SEARCHING FOR SOLUTIONS

- Many AI problems can be solved by **exploring the so called solution space**. It contains all possible sequences of actions that might be applied by an agent. Some of these sequences lead to a solution.
  - The agent examines alternative sequence of actions that lead to known states and chooses, then, the best one.
  - The process of trying this sequence is called **SEARCH**.
  - It is useful to think about the search process like building a search tree whose nodes are states and whose branches are operators/actions.

    The initial node is called root node

- A search algorithm takes as input a problem and returns a solution in the form of a sequence of actions.

- Once the solution is found, the suggested actions can be performed.
  - This is called **EXECUTION**.

SEARCHING FOR SOLUTIONS

Generate sequences of actions.

- ***Expansion:*** one starts from a state, and applying the operators (or successor function) will generate new states.
- ***Search strategy:*** at each step, choose which state to expand.
- ***Search tree:*** It represents the expansion of all states starting from the initial state (the root of the tree).
- The leaves of the tree represent either states to expand or solutions or dead-ends

SEARCH TREES

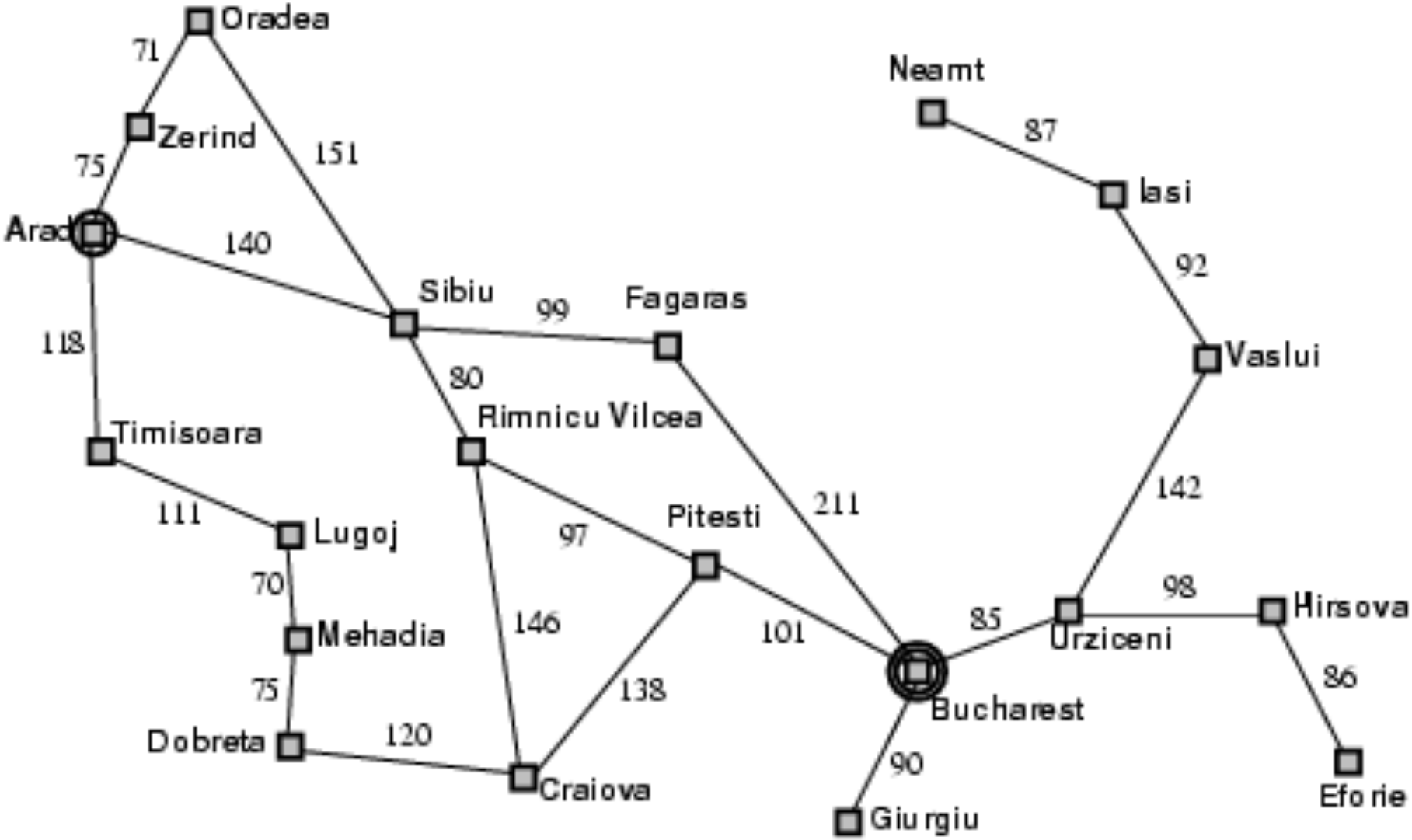
- Basic idea:
  - Off-line, simulated exploration obtained by expanding states that have been already explored.

```
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
```

EXAMPLE

- A holiday in Romania; currently we are in Arad.
- Flights landing in Bucharest tomorrow
- goal:
  - To be in Bucharest
- problem:
  - States: Being in a city
  - actions: Travel between two cities
- solution:
  - Sequence locations, eg., Arad, Sibiu, Fagaras, Bucharest.

EXAMPLE


Oradea
71
Neamt
87
Zerind
151
75
Iasi
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
Pitesti
211
Lugoj
97
98
70
Hirsova
Mehadia
146
101
85
Urziceni
75
86
138
Bucharest
Dobreta
120
90
Craiova
Efo rie
Giu rgiu

PROBLEM FORMULATION

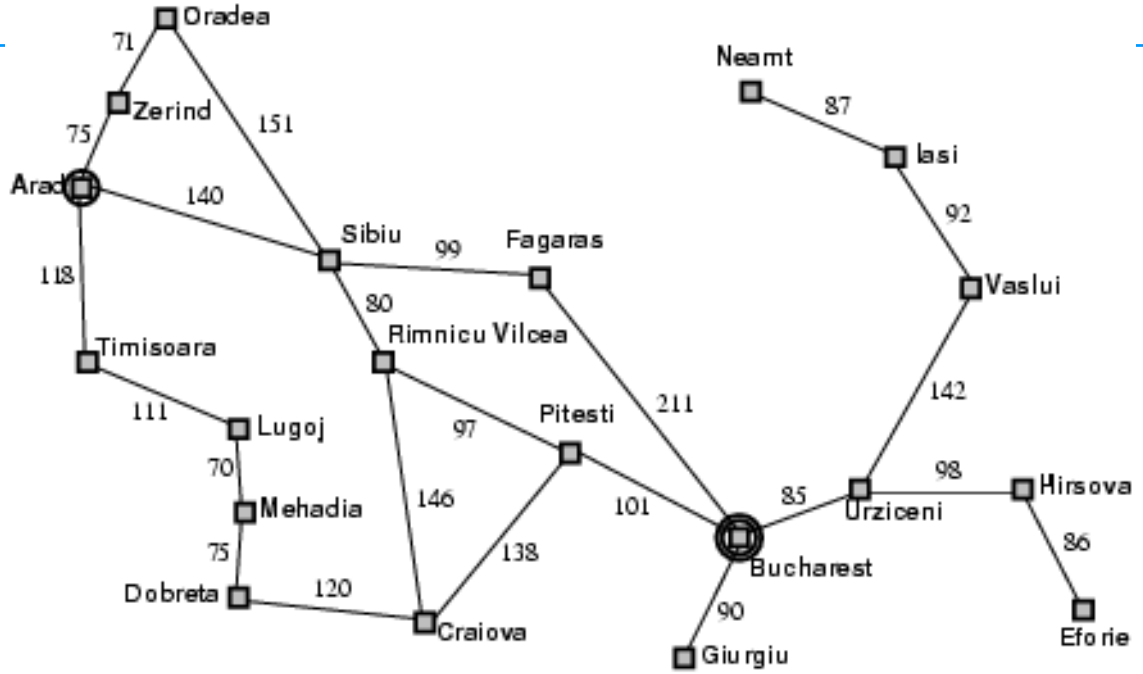
The problem is defined by four points:

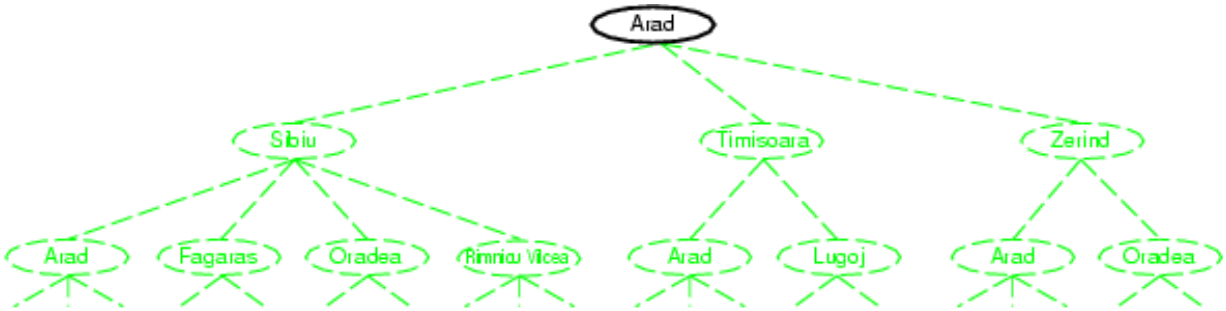
In order to find the solution we are going to find a pattern with less cost

1. Initial state e.g., "at Arad"
2. Successor functions *S (x)* = Set of action-state pairs
   - e.g., *S (Arad)* = {<*Arad* → *Zerind, Zerind*>, ... }
- test goal, could be
  - explicit, Eg., *x* = "At Bucharest"
  - implicit, Eg., *checkmap (x)*
- Cost of the path e.g., distance travelled, the number of actions (hops) performed etc. $c\ (x, a, y) \geq 0$
- A solution is a sequence of actions that lead from the initial state to the goal.

Non informed strategy; are strategy blind, do not consider the problem in question, and they can be applied everytime
Informed strategy: They have information about our problem, ad they are going to consider the cost of our operations

