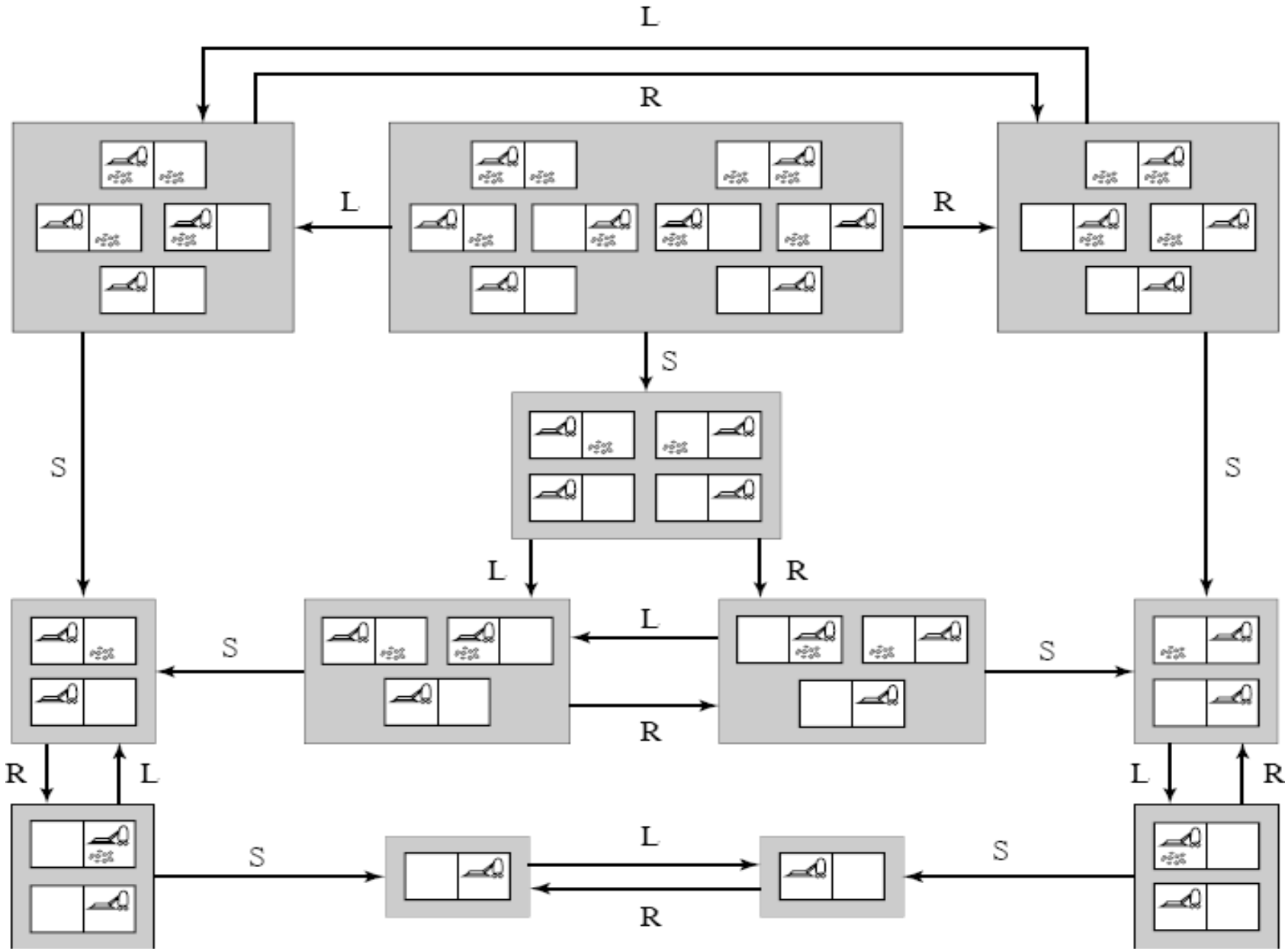
- Fringe can itself be implemented as a LIFO or a FIFO Queue making all types of previous algorithms applicable here
- Detecting the repeated states requires more memory but can reduce time complexity

