Solving Problems by Searching

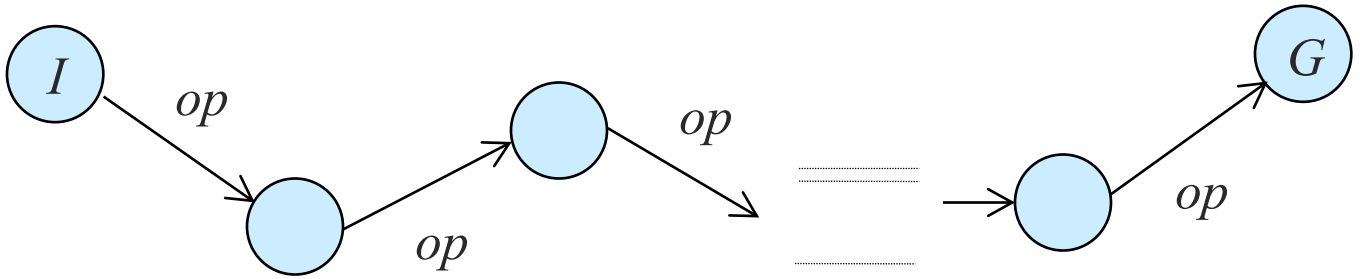
Outline

- What is Problem Solving?
- Example Problems
- Searching for Solutions
- Uninformed Search Strategies
  - Breadth-first Search
  - Uniform-cost Search
  - Depth-first Search
  - Depth-limited Search
  - Iterative Deepening Search
  - Bidirectional Search
  - Comparing Uninformed Search Strategies
  - Tree Search vs. Graph Search

What is Problem Solving?

- Problem Solving
  - "Problem solving is a search through a state space to find a sequence of operators (path) that can transform an initial state to a goal state"

Either one of these is a solution

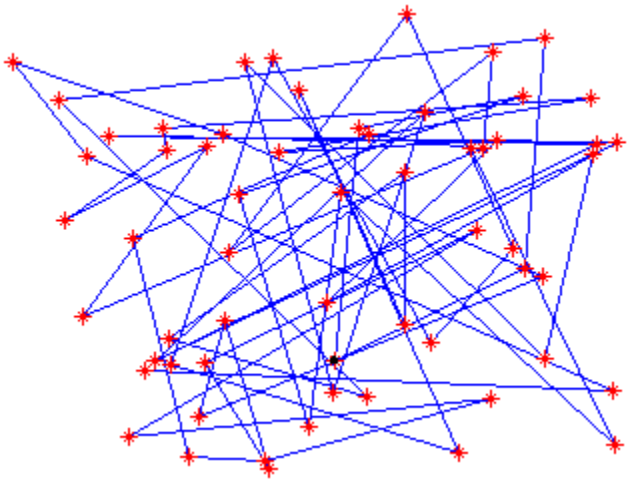


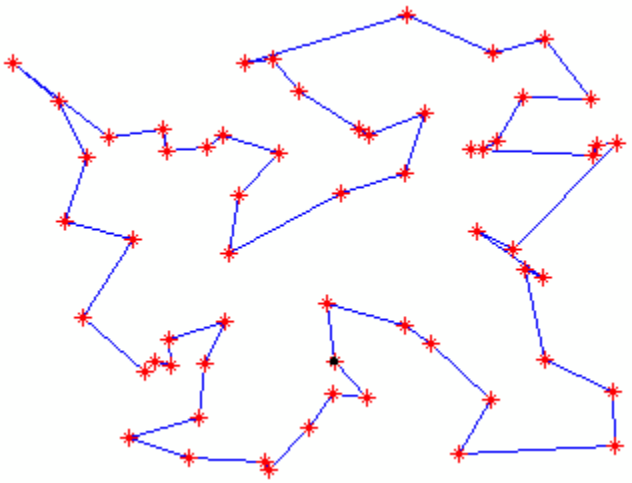
forward vs. backward

What is Problem Solving?

Example: Traveling Salesperson Problem:

The salesperson wants to minimize the total traveling cost required to visit all the cities in a territory, and return to the starting point





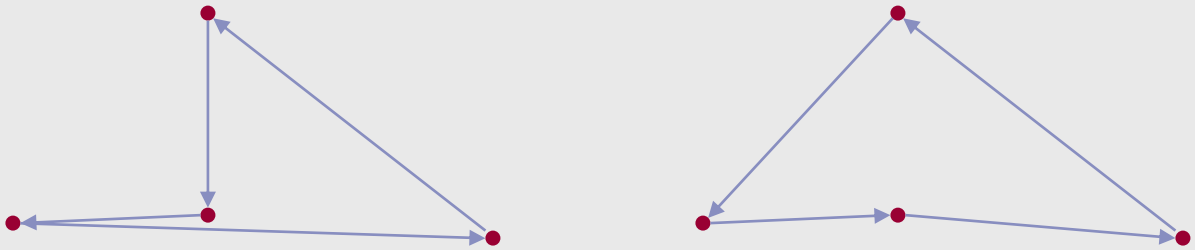
What is Problem Solving?

- Combinatorial Explosion: $O(n!)$

  $\rightarrow$ Efficiency is the issue!

- Nearest neighbor heuristic: $O(n^2)$
  - Can find a near optimal solution in a much shorter time

NN fails to find the best tour



The tour on the right costs less

What is Problem Solving?

- How thick would it be if you fold a newspaper 50 times?

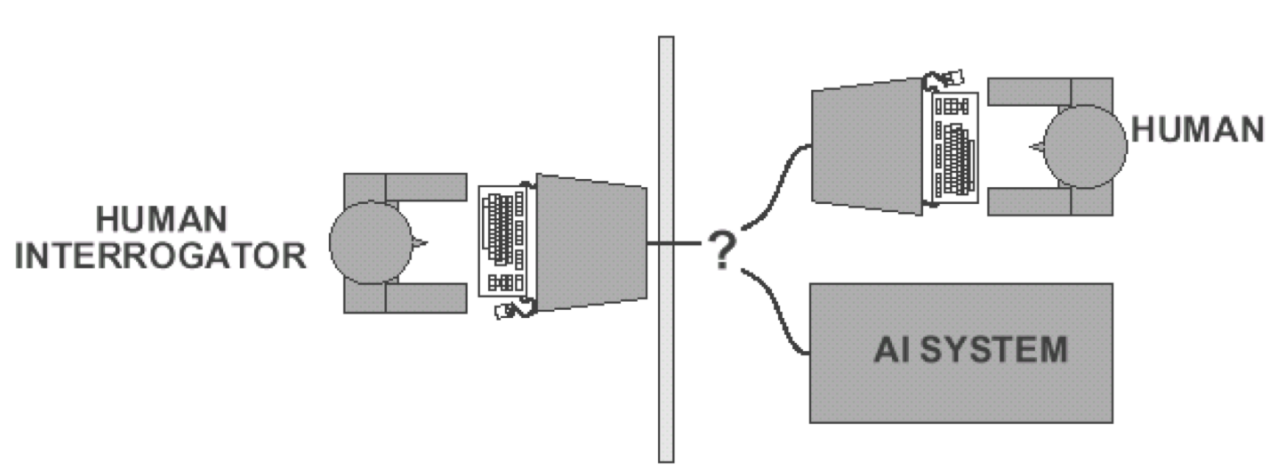
$$
\begin{aligned}
2^{50} \cdot 10^{-4} &= (2^{10})^5 \cdot 10^{-4} \\
&\approx (10^3)^5 \cdot 10^{-4} \\
&= 10^{15} \cdot 10^{-4} \\
&= 10^{11} \ \text{(m)}
\end{aligned}
$$

$1\text{AU} \approx 1.5 \cdot 10^{11}$ m

- Note: $n! >> 2^n$

What is Problem Solving?

- The Turing test (Alan Turing, 1950):
  - If the evaluator cannot reliably tell the machine from the human, the machine is said to have passed the test
  - The test does not check the ability to give correct answers to questions, only how closely answers resemble those a human would give



What is Problem Solving?

- To formalize or operationalize,

  (informal problem description → formal problem description)

  we need to define:

  - Initial (start) state
  - Operators (successor functions)

  } constitute the state space

  - Goal test
  - Path cost function: sum of the costs of the individual actions along the path
  - ➡ Input to the search algorithms
- Solution: a path from the initial state to a goal state
  - Optimal solution: a solution with the lowest path cost

Example Problems

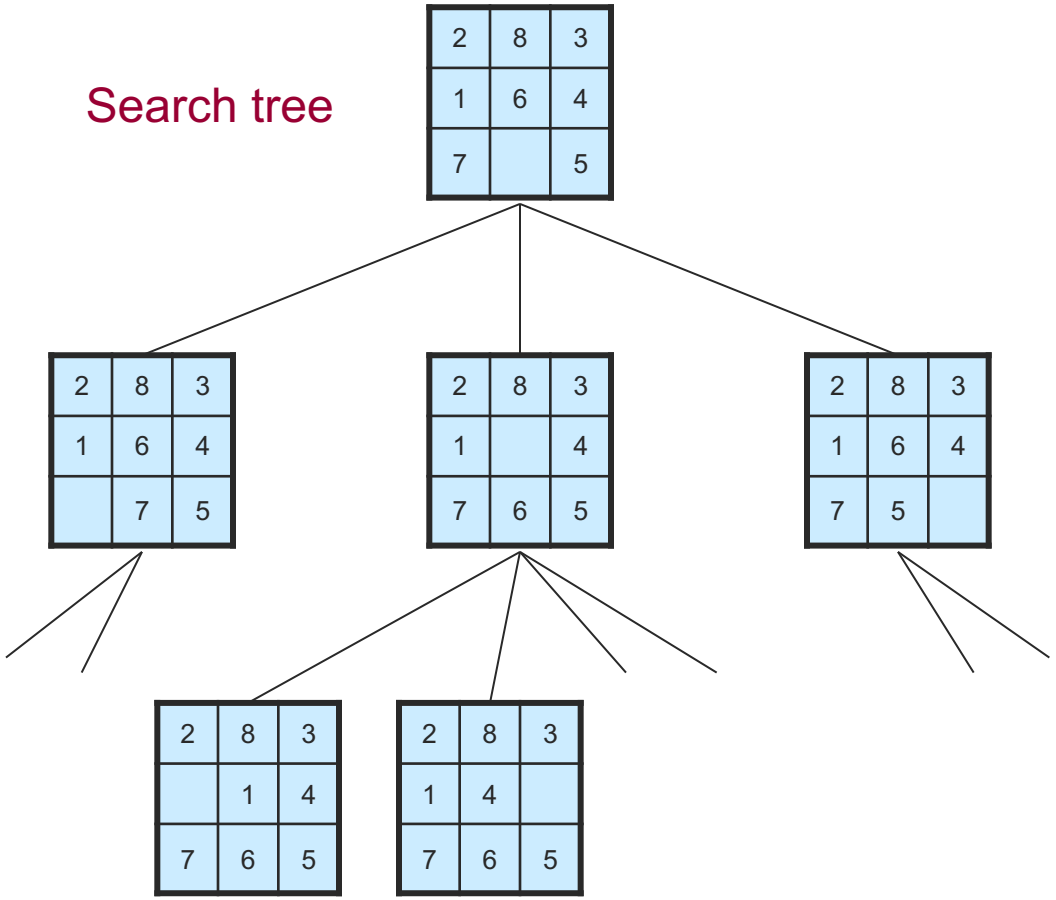
- The 8-puzzle:

2	8	3
1	6	4
7		5

initial state

1	2	3
8		4
7	6	5

goal state

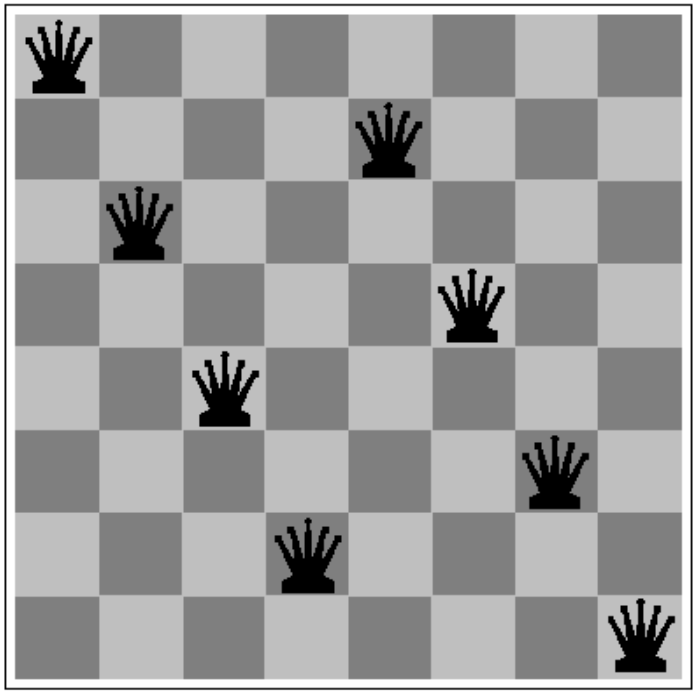


Example Problems

- The 8-puzzle:
  - States:
    - the location of each of the eight tiles in one of the nine squares
  - Operators:
    - blank moves left, right, up, or down
  - Goal test:
    - state matches the goal configuration
  - Path cost:
    - each step costs 1

Example Problems

- The 8-queens problem:
  - States:
    - arrangements of 0 to 8 queens on the board
  - Successor function:
    - add a queen to any empty square
  - Goal test:
    - 8 queens on board, none attacked



➡ $64 \cdot 63 \cdot \cdots \cdot 57 \approx 3 \times 10^{14}$ possible sequences

Example Problems

- The 8-queens problem: A better formulation
  - States:
    - Arrangements of n queens ($0 \leq n \leq 8$), one per row in the topmost n rows, with no queen attacking another
  - Successor function:
    - Add a queen to any square in the topmost empty row such that it is not attacked by any other queen
  - ➡ The state space is reduced from 3×10^{14} to $2{,}057$

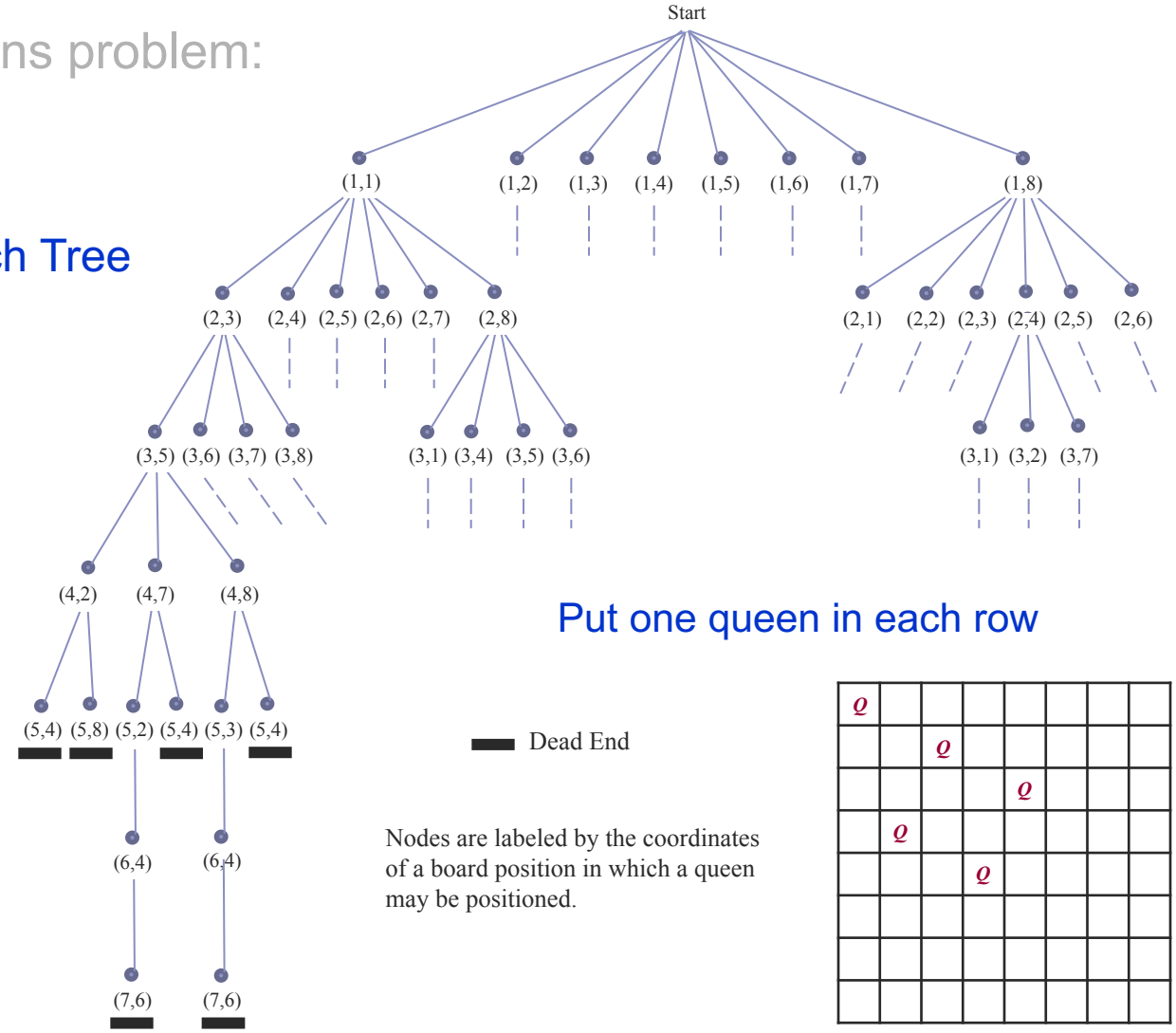
For 100 queens, the reduction is from 10^{400} to 10^{52}

➡ Huge reduction but still too big!

Example Problems

- The 8-queens problem:

Search Tree



Put one queen in each row

Dead End

Nodes are labeled by the coordinates of a board position in which a queen may be positioned.

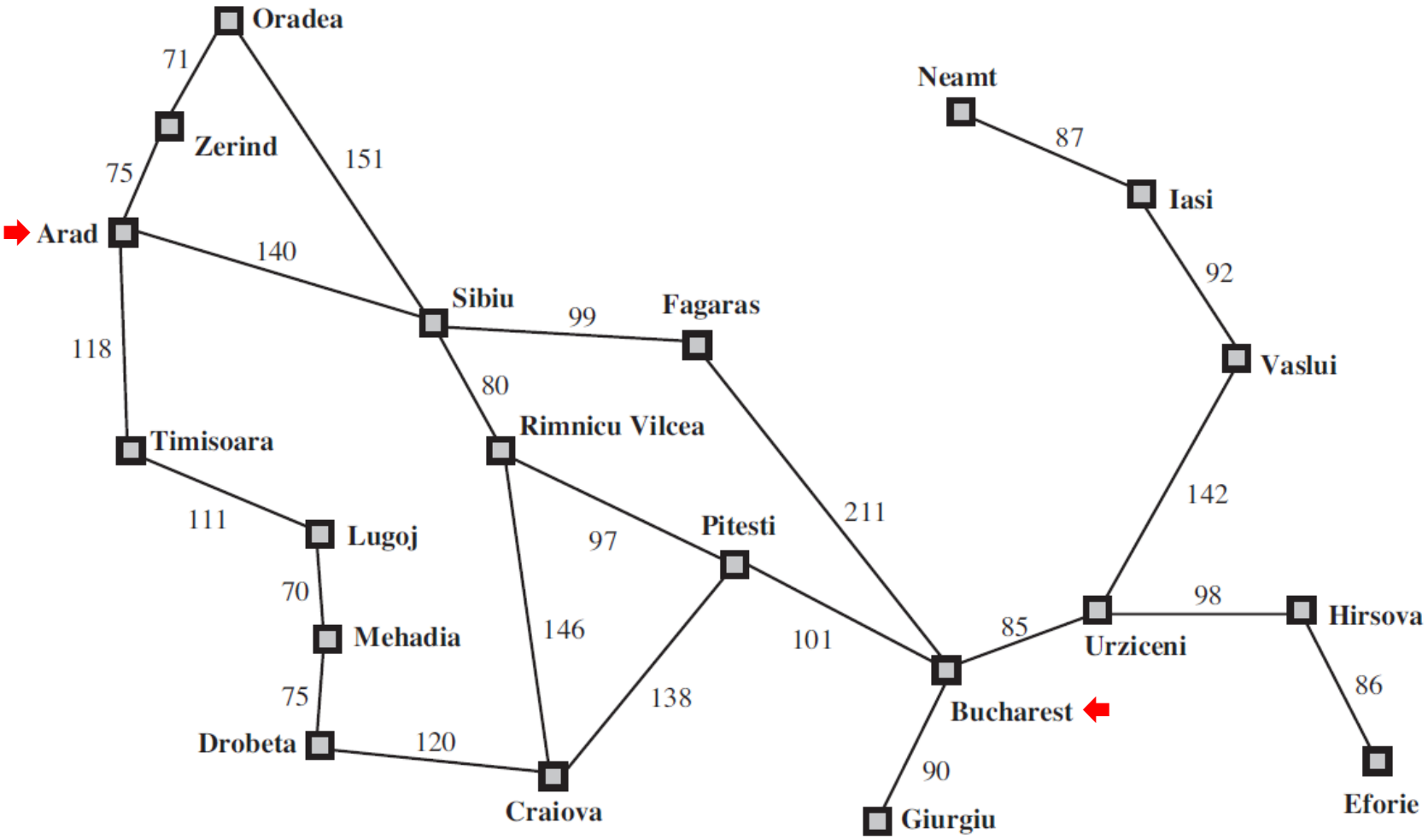
Example Problems

- Real World Problems:
  - Precision Medicine (Medical Diagnosis)
  - Discovery of new drugs
  - Natural Language Processing
  - Financial analysis
  - Robot navigation
  - Perception
    - Speech recognition
    - Computer vision
  - VSLI layout
    - Cell layout
    - Channel routing

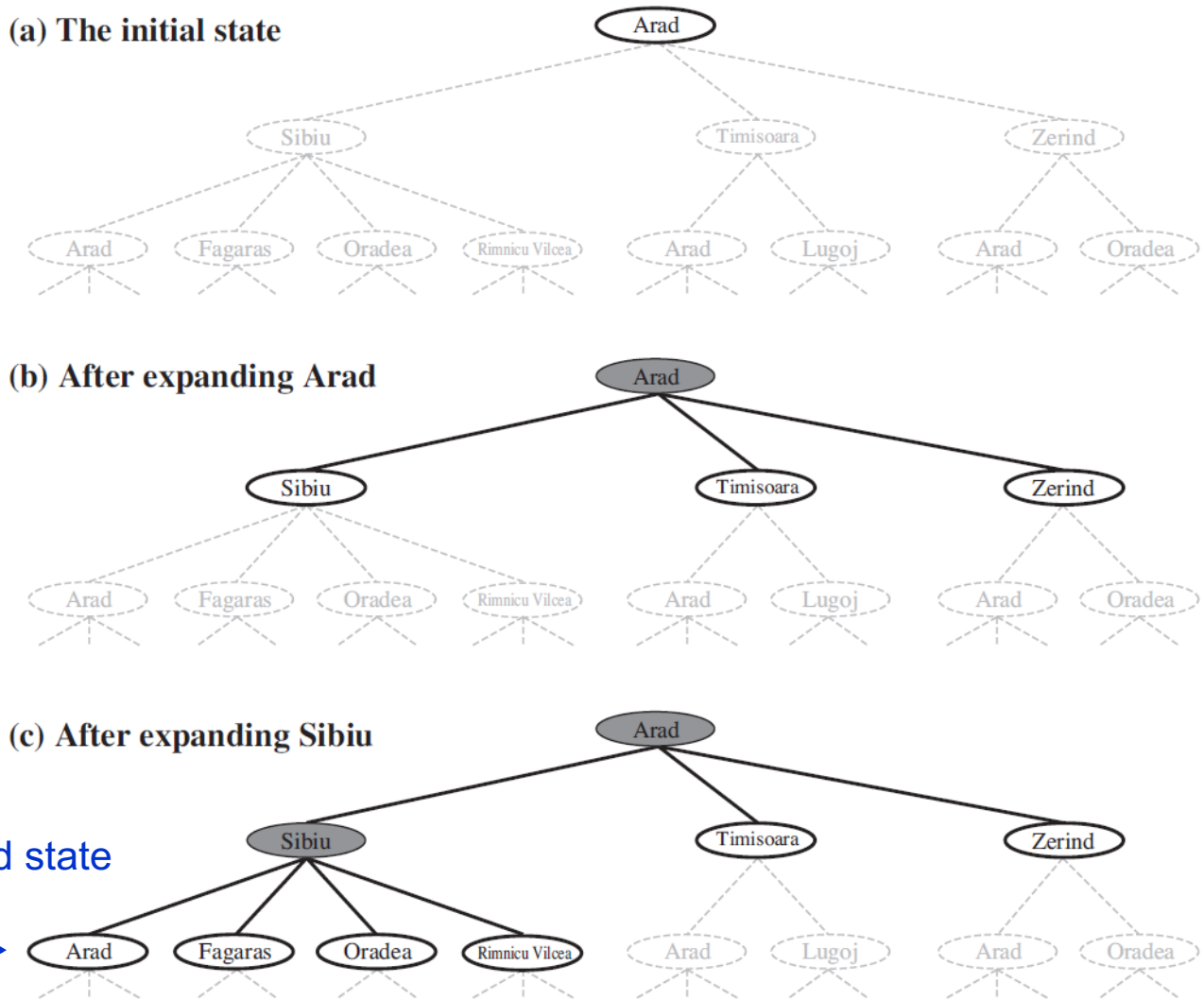
Searching for Solutions

- Considering various possible action sequences:
  - We form a search tree with the initial state at the root
    - The branches are actions
    - The nodes correspond to states in the state space of the problem
  - We apply operators to the current state (expanding the state), thereby generating a new set of states
    - Parent and child nodes
  - The collection of leaf nodes waiting to be expanded is called the frontier (aka open list)