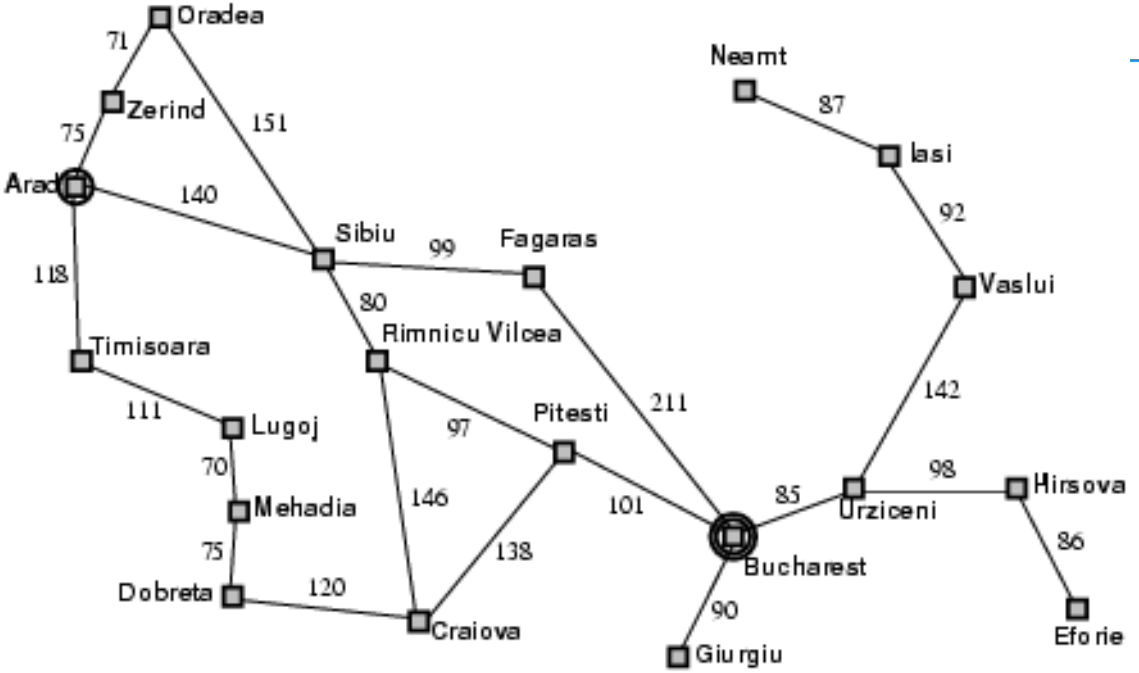
EXAMPLE

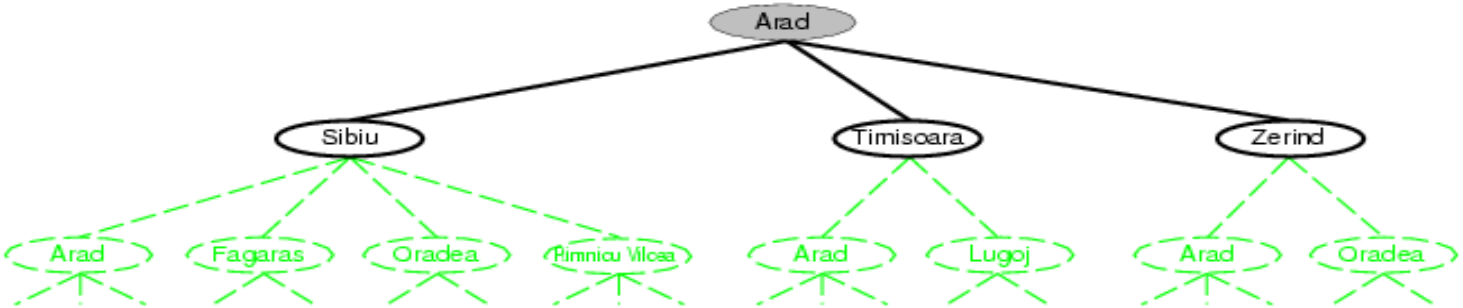




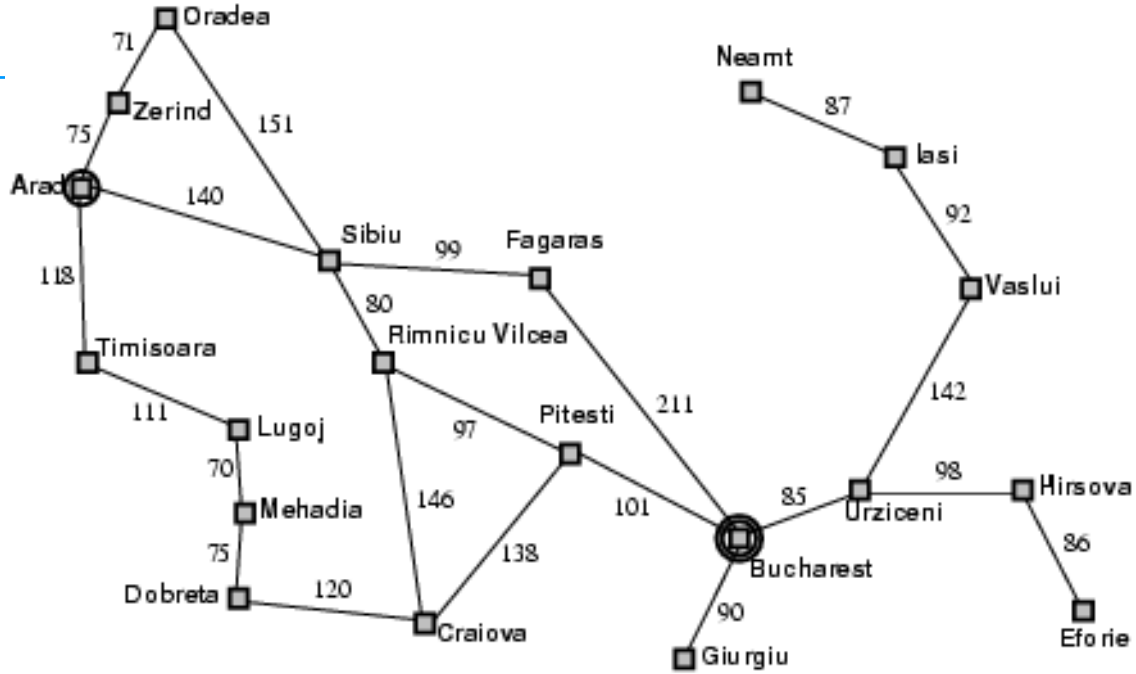
This is the search tree

EXAMPLE

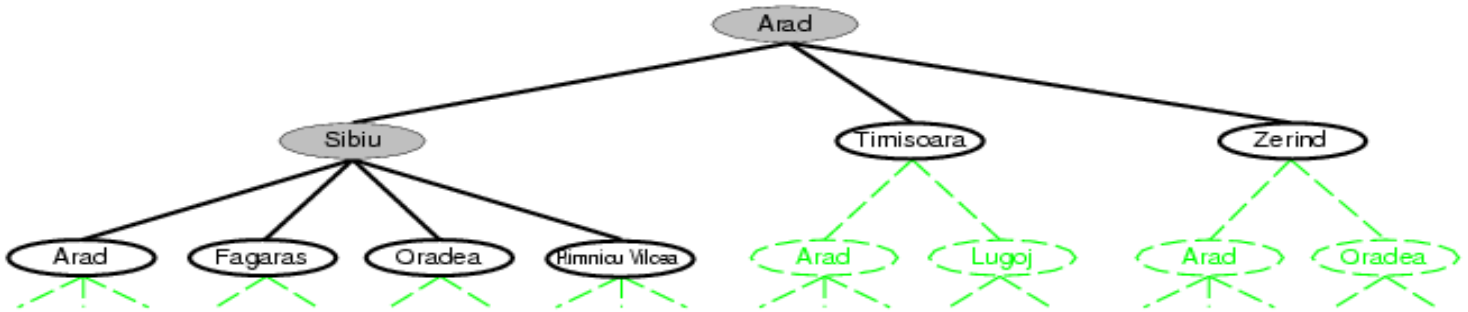

Oradea
71
Neamt
87
Zerind
151
Iasi
75
Arad
140
92
Sibiu
99
Fagaras
118
Vaslui
80
Rimnicu Vilcea
Timisoara
142
111
Pitesti
211
Lugoj
97
98
Hirsova
70
146
85
Mehadia
101
Urziceni
75
138
86
Bucharest
Dobreta
120
90
Craiova
Efo rie
Giu rgiu


Arad
Sibiu
Timisoara
Zerind
Arad
Fagaras
Oradea
Rimnicu Vilcea
Arad
Lugoj
Arad
Oradea

EXAMPLE

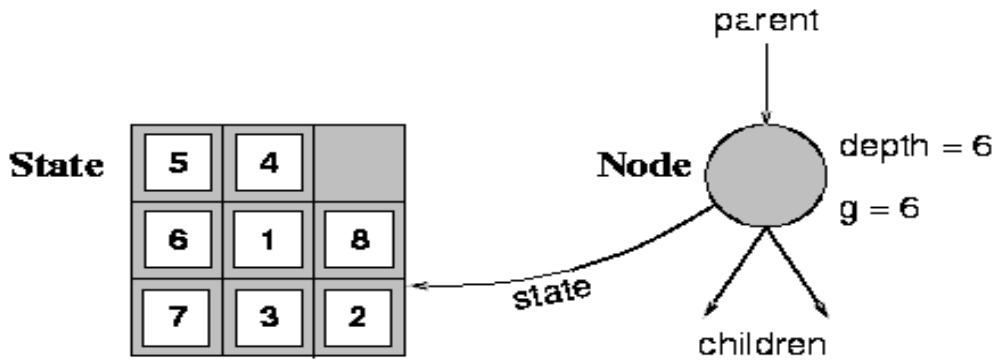


In the final the search tree that we have to write it is only the black part



NODE

- Each node in the search tree corresponds to a data structure containing
  - The state
  - The parent node.
  - The operator which has been applied to obtain the node.
  - The depth of the node.
  - The cost of the path from the initial state to the node



IMPLEMENTATION

```
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)
```

fringe is the set of nodes created but not expanded yet, it contains all the possibilities that we can choose

At the begginig we contain only the root, the initial state

here we insert in fridge all the children of the node that we choose

```
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
```

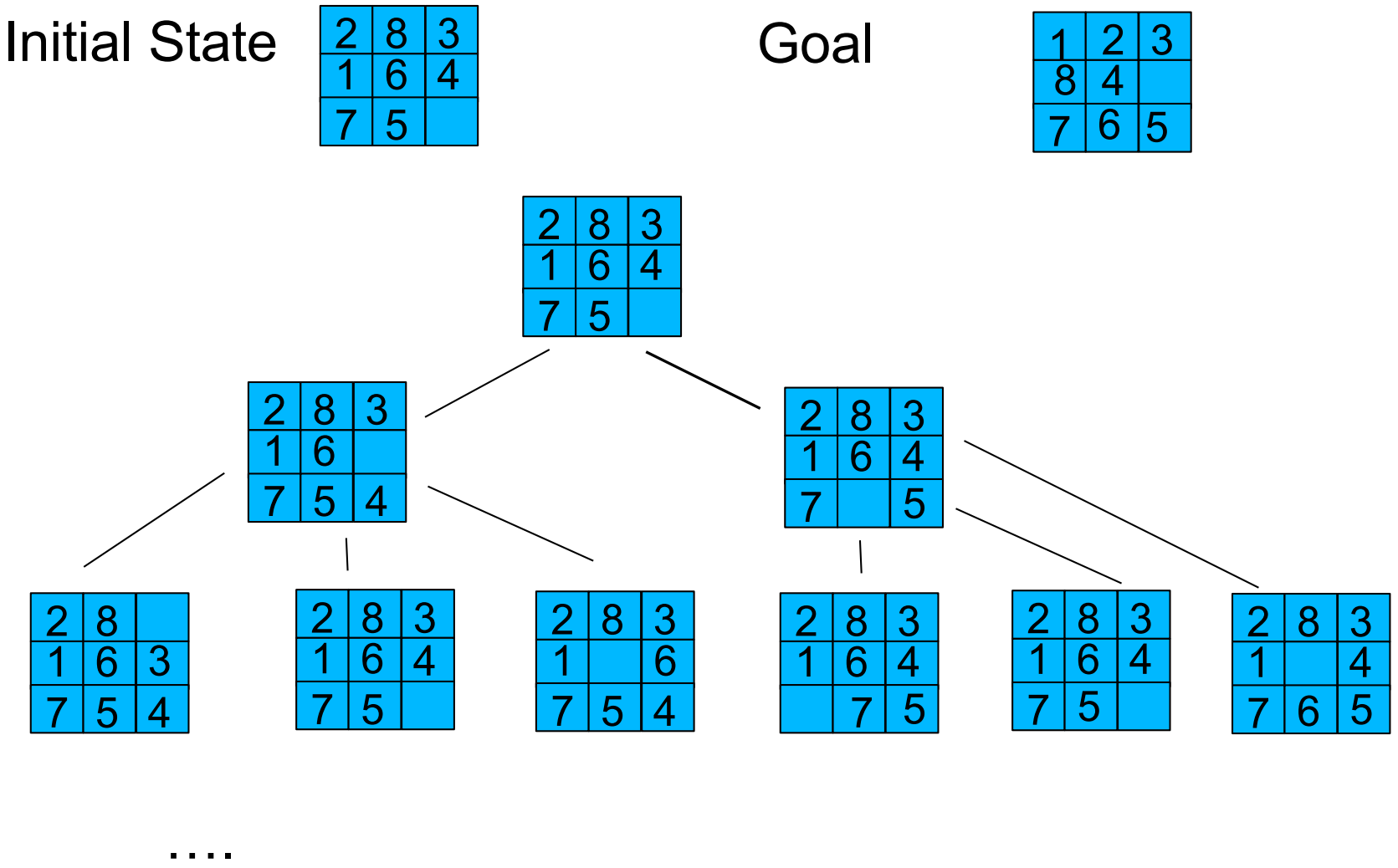
SEARCH STRATEGY: EFFECTIVENESS

- Can we find a solution?
- Is it a good solution?
  - minimum path solution – on-line cost
- What is the cost of the search?
  - Time to find a solution - off-line cost
- Total cost = search path cost + cost of the search.
- Choosing states and actions → Importance of abstraction

EXAMPLE: THE GAME OF 8

- We have a three by three board with eight tiles and one empty space (blank). We have to find the moves that lead to a given position as goal
- States: position of each of the tiles;
- Operators: the blank moves to the right, left, up and down;
- Test target: description of the final state;
- Walk cost: each move costs 1.

