Block 1 – Knowledge Representation and Search

Chapter 4:
Solving problems by searching.
Uninformed Search.

Chapter 4- Uninformed search

1. Problem Formulation: reminder and example
2. Searching for Solutions
3. Breadth-first search
4. Uniform-cost search
5. Depth-first search
6. Search in CLIPS
7. Iterative deepening depth-first search

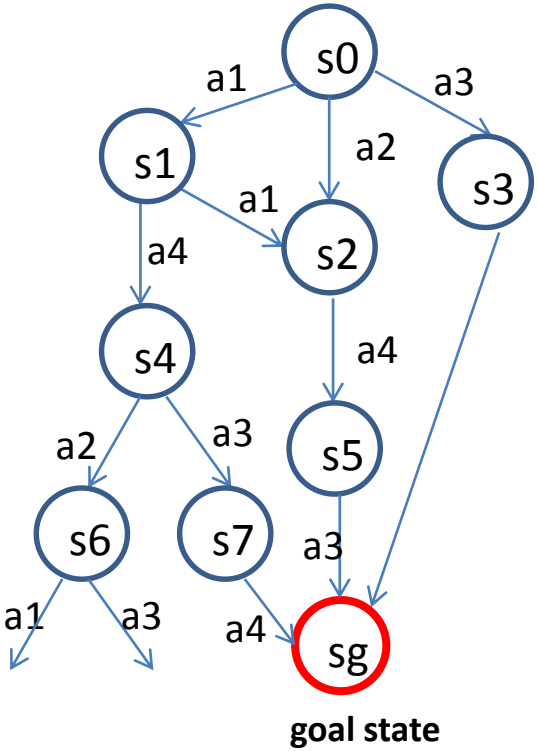
Bibliography

S. Russell, P. Norvig. ***Artificial Intelligence . A modern approach.*** Prentice Hall, 3rd edition, 2010 (Chapter 3) http://aima.cs.berkeley.edu/

1. Problem Formulation: a reminder

Concepts:

- State-based representation
- Initial state: *s0*
- Goal state: *sg*
- Actions; e.g {a1,a2,a3,a4}
- Transition model: Result(s,a)
- State space: directed network or graph
- Path



1. Problem Formulation: a reminder

Path cost function assigns a numeric cost to each path

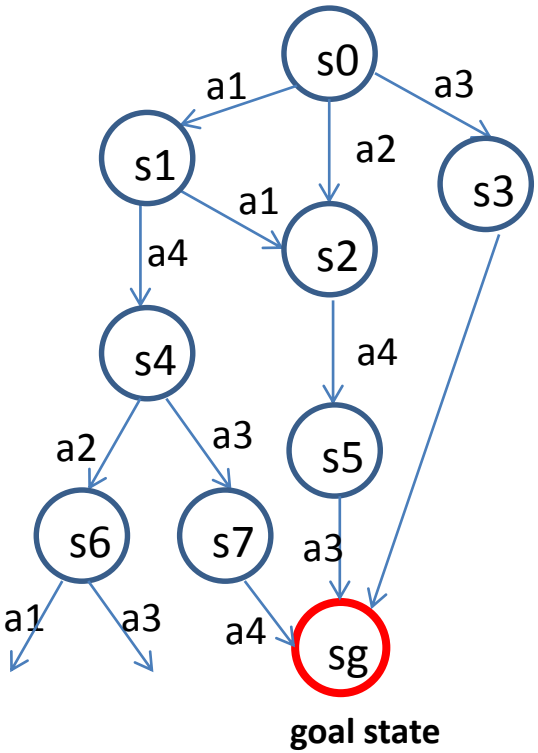
- the step cost of taking action **a** in state **s** to reach **s'**: c(s,a,s'); for simplicity, we will use c(a) when each action is assigned a fixed cost.

- the cost of a path is described as the sum of the costs of the individual actions along the path

- the cost of a state sn (g(sn)) is the cost of the path to go from the initial state s0 to such a state sn

$$g(sn)=c(s0,a1,s1)+c(s1,a2,s2)+ \dots + c(sn-1,an, s_n))= c(a1)+c(a2)+\dots+c(an)$$

- Examples:
  - g(s5)= c(s0,a2,s2)+c(s2,a4,s5)=c(a2)+c(a4)
  - There are two different paths to reach s2:
    - g(s2)=c(a2)
    - g(s2)=c(a1)+c(a1)
    - We will be interested in the minimum path cost



1. Problem formulation: an example

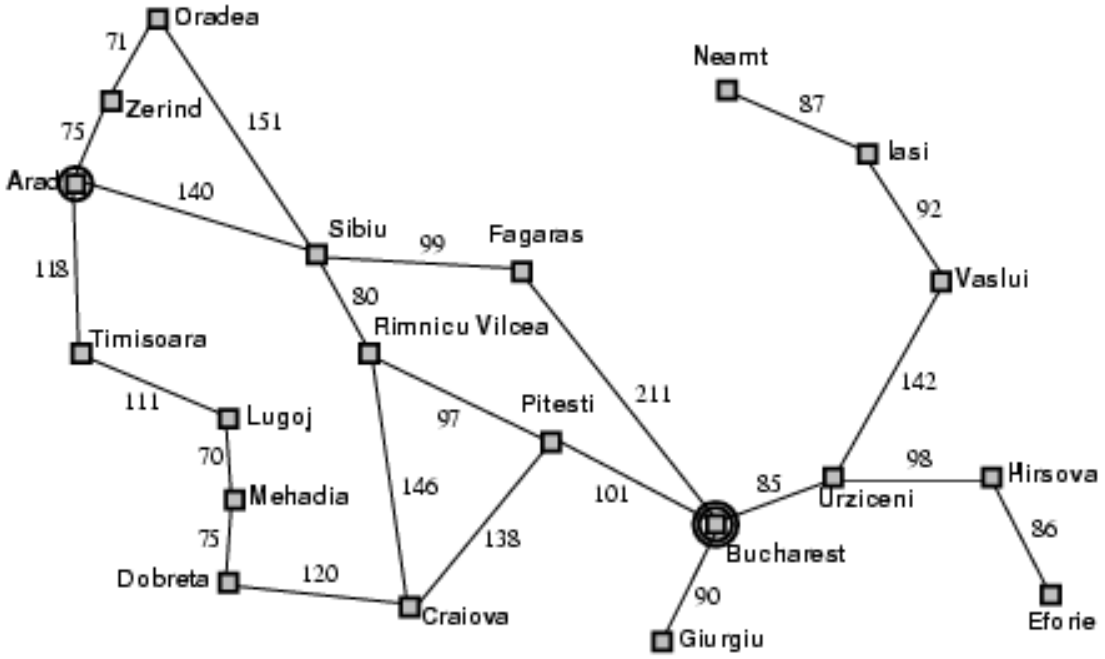
On holiday in Romania; currently in Arad; I want to reach Bucharest [Russell&Norvig, pp. 67-70]