Searching for Solutions



Searching for Solutions



Searching for Solutions

function TREE-SEARCH (*problem*) **returns** a solution, or failure

initialize the frontier using the initial state of *problem*

loop do

if the frontier is empty **then return** failure

choose a leaf node and remove it from the frontier

if the node contains a goal state **then return** the corresponding solution

expand the chosen node, adding the resulting nodes to the frontier

Search algorithms vary according to how they choose which state to expand next (search strategy)

Searching for Solutions

- Repeated states:
  - *Arad* is a repeated state in the previous search tree
    - A repeated state is generated by a loopy path causing an infinite loop
    - A loopy path is a special case of redundant paths
  - Redundant paths can be avoided by remembering every expanded node in the explored set (aka closed list)
    - Newly generated nodes are discarded if they match previously generated nodes—ones in the explored set or the frontier
    - ➡ GRAPH-SEARCH algorithm

Searching for Solutions

function GRAPH-SEARCH (*problem*) **returns** a solution, or failure

initialize the frontier using the initial state of *problem*

initialize the explored set to be empty

loop do

if the frontier is empty **then return** failure

choose a leaf node and remove it from the frontier

if the node contains a goal state **then return** the corresponding solution

add the node to the explored set

expand the chosen node, adding the resulting nodes to the frontier

only if not in the frontier or explored set

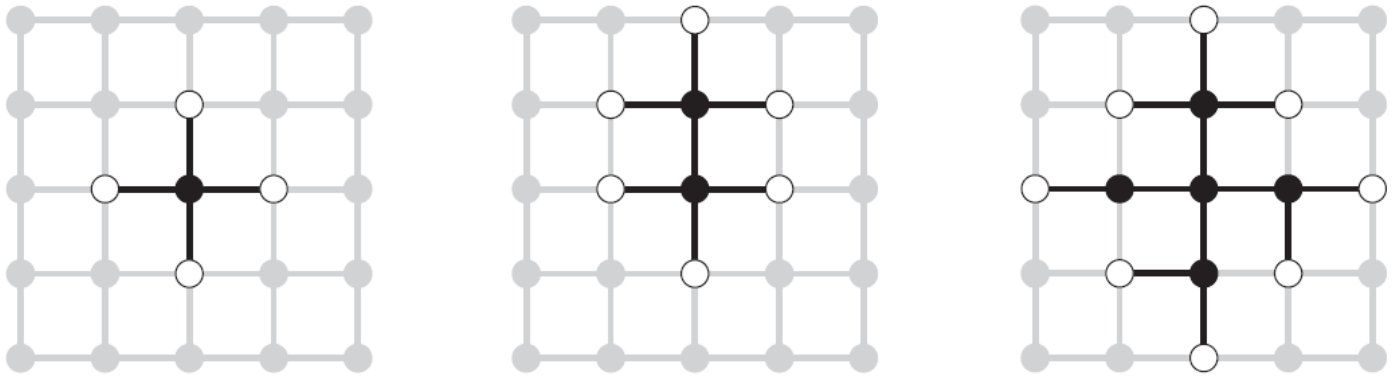
Searching for Solutions

- Repeated states:
  - The search tree constructed by GRAPH-SEARCH contains at most one copy of each state
    - The frontier separates the state-space graph into the explored region and the unexplored region
    - ➡ Every path from the initial state to an unexplored state has to pass through a state in the frontier

Searching for Solutions

At each state, each path is extended by one step

Searching for Solutions

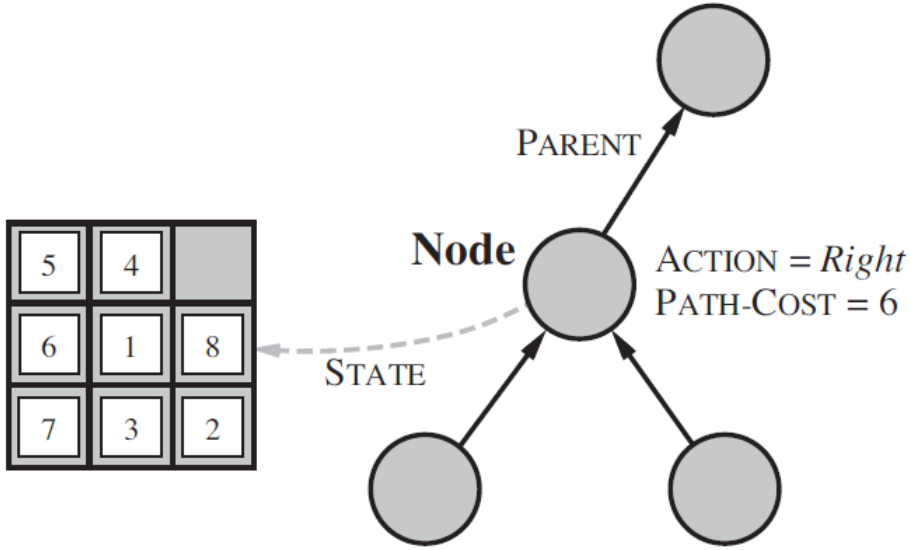


(a) (b) (c)

- The separation property of GRAPH-SEARCH:
  - (a) Just the root has been expanded
  - (b) One leaf node has been expanded
  - (c) The remaining successors of the root has been expanded in clockwise order

Searching for Solutions

- Infrastructure for search algorithms:
  - A node n is a structure of four components:
    - n.STATE
    - n.PARENT
    - n.ACTION
    - n.PATH-COST



Searching for Solutions

- Infrastructure for search algorithms:

```
function CHILD-NODE (problem, parent, action) returns a node
    return a node with
        STATE = problem.RESULT(parent.STATE, action)
        PARENT = parent
        ACTION = action
        PATH-COST = paernt.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

- The function CHILD-NODE takes a parent node and an action and returns the resulting child node
- The function SOLUTION returns the sequence of actions obtained by following parent pointers back to the root

Searching for Solutions

- Infrastructure for search algorithms:
  - The frontier is stored in a queue
    - FIFO queue
    - LIFO queue (stack)
    - Priority queue
  - The explored set can be implemented with a hash table to allow efficient checking for repeated states

Searching for Solutions

- Evaluation criteria (measures of problem-solving performance):
  - Completeness: Guaranteed to find a solution?
  - Optimality: Does it find the optimal solution?
  - Time complexity: How long does it take?
    - Number of nodes generated
  - Space complexity: How much memory does it need?
    - Maximum number of nodes stored in memory

Searching for Solutions

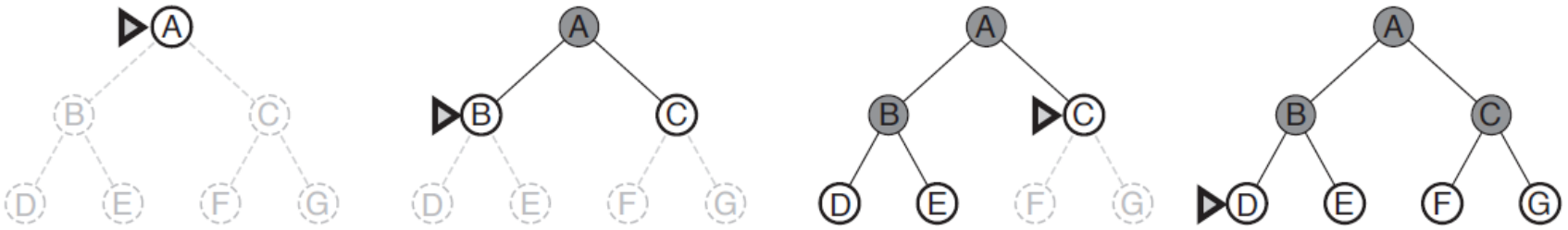
- Evaluation criteria (measures of problem-solving performance):
  - Since the graph is often represented implicitly and is frequently infinite, complexity is expressed in terms of
    - b, the branching factor or maximum number of successors of any node
    - d, the depth of the shallowest goal node
    - m, the maximum length of any path in the state space (may be ∞)

Uninformed Search Strategies

- Uninformed search (blind search)
  - No information about the number of steps or the path cost from the current state to the goal
- Informed search (heuristic search)
  - Knows which state is more promising
  - More efficient

Breadth-first search

- Expand the shallowest unexpanded node in the frontier
- Implementation:
  - Frontier is a FIFO queue, i.e., new successors go at end
  - TREE-SEARCH-based implementation?
    - O.K., but less efficient



[A] [B, C] [C, D, E,] [D, E, F, G]

Breadth-first search on a graph

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

Breadth-first search

- Evaluation:
  - Complete and optimal
    - Optimal only when all operators have the same cost
  - Time & memory complexity
    - Branching factor: b
    - Solution path length: d
    - The total number of nodes generated before finding a solution in the worst case

      $$b + b^2 + b^3 + \ldots + b^d \qquad O(b^d)$$

      If the goal test is applied to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected $\rightarrow O(b^{d+1})$
    - ➡ Exponential time and memory complexity

Breadth-first search

Number of nodes generated

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

($b = 10$; 10^6 nodes/second; 1,000 bytes/node)

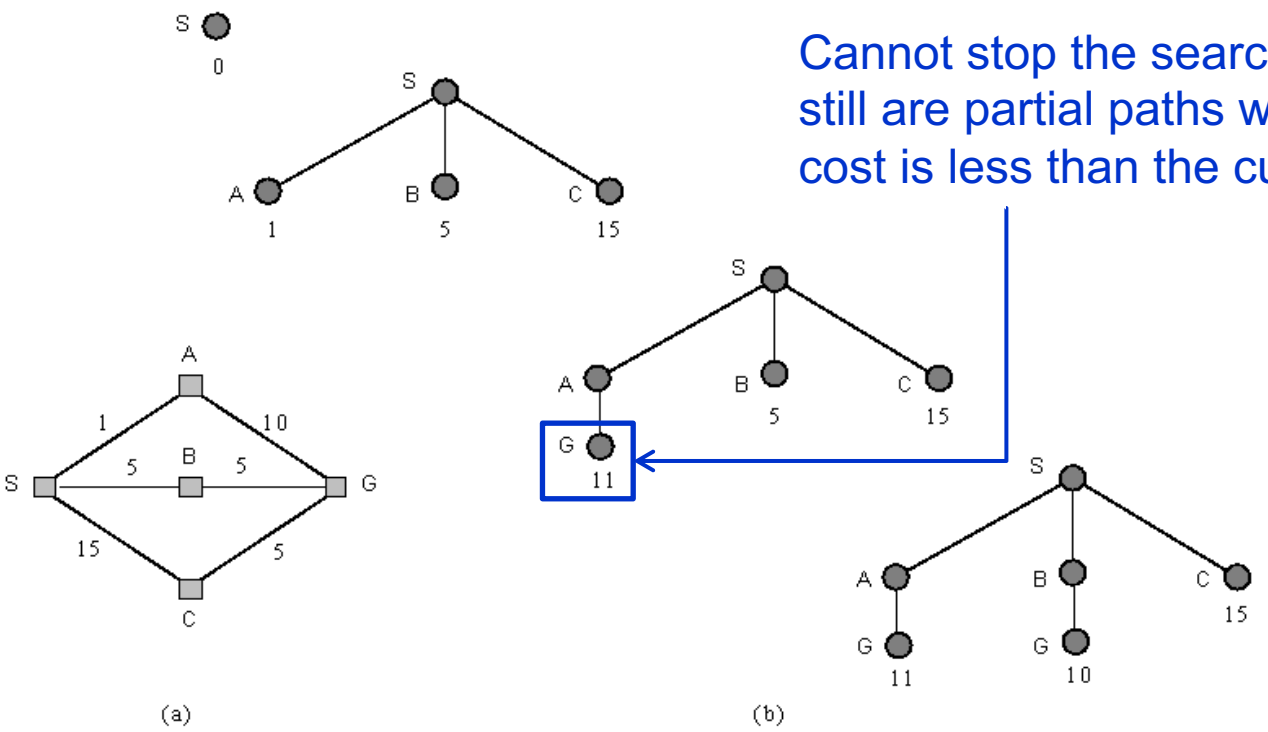
- Memory requirements are a bigger problem
- Blind search cannot solve exponential problems

Uniform-cost search

- Expand the least-cost unexpanded node on the frontier rather than the shallowest node
- Implementation:
  - Frontier is a priority queue ordered by path cost, lowest first (allows redundant path if better than old)
  - TREE-SEARCH-based implementation?
    - O.K. but less efficient
- Finds the least-cost optimal solution rather than the shallowest goal state, as long as

  $path\text{-}cost(Successor(n)) \geq path\text{-}cost(n)$ (i.e., step cost $\geq \varepsilon > 0$)
  - Can get stuck in an infinite loop (i.e., incomplete) if there is a path with an infinite sequence of zero-cost actions

Uniform-cost search



Cannot stop the search if there still are partial paths whose cost is less than the current

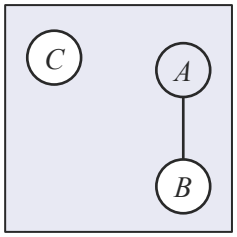
Uniform-cost search on a graph