EXAMPLE: THE GAME OF 8



SEARCH STRATEGIES

- The choice on how to expand a search tree is called **strategy**.

- At each node more actions may be performed (action: operators of applications)

- STRATEGY. Two possibilities
  - **Non informed strategies** Do not use any domain knowledge: apply rules arbitrarily and do an exhaustive search strategy
    - Impractical for some complex problems.
  - **Informed strategies** Use domain knowledge: apply rules following a heuristics
    - If we had a perfect heuristics you do not need search

SEARCH STRATEGIES

- The Strategies are evaluated according to four criteria:
  - **Completeness**: does the strategy guarantees to find a solution if one exists?
  - **Time complexity**: how long does it take to find a solution?
  - **Space complexity**: how much memory is needed to carry out the search?
  - **Optimality**: does the strategy find the *best* solution when there are more solutions?

SEARCH STRATEGIES

- NON-INFORMED SEARCH STRATEGIES:
  - breadth-first (at a uniform cost);
  - depth-first;
  - depth-first limited depth;
  - iterative deepening.

THE GENERAL SEARCH STRATEGY

```
function GENERAL-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
```

THE GENERAL SEARCH STRATEGY

```
function GENERAL-SEARCH(problem, QUEUING-FN) returns a solution, or failure

    nodes ← MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))
    loop do
        if nodes is empty then return failure
        node ← REMOVE-FRONT(nodes)
        if GOAL-TEST[problem] applied to STATE(node) succeeds then return node
        nodes ← QUEUING-FN(nodes, EXPAND(node, OPERATORS[problem]))
    end
```

Queuing-Fn is a function passed to append the nodes obtained by the expansion

BREADTH-FIRST SEARCH

- Definition DEPTH:
  - The depth of the root node is equal to 0;
  - the depth of any other node is the depth of its parent plus 1

- Breadth-first search always EXPANDS LESS DEEP tree nodes. picture pag 24

  It basically means that it procede level by level

- In the worst case, if depth is d and the branching factor is b then the maximum number of nodes expanded in the worst case will be b^d. Time and space complexity (many paths stored in memory)

  - $1 + b + b^2 + b^3 + ... + (b^d - 1) \rightarrow b^d$

- This strategy ensures COMPLETENESS, but we don't have EFFICIENT IMPLEMENTATIONS on single-processor systems (multi-processor architectures).

  It is very costly so not really used

BREADTH-FIRST SEARCH

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

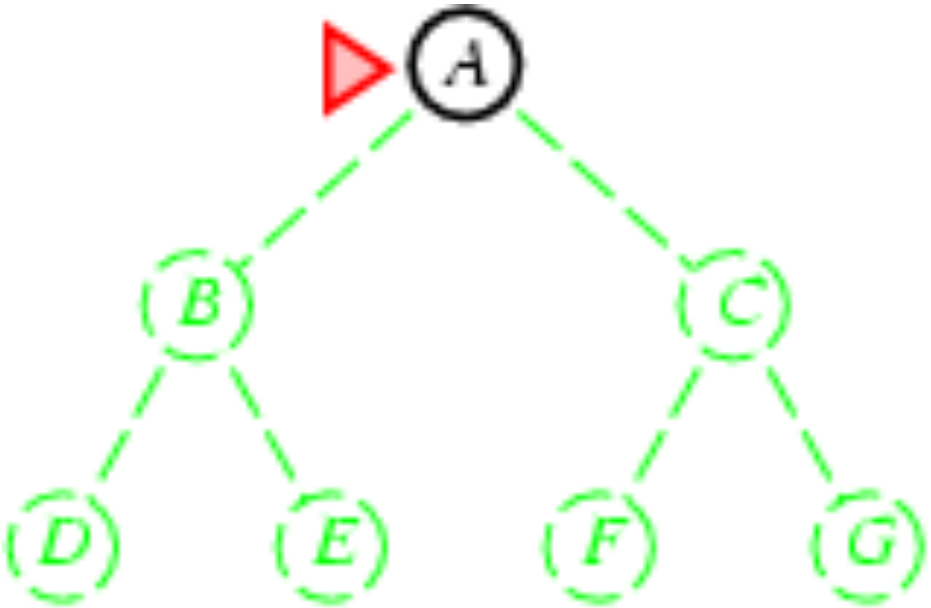
The main disadvantage is the excessive memory footprint. The example assumes that the branching factor is b = 10. It expands 1000 nodes/ second. Each node uses 100 bytes of memory.

BREADTH-FIRST SEARCH

- The problem of memory seems to be the most serious.
- Breadth-first search always finds the min-cost path if the cost equals the depth (otherwise we should use another strategy that always expands the least cost node → uniform cost strategy).
- The uniform-cost strategy is comprehensive and, unlike the breadth-first search, ideal when operators have not uniform costs. (Same temporal and spatial complexity as breadth-first search).

BREADTH-FIRST SEARCH

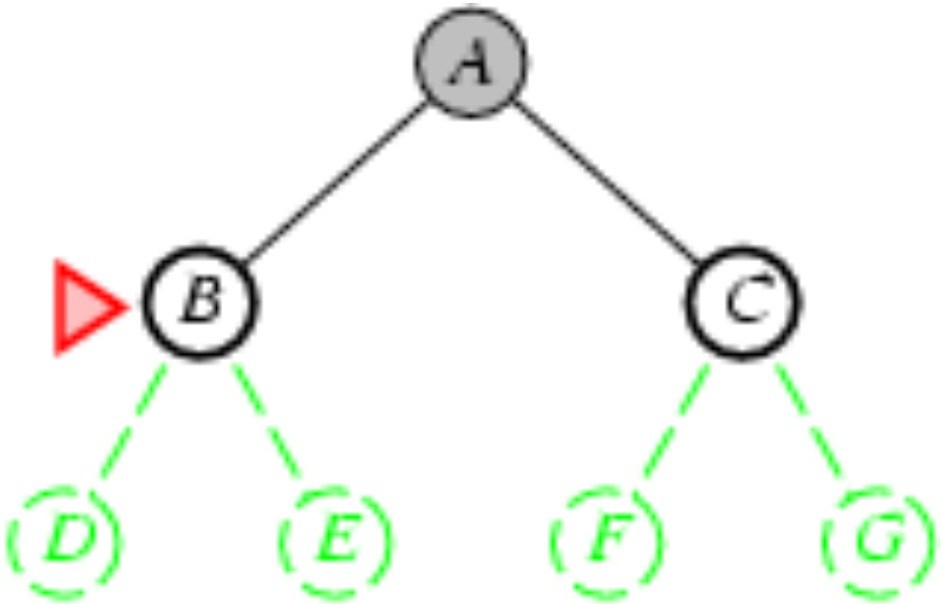
- Expands The nodes at lower depths
- Implementation:
  - *fringe* is a FIFO queue, i.e successors at bottom.



BREADTH-FIRST SEARCH

- Expands The nodes at lower depths
- Implementation:
  - *fringe* is a FIFO queue, i.e successors at bottom

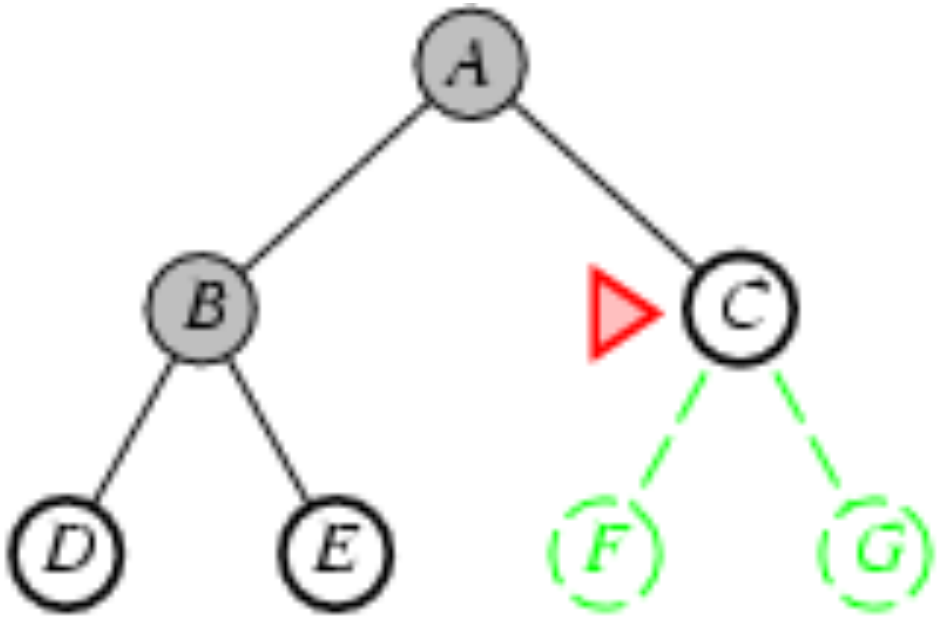
Fringe {A}



BREADTH-FIRST SEARCH

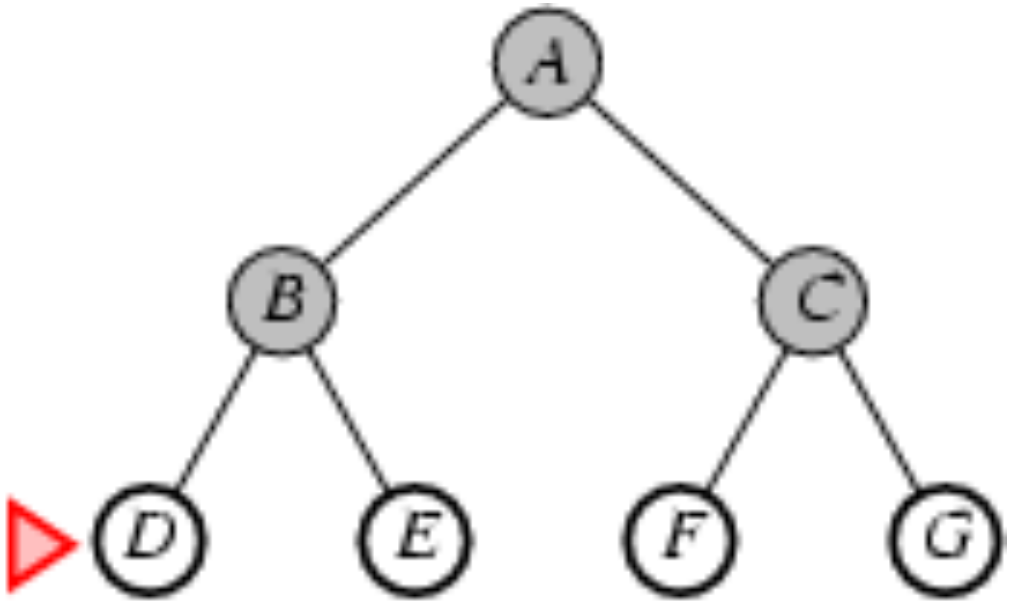
- Expands The nodes at lower depths
- Implementation:
  - *fringe* is a FIFO queue, i.e successors at bottom