Problem formulation

- State representation: be at various cities (Arad, Zerind, Fagaras, Sibiu, Bucharest …).
- Initial state: I am in Arad
- Goal state: be at Bucharest
- Actions: drive between cities
- Find solution: sequence of cities from Arad to Bucharest; e.g. Arad, Sibiu, Fagaras, Bucharest,
- Path cost: number of km. from Arad to Bucharest (sum of the km. of each driving action)

State space forms a graph

- nodes represent the problem states
- arcs represent the problem actions



2. Searching for Solutions

A solution is an action sequence, so search algorithms work by considering various possible action sequences

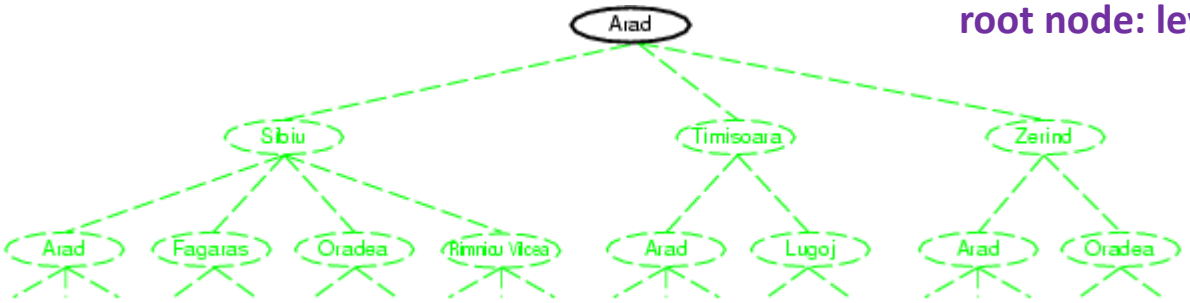
The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem

General Search Process

1. current node <- initial state
2. Check whether current node is a goal state; if so, finish
3. Apply each legal action to the current state, thereby generating a new set of states; we add the branches from the parent node to the new child node
4. Choose next node from the frontier (nodes which are not yet expanded)
5. Start all over again from step 2

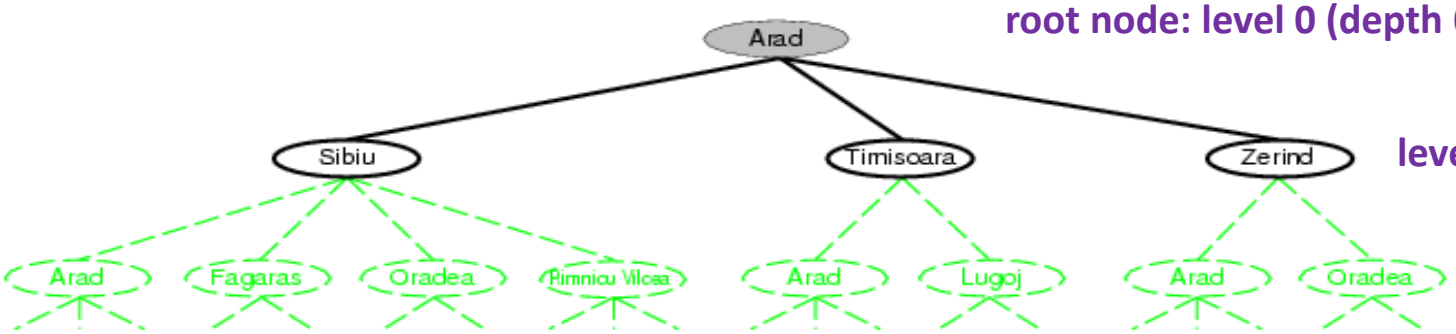
Frontier also called *leaf nodes*, OPEN LIST or fringe.

2. Searching for solutions: tree-search example



root node: level 0 (depth 0)

OPEN={Arad} ⇨ Pick a node and remove it from OPEN (expansion of the node) ⇨ Arad goal?: NO ⇨ Generate children

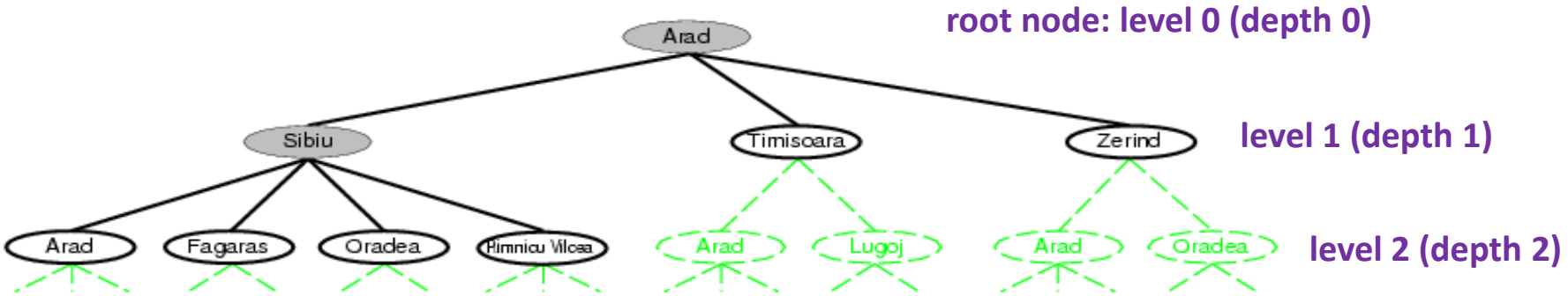


root node: level 0 (depth 0)

level 1 (depth 1)

OPEN={Sibiu Timisoara Zerind} ⇨ Pick a node and remove it from OPEN (expansion of the node) ⇨ Sibiu goal?: NO ⇩ Generate children