```
function UNIFORM-COST-SEARCH (problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored        /* Goal test at this point guarantees least-cost goal */
    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
```

Uniform-cost search

- If a child node B generated by expanding a node A has the same state as a node C in the explored set, then B can be simply discarded because PATH-COST(C) < PATH-COST(B)
  - Note that PATH-COST(C) ≤ PATH-COST(A) because C was expanded before A by the uniform cost search

    But, PATH-COST(A) < PATH-COST(B) because B is a child of A

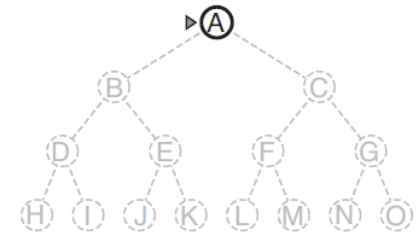


- Evaluation:
  - Optimal and complete if every action costs at least $\varepsilon > 0$
  - Time & memory complexity: $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
    - C^*: the cost of the optimal solution
    - When all the step costs are equal, $b^{1+\lfloor C^*/\varepsilon \rfloor}$ becomes b^{1+d} (Uniform-cost search does more work than breadth-first search by expanding nodes at depth d unnecessarily)

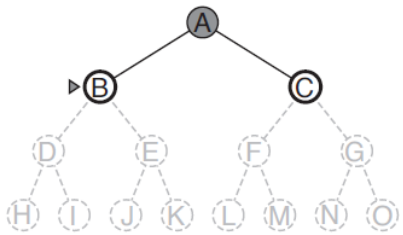
Depth-first Search

- Expand the deepest unexpanded node on the frontier
  - Frontier is a stack, i.e., put successors at front
- TREE-SEARCH-based implementation:
  - Additional feature: a new state is discarded if it is the same as one of the ancestors (to avoid an infinite loop)
    - Redundant paths still cannot be avoided
  - Modest memory requirement
  - Often implemented as a recursion on each of the children in turn (See RECURSIVE-DLS)
- GRAPH-SEARCH-based Implementation?
  - Meaningless because the explored set requires exponential memory

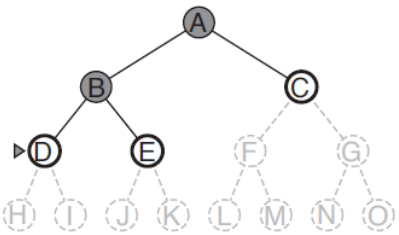
Depth-first search



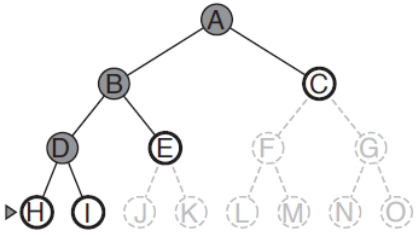
[A]



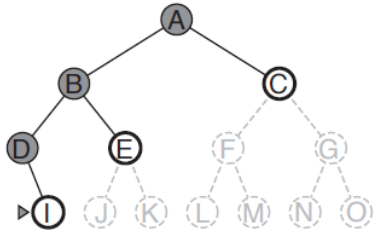
[B, C]



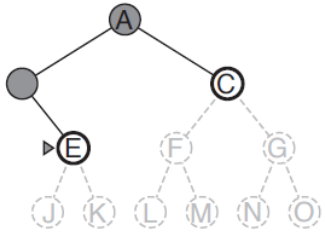
[D, E, C]



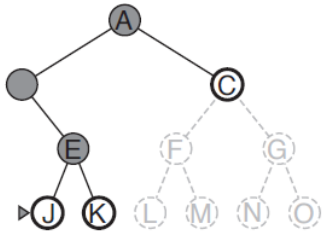
[H, I, E, C]



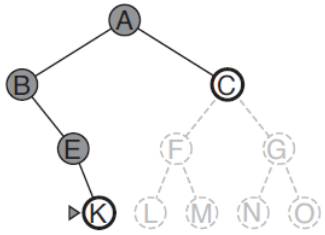
[I, E, C]



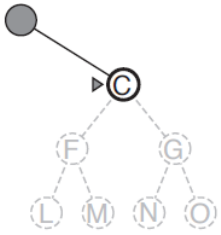
[E, C]



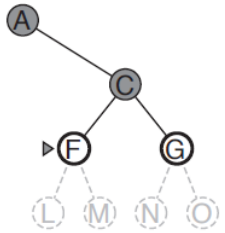
[J, K, C]



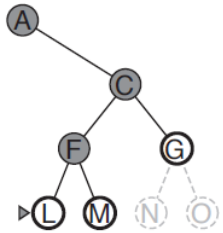
[K, C]



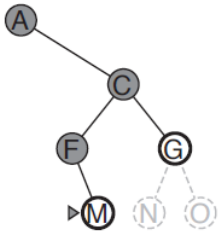
[C]



[F, G]