Searching with Partial Information

- Our problem solving agent discussed previously could only solve problems in a deterministic, static and fully observable environment
- But most practical problems require us to go beyond these limitations
- There are 3 distinct problem types in the presence of partial information
  - Sensor-less (Conformant) Problems
    - If the agent has no sensors. It can be in one of several possible initial states and each action might therefore lead to one of several possible successor states
  - Contingency Problem
    - If the environment is partially observable or the actions are uncertain then the percepts provide new information after each action. Adversarial problems involve another agent.
  - Exploration Problem
    - When the states and actions are unknown and the agent must explore to know its environment

Sensor-less Problem

- When the world is not fully observable then the agent must reason about sets of states that it might get to
- Such a set is called a Belief State and it represents the agent's current belief about the possible physical state it might be in
- Consider a sensor-less vacuum cleaner agent in a deterministic environment…



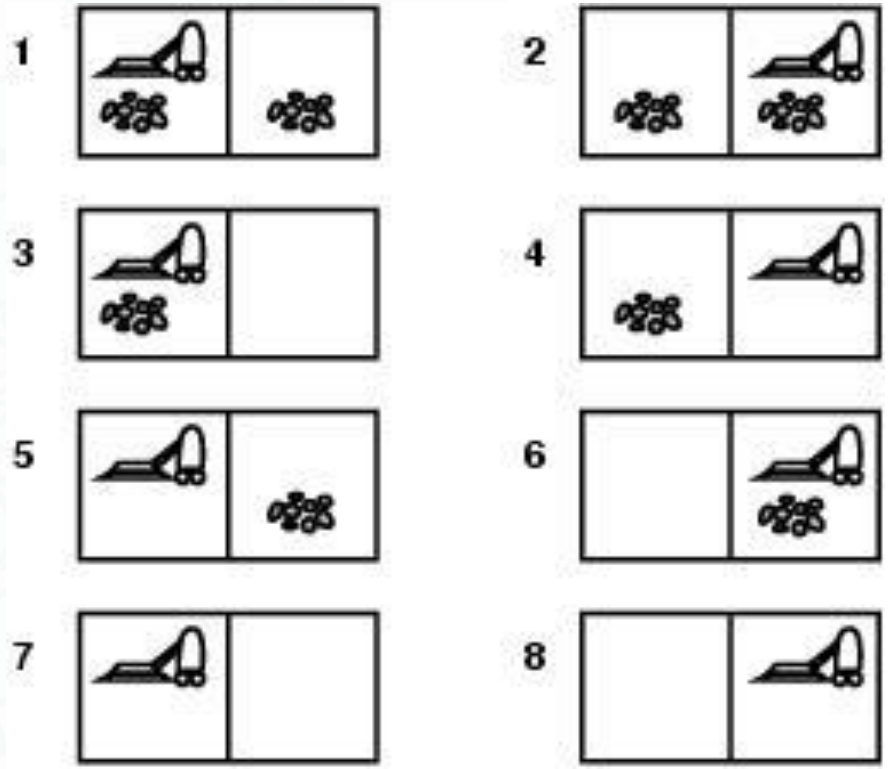
All Members of a Goal Belief State are goal states

Sensor-less Problem…

- The situation gets worse when the environment is non-deterministic
  - Suppose that the vacuum cleaner agent can, at times, deposit dirt in a clean floor when it executes ‘Suck’
  - Does such a problem have solution?

Contingency Problems

- In contingency problem, a new percept provides newer information after each action
- Consider a fully observable vacuum cleaner agent in a non-deterministic world in which 'Suck' can dirty a clean carpet
- start in {1,3} and execute sequence [S,R,S]
  - {1,3}: Percept = [L,Dirty]
  - {1,3} →[Suck] → {5,7}
  - {5,7} →[Right] →{6,8}
  - {6} → [Suck] → {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.



Tasks

- When will complexity of UCS be equal to that of BFS?
- How Can DFS be implemented as a recursion?
- What is the possible number of belief states in a problem with a total of S states? Explain your answer.
- Will a sensor-less vacuum cleaner agent that deposits dirt on a clean floor be able to achieve a clean floor? Explain and propose a possible way out of this problem with a faulty vacuum cleaner.
- Implement BFS, UCS, DFS, Backpropagation DFS, ULS, IDS, ILS while avoiding repeated states for solving the 8 queens problem. (To be done in C++ or Matlab and in Groups)

End of Lecture

It is the true nature of mankind to learn from mistakes, not from example.

Fred Hoyle (lived 1915), British astronomer, mathematician, and writer.
Into Deepest Space (1975).