2. Searching for solutions: tree-search example



OPEN = {Timisoara Zerind Arad Fagaras Oradea Rimnicu Vilcea}

2. Searching for Solutions: tree-search algorithm

- A search tree is the set of nodes generated during search
- Search algorithms all share the basic structure seen before; they vary primarily according to how they choose which state to expand next – the so-called search strategy

function TREE-SEARCH (*problem*) **return** a solution or failure
    Initialize the OPEN LIST using the *initial state* of the *problem*
    do
        if the OPEN LIST is empty **then return** *failure*
        expand a node: choose a leaf node for expansion and remove it from the OPEN LIST
        if node contains goal state **then return** the corresponding *solution*
        generate children and add the resulting nodes to the OPEN LIST
    enddo

search-strategy

2. Searching for Solutions: repeated states

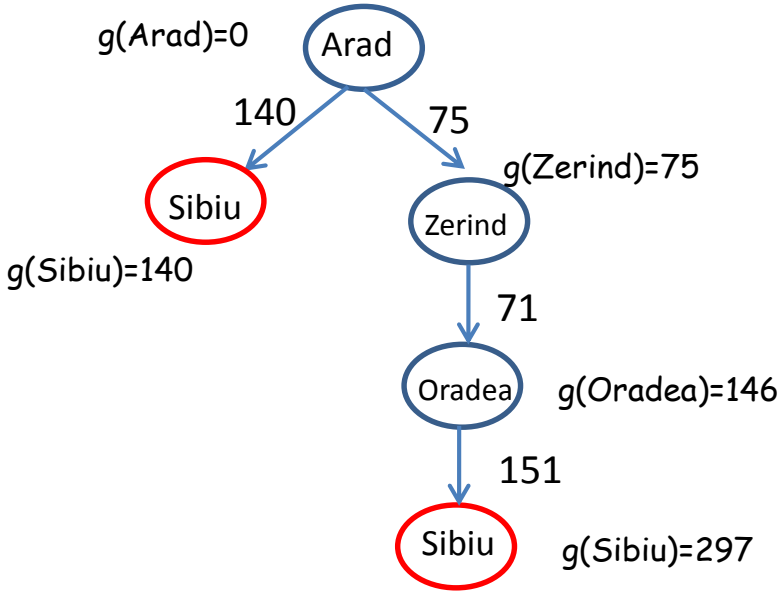
Search space ≠ State Space (search tree may include repeated states and loopy paths); e.g.: the state space of the Romania example has only 20 states whereas the search tree contains multiple repeated states and loopy paths (the complete search tree is infinite)

- Repeated states are due to:
  - reversible actions: loopy paths

repeated node:
ancestor node
(grandparent)

repeated node: ancestor node in the path

- redundant paths: more than one way to get from one state to another



2. Searching for Solutions: repeated states

Since we are concerned about reaching the goal, there is no reason to keep more than one path to any given state because any goal that is reachable by extending one path is also reachable by extending the other.

How to avoid repeated states:

- remember where one has been,
- augment the TREE-SEARCH algorithm with a data structure called *explored set* (CLOSED LIST) which remembers every expanded node,
- newly generated nodes that match a node in the CLOSED LIST (already expanded) or in the OPEN LIST (generated but not yet expanded) can be discarded instead of being added to the OPEN LIST,
- new algorithm: GRAPH-SEARCH algorithm, which separates the state-space graph into the explored/expanded region and the unexplored/unexpanded region.

In general, avoiding repeated states and redundant paths is a matter of efficiency.

2. Searching for solutions: GRAPH-SEARCH algorithm

The OPEN list is implemented as a priority queue:

- elements are always ordered in the queue according to some ordering function (in increasing order, function minimization)
- nodes with lower values go to the front of the queue; nodes with higher values go to the back of the queue
- we always pop the element of the queue with the highest priority (lowest value of the ordering function)

For each search strategy, we define an evaluation function ($f(n)$) that returns a numerical value for each node n such that nodes are inserted in the queue in the same order as they would be expanded by the search strategy (increasing order of $f(n)$) .

2. Searching for solutions: GRAPH-SEARCH algorithm

function GRAPH-SEARCH (*problem*) **return** a solution or a failure
    Initialize the OPEN list with the *initial state* of the problem
    Initialize the CLOSED list to empty
    do
        if OPEN is empty **then return** *failure*
        p <- pop (OPEN list)
        add p to CLOSED list
        if p = *final state* **then return** *solution* p
        generate the children of p
        for each child n of p:
            apply f(n)
            if n is not in CLOSED **then**
                if n is not in OPEN or (**n** is a repeated node in OPEN and f(n) is lower than the value of the node in OPEN) **then**
                    insert **n** in increasing order of f(n) in OPEN*
            else if f(n) is lower than the value of the repeated node in CLOSED
                then choose between re-expanding **n** (insert it in OPEN and eliminate it from CLOSED) or discard **n**
    enddo

* Since we only want to find the first solution, the repeated node in OPEN can be eliminated

2. Searching for Solutions: GRAPH-SEARCH algorithm

Search in a tree:

- maintains the OPEN LIST but not the CLOSED LIST (less storage requirements)
- can only avoid repeated states in the OPEN LIST (if we are only concerned about reaching the goal)
- re-expands nodes already explored (waste of time and memory repeating the same search)

Search in a graph:

- maintains the OPEN LIST and CLOSED LIST (bigger memory requirements)
- control of all repeated states, avoids redundant paths (exponential reduction in search)

2. Searching for Solutions

We will evaluate search strategies performance in four ways:

1. Completeness: is the strategy guaranteed to find a solution when there is one?
2. Time complexity: how long does it take to find a solution?
3. Space complexity: how much memory does it need to perform the search?
4. Optimality: does the strategy find the highest-quality solution when there are several solutions? solution with the lowest path cost?