[L, M, G]



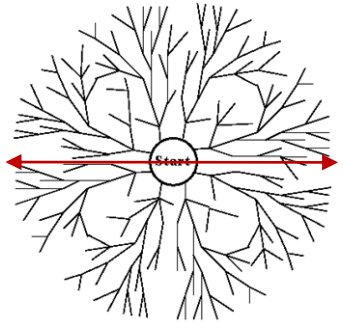
[M, G]

Depth-first search

- Evaluation (depth-first tree search):
  - Modest memory requirements: $O(bm)$
    (b: branching factor, m: maximum depth)
  - Takes $O(b^m)$ time
    - Terrible if m is much larger than d
    - But if solutions are dense, may be much faster than breadth-first
  - Neither complete nor optimal in infinite-depth spaces
  - Complete in finite spaces
  - Not optimal in finite spaces because of the redundant paths

Depth-limited Search

- Equal to the depth-first search with depth-limit l
  - Nodes at depth l has no successors
  - Usually TREE-SEARCH-based
- What is a good limit ?
  - We want to know the diameter of the state space



- Evaluation:
  - Takes $O(b^l)$ time
  - Takes $O(bl)$ space
  - Neither complete nor optimal unless $l \geq d$
    (d is the depth of solution)

Depth-limited Search

function DEPTH-LIMITED-SEARCH(*problem*, *limit*) **returns** a solution, or failure/cutoff

  return RECURSIVE-DLS(MAKE-NODE(*problem*.INITIAL-STATE), *problem*, *limit*)

function RECURSIVE-DLS(*node*, *problem*, *limit*) **returns** a solution, or failure/cutoff

  if *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

  else if *limit* = 0 **then return** *cutoff*

  else

    cutoff_occurred? ← false

    for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

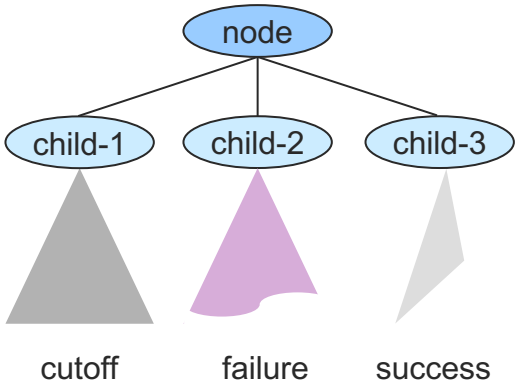
      child ← CHILD-NODE (*problem*, *node*, *action*)

      result ← RECURSIVE-DLS(*child*, *problem*, *limit* – 1)

      if *result* = *cutoff* **then** *cutoff_occurred*? ← true

      else if *result* ≠ *failure* **then return** *result*

    if *cutoff_occurred*? **then return** *cutoff* **else return** *failure*



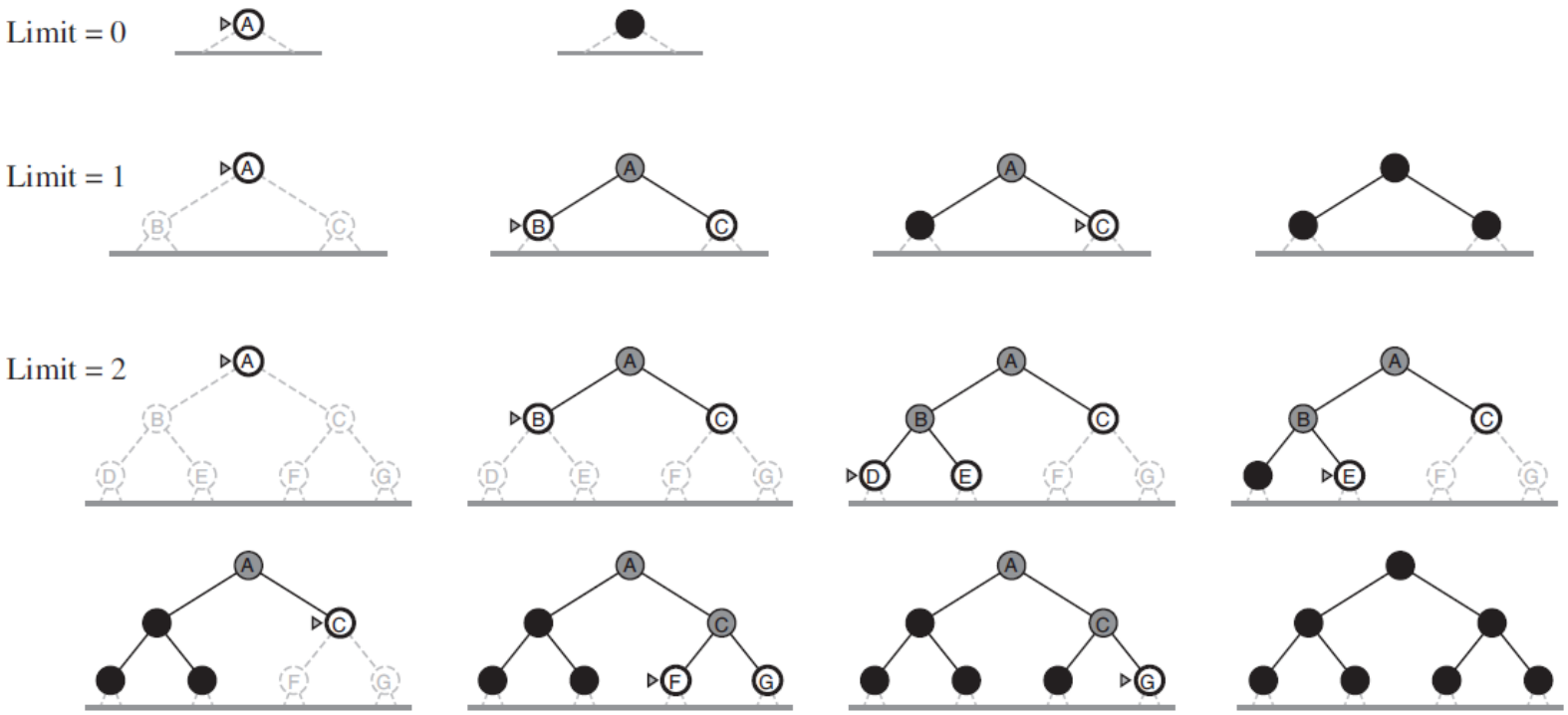
Iterative Deepening Search

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

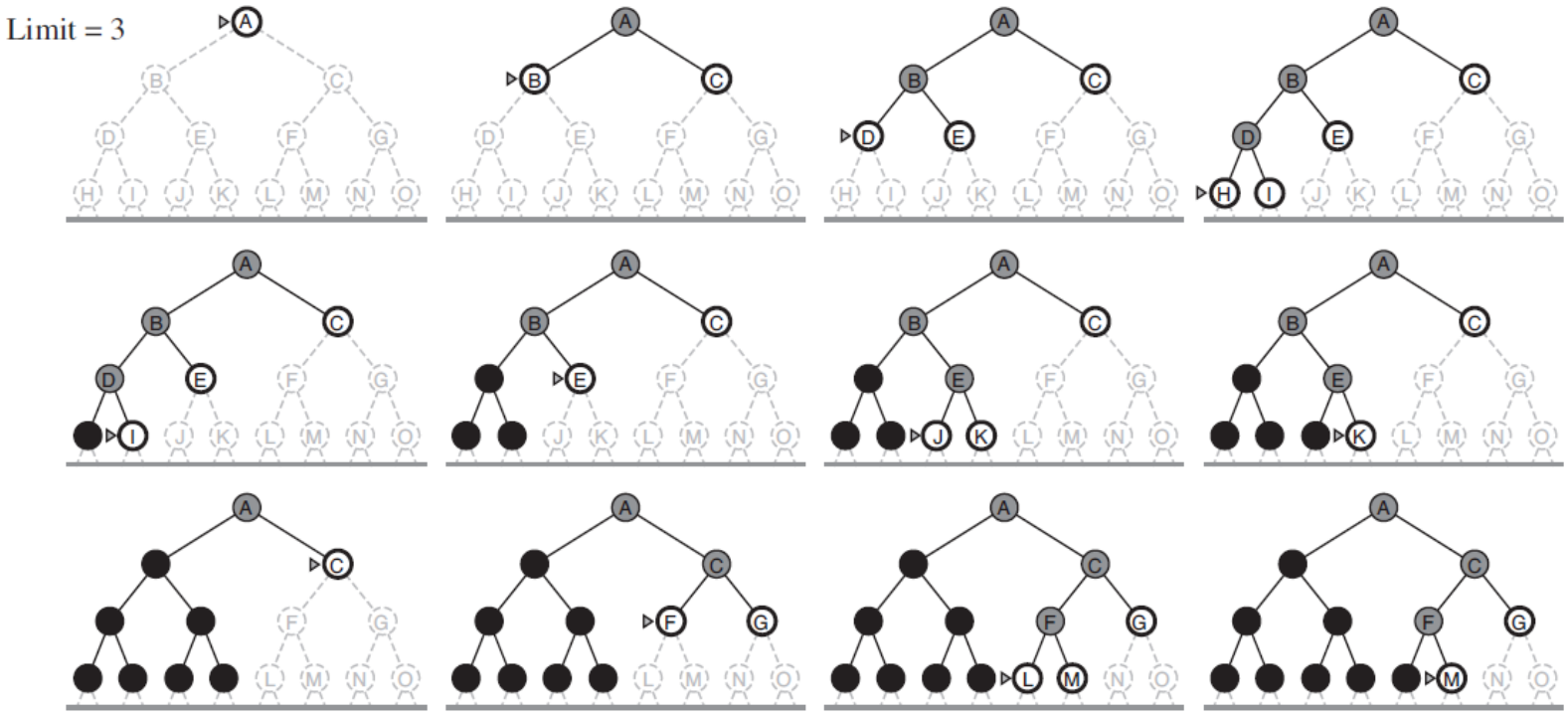
- Tries increasing depth limit when we don't know the diameter of the state space
- GRAPH-SEARCH-based implementation is meaningless because it requires exponential memory (BFS is faster!)
- Iterative-lengthening-search:
  - Uses increasing path-cost limit

    (iterative analog to uniform-cost search)