fringe {A,B}

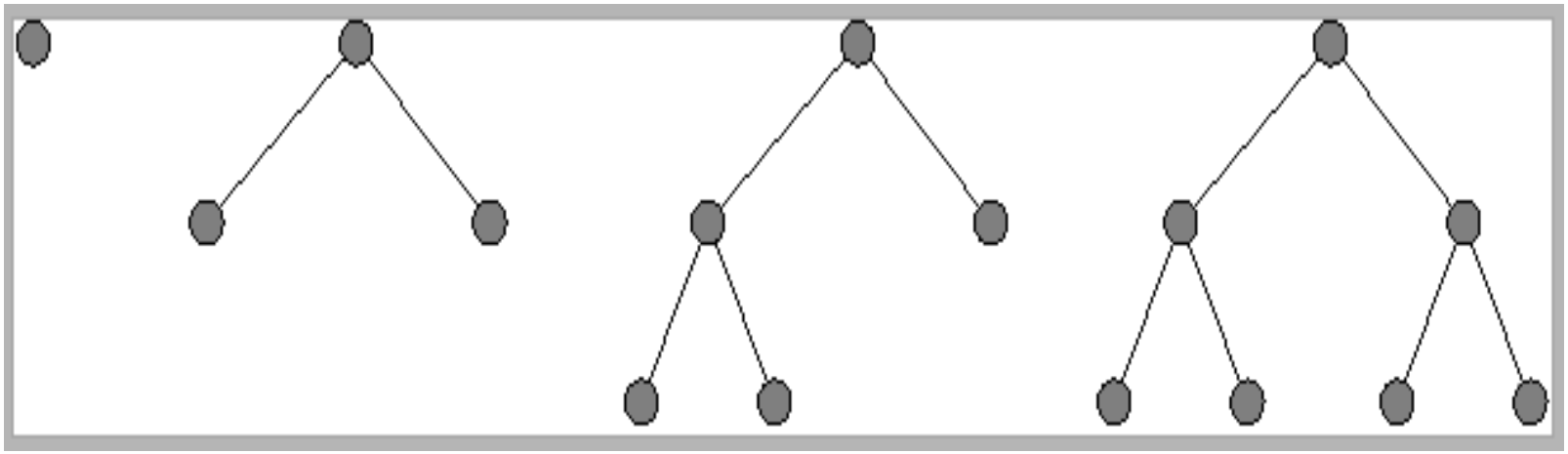


BREADTH-FIRST SEARCH

- Expands The nodes at lower depths
- Implementation:
  - *fringe* is a FIFO queue, i.e successors at bottom



BREADTH-FIRST SEARCH



QueueingFn = Add the successors to the end of the queue

Properties of breadth-first search

Complete?? Yes (if b is finite)

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

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

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

1 or euqal for every step

Space is the big problem; can easily generate nodes at 1MB/sec

so 24hrs $=$ 86GB.

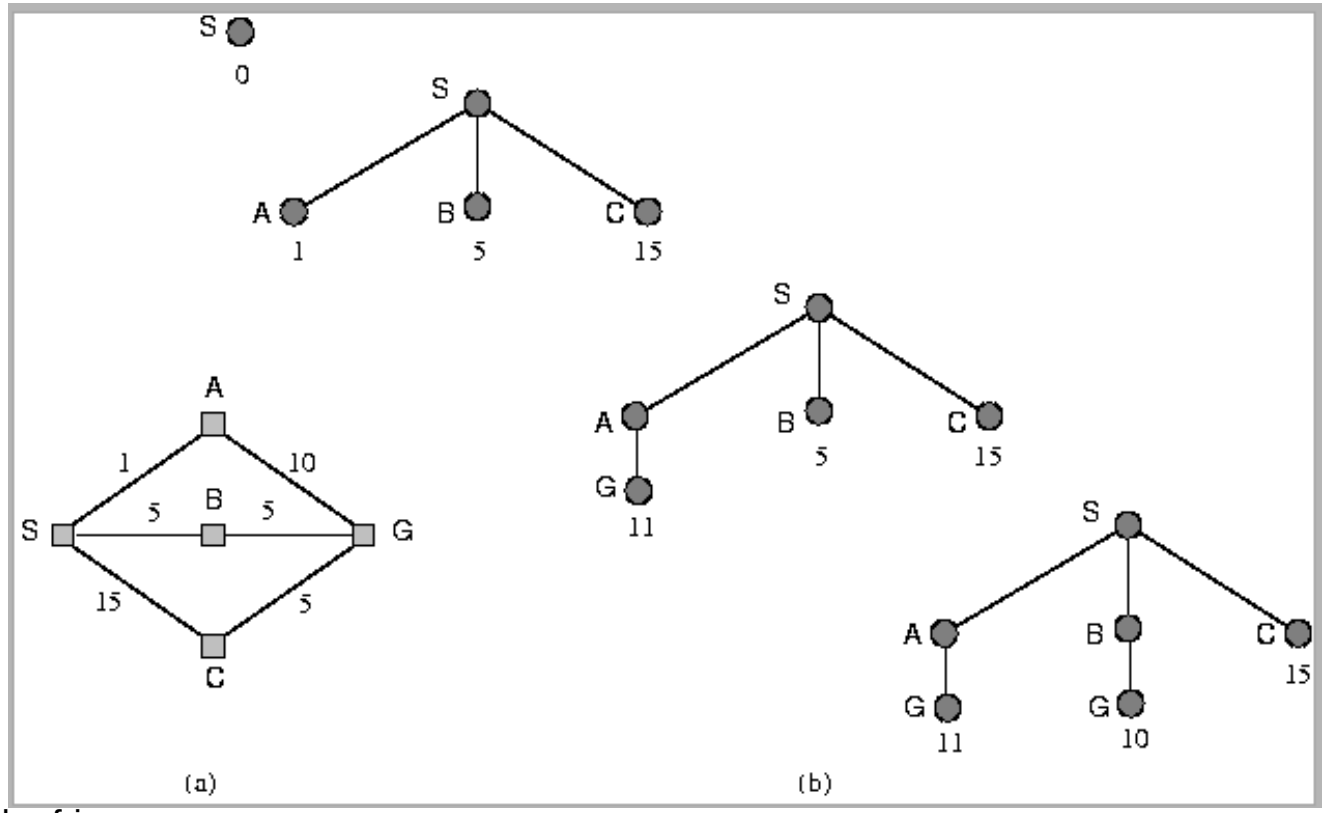
b - Maximum of the search tree branching

d - depth of the least-cost solution

m - maximum depth of the state space (may be infinite)

UNIFORM-COST SEARCH

Each node is labelled with a cost g(n)



Before insert them in the fringe
we order them in a "lower-ordered array"

QueueingFn = Enter the successors in order of increasing path cost

DEPTH FIRST SEARCH

It does the opposite of the breadth

- Depth-first EXPANDS deepest nodes first;
- Nodes at equal depth are ARBITRARILY selected (leftmost).
- Depth-first search requires a modest memory occupation
- For a state space with branching factor b and maximum search depth d we have to store b * d nodes.
- The temporal complexity is rather similar to that of breadth-first search.
- In the worst case, if the maximum depth is d and the branching factor is b the maximum number of nodes expanded is b^d. (Time complexity).

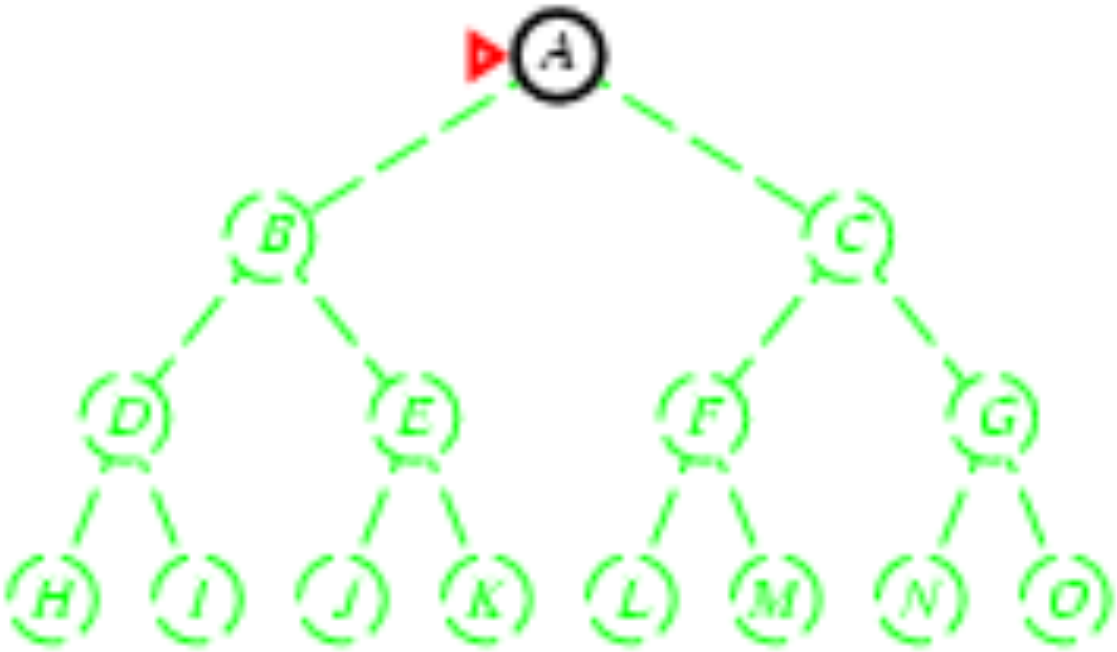
DEPTH FIRST SEARCH

- Depth-first search is EFFICIENT from an implementation point of view: one path at a time is stored (a single stack)

- Depth-first search can be NON-COMPLETE with possible loops in the presence of infinite branches.