We might be interested in finding trade-off solutions between:

- cost of the solution path: sum of the costs of the individual actions along the path ($g(goal\_state)$) (optimal path, close to optimal, far from optimal …)
- search cost: cost to find the solution path

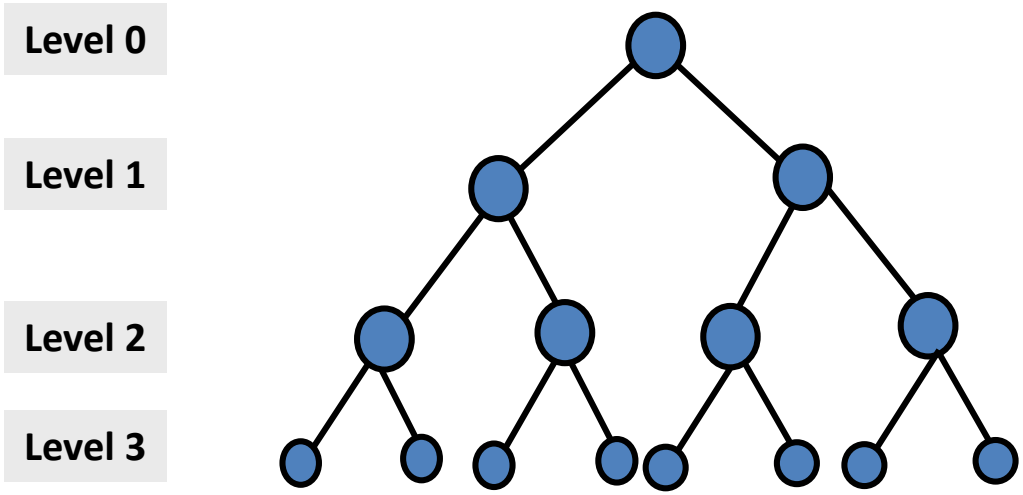
TOTAL COST is a combination of the search cost and the path cost of the solution found

Types of search strategies

1. uninformed or blind search (expansion order of nodes)
2. informed or heuristic search ("intelligent" expansion of nodes)

3. Breadth-first search

Breadth-first: expand the shallowest unexpanded node



Priority queue (OPEN list):

which ordering function can we use to insert the shallowest nodes at the front of the queue? which evaluation function can we use to make shallower nodes take on lower values?

f(n)=depth(n)=level(n)

3. Breadth-first search

- Complete
- Optimal :
  - breadth-first always returns the shortest solution path (shallowest goal node)
  - shortest solution is optimal if all actions have the same cost and the path cost is a non-decreasing function of the depth of the node (non-negative action costs)

- Time complexity (execution time): for a branching factor b and depth d:

  expanded nodes $1 + b + b^2 + b^3 + b^4 + \ldots\ldots\ldots + b^d$ $O(b^d)$

  generated nodes $1 + b + b^2 + b^3 + b^4 + \ldots\ldots\ldots + b^d + (b^{d+1} - b)$ $O(b^{d+1})$

- Space complexity (memory requirements): for a branching factor b and depth d:
  - for breadth-first GRAPH-SEARCH in particular, every node generated remains in memory
  - there will be $O(b^d)$ in the CLOSED LIST and $O(b^{d+1})$ in the OPEN LIST (dominated by the size of the OPEN LIST)

- Take-home lessons:
  - the memory requirements are a bigger problem for breadth-first than is the execution time
  - time requirements are still a major factor

3. Breadth-first search

Time and memory requirements for breadth-first search

b = 10
100000 nodes generated/sec
1000 bytes of storage/node

(Figures taken from the 3rd edition of the *Artificial Intelligence. A modern approach textbook**)

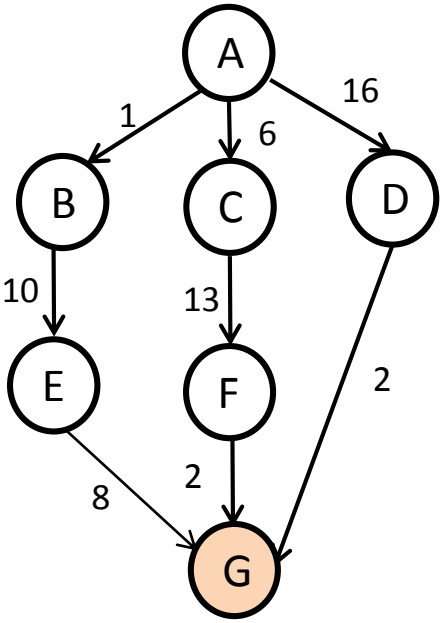
Depth	Nodes	Time	Memory
2	110	1.1 ms	107 kilobytes
4	11110	111 ms	10.6 megabytes
6	10^6	11 sec.	1 gigabytes
8	10^8	19 min.	103 gigabytes
10	10^{10}	31 hours	10 terabytes
12	10^{12}	129 days	1 petabytes
14	10^{14}	35 years	99 petabytes
16	10^{16}	3500 years	10 exabytes

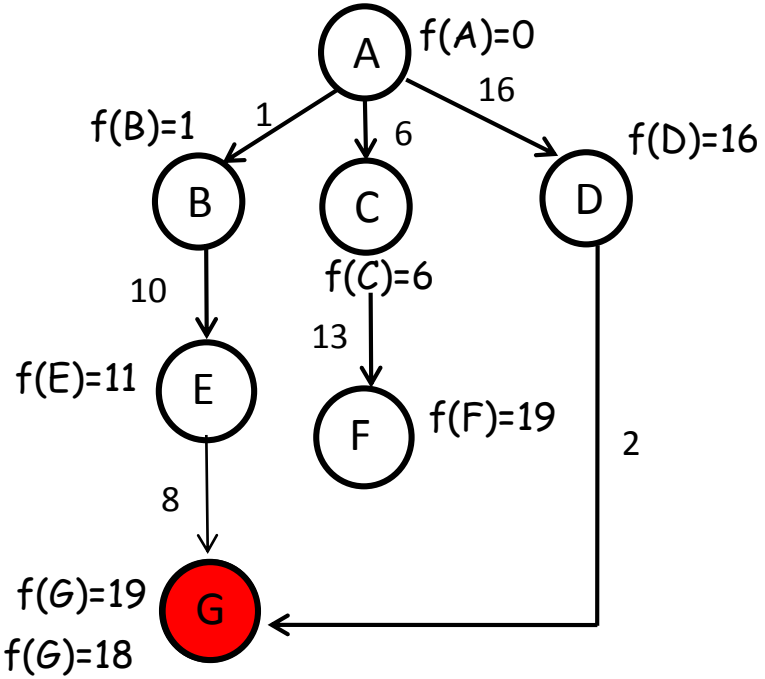
* This table is slightly different in the 1st and 2nd edition because different parameters are assumed

4. Uniform-cost search

Uniform cost: expand the unexpanded node with the lowest cost (lowest value of $g(n)$)