Iterative Deepening Search



Iterative Deepening Search



Iterative Deepening Search

- Evaluation:
  - Optimal and complete like breadth-first, and requires modest memory like depth-first
  - The overhead of multiple expansion is rather small
    - Total number of nodes generated in a depth-limited search

      $$b + b^2 + b^3 + \ldots + b^{d-1} + b^d \qquad (= 111{,}110 \text{ when } b = 10,\ d = 5)$$

    - Total number of nodes generated in an iterative-deepening search

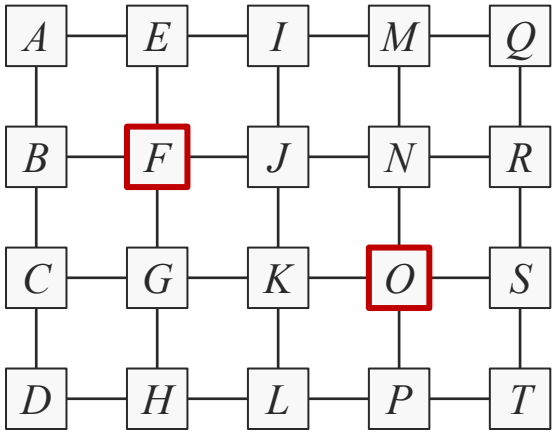
      $$(d)b + (d-1)b^2 + b^3 + \ldots + 2b^{d-1} + 1b^d \quad (= 123{,}450)$$

  - Takes $O(b^d)$ time and $O(bd)$ space (d : depth limit)

Iterative Deepening Search

Example:

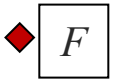
- To find a shortest path from F to O
- Use an iterative deepening search based on TREE-SEARCH

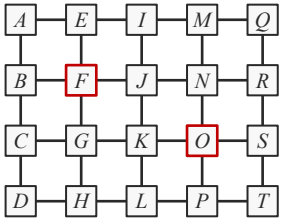


Iterative Deepening Search

Example:

Limit = 0

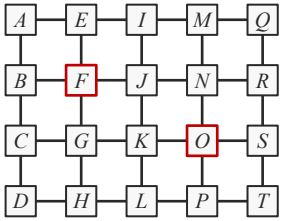




Iterative Deepening Search

Example:

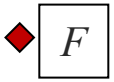
Limit = 0

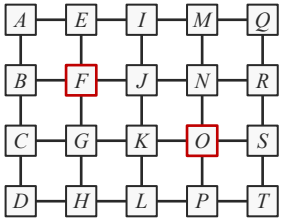


Iterative Deepening Search

Example:

Limit = 1

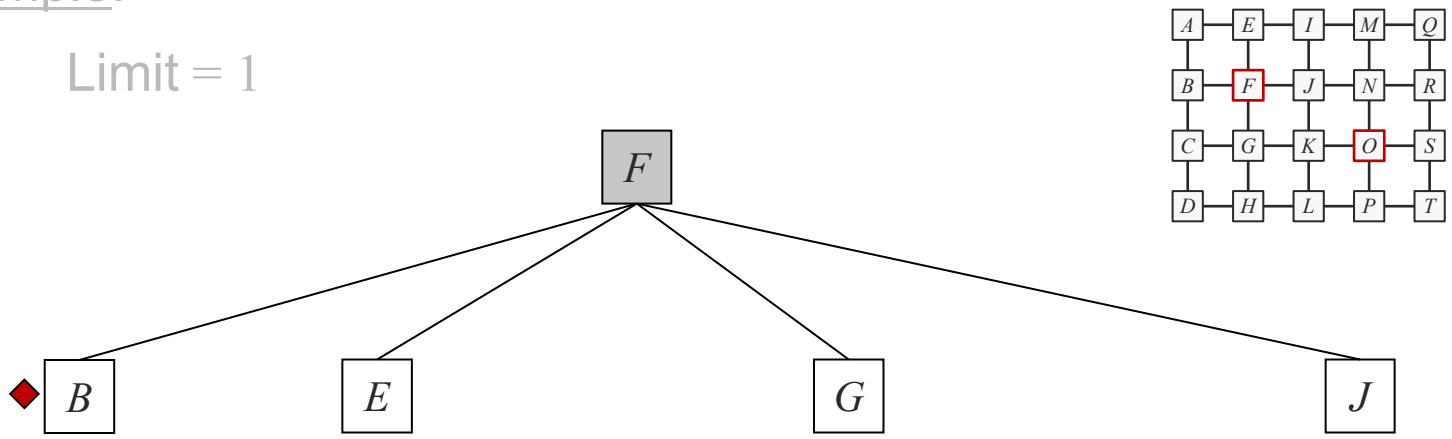




Iterative Deepening Search

Example:

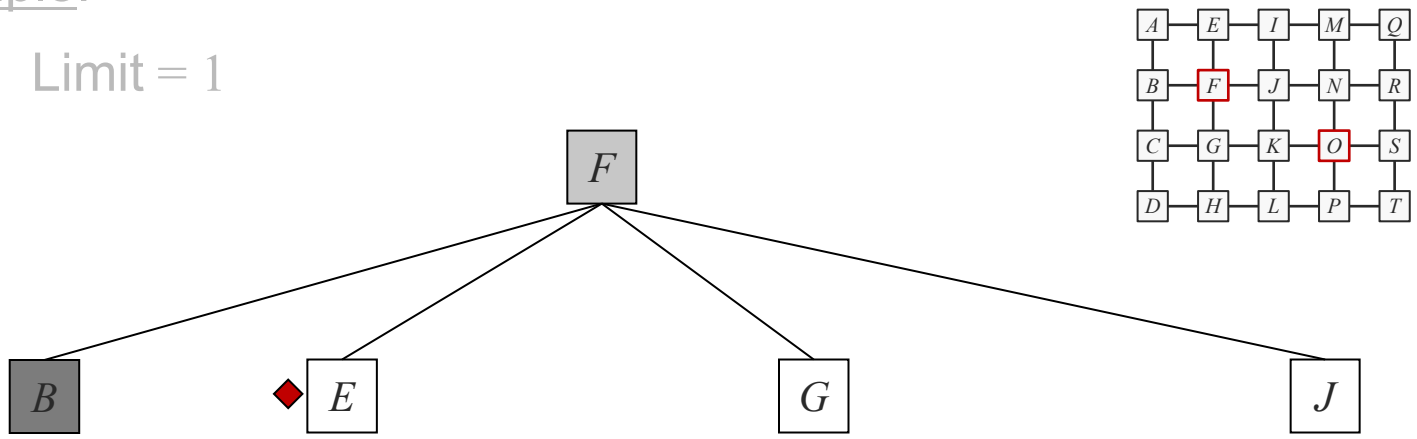
Limit = 1



Iterative Deepening Search

Example:

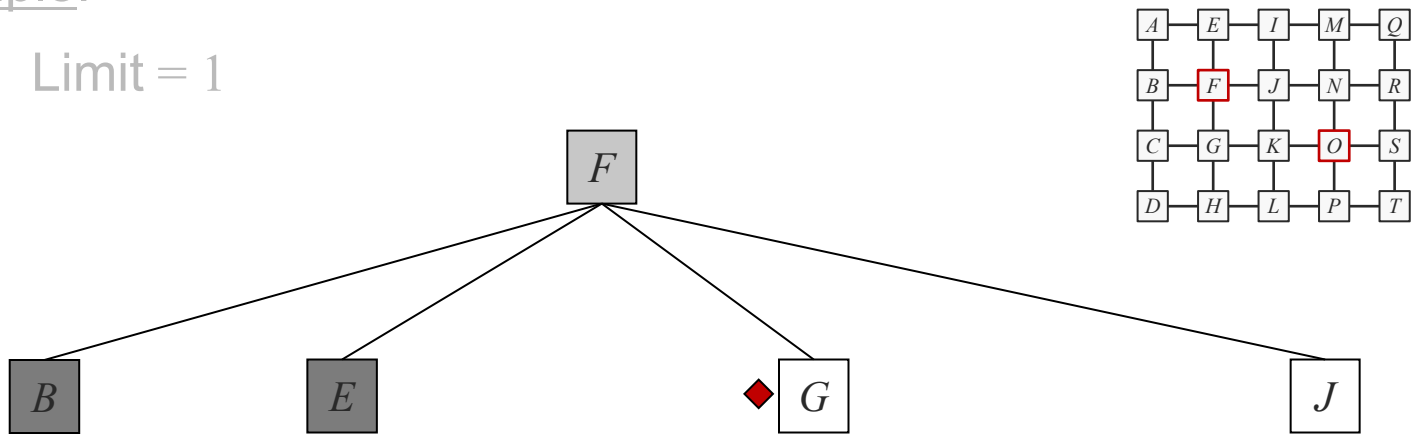
Limit = 1



Iterative Deepening Search

Example:

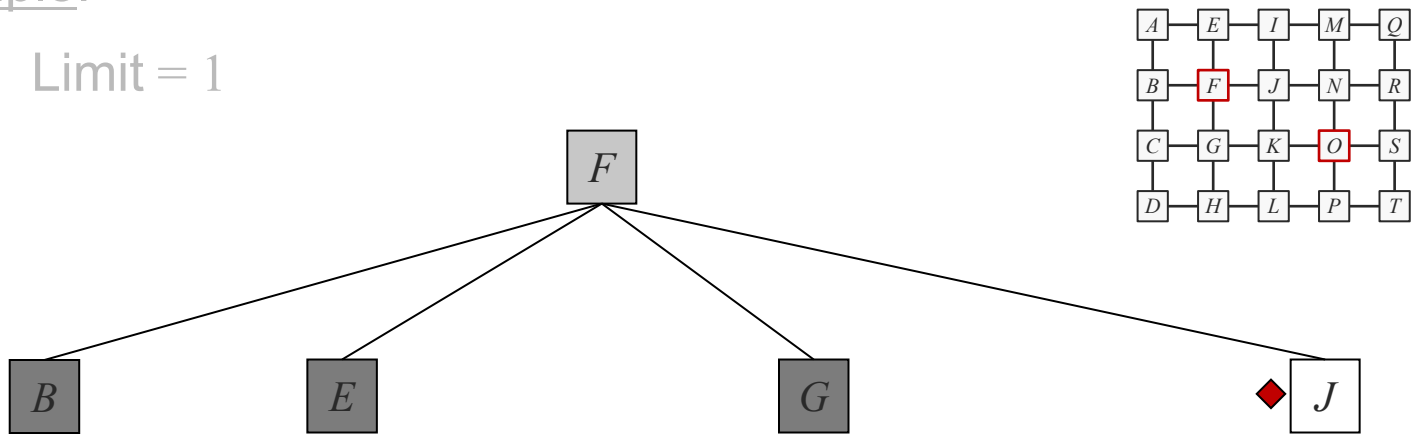
Limit = 1



Iterative Deepening Search

Example:

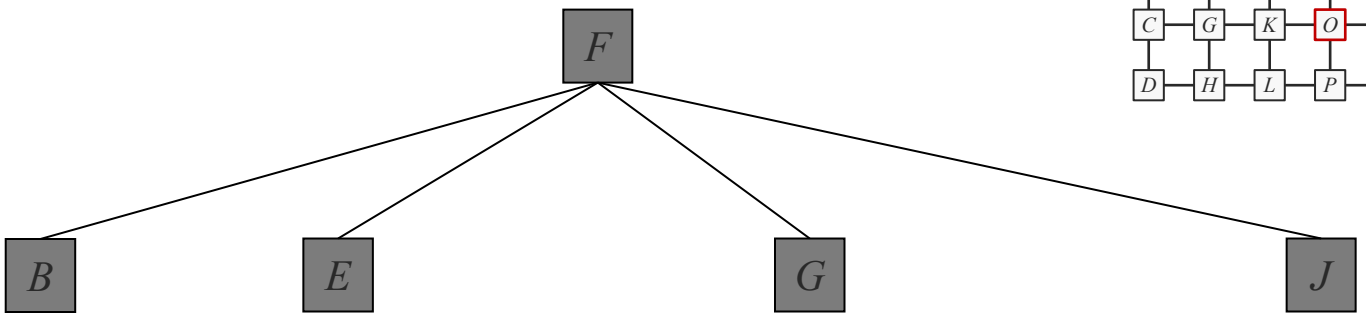
Limit = 1

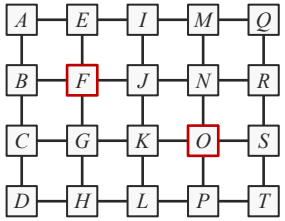


Iterative Deepening Search

Example:

Limit = 1

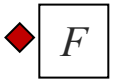


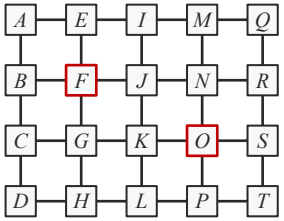


Iterative Deepening Search

Example:

Limit = 2

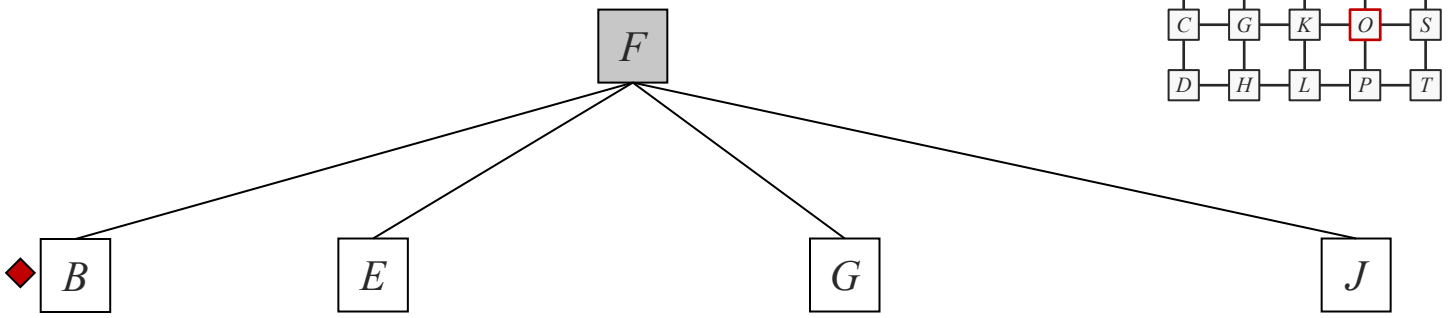


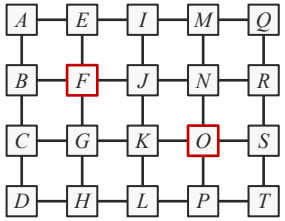


Iterative Deepening Search

Example:

Limit = 2

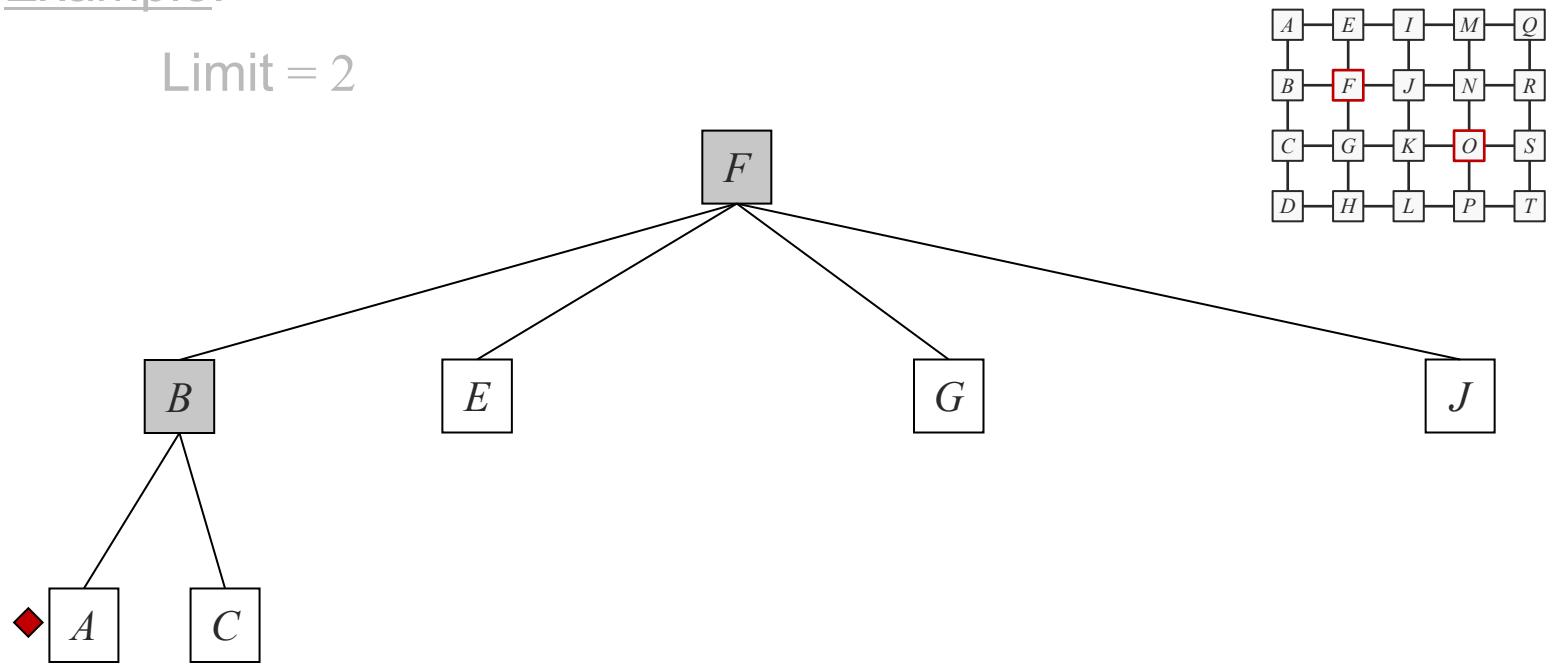




Iterative Deepening Search

Example:

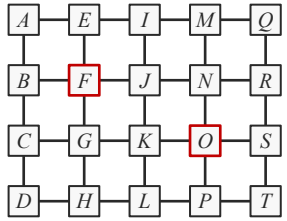
Limit = 2

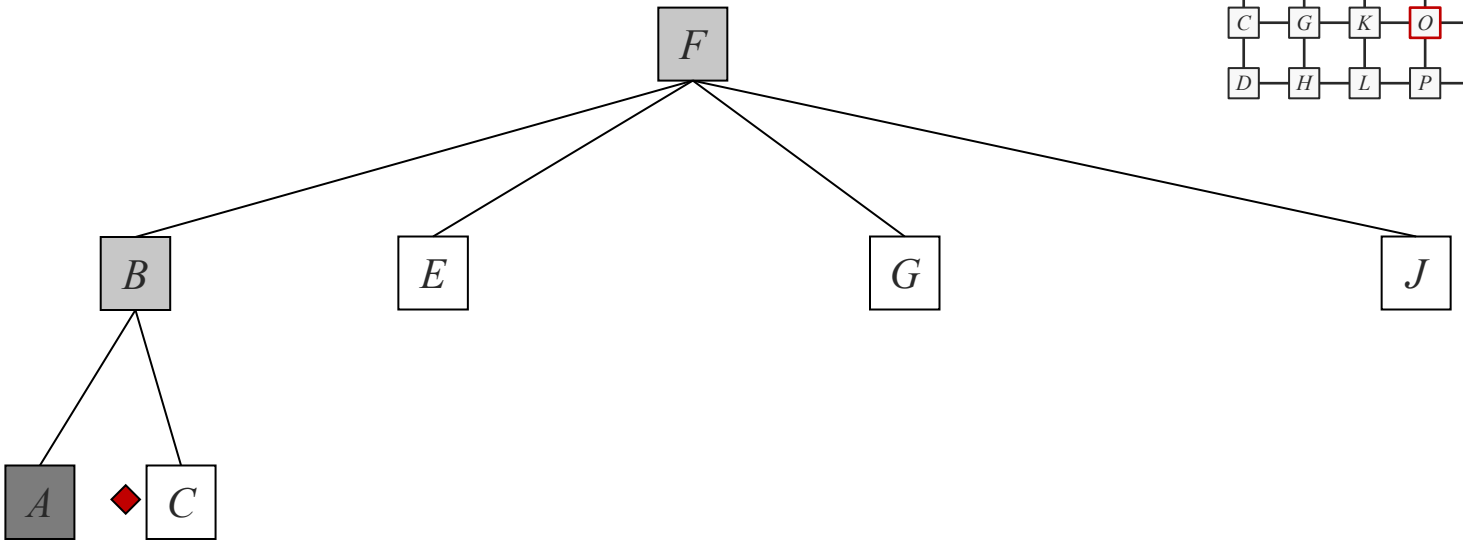


Iterative Deepening Search

Example:

Limit = 2