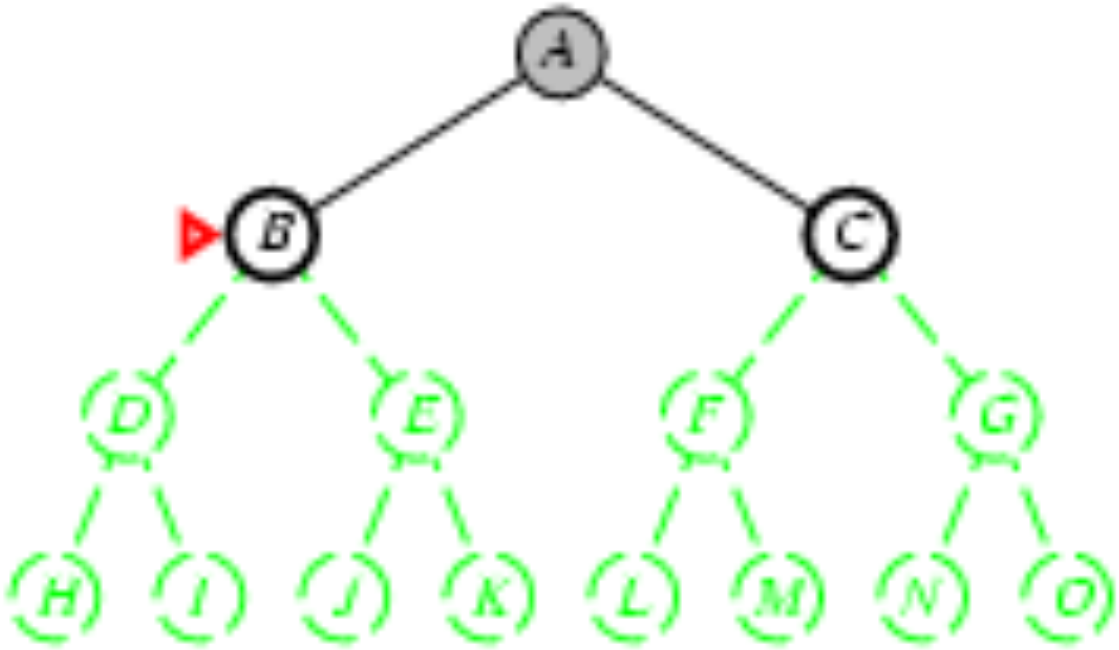
DEPTH FIRST SEARCH

- Expands The deeper nodes first
- Implementation:
  - *fringe* = LIFO stack



DEPTH FIRST SEARCH

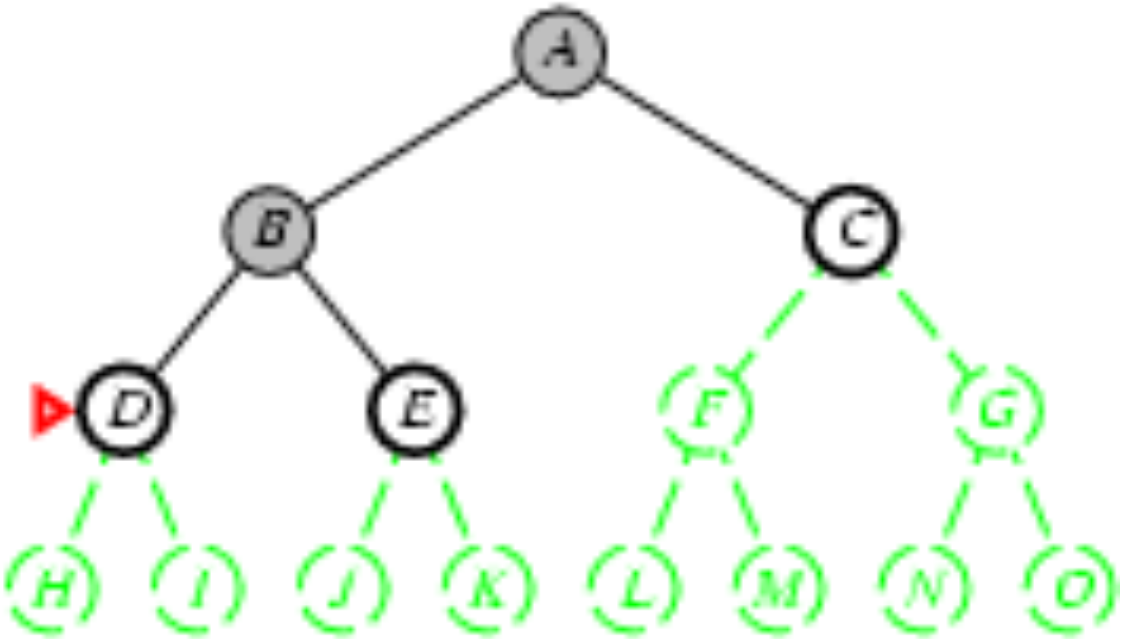
- Expands The deeper nodes first
- Implementation:
  - *fringe* = LIFO stack



DEPTH FIRST SEARCH

- Expands The deeper nodes first
- Implementation:
  - *fringe* = LIFO stack

fringe {D, E, C}

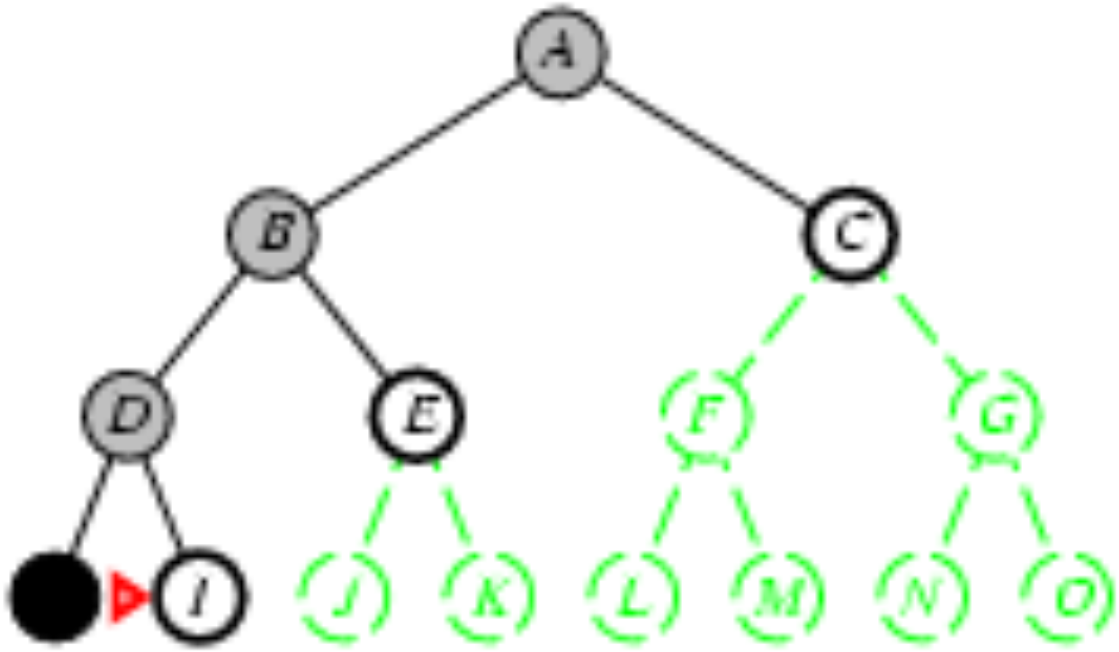


DEPTH FIRST SEARCH

- Expands The deeper nodes first
- Implementation:
  - *fringe* = LIFO stack

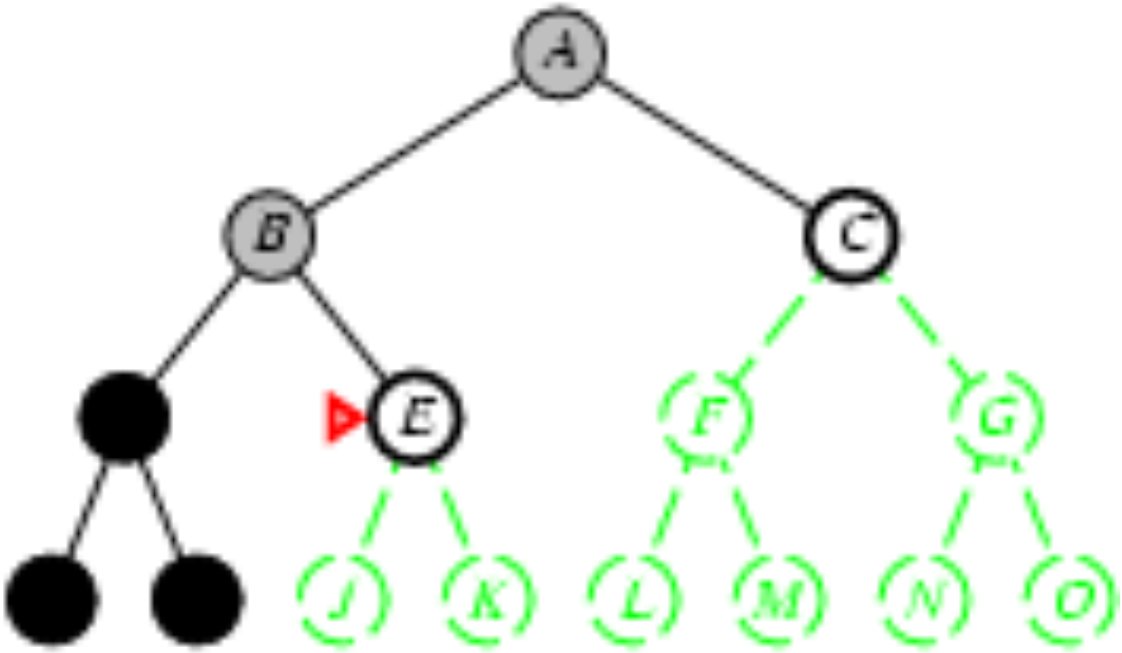
DEPTH FIRST SEARCH

- Expands The deeper nodes first
- Implementation:
  - *fringe* = LIFO stack



DEPTH FIRST SEARCH

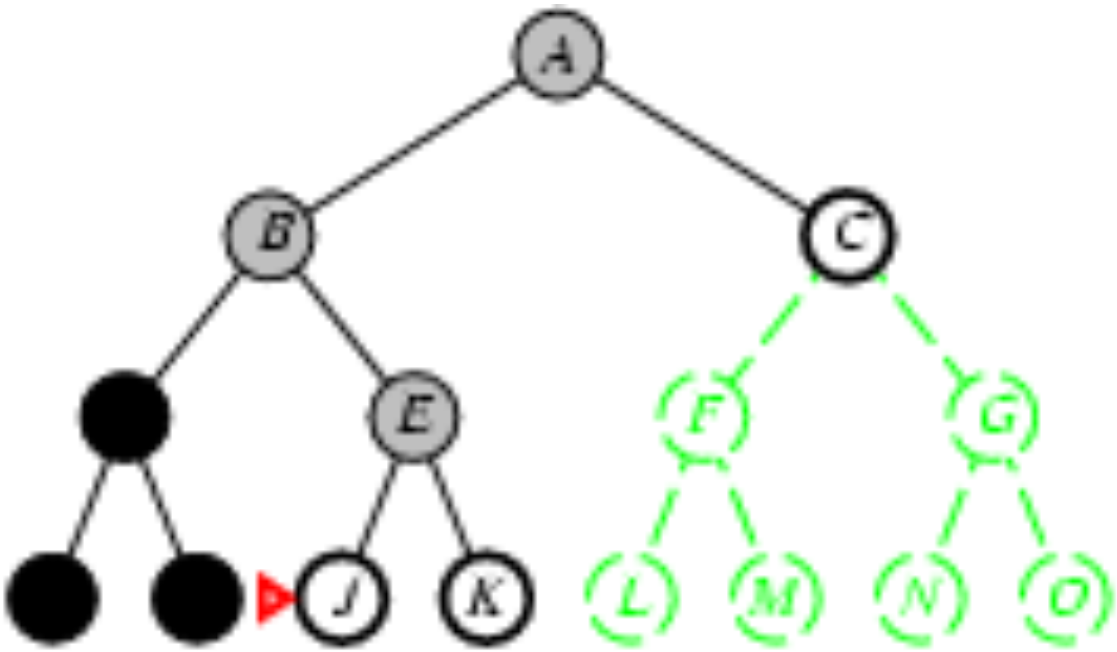
- Expands The deeper nodes first
- Implementation:
  - *fringe* = LIFO stack