Evaluation function (priority queue, OPEN list): $f(n)=g(n)$





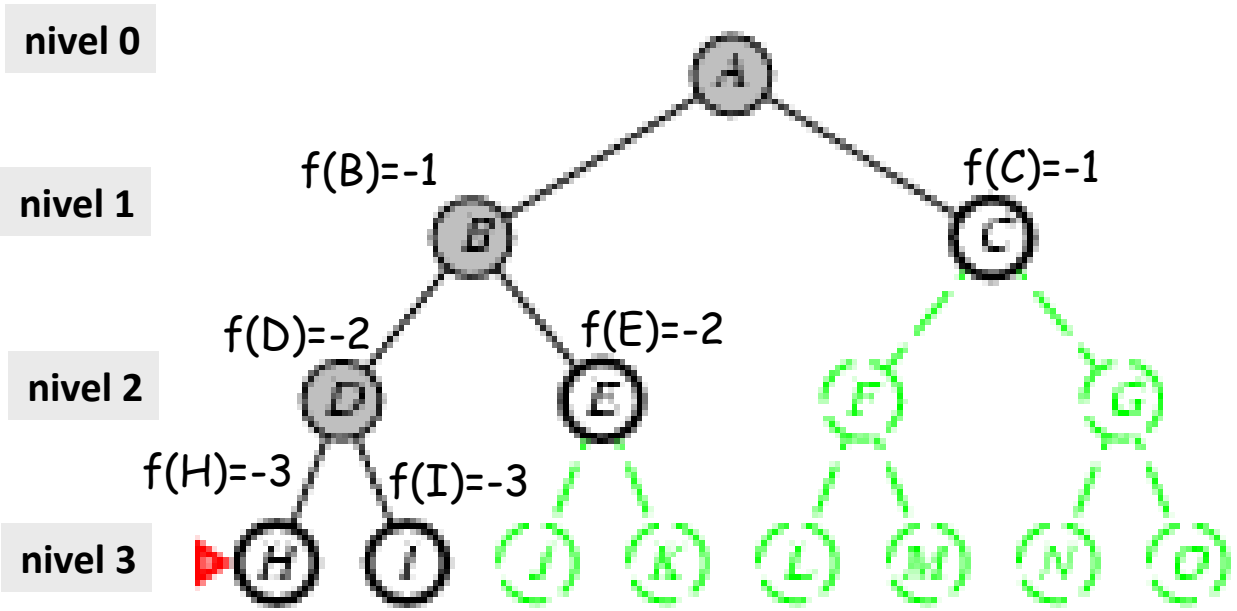
4. Uniform-cost search

- Complete
  - It is complete if action costs ≥ ε (small positive constant, non-zero costs)
- Optimal :
  - Optimal if action costs are non-negative $g(successor(n)) > g(n)$
  - Uniform-cost search expands nodes in order of their optimal path cost
- Time and space complexity (execution time and memory requirements):
  - let C^* be the cost of the optimal solution
  - assume that every action costs at least ε
  - the algorithm's worst-case time and space complexity is $O(b^{C^*/\varepsilon})$

5. Depth-first tree search

Depth-first: expand the deepest unexpanded node

Evaluation function (priority queue, OPEN list): f(n)=-level(n)



Backtracking :

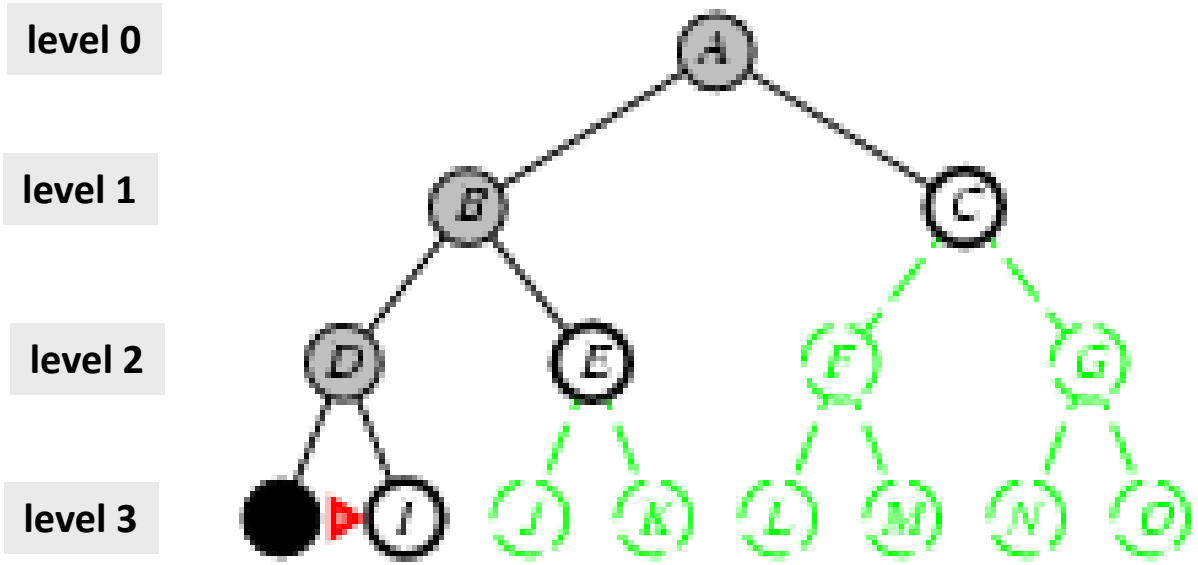
1. dead-end node, non-goal node with no applicable actions (no children)
2. set a depth limit (user-defined; maximum depth limit in this example m=3)
3. repeated state (optional, if control of repeated states)

A list called PATH is used to apply Backtracking: it only stores the expanded nodes of the current path and the nodes are released when the function *bactracking* is invoked

5. Depth-first tree search

BACKTRACKING(n):

1. Remove n from PATH
2. If parent(n) has more children in OPEN => select the next node in the OPEN list
3. If parent(n) has no more children in OPEN => BACKTRACKING (parent(n))