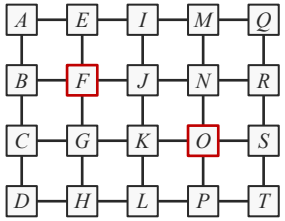


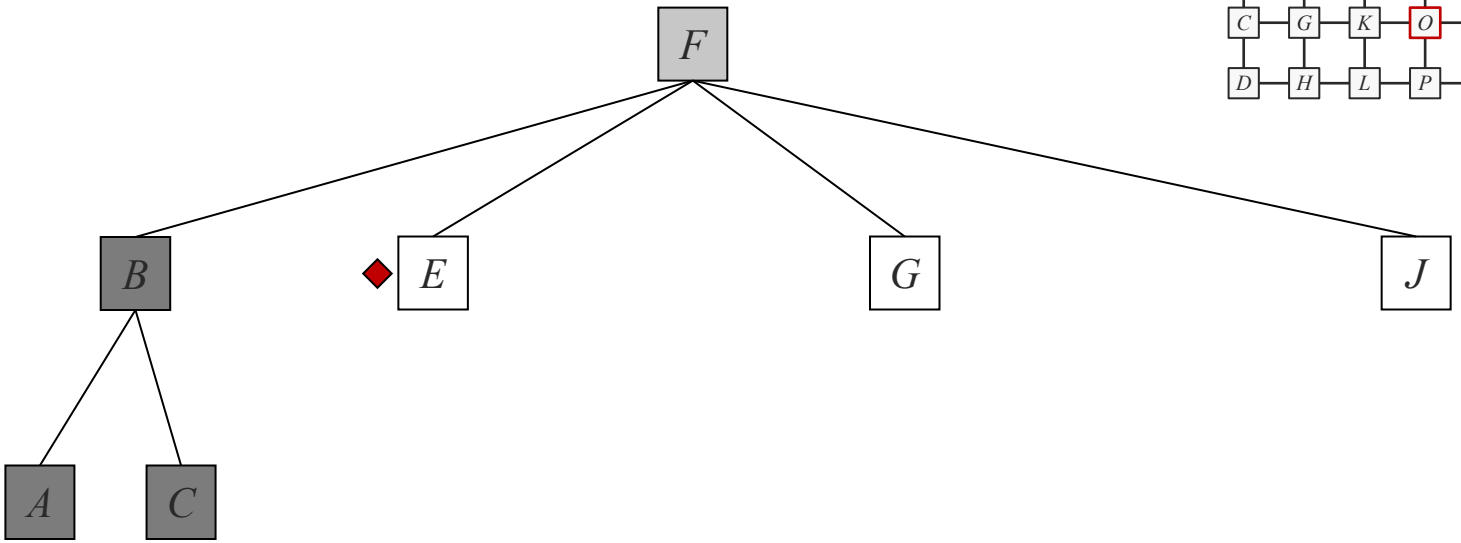


Iterative Deepening Search

Example:

Limit = 2

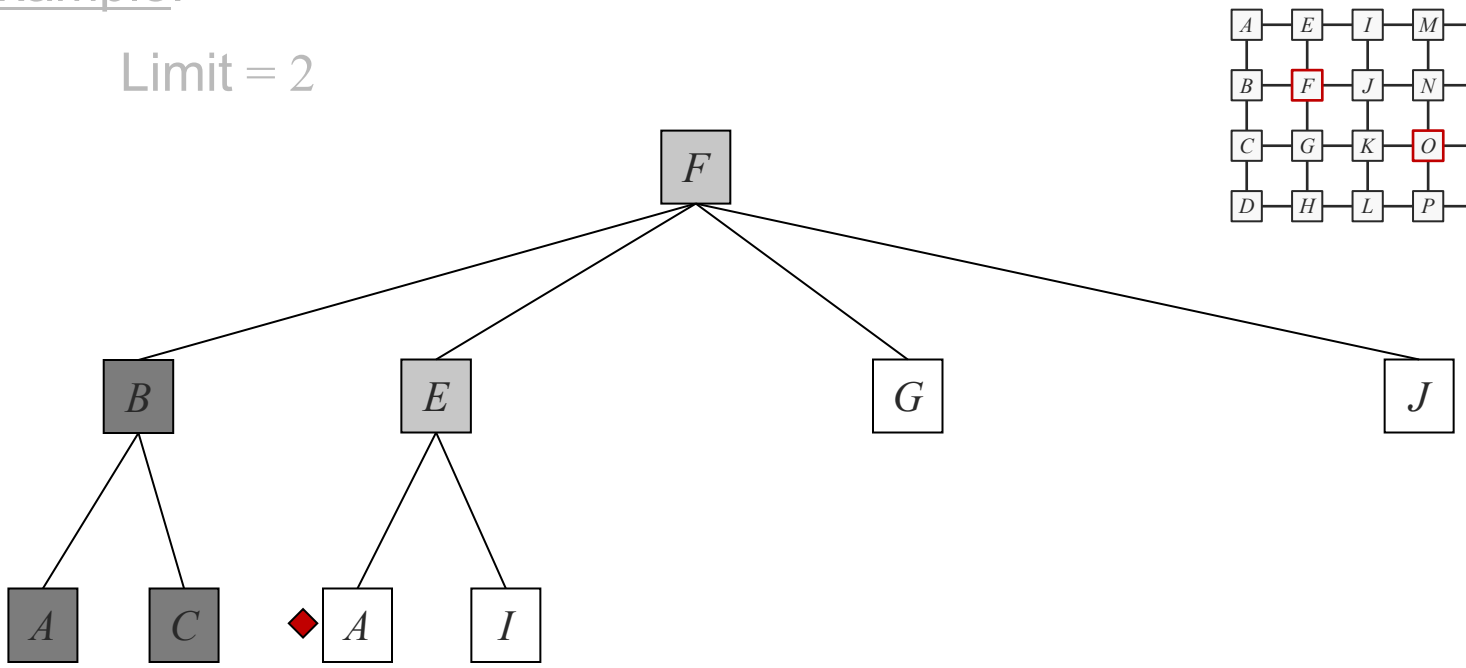




Iterative Deepening Search

Example:

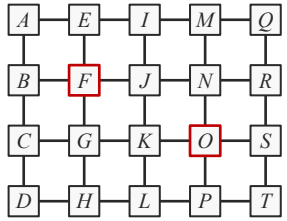
Limit = 2

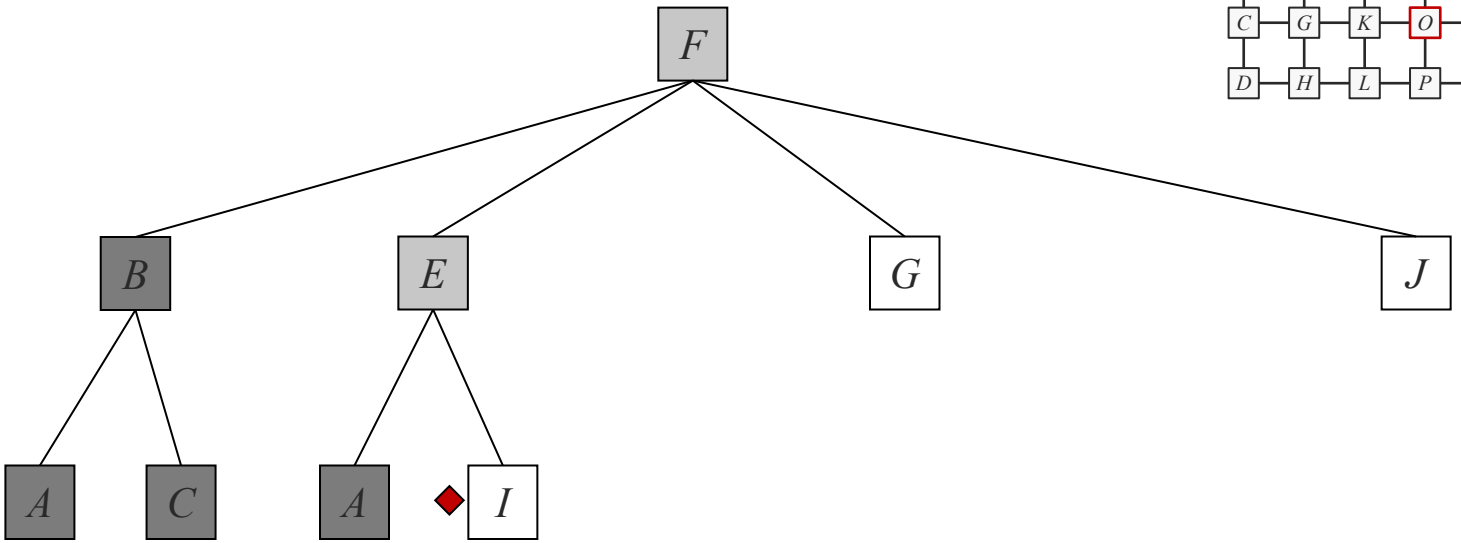


Iterative Deepening Search

Example:

Limit = 2

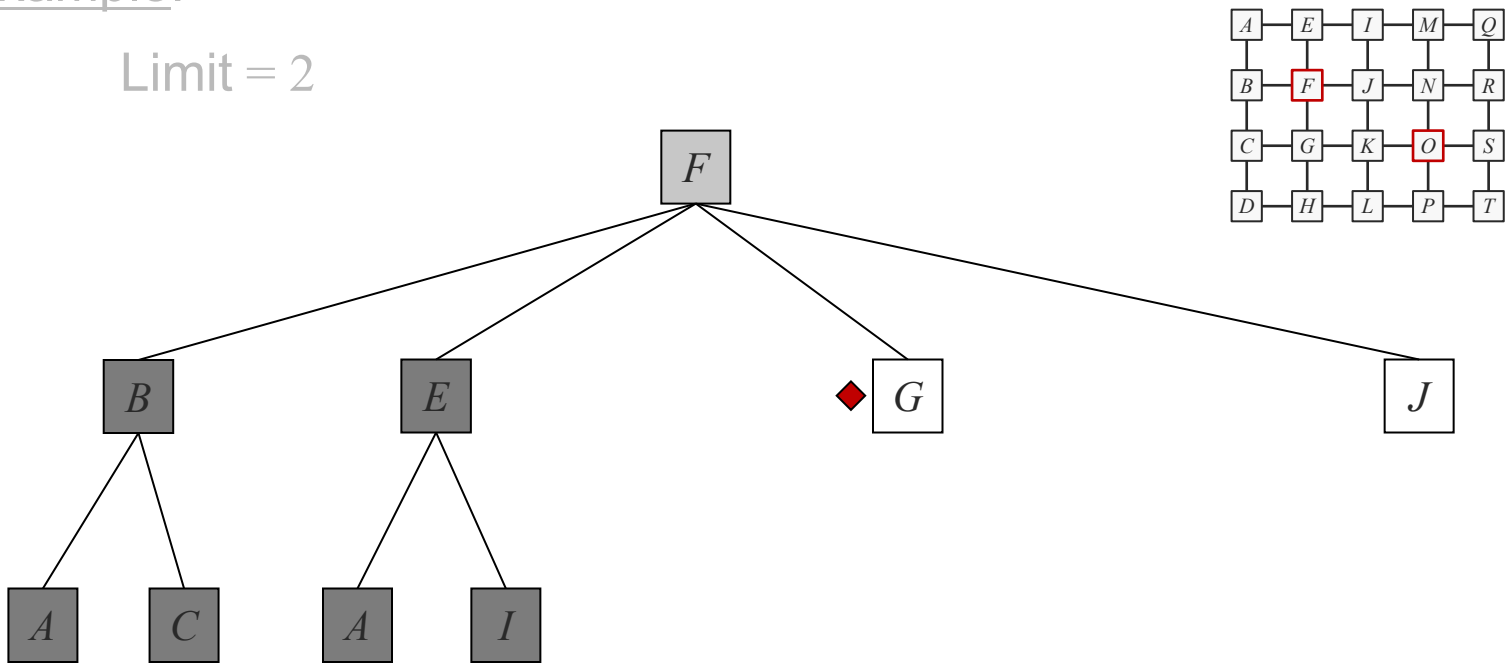




Iterative Deepening Search

Example:

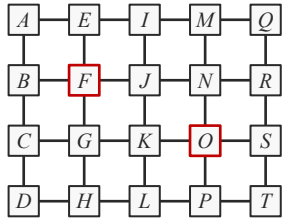
Limit = 2

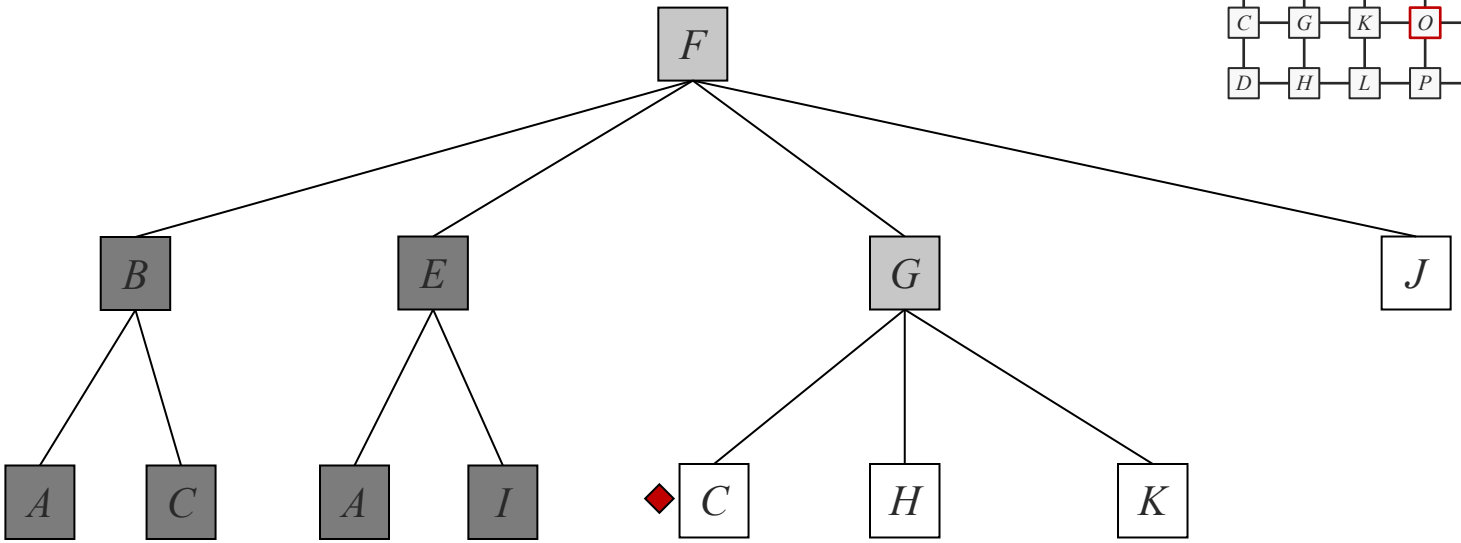


Iterative Deepening Search

Example:

Limit = 2

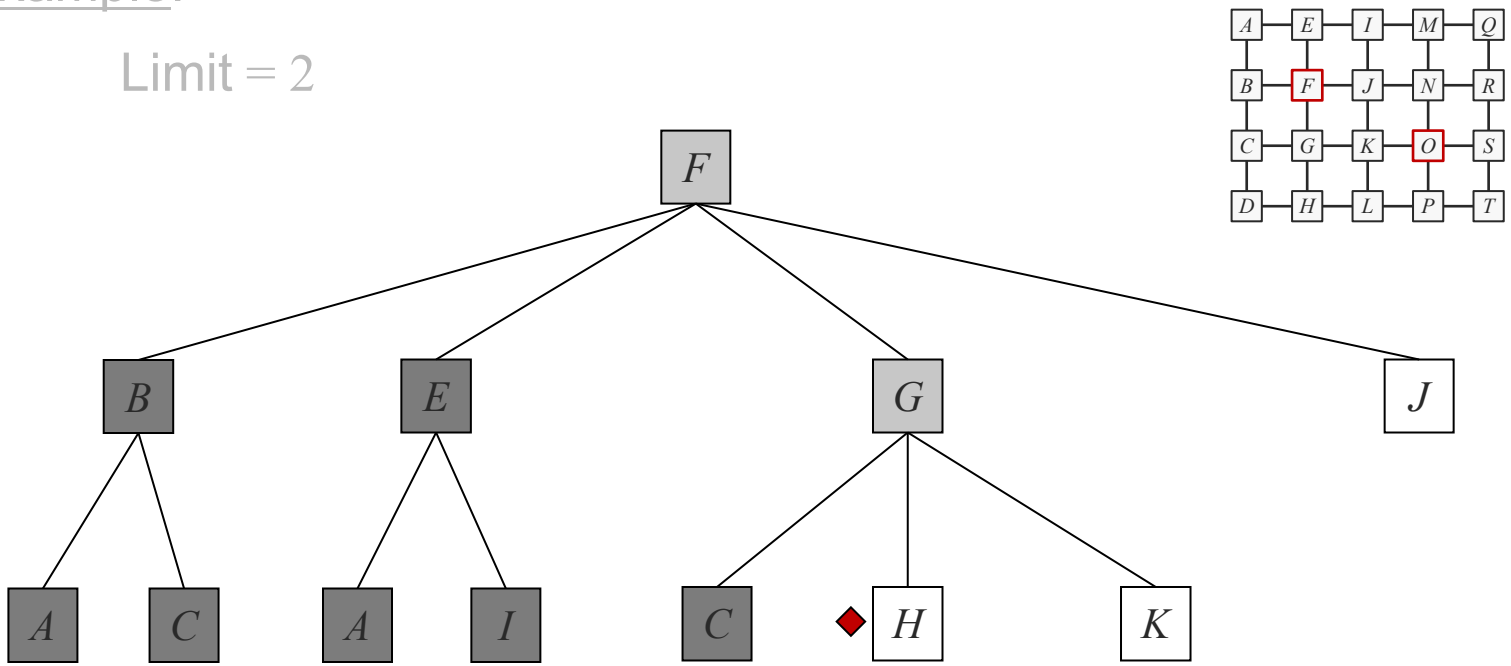




Iterative Deepening Search

Example:

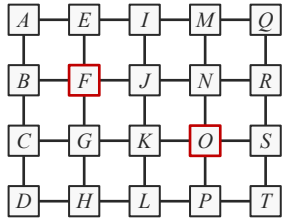
Limit = 2

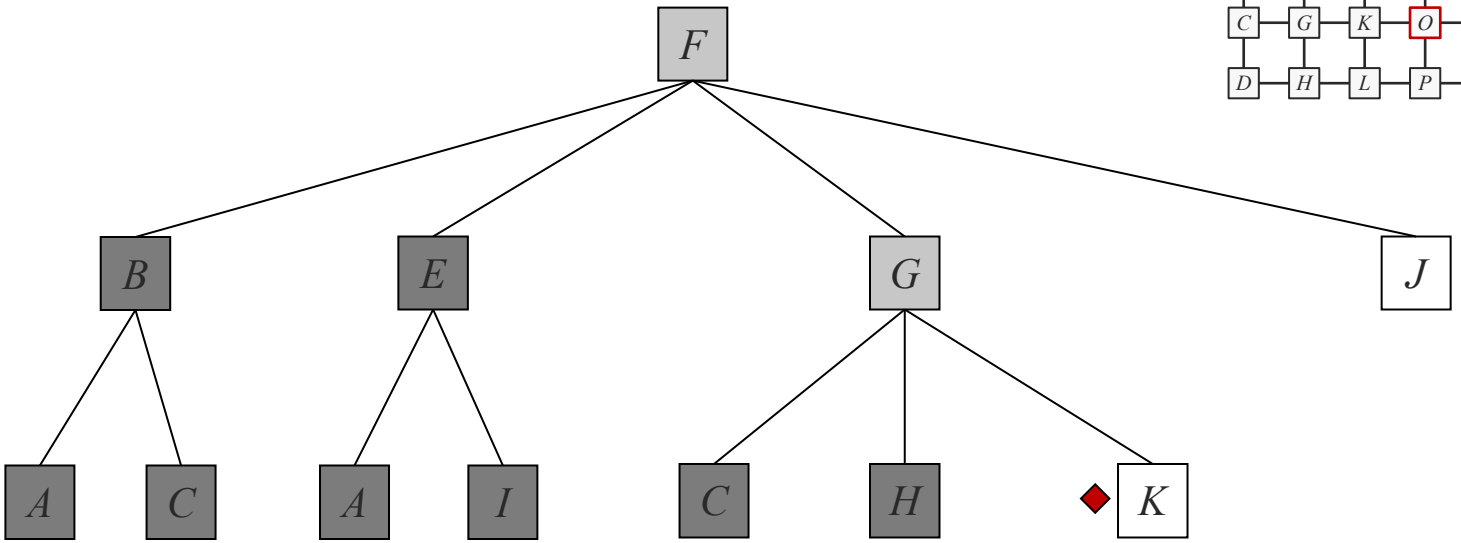


Iterative Deepening Search

Example:

Limit = 2

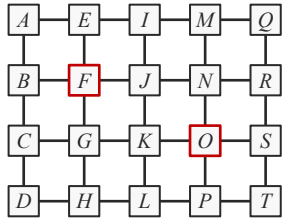


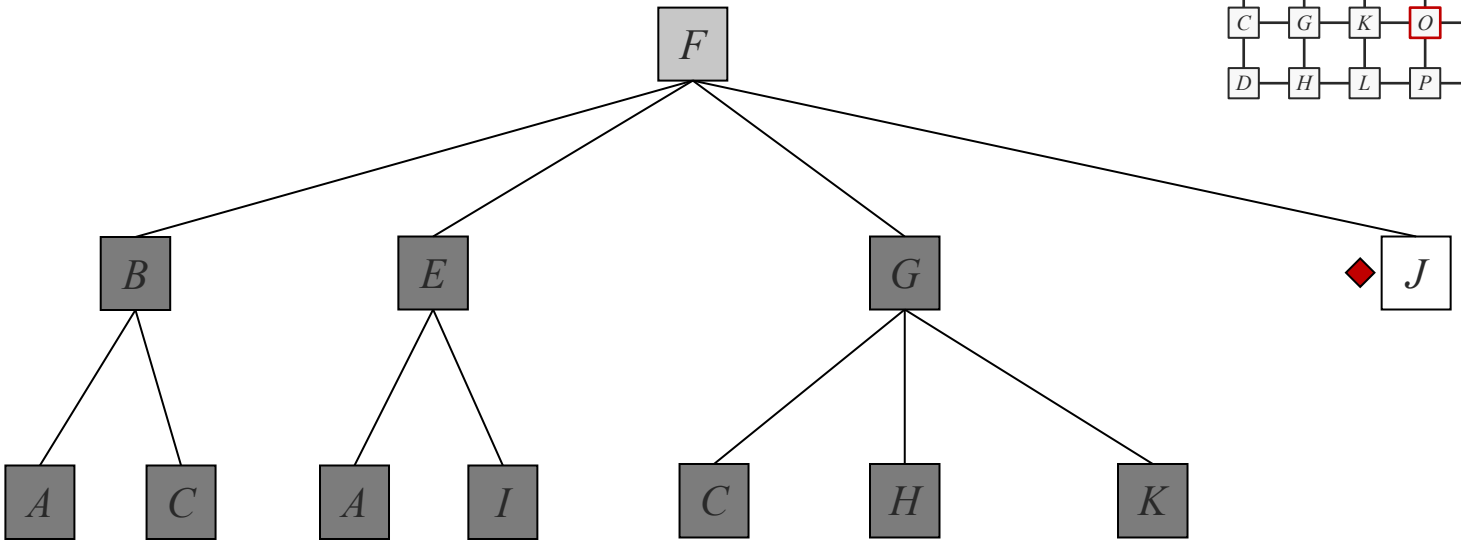


Iterative Deepening Search

Example:

Limit = 2

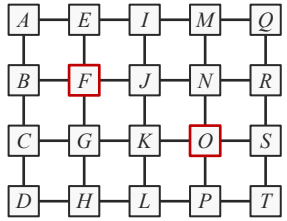


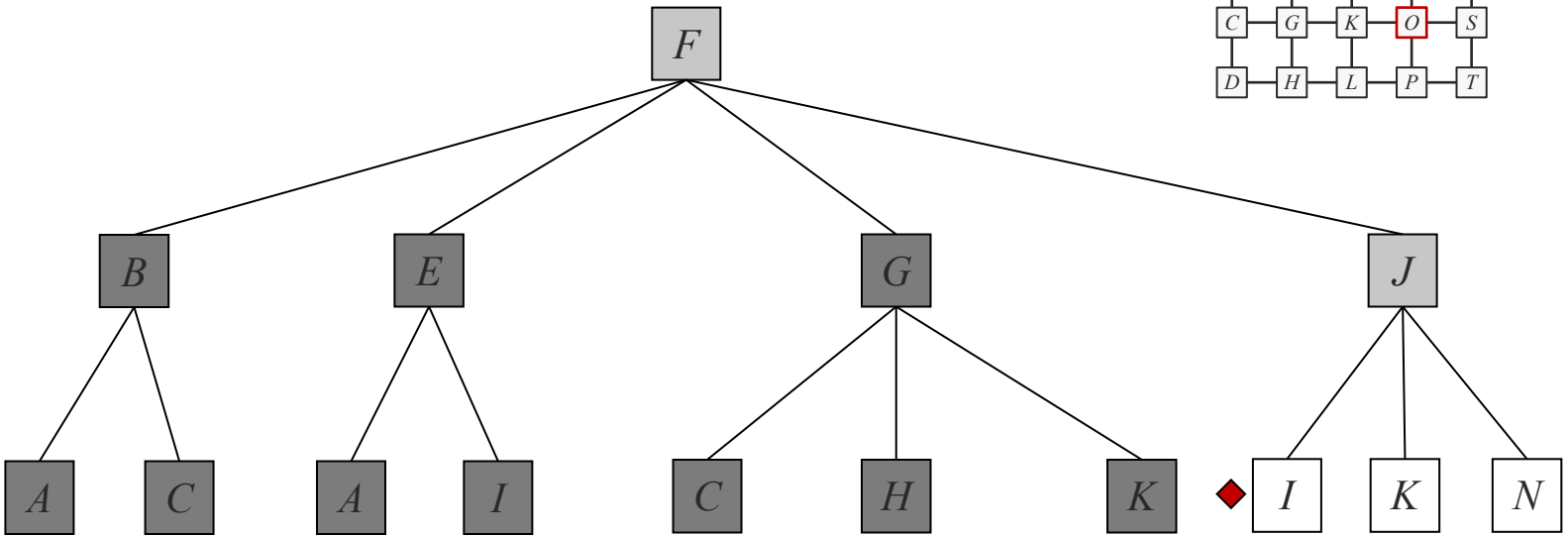


Iterative Deepening Search

Example:

Limit = 2

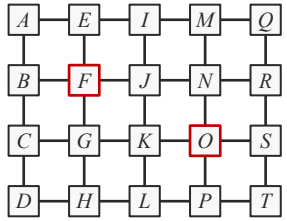


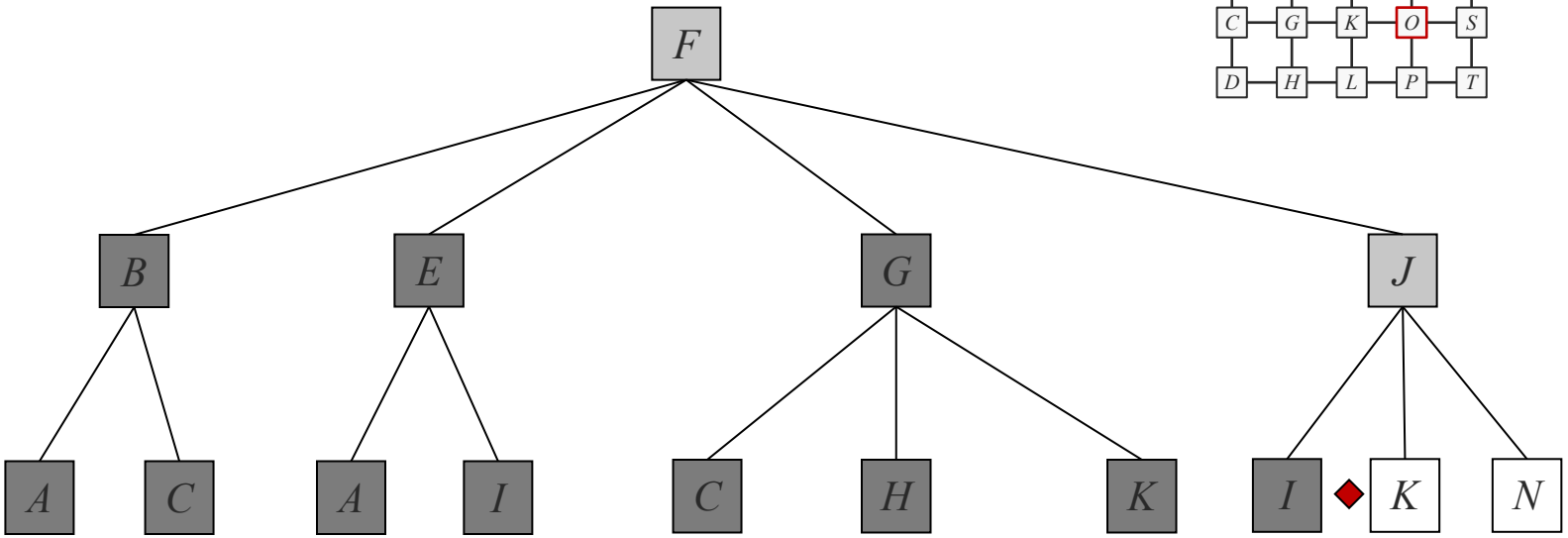


Iterative Deepening Search

Example:

Limit = 2

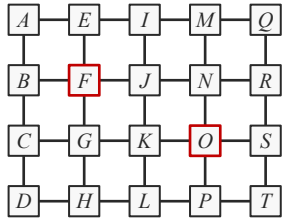


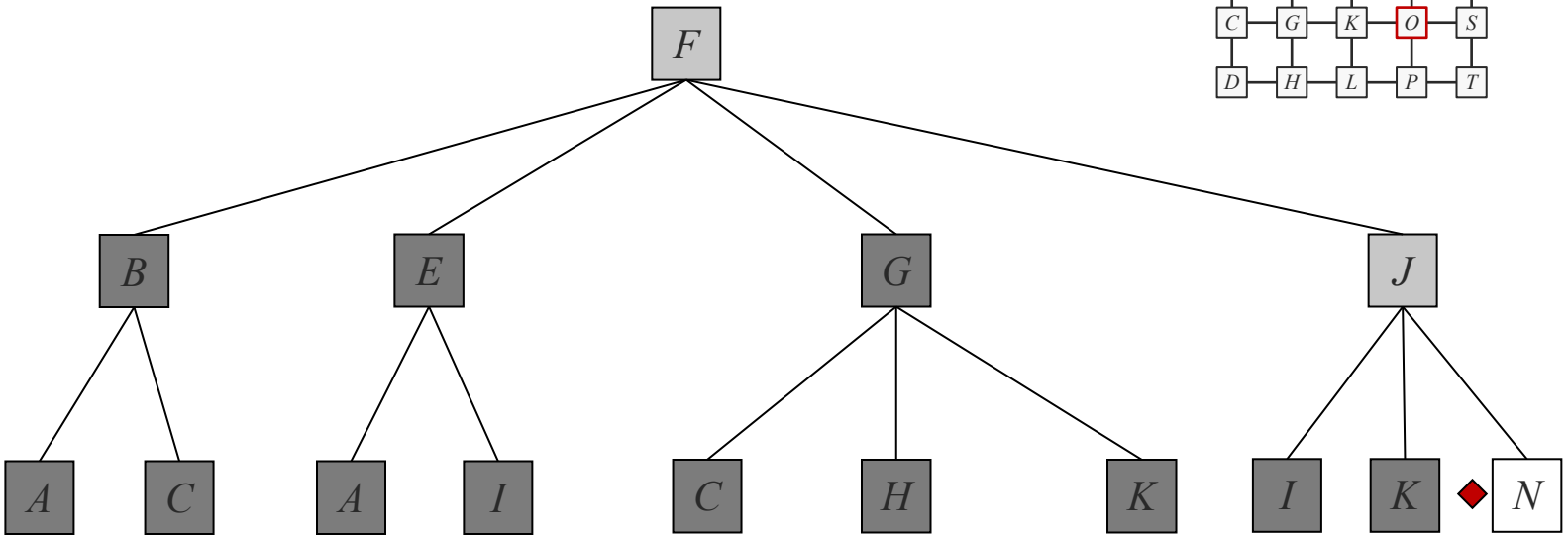


Iterative Deepening Search

Example:

Limit = 2

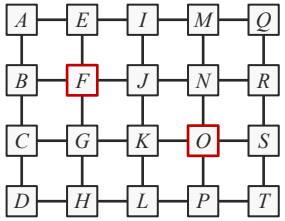


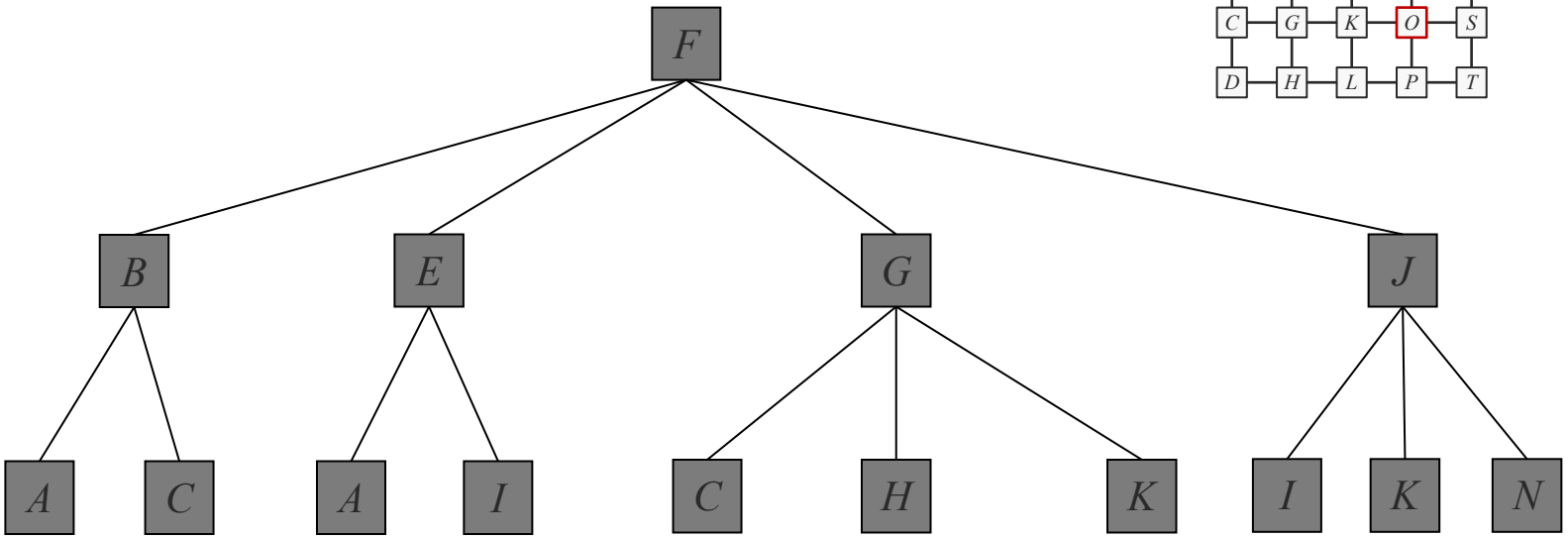


Iterative Deepening Search

Example:

Limit = 2

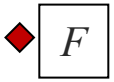


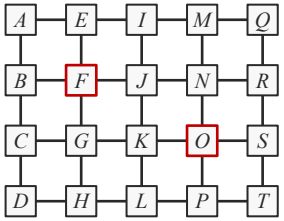


Iterative Deepening Search

Example:

Limit = 3

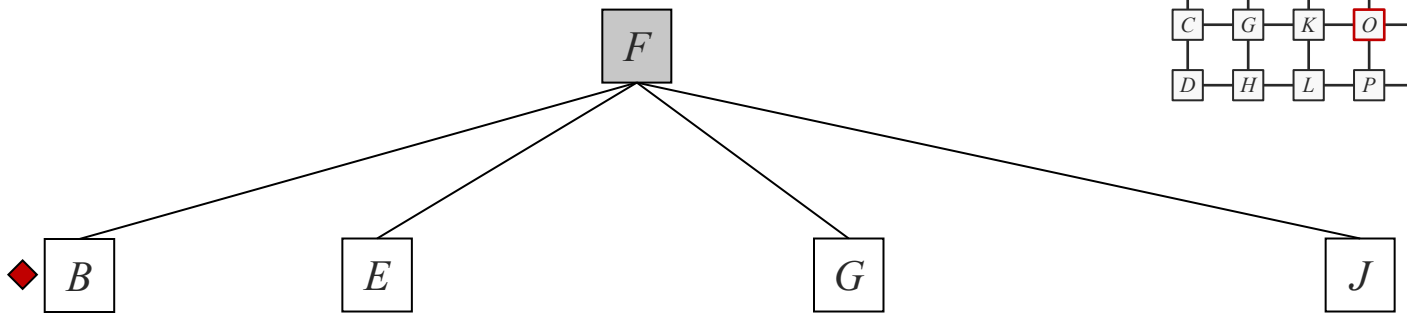


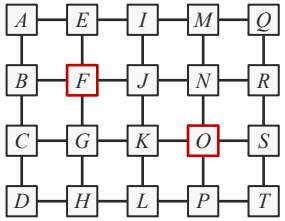


Iterative Deepening Search

Example:

Limit = 3

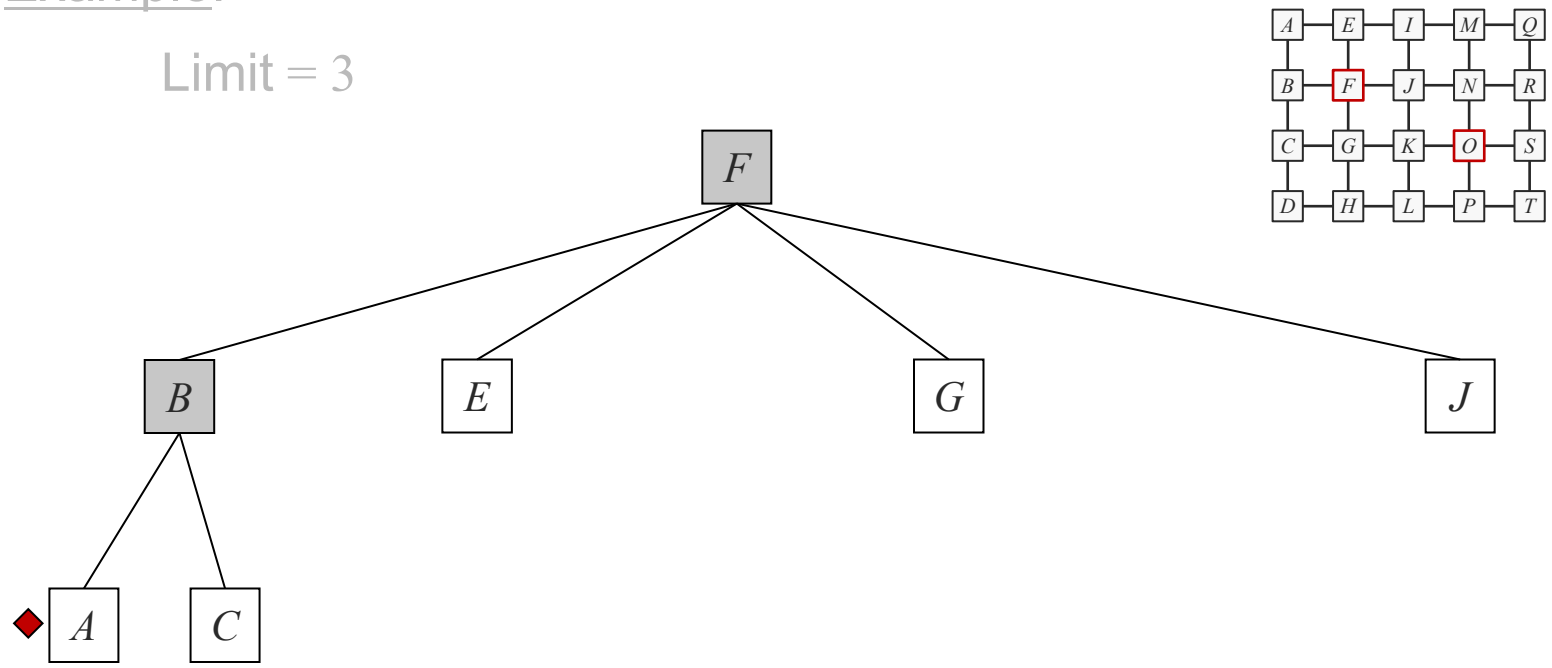




Iterative Deepening Search

Example:

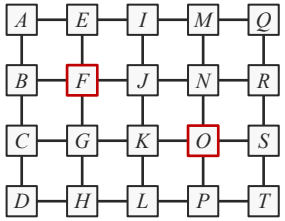
Limit = 3

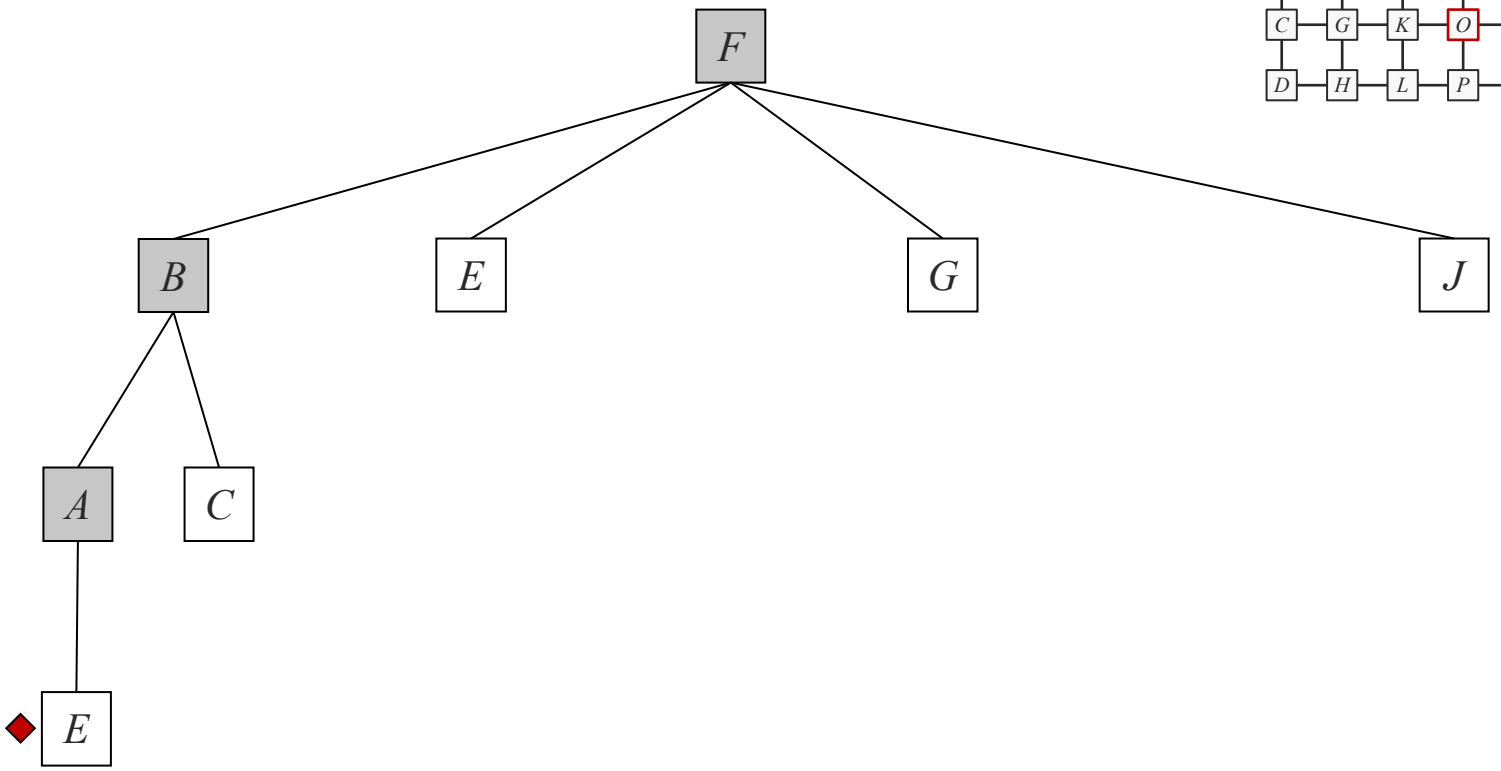


Iterative Deepening Search

Example:

Limit = 3

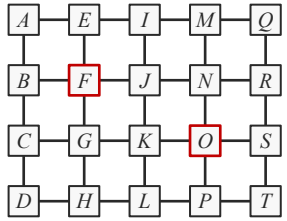


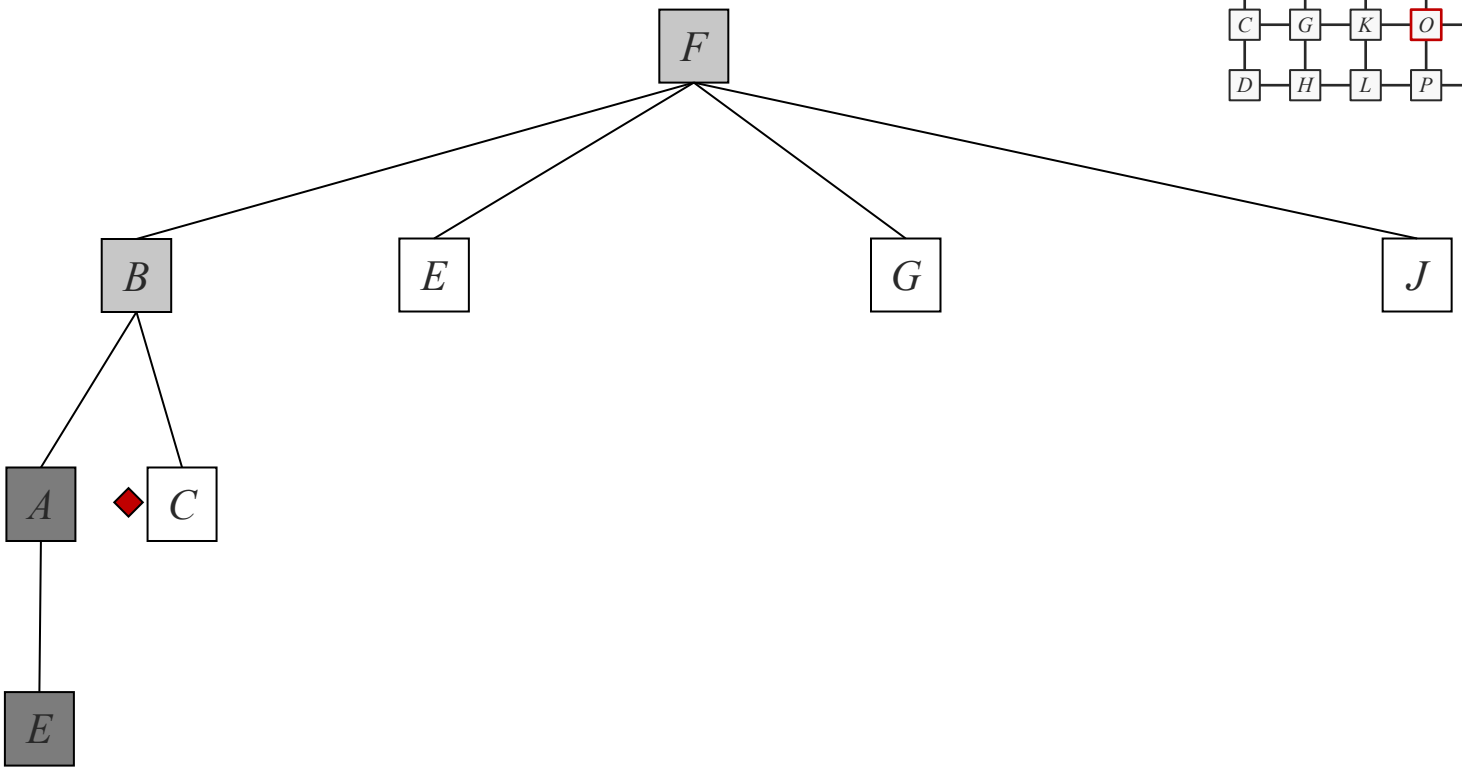


Iterative Deepening Search

Example:

Limit = 3

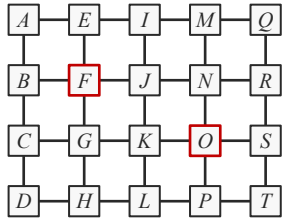


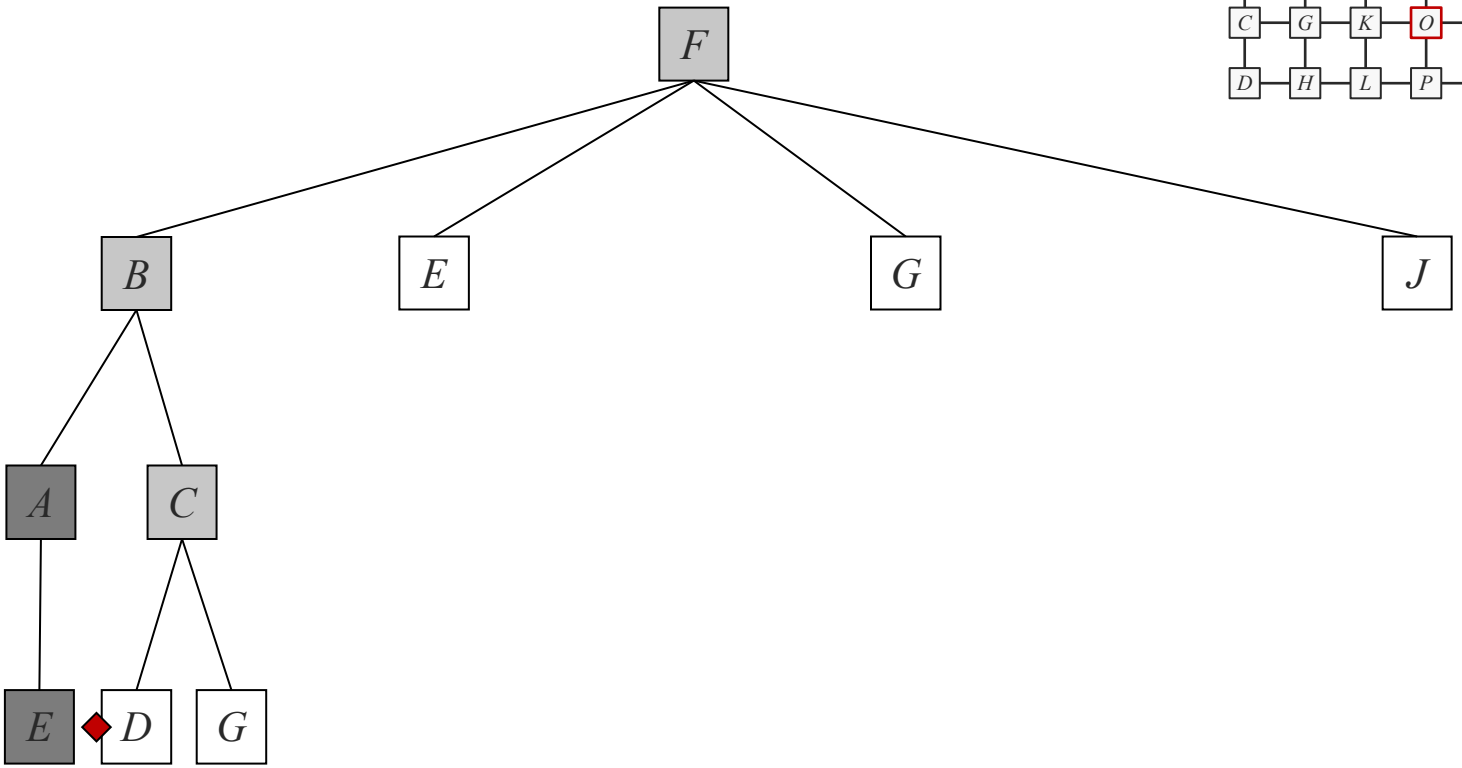