DEPTH FIRST SEARCH

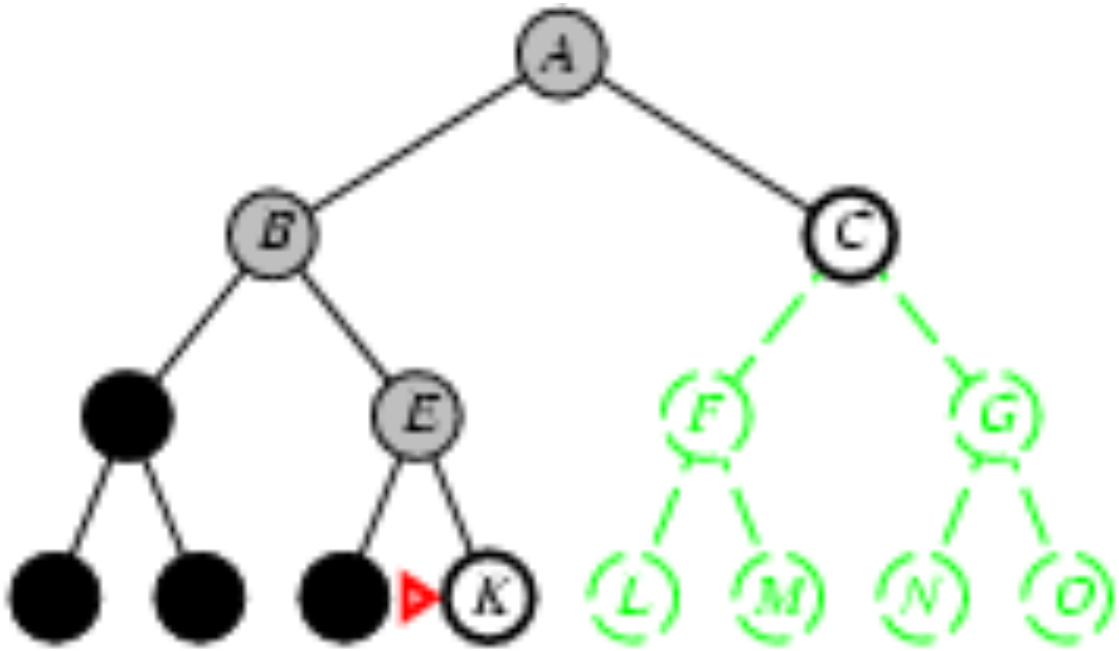
- Expands The deeper nodes first
- Implementation:
  - *fringe* = LIFO stack