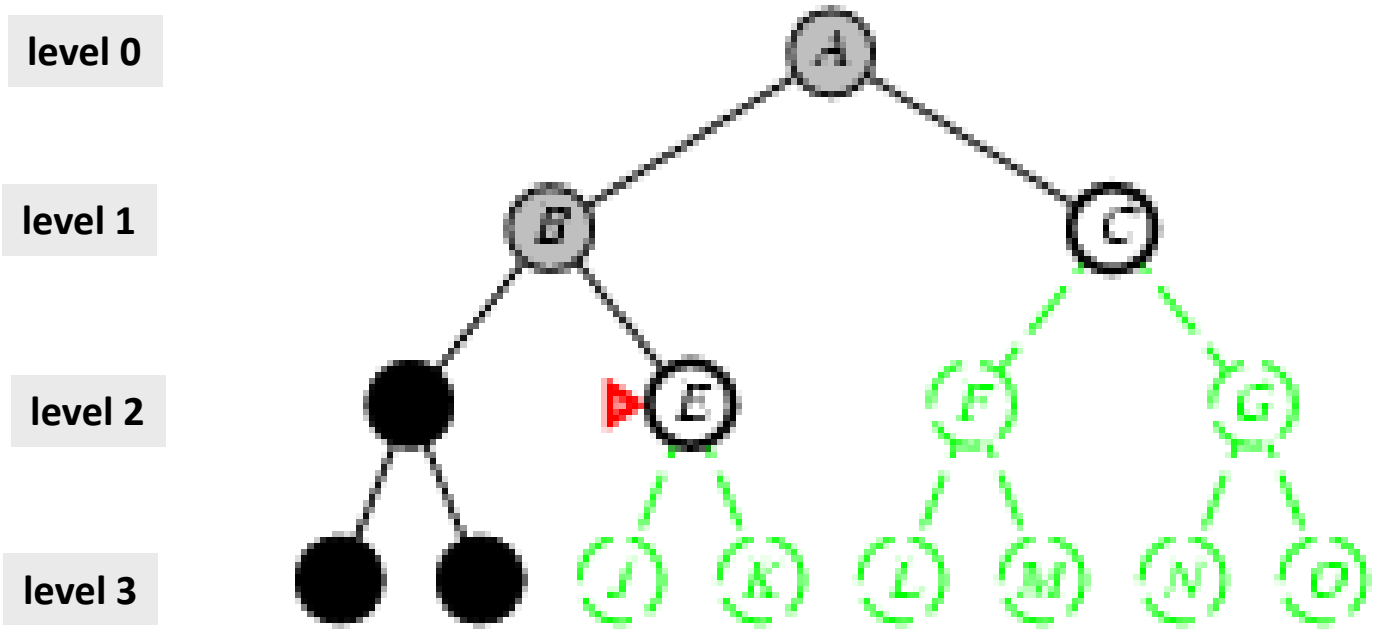
Remove node H from the OPEN list:

1. put H in the PATH list
2. check if H is a goal state: NO
3. check if H is at the maximum depth level (m=3): YES => BACKTRACKING (H)

OPEN list = {I(-3),E(-2),C(-1)}

PATH list = {A,B,D}

5. Depth-first tree search



Remove node I from the OPEN list:

1. put I in the PATH list
2. check if I is a goal state: NO
3. Check if I is at the maximum depth level (m=3): SI => BACKTRACKING (I)

OPEN list = {E(-2),C(-1)}

PATH list = {A,B}

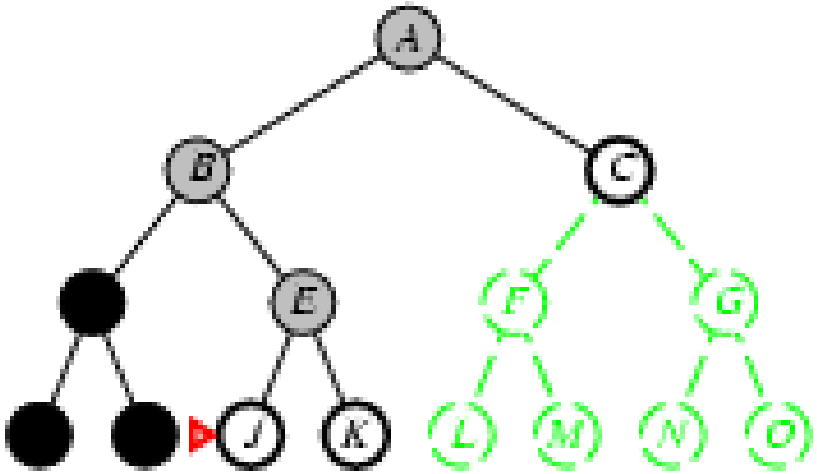
5. Depth-first search tree

level 0

level 1

level 2

level 3



f(J)=-3 f(K)=-3

OPEN list = {J(-3),K(-3),C(-1)}

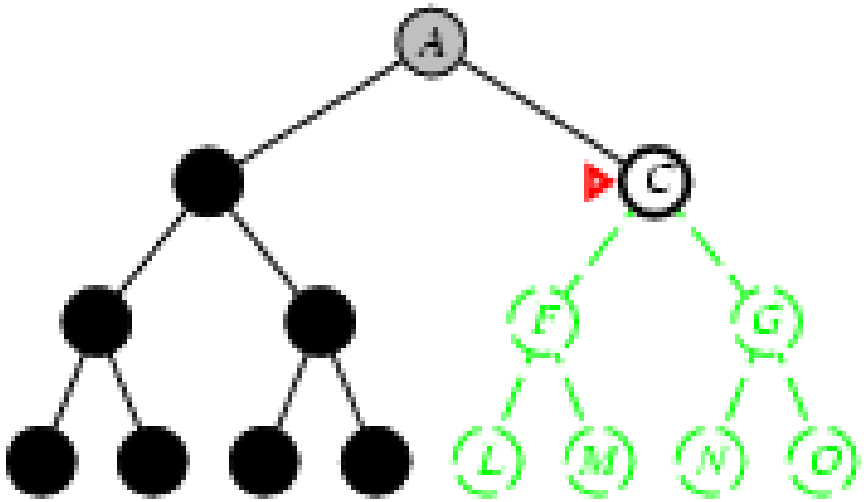
PATH list = {A,B,E}

level 0

level 1

level 2

level 3



OPEN list = {C(-1)}

PATH list = {A}

5. Depth-first tree search

- Time complexity (execution time):
  - For a bounded tree with maximum depth level **m**, $O(b^m)$
  - If **m=d** then depth-first explores as many nodes as breadth-first. But **m** itself can be much larger than **d**