Iterative Deepening Search

Example:

Limit = 3

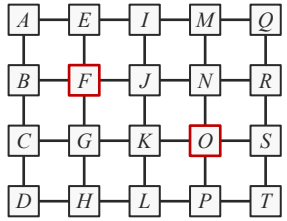


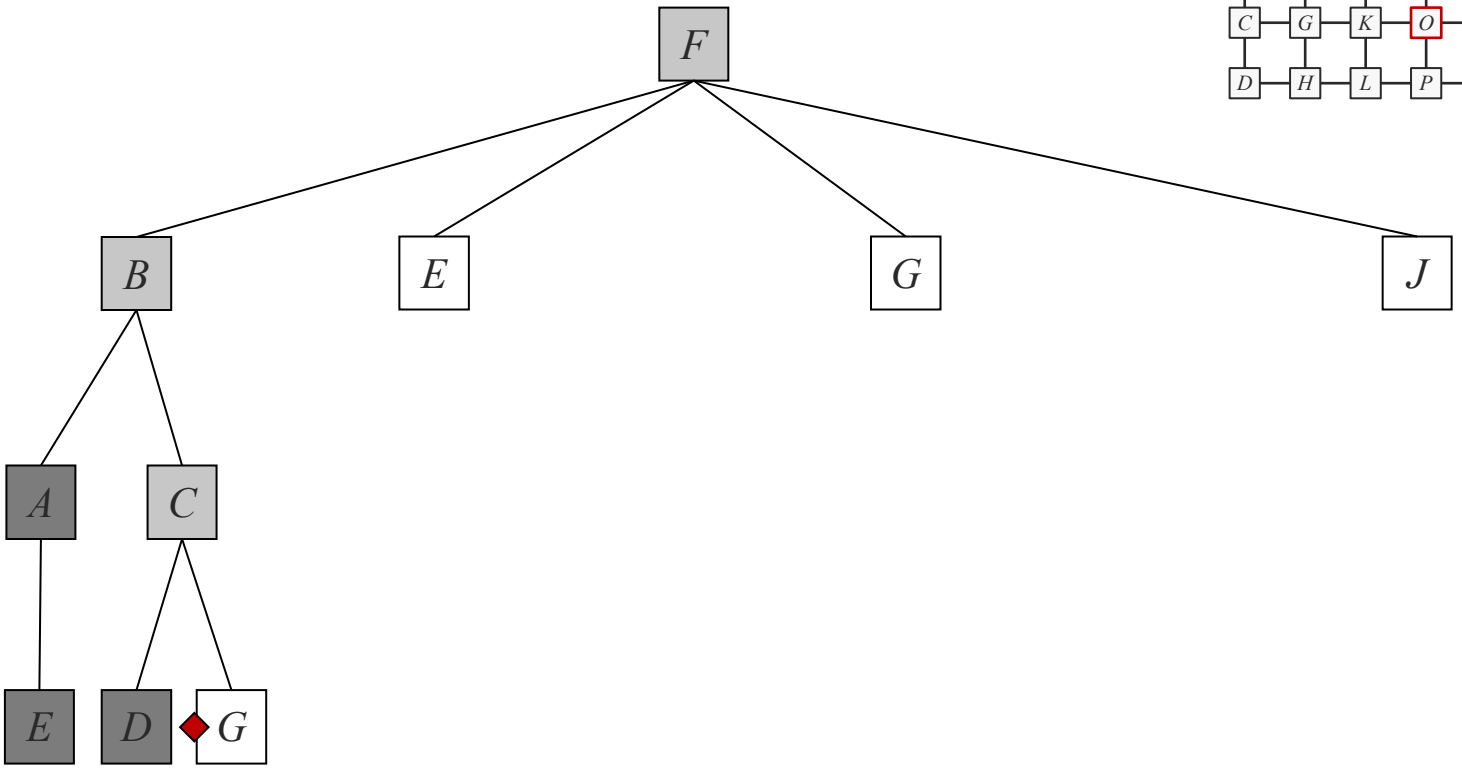


Iterative Deepening Search

Example:

Limit = 3

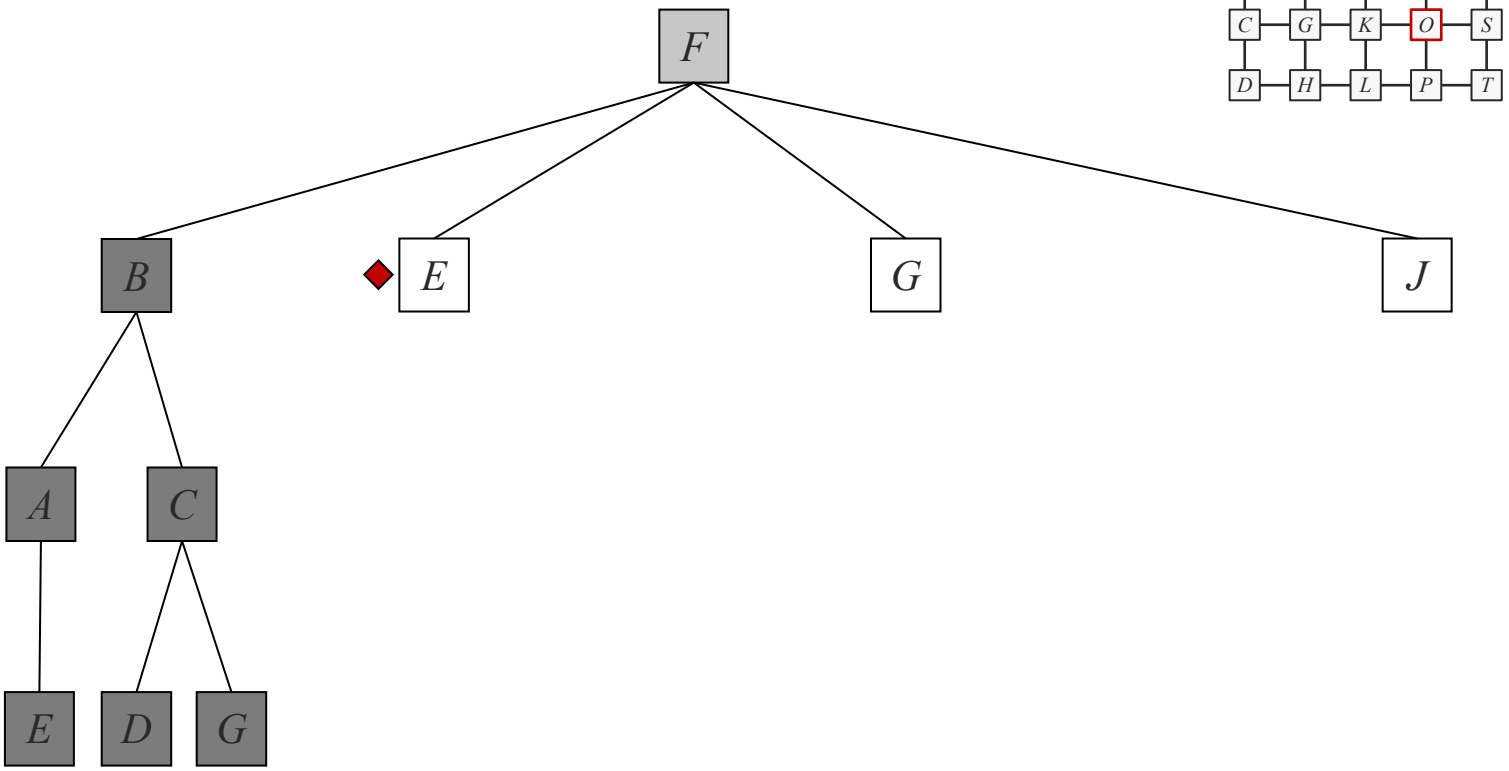


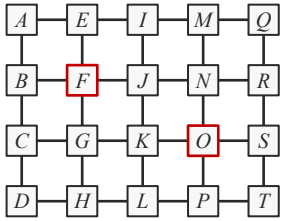


Iterative Deepening Search

Example:

Limit = 3

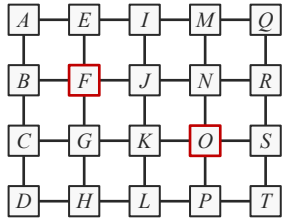


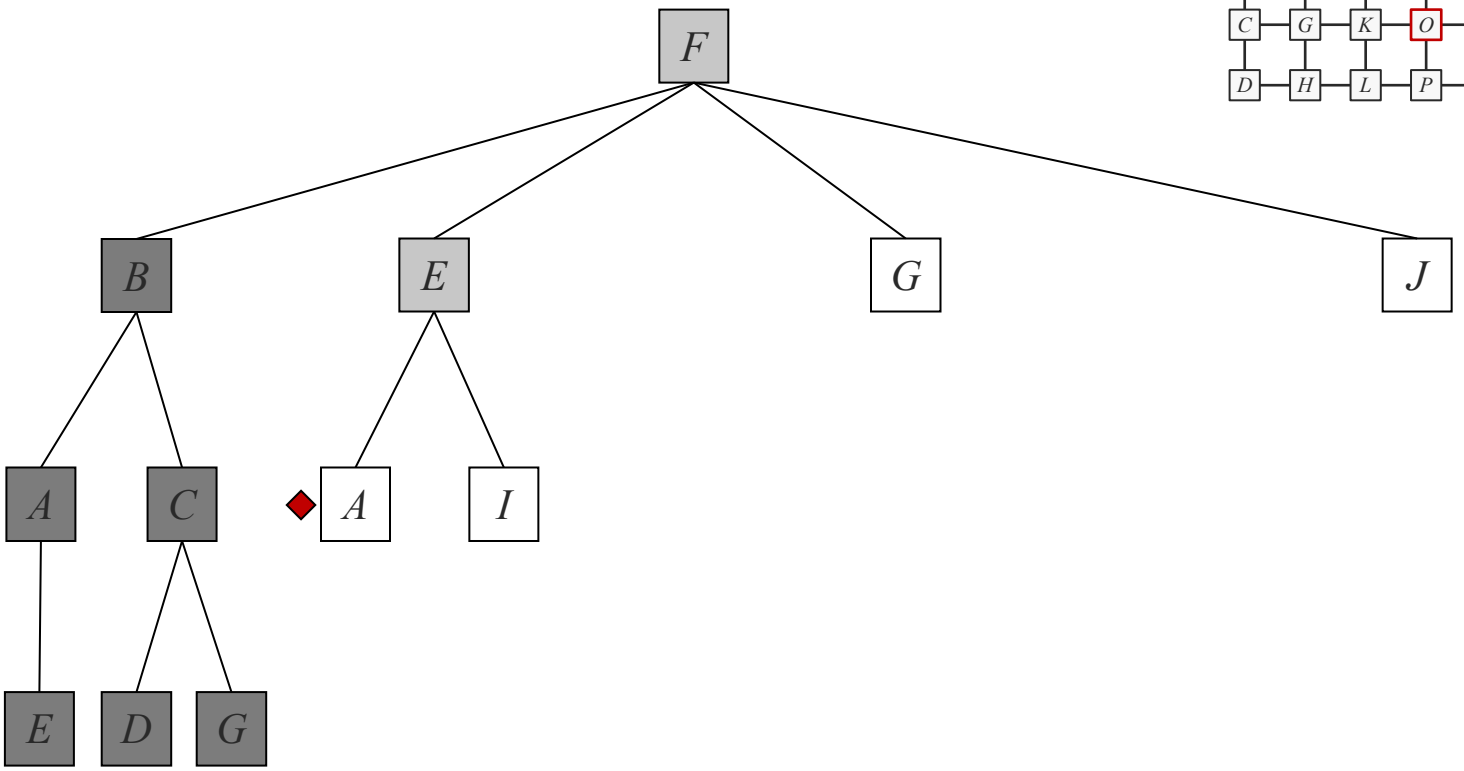


Iterative Deepening Search

Example:

Limit = 3

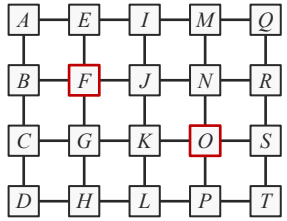


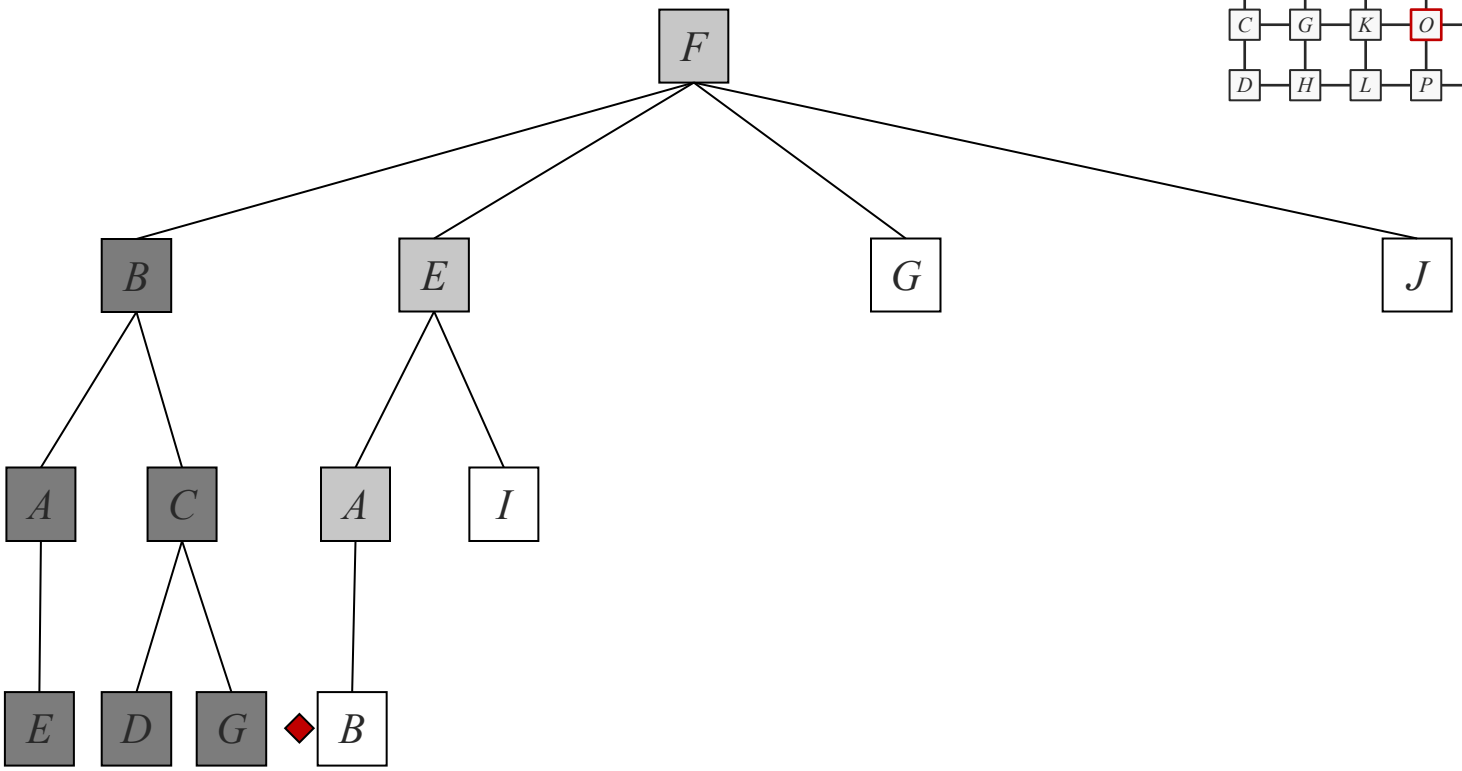


Iterative Deepening Search

Example:

Limit = 3

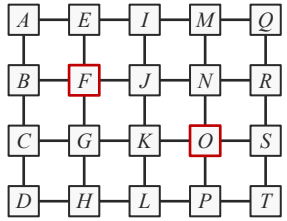


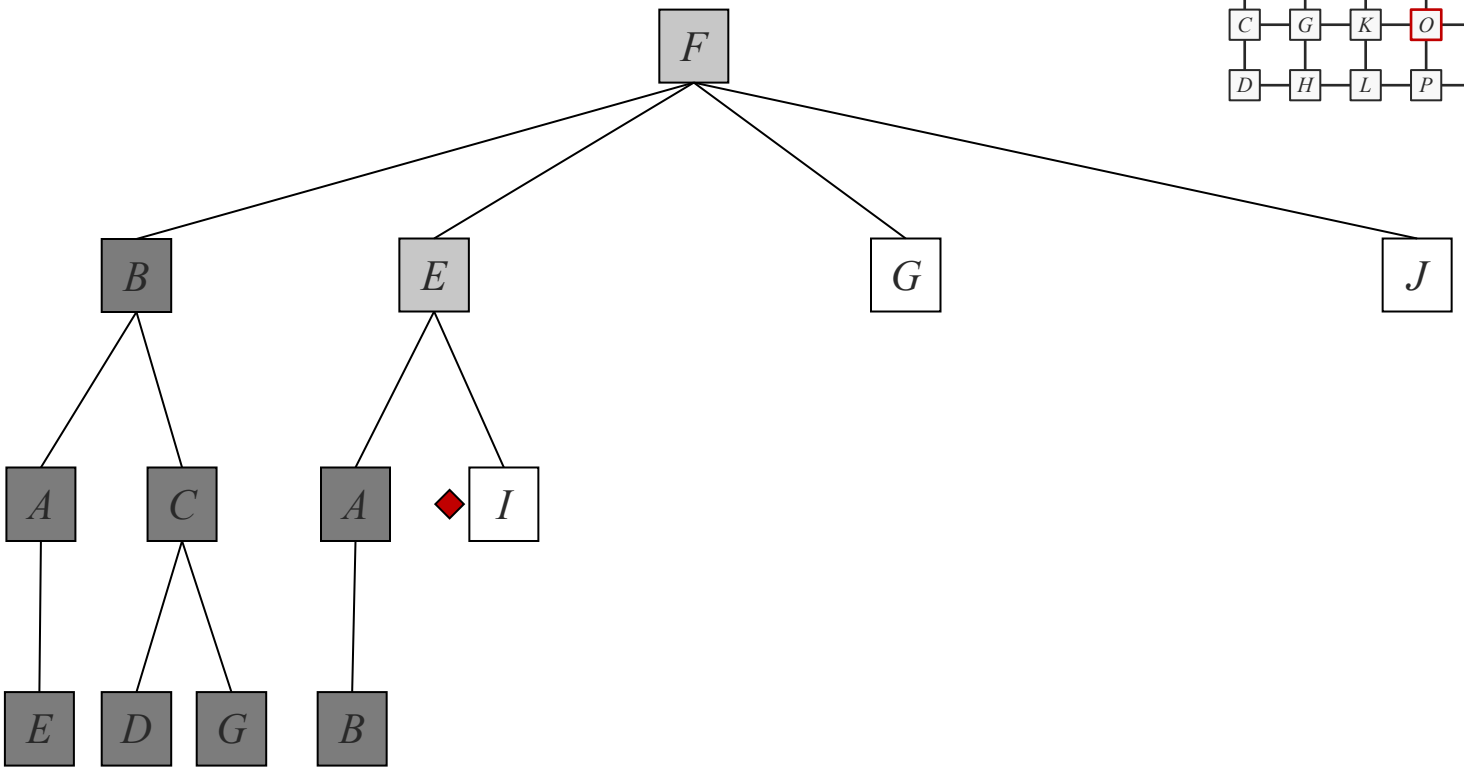


Iterative Deepening Search

Example:

Limit = 3

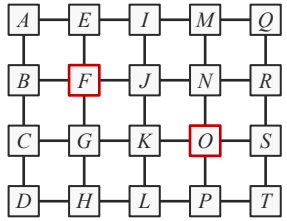


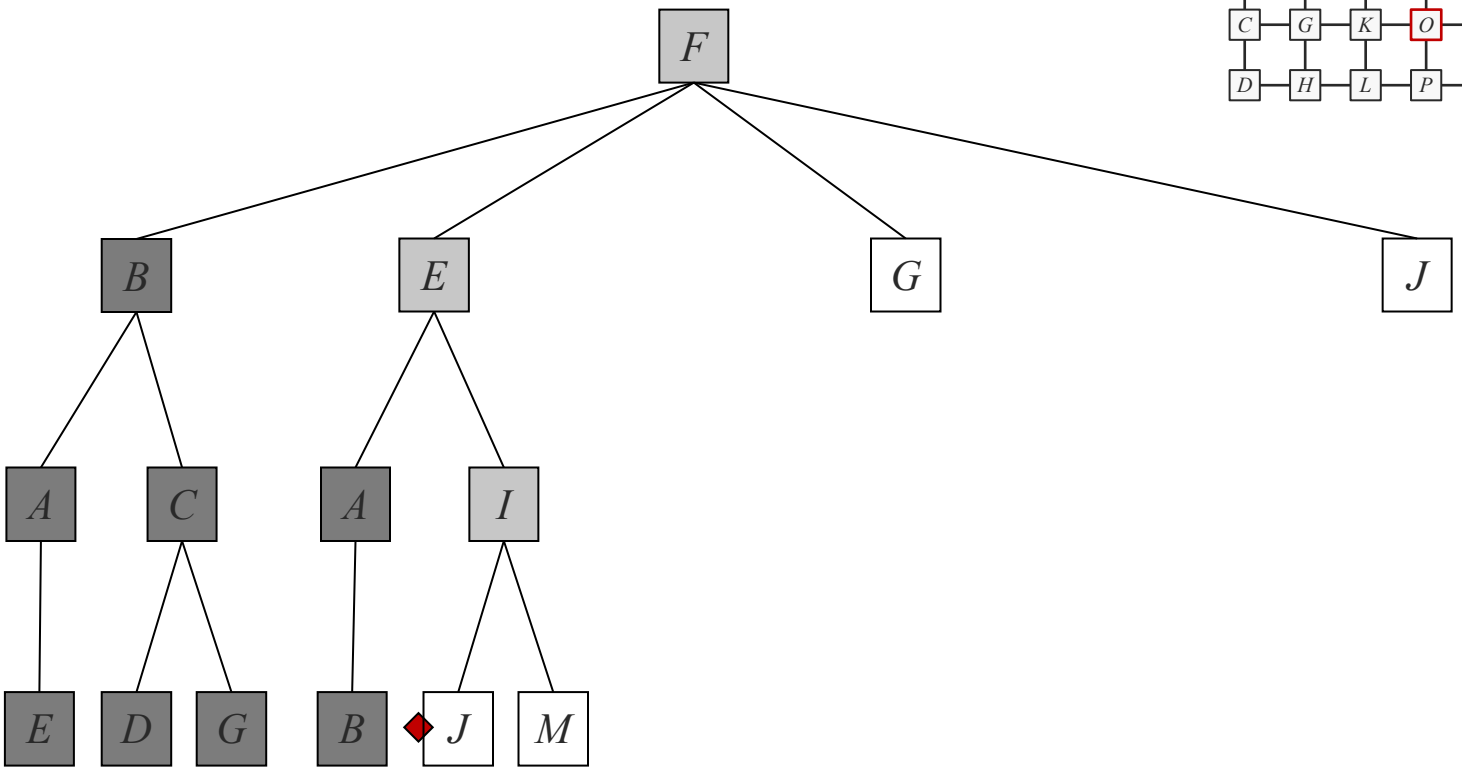


Iterative Deepening Search

Example:

Limit = 3

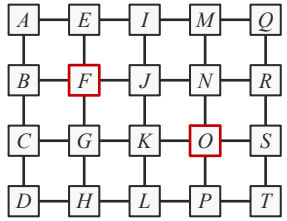


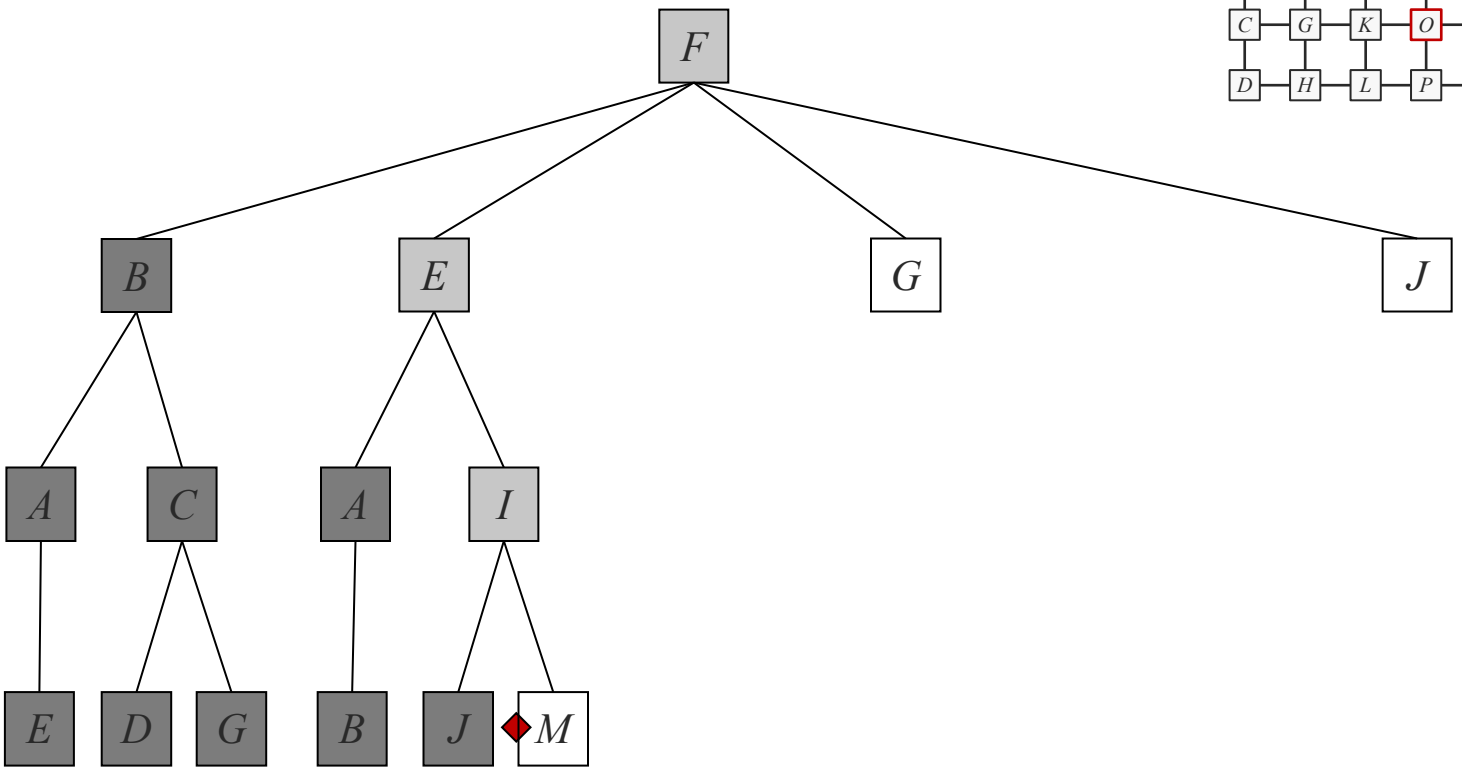


Iterative Deepening Search

Example:

Limit = 3

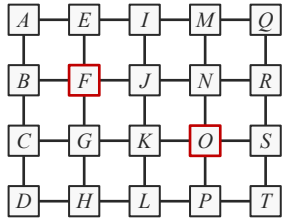


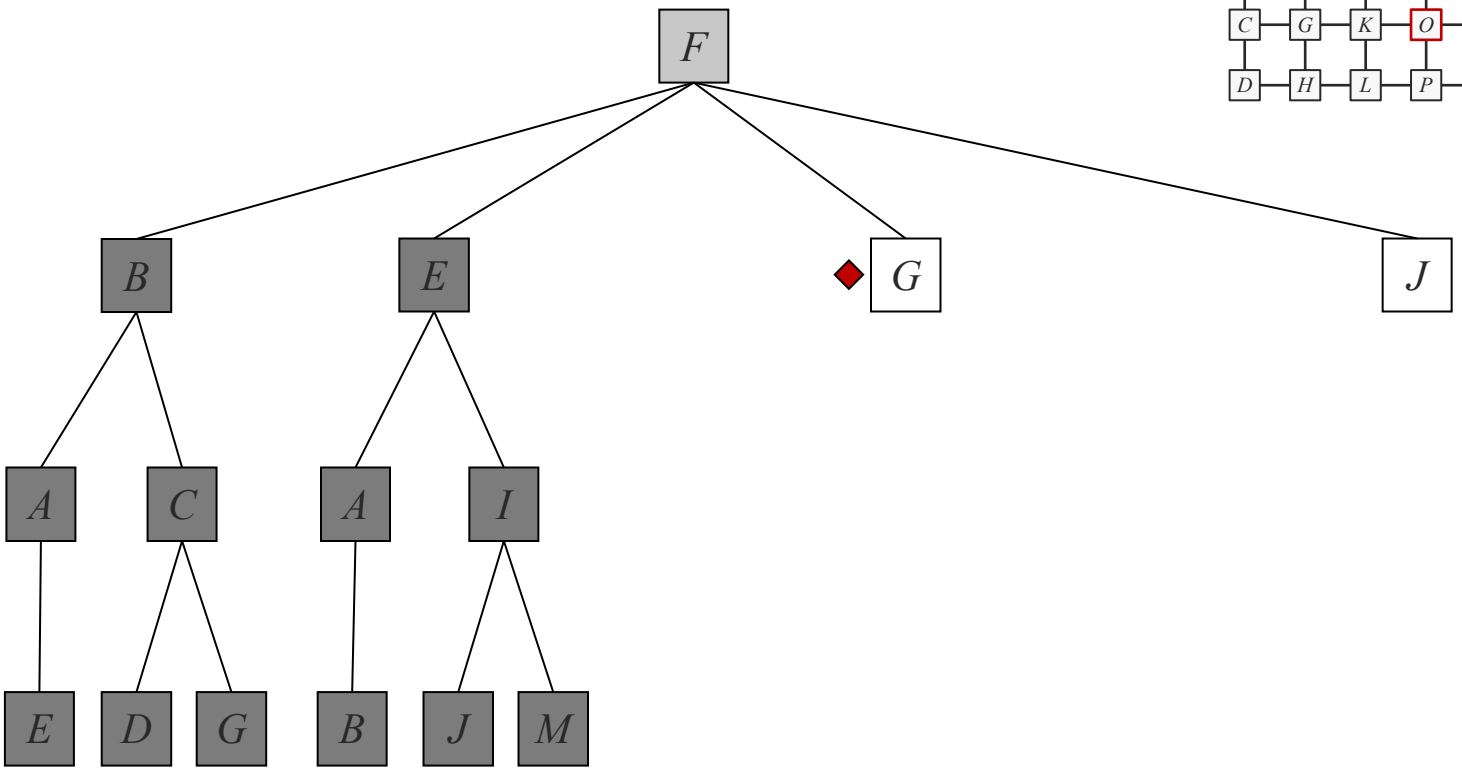


Iterative Deepening Search

Example:

Limit = 3

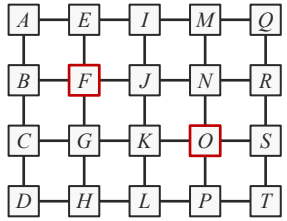


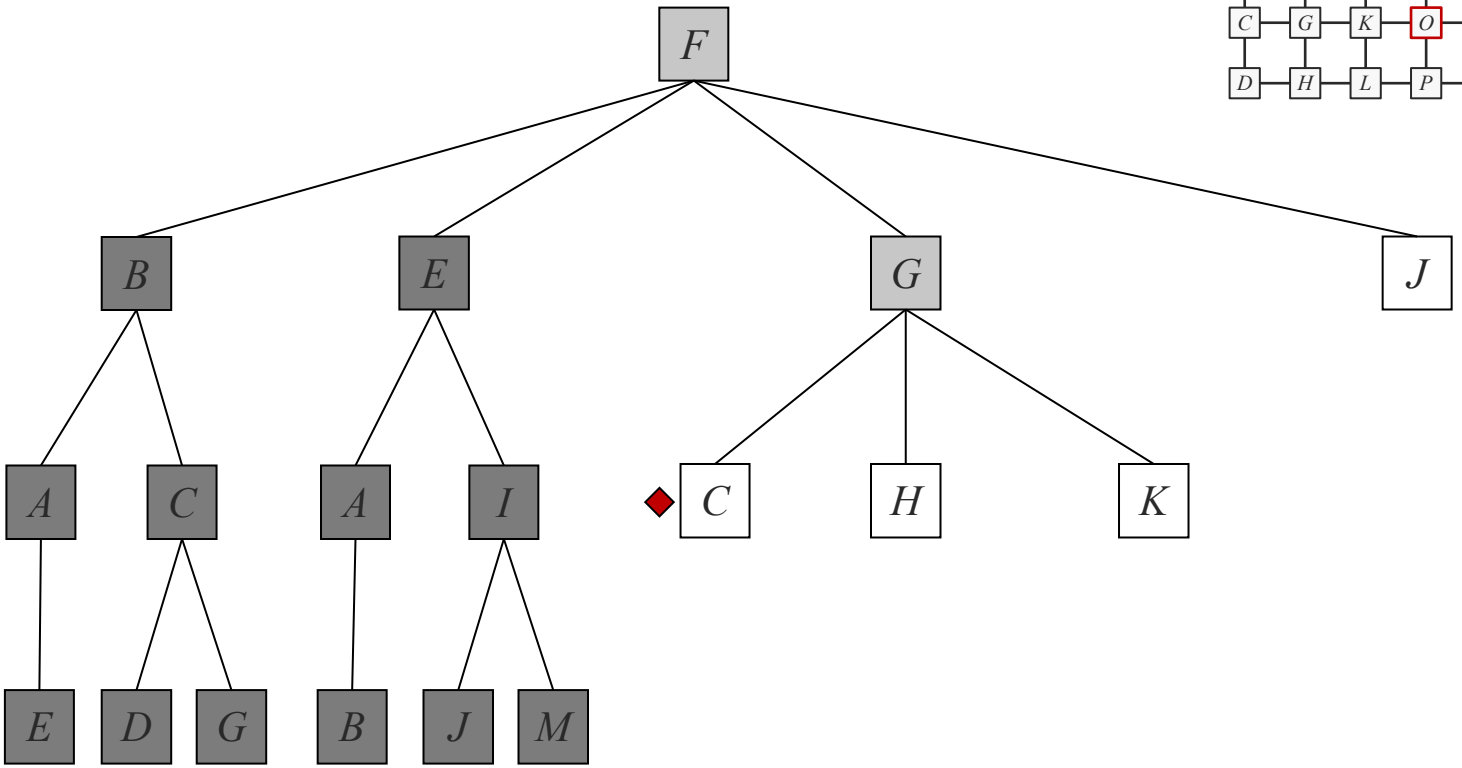


Iterative Deepening Search

Example:

Limit = 3

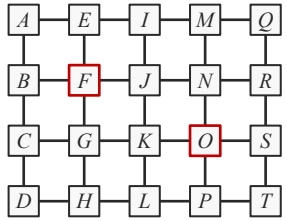


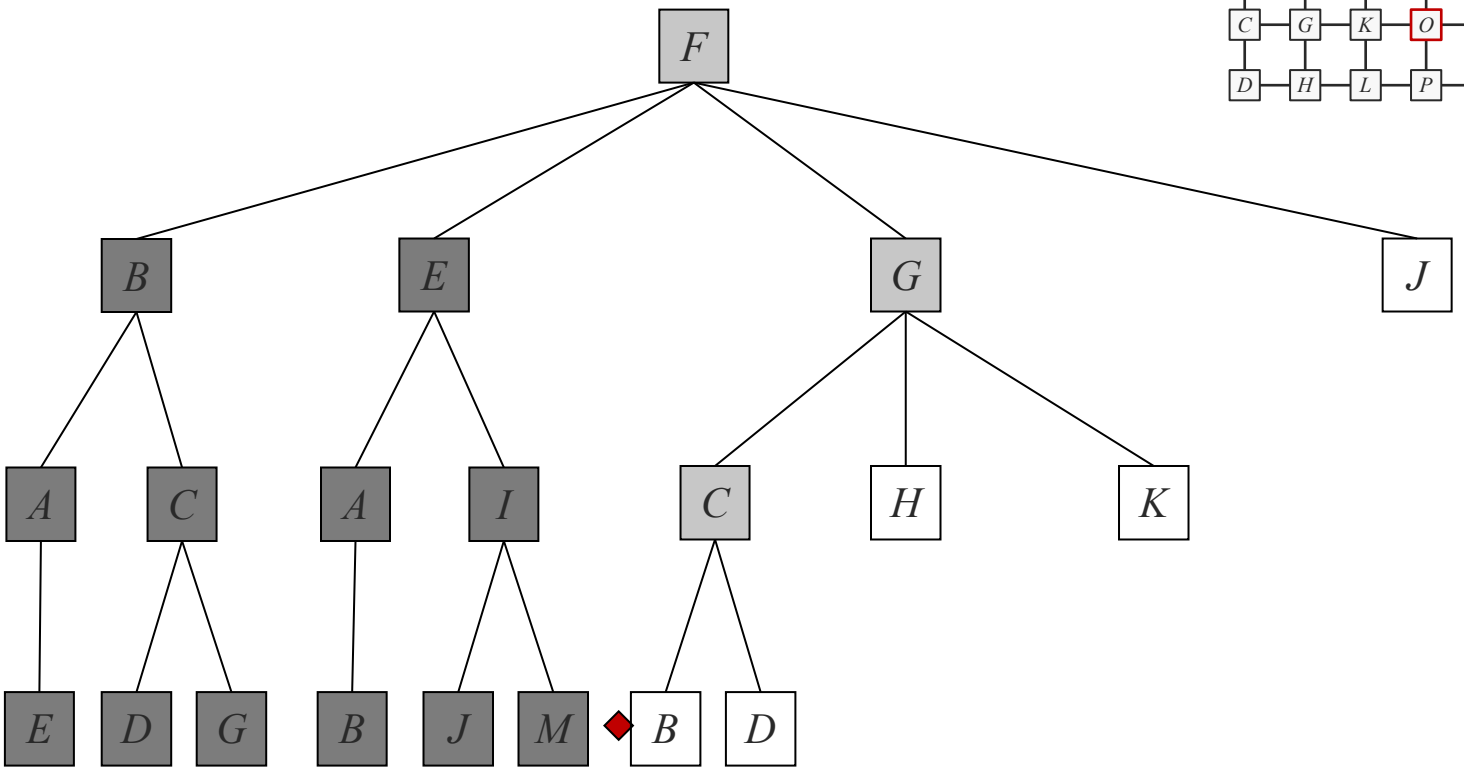


Iterative Deepening Search

Example:

Limit = 3

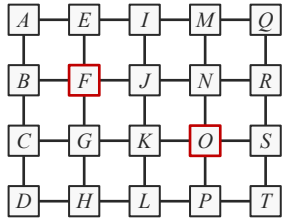


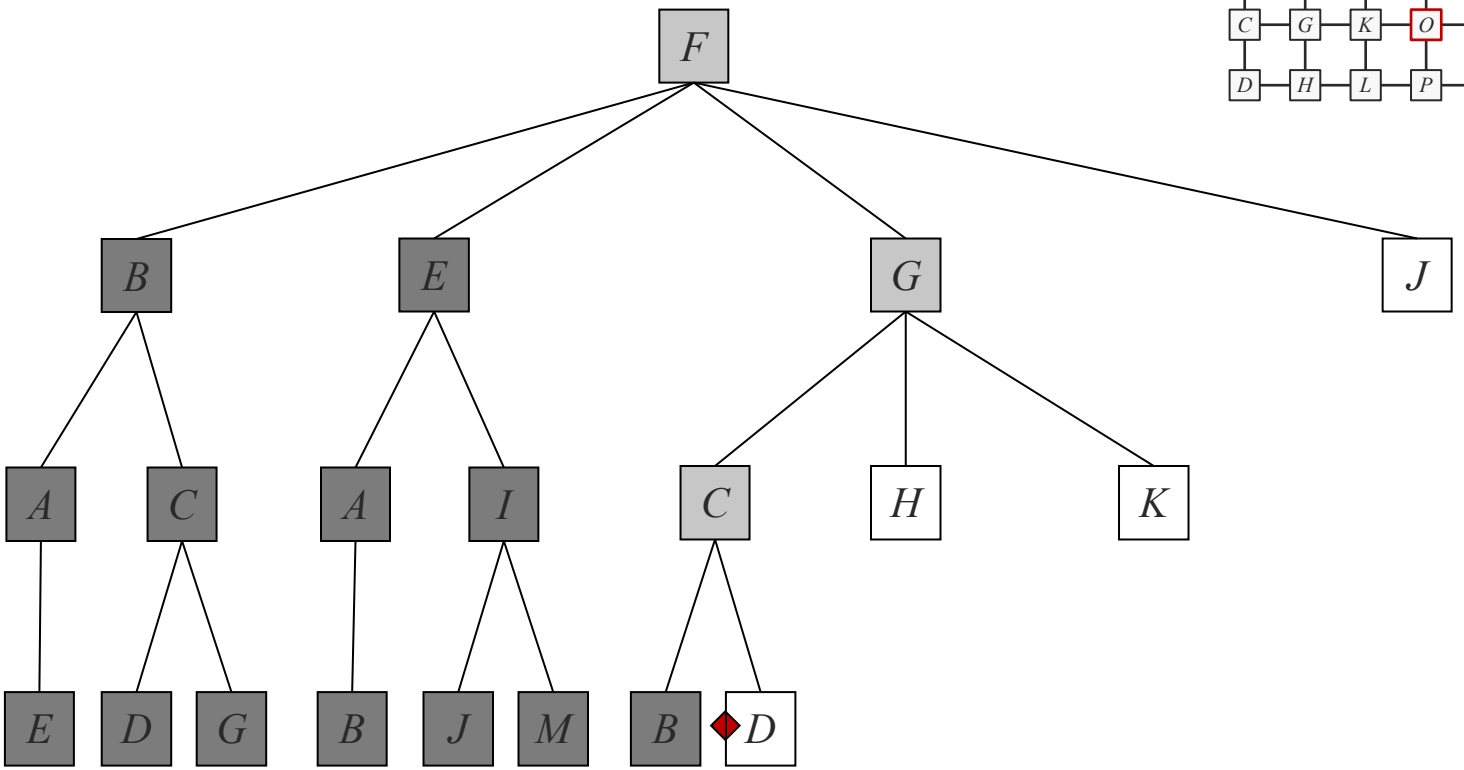


Iterative Deepening Search

Example:

Limit = 3

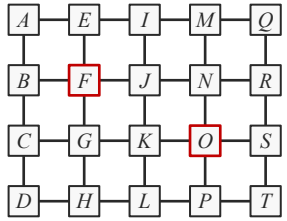


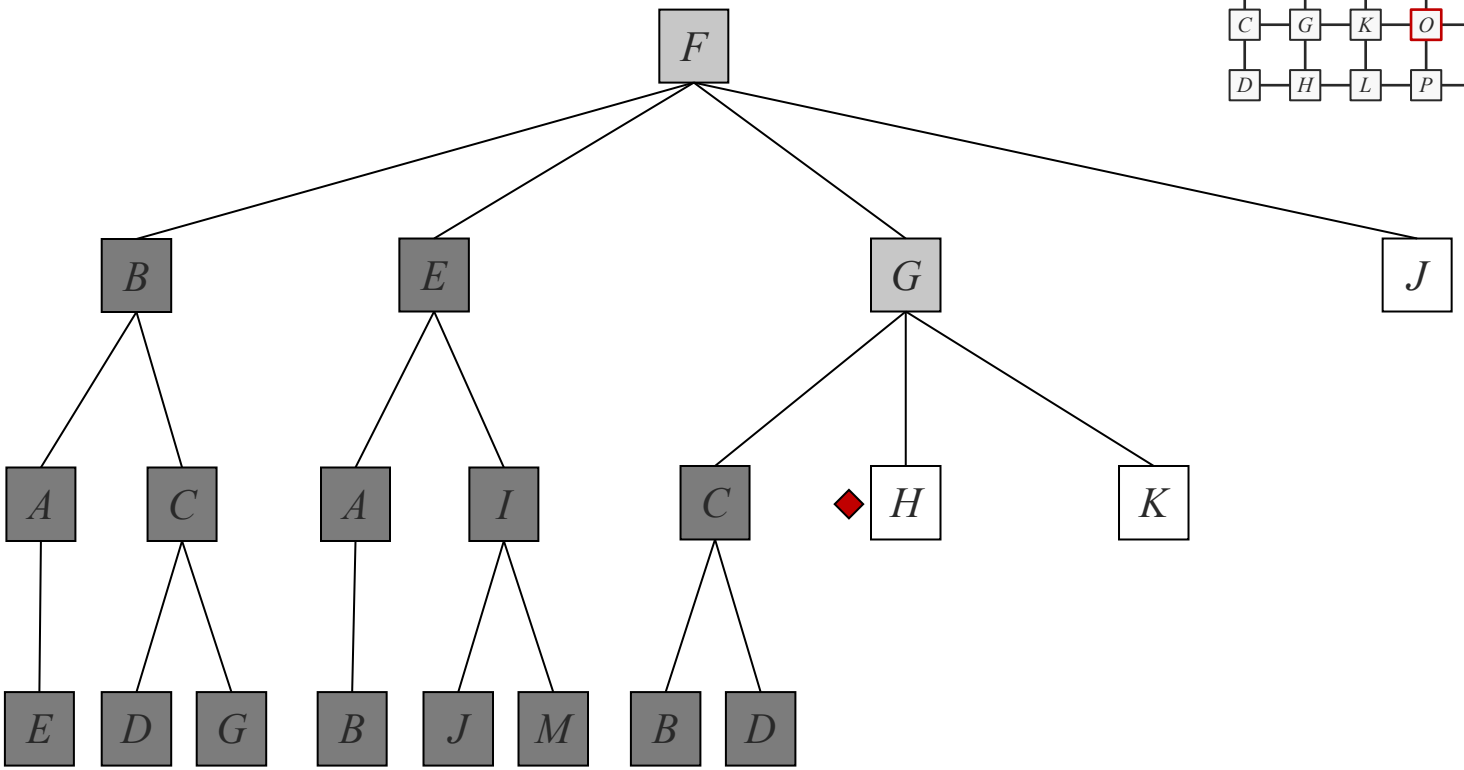


Iterative Deepening Search

Example:

Limit = 3

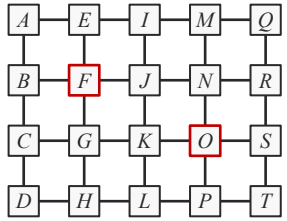


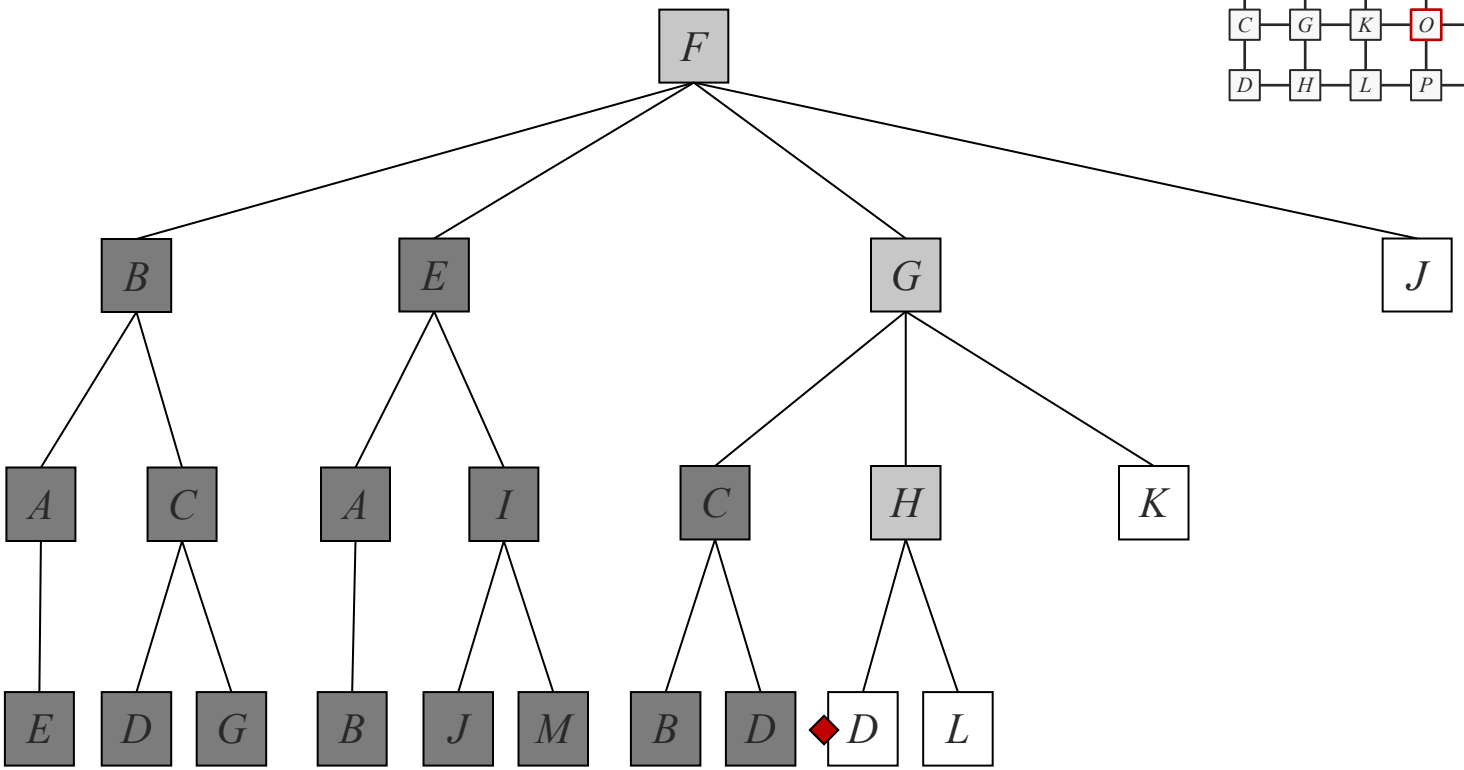


Iterative Deepening Search

Example:

Limit = 3

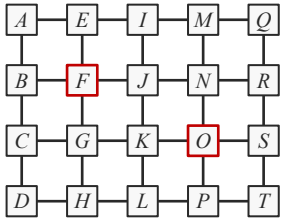


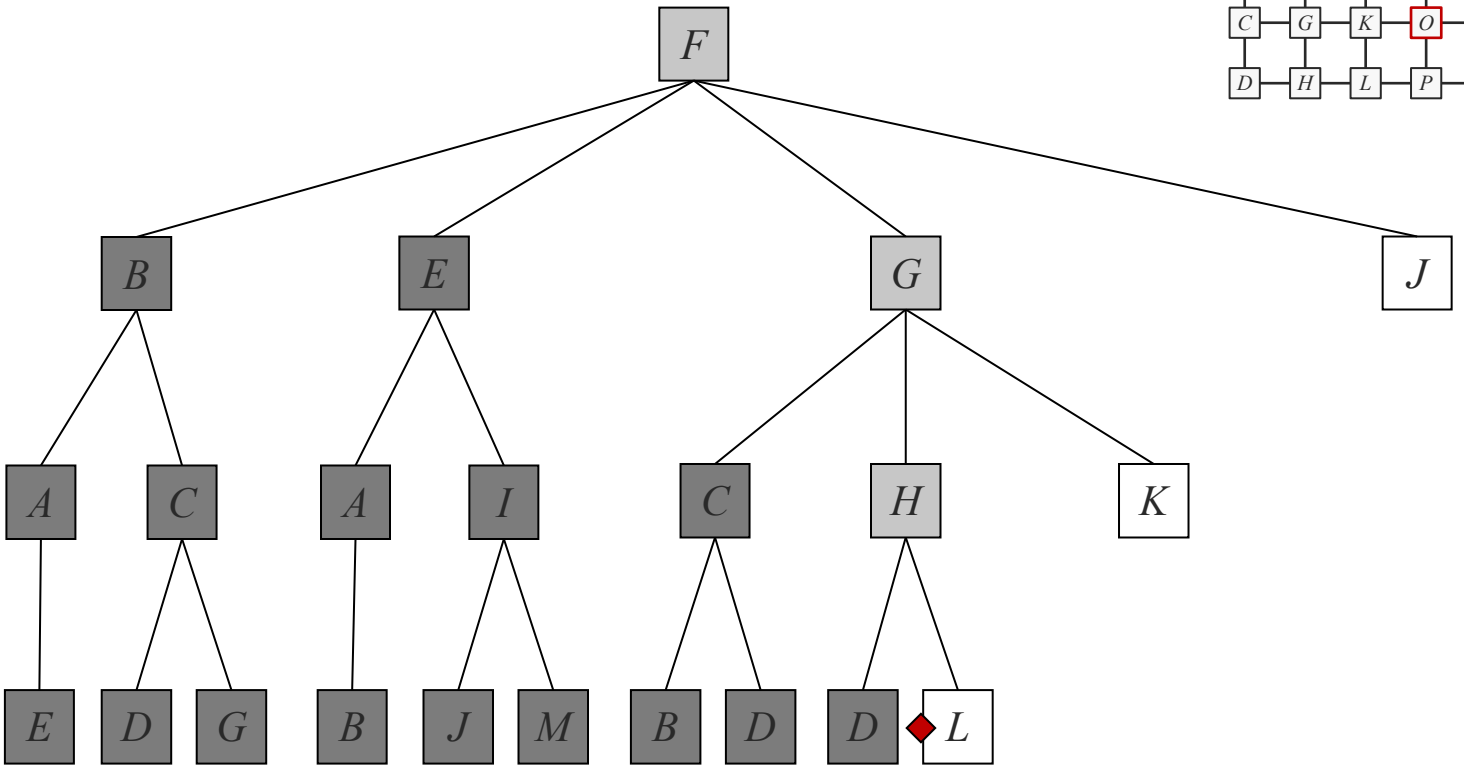


Iterative Deepening Search

Example:

Limit = 3

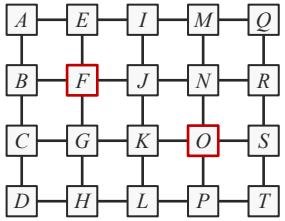


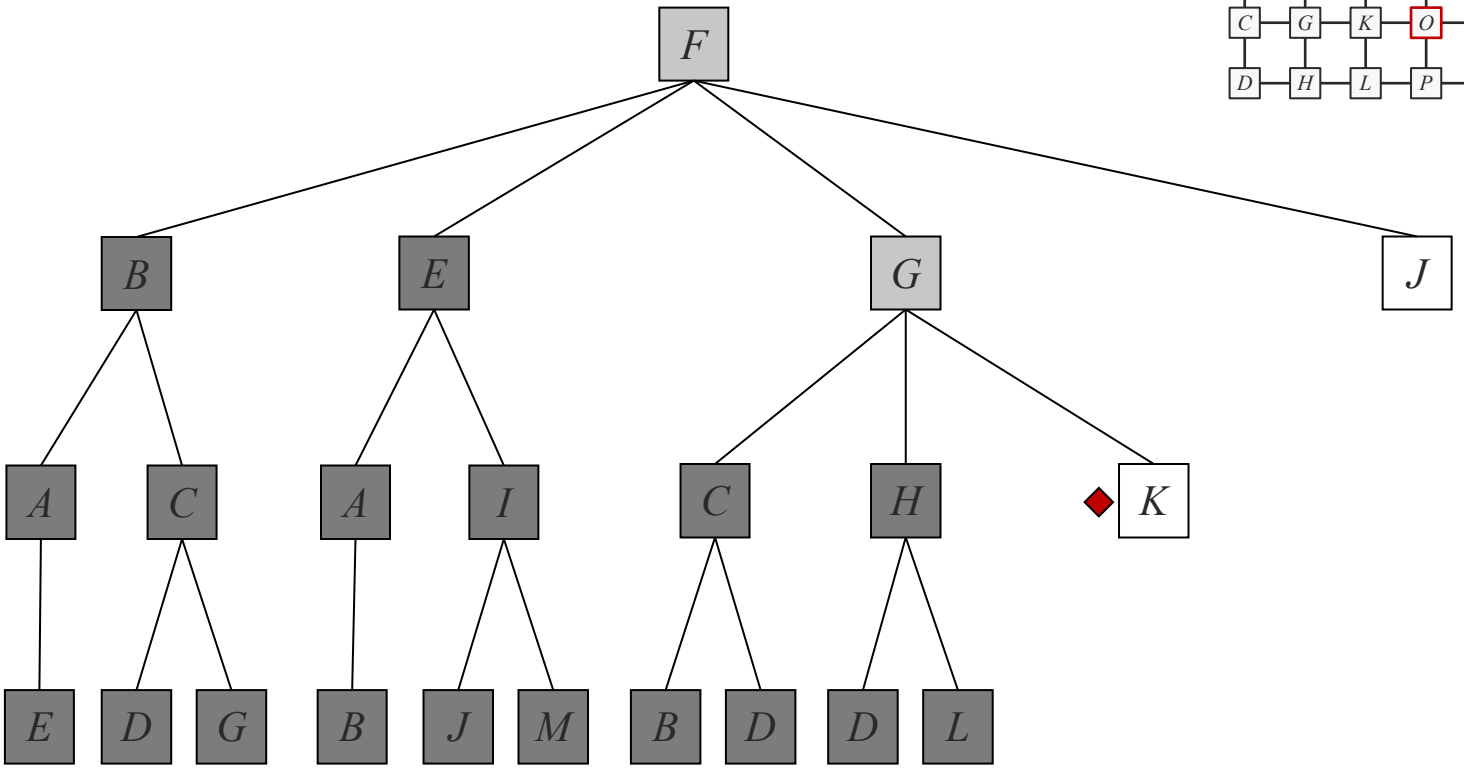


Iterative Deepening Search

Example:

Limit = 3

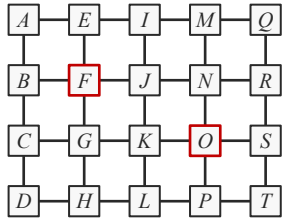


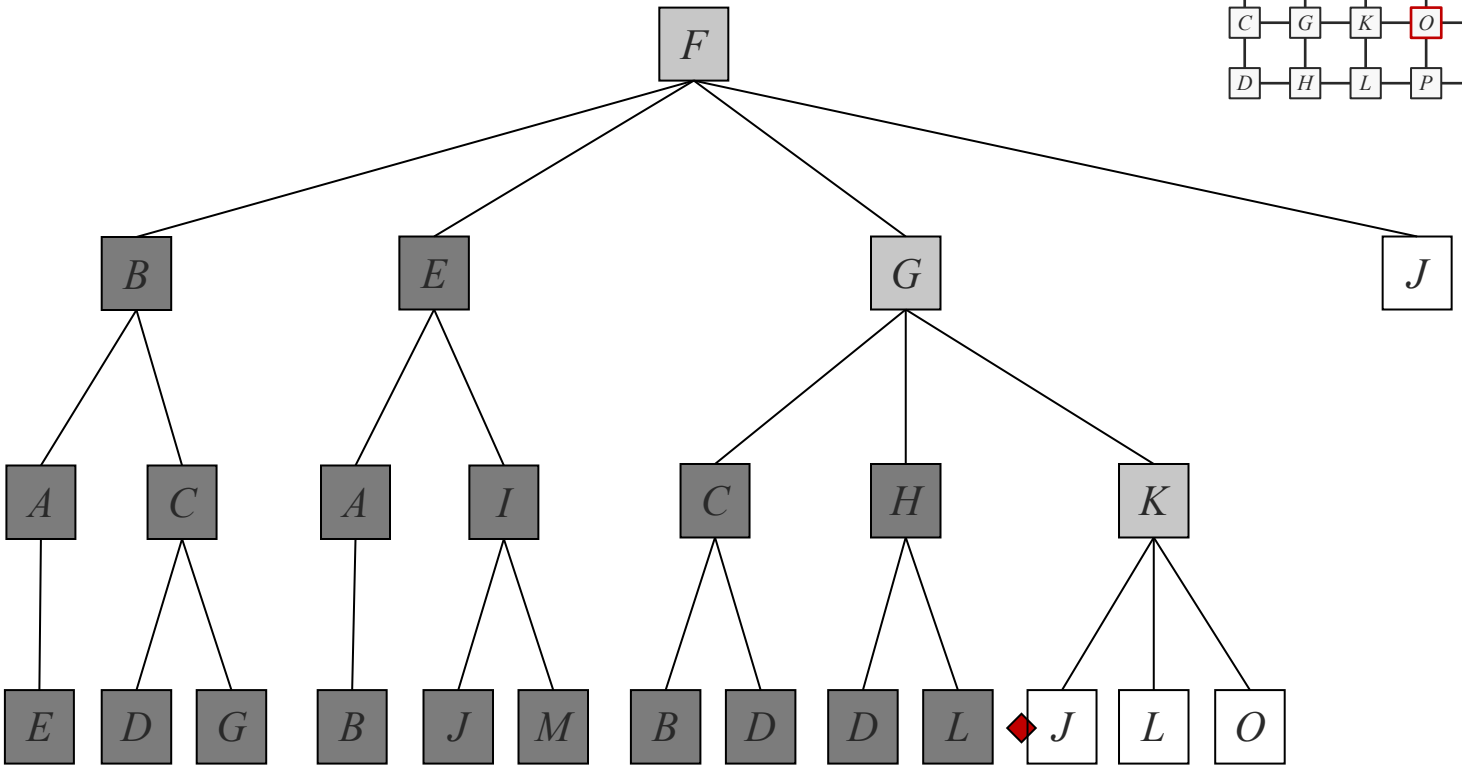


Iterative Deepening Search