fringe {J,K,C}



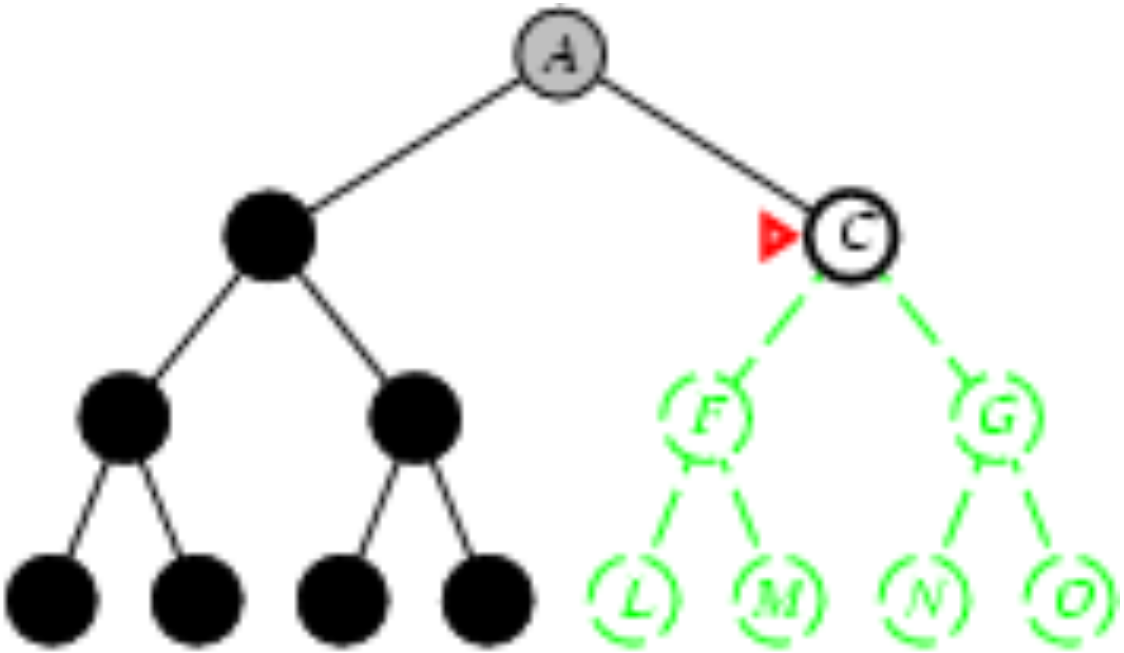
DEPTH FIRST SEARCH

- Expands The deeper nodes first
- Implementation:
  - *fringe* = LIFO stack



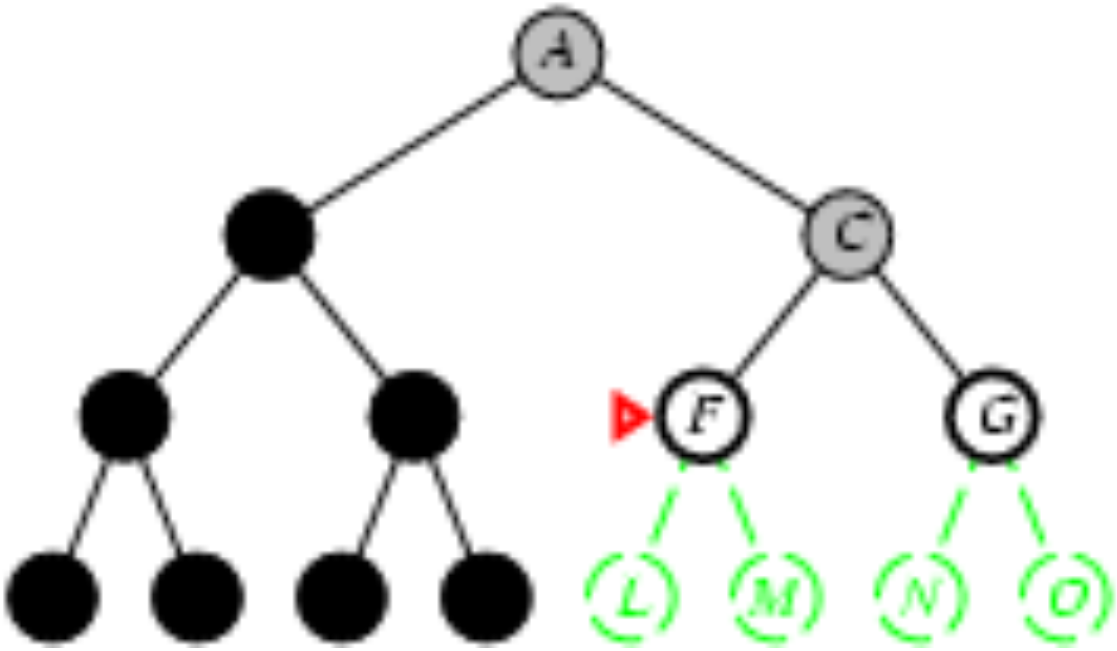
DEPTH FIRST SEARCH

- Expands The deeper nodes first
- Implementation:
  - *fringe* = LIFO stack



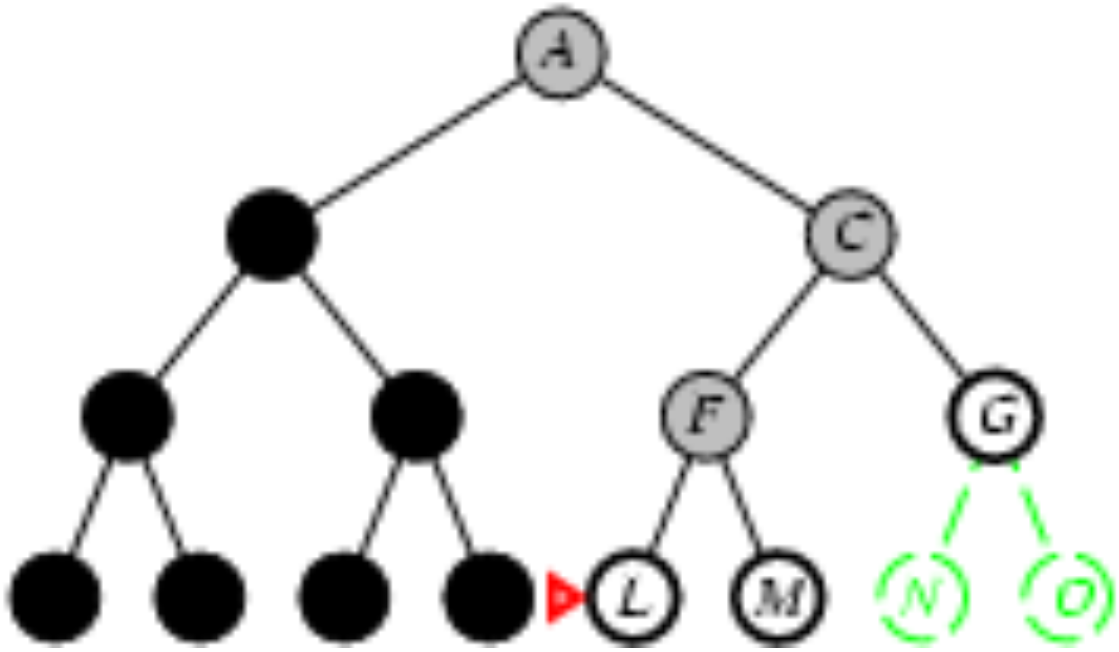
DEPTH FIRST SEARCH

- Expands The deeper nodes first
- Implementation:
  - *fringe* = LIFO stack

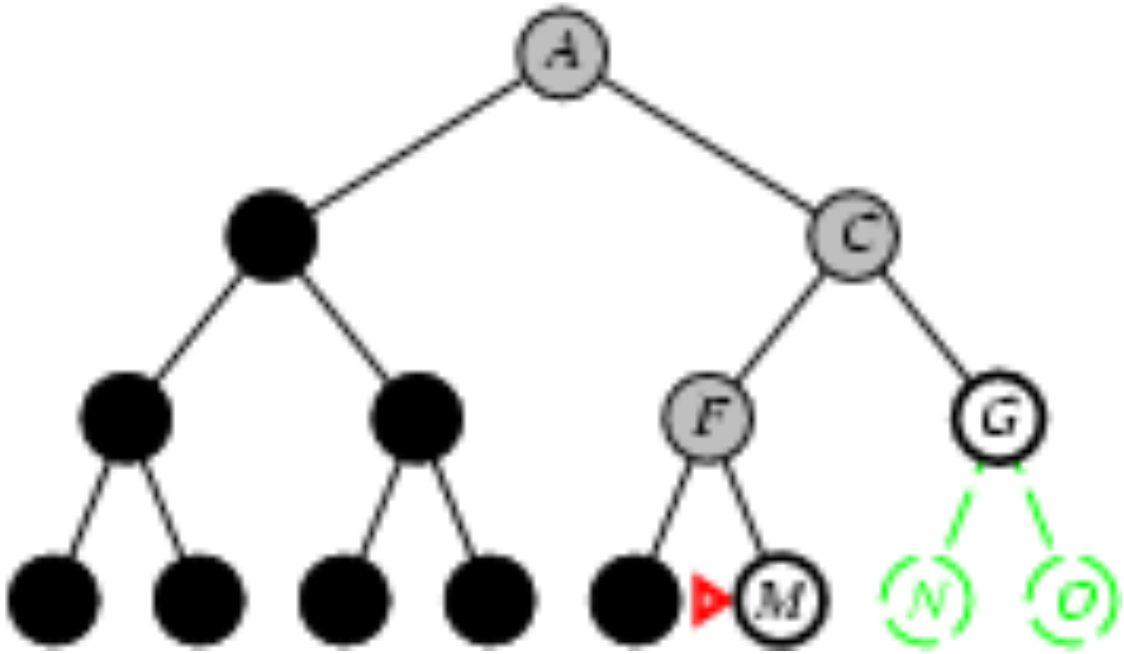


DEPTH FIRST SEARCH

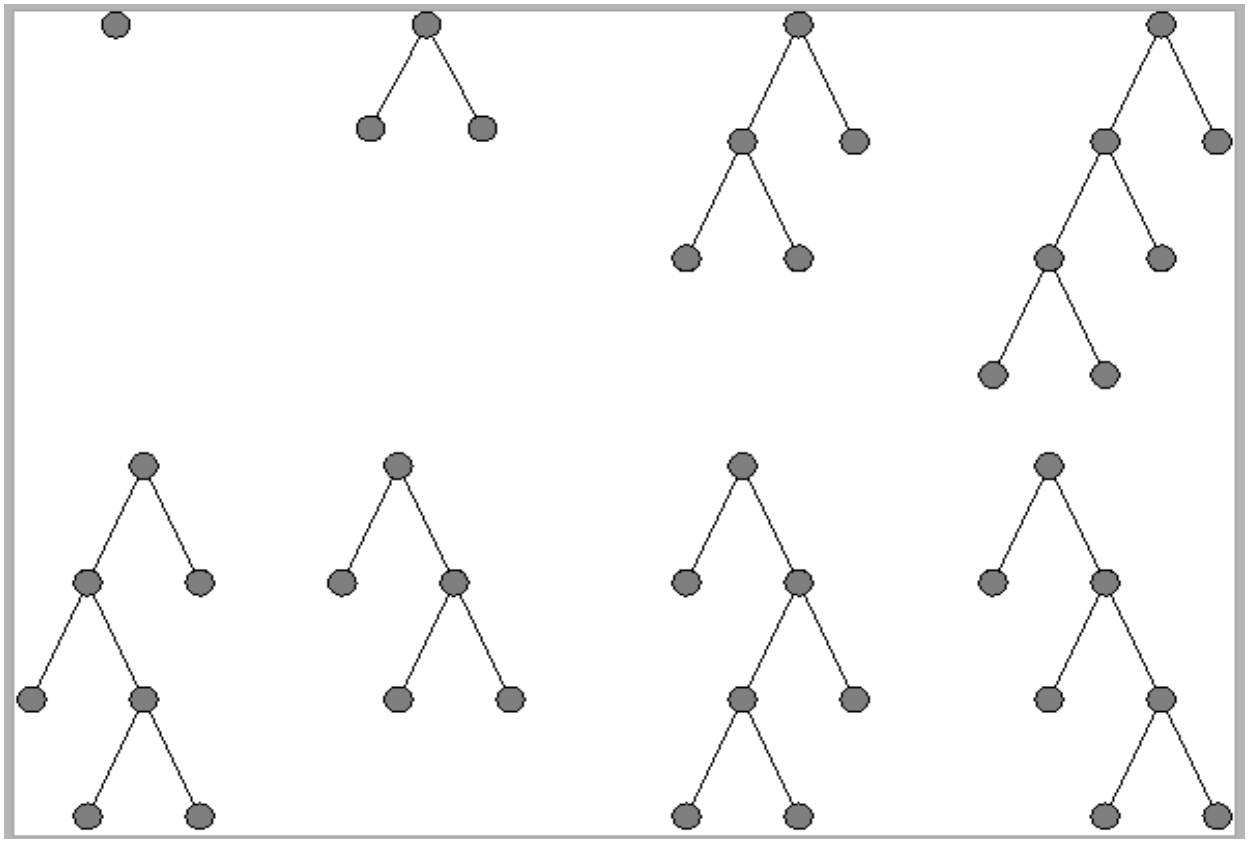
- Expands The deeper nodes first
- Implementation:
  - *fringe* = LIFO stack



DEPTH FIRST SEARCH



DEPTH FIRST SEARCH



QueueingFn = Enter the successors to the top of the stack.
It is assumed that the nodes of depth 3 does not have successors

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear space!

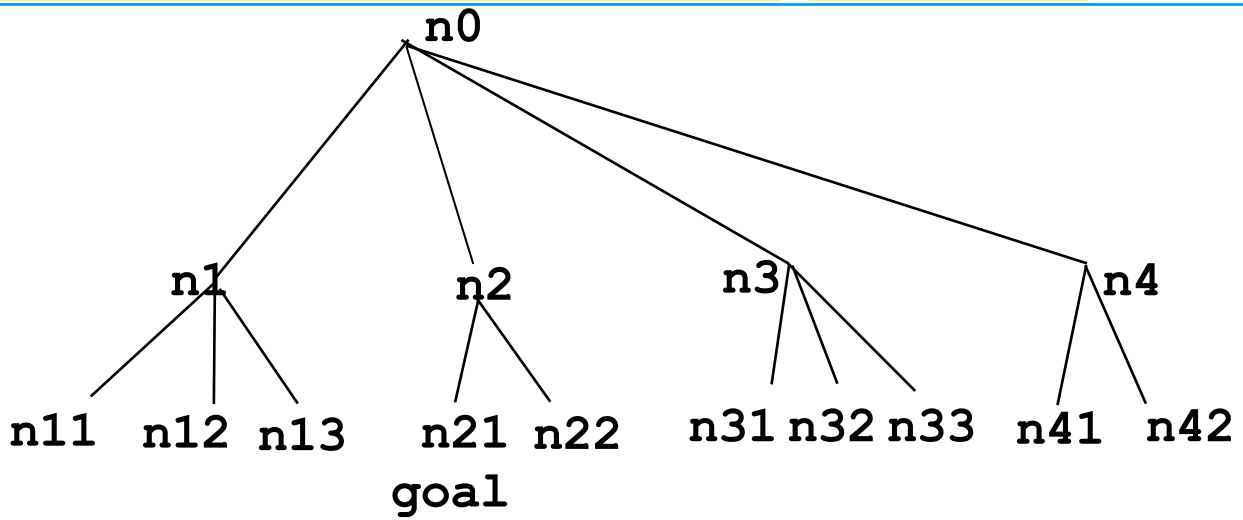
Optimal?? No

b - Maximum of the search tree branching

d - depth of the least-cost solution

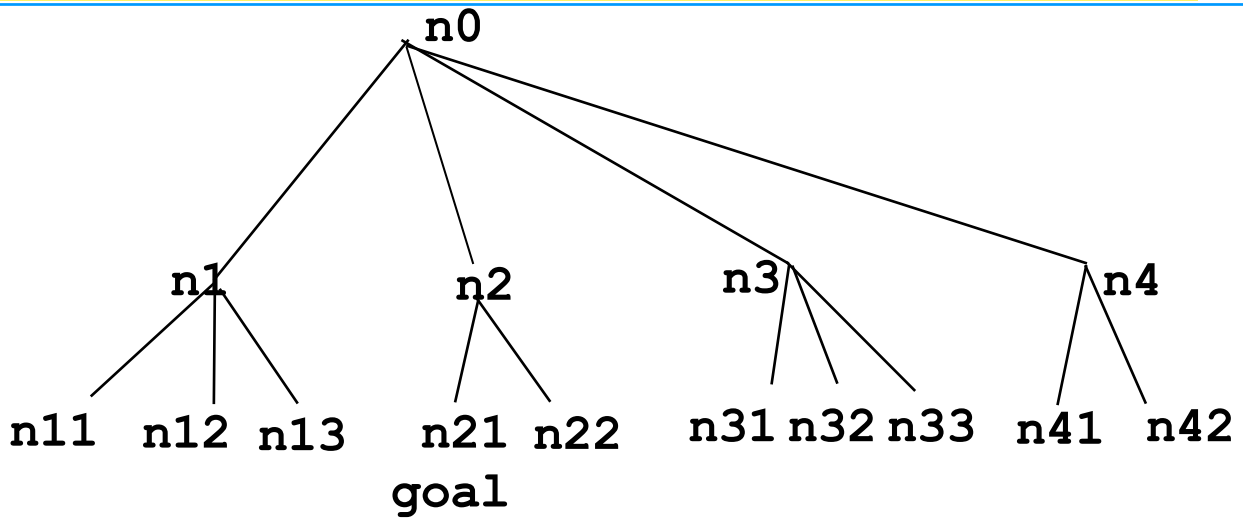
m - maximum depth of the state space (may be infinite)

EXAMPLE: DEPTH-FIRST



- Depth-first: nodes expanded added at the head of L
  - n0
  - n1, n2, n3, n4
  - n11, n12, n13, n2, n3, n4
  - n11, n12, n13, n2, n3, n4
  - n12, n13, n2, n3, n4
  - n13, n2, n3, n4
  - n2, n3, n4
  - **n21**, n22, n3, n4 Success

EXAMPLE: BREADTH-FIRST



- Breadth-first: nodes expanded appended to L.
  - n0
  - n1, n2, n3, n4
  - n2, n3, n4, n11, n12, n13
  - n3, n4, n11, n12, n13, n21, n22
  - n4, n11, n12, n13, n21, n22, n31, n32, n33
  - n11, n12, n13, n21, n22, n31, n32, n33, n41, n42
  - n12, n13, n21, n22, n31, n32, n33, n41, n42
  - n13, n21, n22, n31, n32, n33, n41, n42
  - **n21**, N22, n31, n32, n33, n41, n42 Success

LIMITED DEPTH SEARCH

- It is a depth-first variant
- It includes a MAXIMUM DEPTH parameter
- When you reach the maximum depth or a failure, it explores alternative paths (if they exist), then alternative paths at less than one unit of depth, and so forth (backtracking).
- You may establish a maximum limit of depth (it does not necessarily solve the problem of completeness).
- Avoids infinite branches

LIMITED DEPTH SEARCH

The nodes at depths l have no successors.