- Space complexity (memory requirements):
  - The TREE-SEARCH version has very modest memory requirements. It only stores a single path from the root node to a leaf node (PATH list), along with the remaining unexpanded sibling nodes for each node on the path (OPEN list).
  - For a branching factor **b** and maximum depth level **m**, $O(b.m)$. **Linear space!!**
  - Very efficient handling of the OPEN and PATH lists (very few elements)
  - But there is hardly control of repeated states because lists contains very few elements. In practice, this means that depth-first generates the same node multiple times
  - Even though, the TREE-SEARCH version of depth-first may actually be faster than breadth-first
  - For a GRAPH-SEARCH version, there is no advantage of depth-first over breadth-first

5. Depth-first tree search

- Completeness:
  - If the tree is unbounded (no maximum depth level) and no control of repeated states, it can get stuck in a loop forever (no complete)
  - If the tree is unbounded and control of repeated states, it avoids infinite loops but does not avoid redundant paths. It will eventually find a solution (complete)
  - If the tree is bounded (maximum depth level) it might lose the solution if the solution is not within the search space defined by limit **m** (no complete)

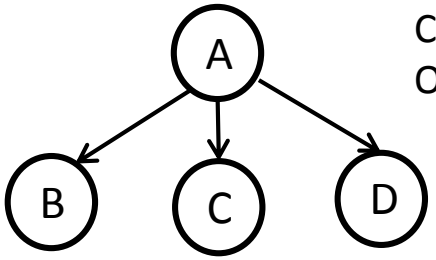
- No optimal

6. Search in CLIPS

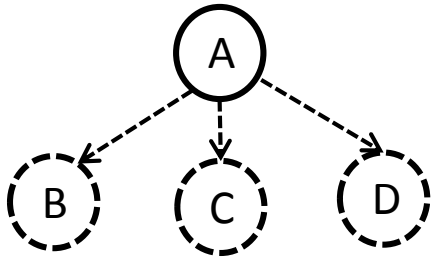
- See practice work for details on how to implement a search in CLIPS

SUMMARY:

- No control of repeated states in CLIPS (facts representing the same state are different due to the fields like *level*). Avoiding fact duplication does not avoid repeated states.
- Applicable rules = Generated nodes = OPEN LIST
- Executed rules = Expanded nodes = CLOSED LIST



CLOSED List= {A}
OPEN List={B, C, D}



WM= {A}
Agenda={B, C, D}

- Breadth-first: set the CLIPS Agenda to Breadth
- Depth-first: set the CLIPS Agenda to Depth.
- CLIPS does not implement de TREE-SEARCH versions but the GRAPH-SEARCH versions of Breadth-first and Depth-first.

7. Iterative deepening depth-first search

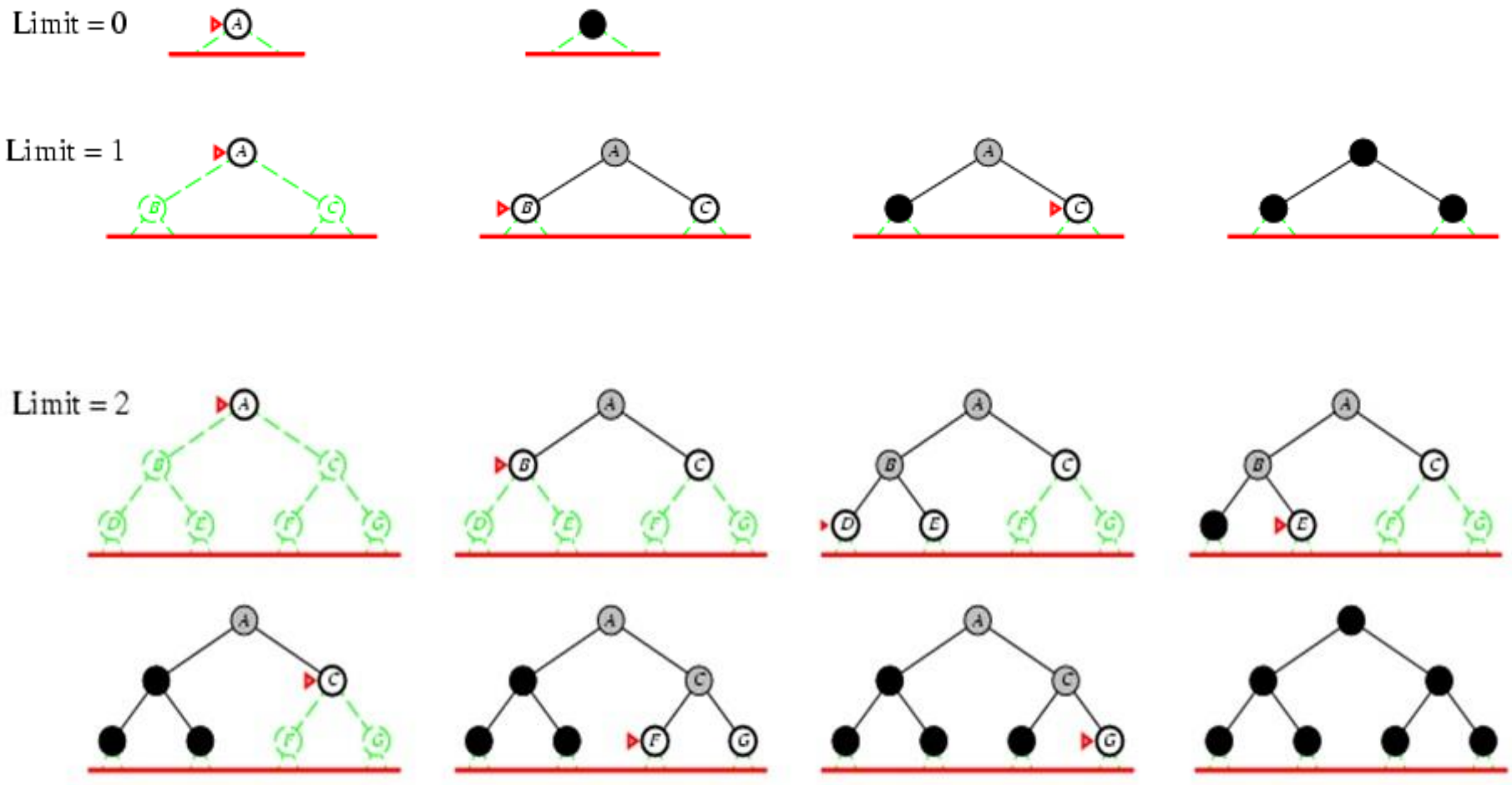
```
function Iterative_Deepening_Search (problem) returns (solution, failure)
    inputs: problem /*a problem*/
      for depth = 0 to ∞ do
          result = depth_limited_search (problem, depth)
          if result <> failure return result
      end
      return failure
end function
```

```
depth_limited_search  (problem, limit)
.....
   if  goal_test(node) then return SOLUTION(node)
        else  if depth(node) = limit then backtracking
                   else   generate_successors (node)
....
```