Example:

Limit = 3

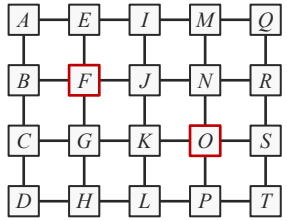


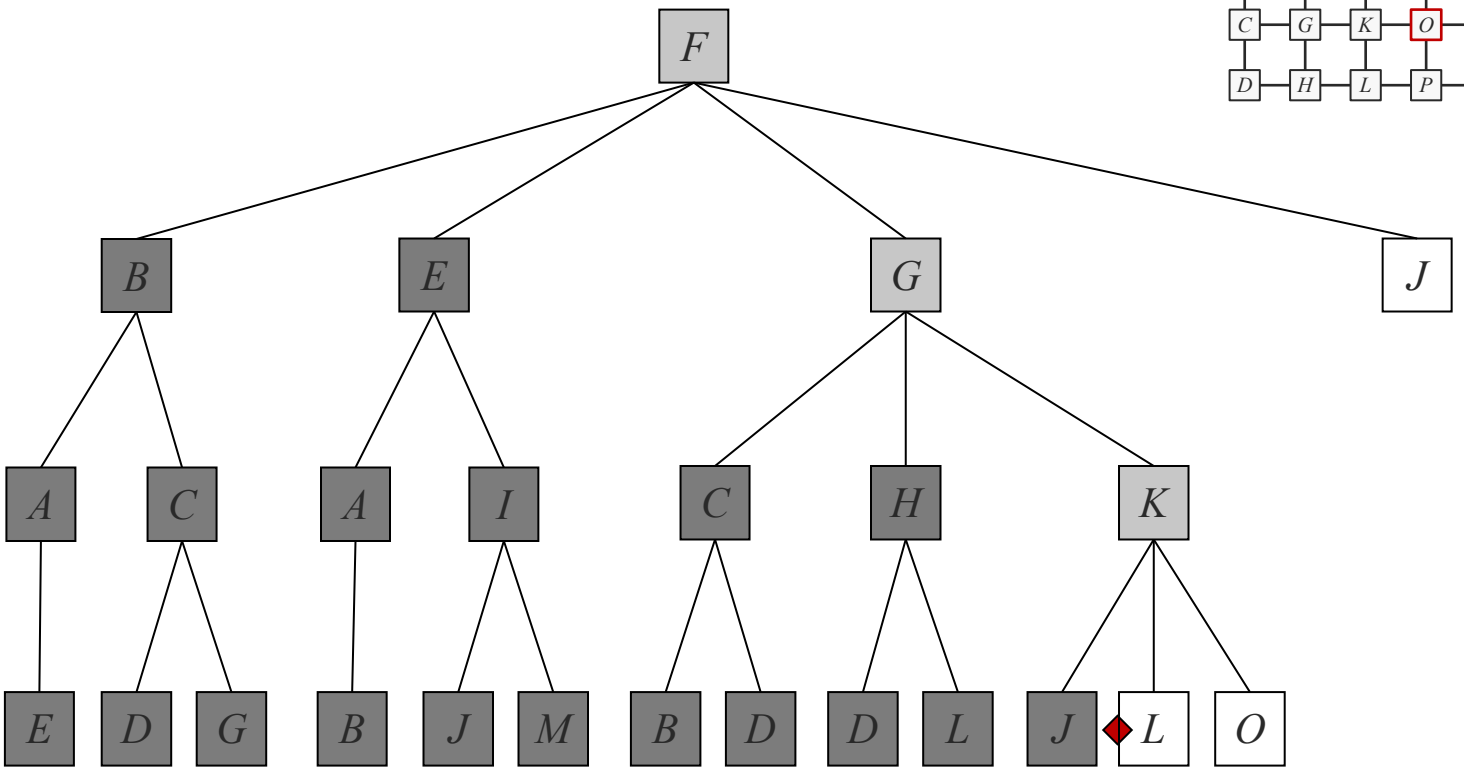


Iterative Deepening Search

Example:

Limit = 3

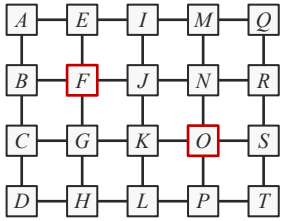


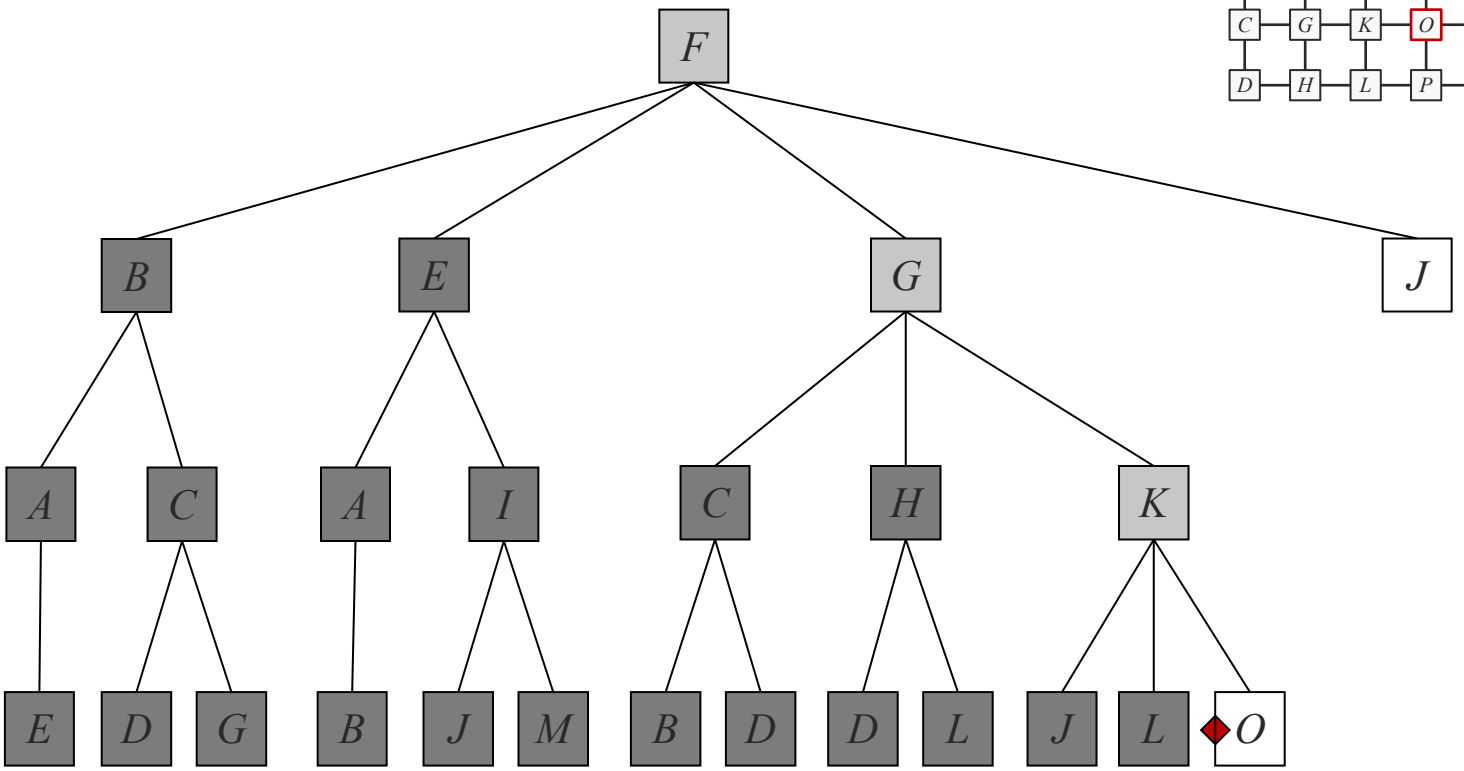


Iterative Deepening Search

Example:

Limit = 3

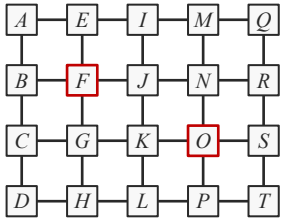


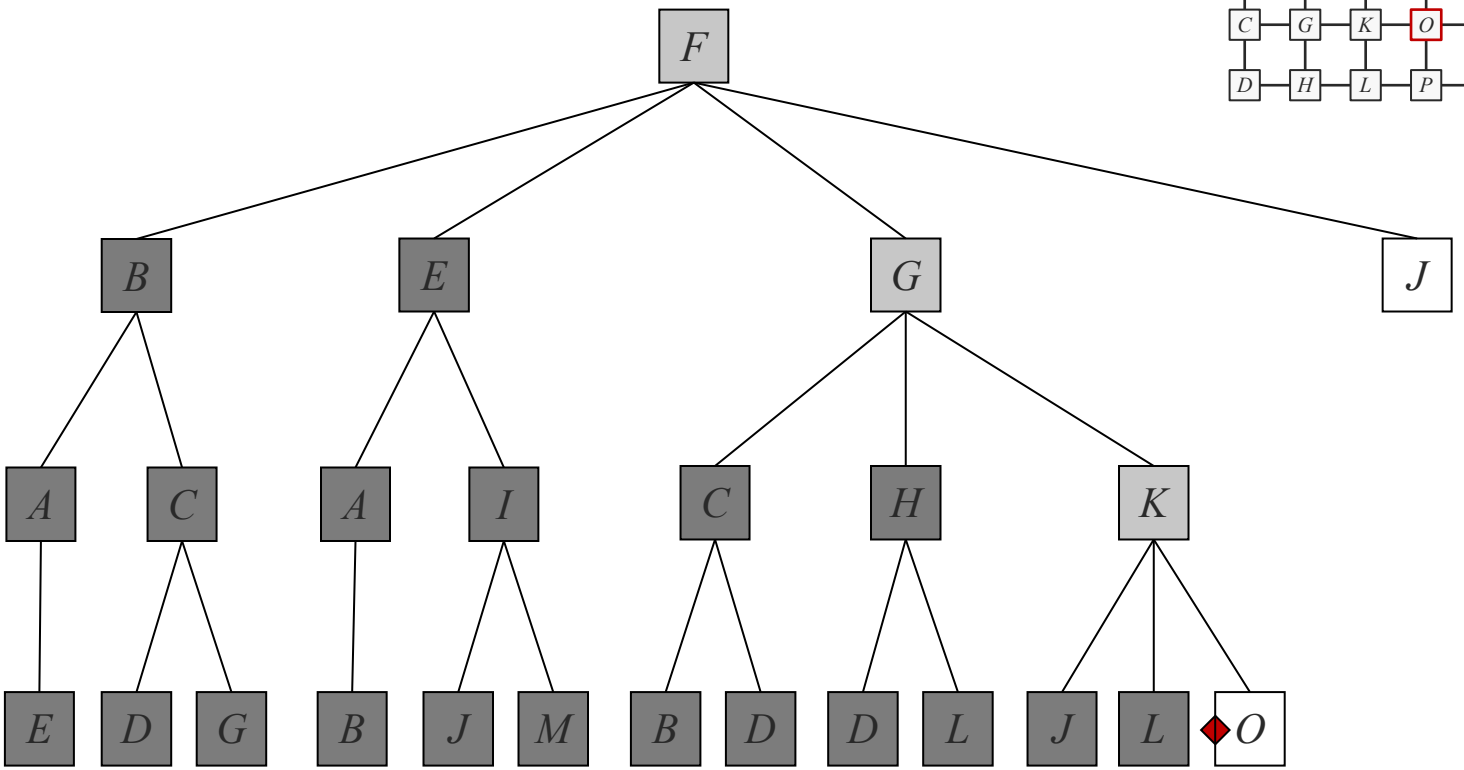


Iterative Deepening Search

Example:

Limit = 3

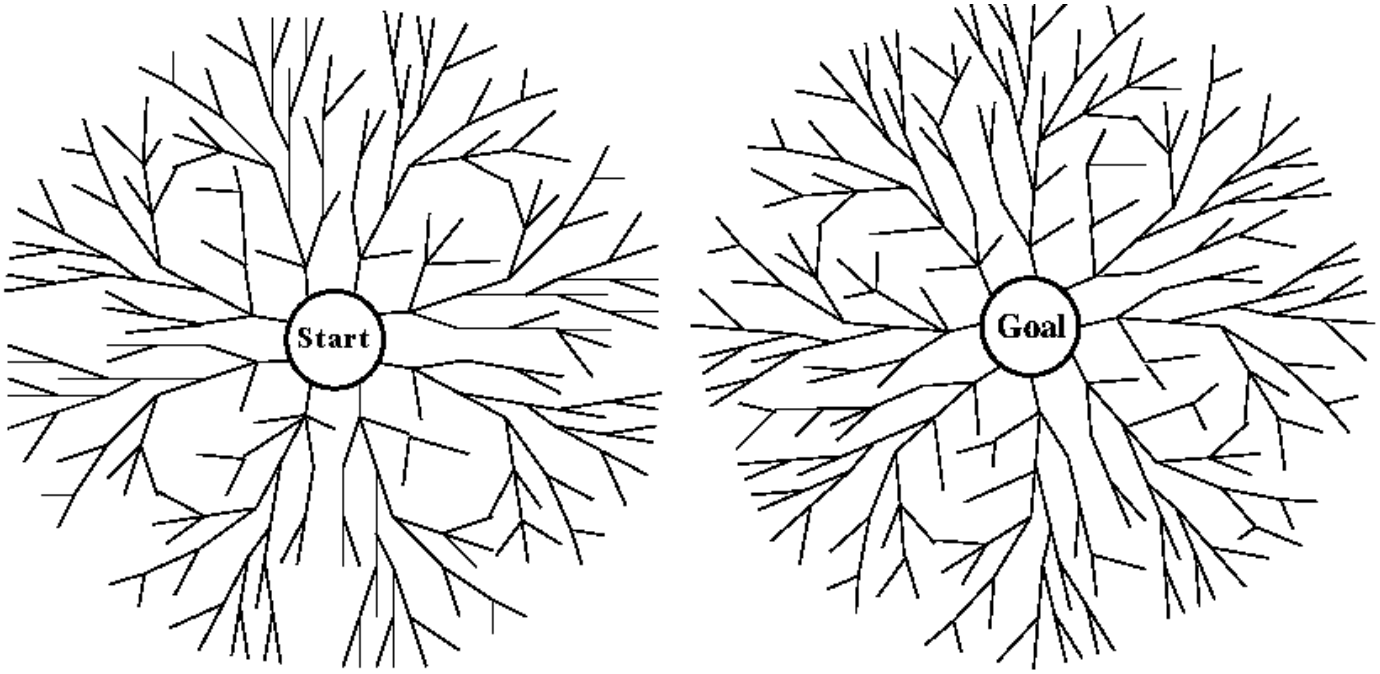




The path is $F - G - K - O$

Bidirectional Search

- Forward from the initial and backward from the goal simultaneously
- The solution will be found in $O(b^{d/2})$ $(2b^{d/2} << b^d)$



Bidirectional Search

- Issues:
  - Need to generate predecessors successively starting from the goal node

    (Calculating predecessors can be difficult)
  - What if there are multiple goal states?
  - What if we only have a description of the goal states?

    (E.g., the checkmate goal in chess)
  - Need an efficient way to check if each new node appears in the other half
  - What kind of search strategy in each half?

Comparing Uninformed Search Strategies

- b : branching factor
- d : depth of solution
- m : maximum depth of the search tree
- l : depth limit
- C^* : path cost of the optimal solution

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

[a] complete if b is finite; [b] complete if step cost $\geq \varepsilon$ for positive ε; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search

Tree Search vs. Graph Search

	May investigate		If implemented for			
	Loopy path	Redun-dant path	Breadth-first Search	Uniform-cost Search	Depth-first Search	Iterative deepening Search
Strict Tree Search	○	○	O.K. but less efficient	O.K. but less efficient	×	O.K. but less efficient
Modified Tree Search [1]	×	○	O.K.	O.K.	Best	Best
Strict Graph Search	×	×	Best	×	Not meaningful [3]	Not meaningful [3]
Modified Graph Search [2]	×	○	O.K.	Best	Not meaningful [3]	Not meaningful [3]

[1] avoids infinite loop by discarding any child that is the same as one of its ancestors;

[2] a child can replace the same one in the frontier if the former is better than the latter;

[3] meaningless because of the memory complexity