- Recursive Implementation:

```
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

- Iterative deepening search avoids the problem of choosing the maximum depth limit by trying all possible depth limits.
  - Start with 0, then 1, then 2 etc ...

- It combines the advantages of depth and breadth-first strategies. It is complete and explores a single branch at a time.

- Many states are expanded multiple times, but this does not worsen considerably the execution time.

- In particular, the total number of expansions is: $(d+1)1 + (d)b + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + b^d$.

- In general it is the favourite search strategy when the search space is very large.

ITERATIVE DEEPENING

- It can emulate the breadth-first search through repeated applications of the depth first search with an increasing depth limit.

1. C = 1
2. Apply depth-first with depth limit of C, if you find a solution stop
3. Otherwise, increase C and go to step 2

ITERATIVE DEEPENING SEARCH

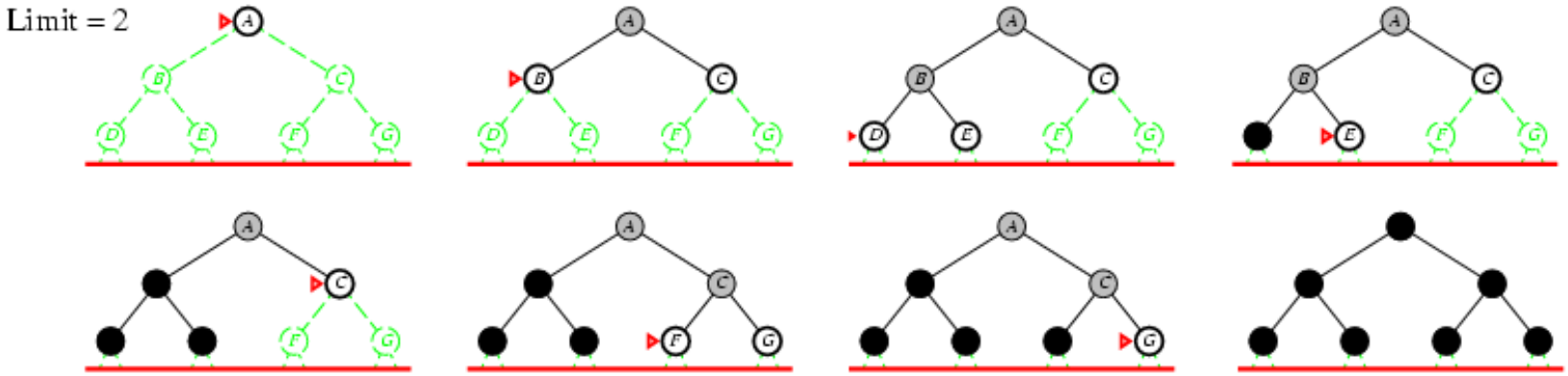
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

ITERATIVE DEEPENING SEARCH L = 0

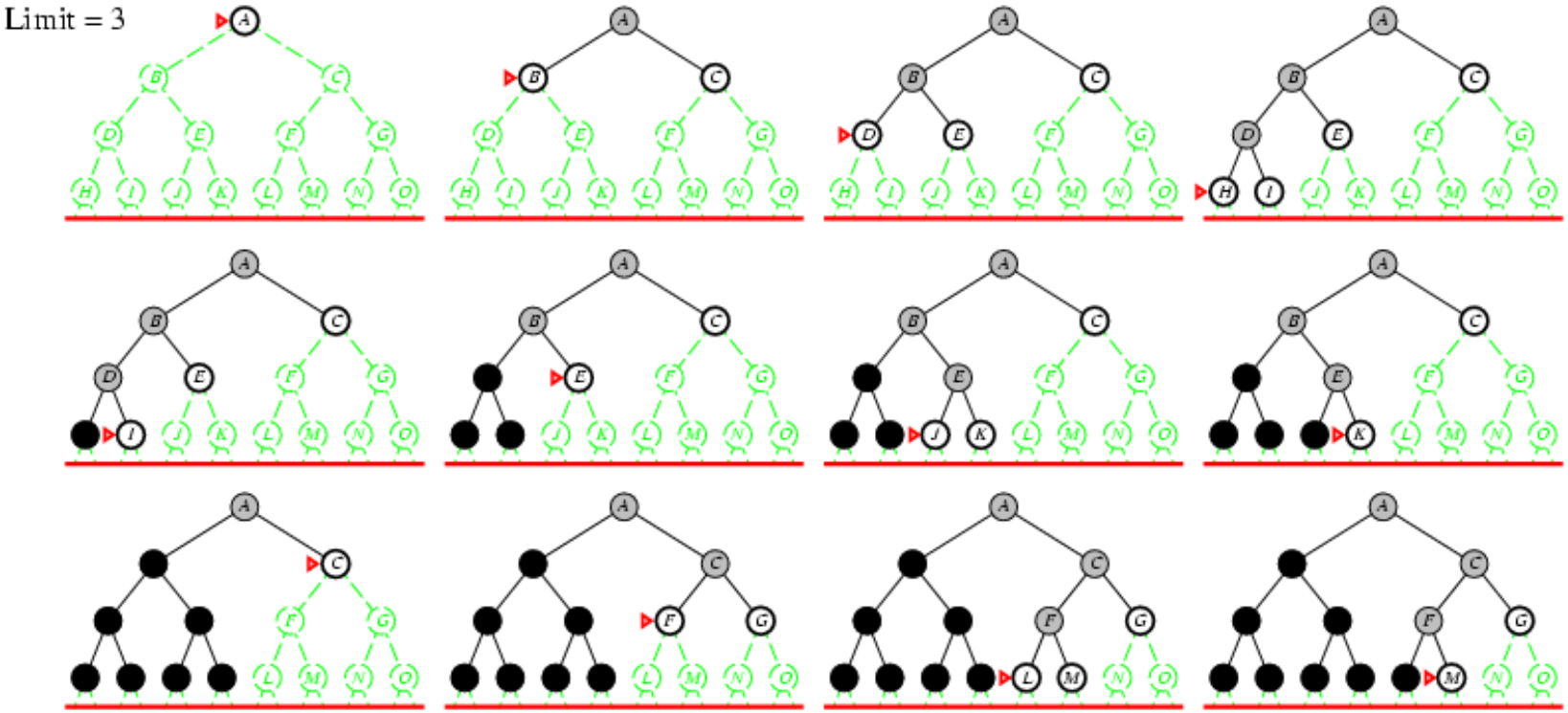
Limit = 0

ITERATIVE DEEPENING SEARCH L = 1

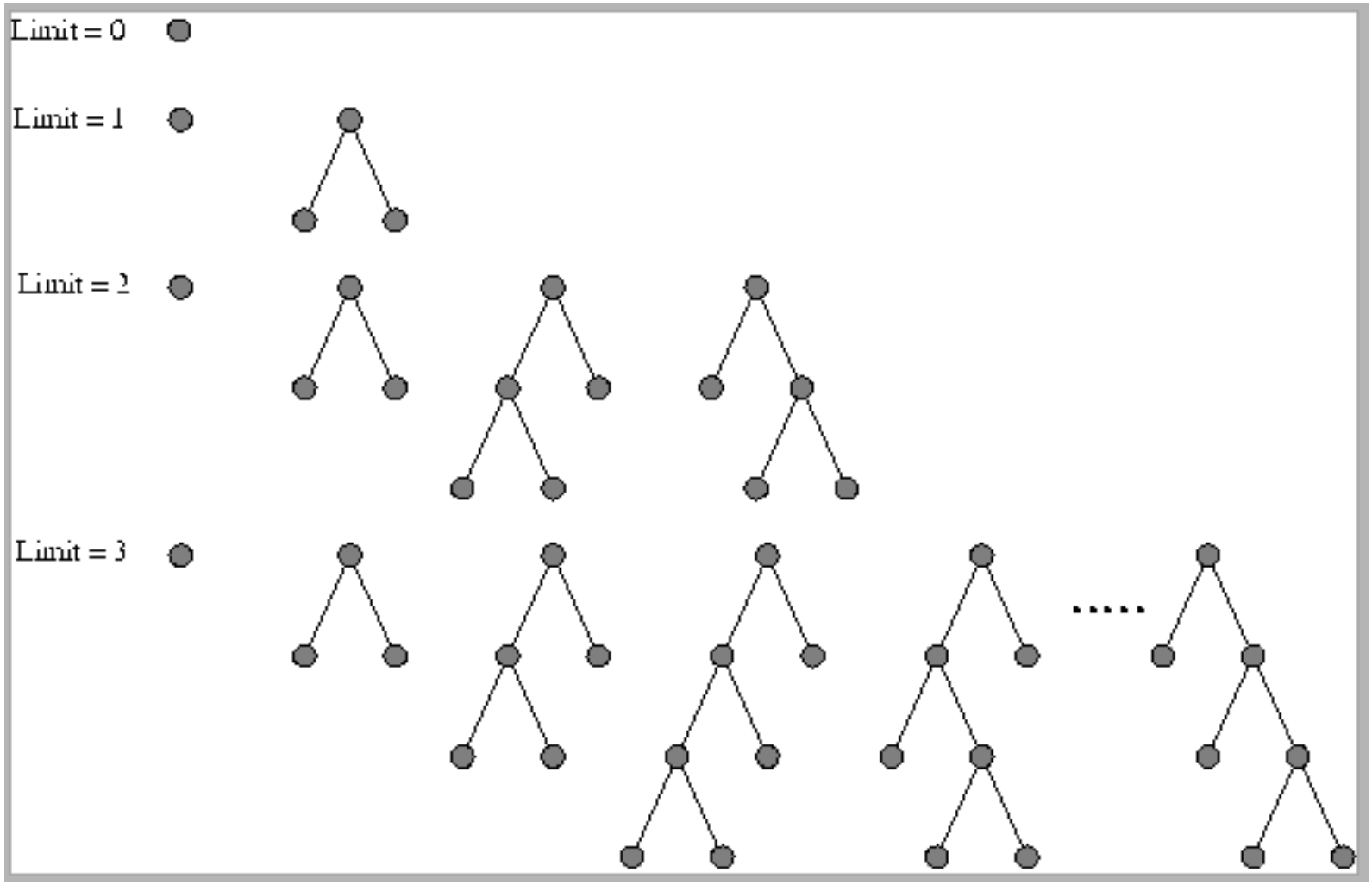
ITERATIVE DEEPENING SEARCH L = 2



ITERATIVE DEEPENING SEARCH L = 3



ITERATIVE DEEPENING SEARCH



Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$

Can be modified to explore uniform-cost tree

b - Maximum of the search tree branching

d - depth of the least-cost solution

m - maximum depth of the state space (may be infinite)

Comparison of search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

b = branching factor; d = profondià of the solution; m = maximum depth of the search tree; l = depth limit.

WHERE TO APPLY SEARCH STRATEGIES: PRODUCTION SYSTEMS

- Set of operators (rules);
- One or more databases (working memories);
- Control Strategy.
- MODULARITY - FLEXIBILITY
- Operators:
  - **`IF <pattern> THEN <body>`**
  - Rules are not invoked by name, but are activated based on pattern-matching

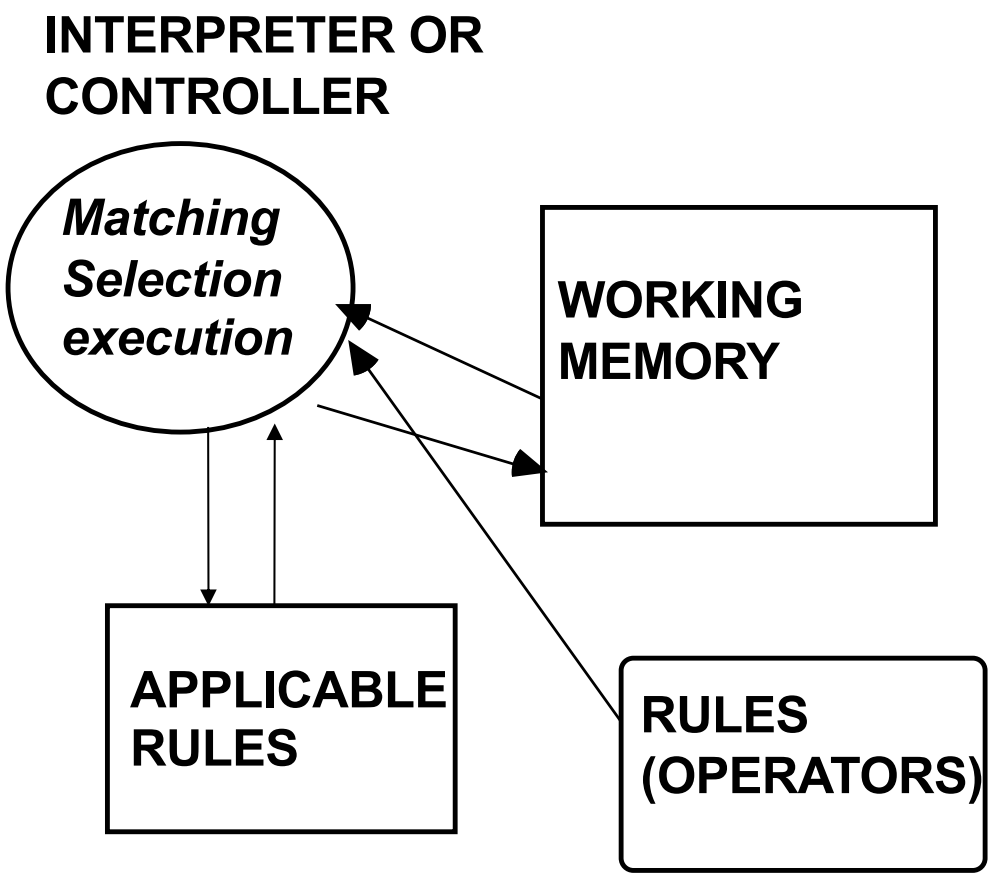
PRODUCTION SYSTEMS

- Definition:

> Programs that perform search methods to problems represented as a state space.

- consist of:
  - A set of rules,
  - a 'Working memory', which contains the current achieved states
  - a control strategy that select rules to apply to states of 'Working memory' (Matching, verification of preconditions and tests the state goal if achieved).

GENERAL ARCHITECTURE:



TWO REASONING MODES

- FORWARD OR DATA-DRIVEN:
  - The working memory in its initial configuration contains the initial knowledge about the problem, namely the known facts.
  - The production rules selected are those whose antecedent can do *matching* with the working memory (F-rules).
  - The process ends with success when the goal to prove is in the working memory

TWO REASONING MODES

- BACKWARD OR GOAL-DRIVEN:
  - The initial working memory contains the goals (or goal) of the problem.
  - The production rules are those whose consequent can do *matching* with the working memory (B-rules).
  - Each time a rule is selected and executed, new subgoals to prove are inserted into the working memory.
  - The process ends with success when in the initial state appears in the working memory

WHEN APPLYING BACKWARD AND FORWARD WHEN?

- What is the average number of branches generated from a single node?
- What is the most natural mode of reasoning? (explanation for the user)

- TWO-WAY OR MIXED:
  - It is the combination of forward and backward methods;
  - The working memory is divided into two parts: one containing the facts and the other the goals or subgoals;
  - Apply simultaneously F-rules and B-rules to the two parts of the working memory. Termination is achieved when the portion of the working memory created by backward chaining is equal to or a subset of the one obtained by means of forward chaining (TERMINATION CONDITION).