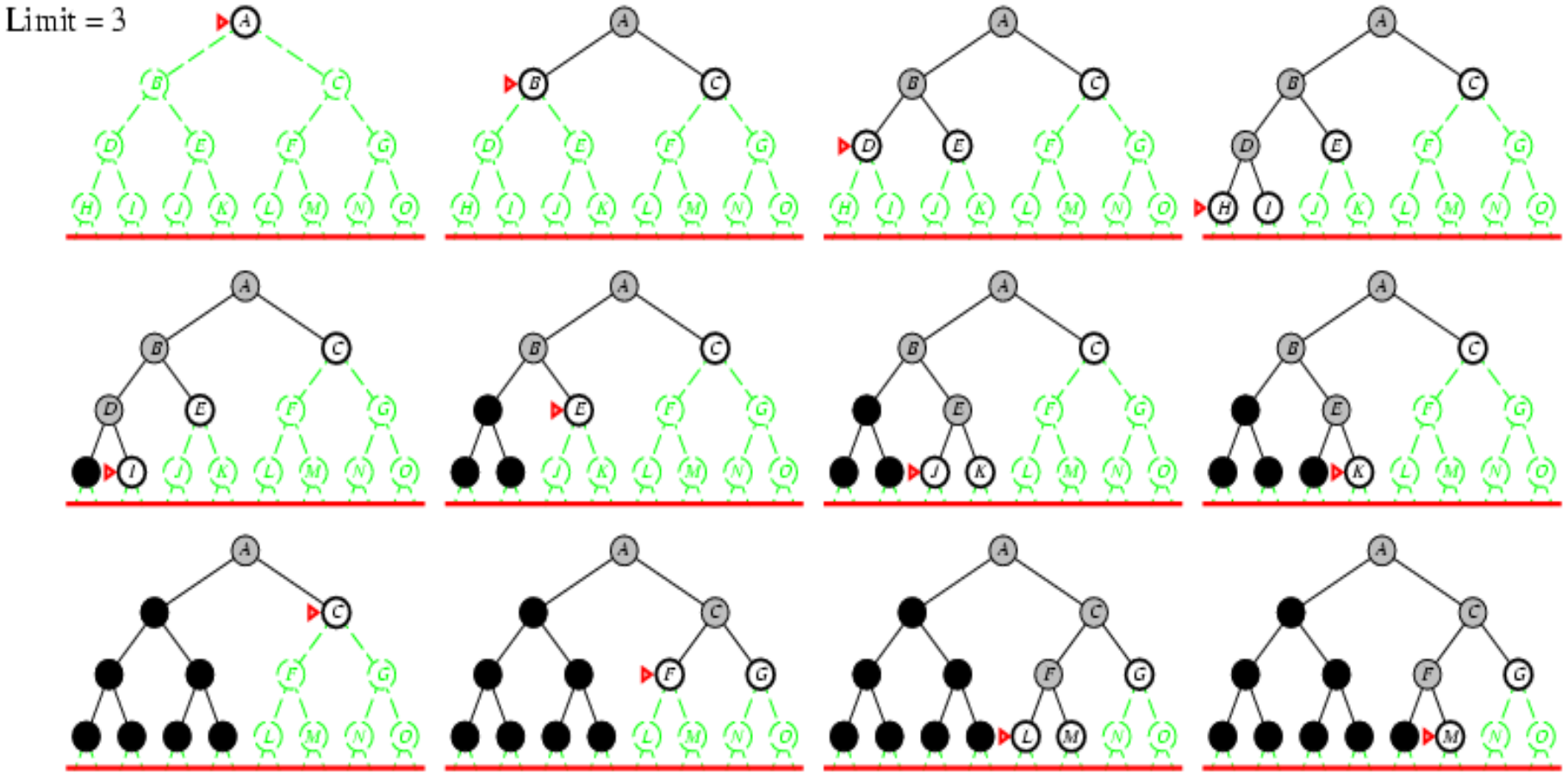
It ***iteratively*** performs a ***depth-limited search*** from depth-limit=0 to depth-limit=∞:

- solves the problem of picking a good depth limit in depth-first search
- combines the benefits of depth-first and breadth-first search.
- complete and optimal (under same conditions of breadth-first search).
- time complexity **$O(b^d)$**; space complexity **$O(b \cdot d)$**
- preferred search method when there is a large search space and the depth of the solution is not known.

7. Iterative deepening depth-first search



7. Iterative deepening depth-first search



7. Iterative deepening depth-first search (ID)

- Number of generated nodes for b=10, d=5:
  - Iterative deepening: $(d)\cdot b + (d-1)\cdot b^2 + (d-2)\cdot b^3 + \ldots\ldots\ldots + 1\cdot b^d$ 123.456
  - Breadth-first: $1 + b + b^2 + \ldots\ldots\ldots\ldots + b^{d-2} + b^{d-1} + b^d + (b^{d+1} - b)$ 1.111.100

- In breadth-first when a solution node is selected for expansion, the tree already contains nodes generated at the next level in depth (d+1). However, this does not occur in ID.

- ID may seem wasteful because states are generated multiple times. It turns out this is not so costly. The reason is that in a search tree with a similar branching factor at each level, most of the nodes are in the bottom level so it does not matter much that the upper levels are generated multiple times.

- ID is actually faster than breadth-first despite the repeated generation of states due to the high memory requirements of breadth-first and the difficulty to access the stores nodes.

- ID is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.

Summary of uninformed search

Criterion	Breadth	Uniform cost	Depth	Iterative Deepening
Temporal	$O(b^{d+1})$	$O(b^{C^*/\varepsilon})$	$O(b^m)$	$O(b^d)$
Espacial	$O(b^{d+1})$	$O(b^{C^*/\varepsilon})$	$O(b.m)$	$O(b.d)$
Optima?	Yes *	Yes	No	Yes *
Completa?	Yes	Yes **	No	Yes

* Optimal if step costs are all identical

** Complete if step costs $\geq \varepsilon$ for positive ε