First Out) order. The node that enters OPEN earlier will be expanded earlier. This amounts to expanding the shallowest nodes first.

2.5.2 BFS illustrated

We will now consider the search space in Figure 1, and show how breadth first search works on this graph.

Step 1: Initially fringe contains only one node corresponding to the source state A.

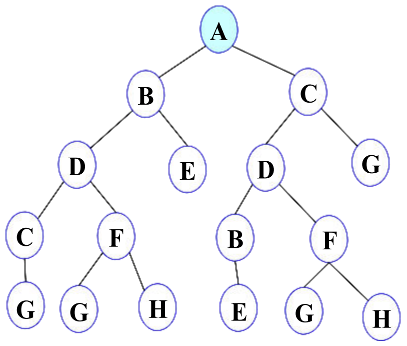


Figure 3

FRINGE: A

Step 2: A is removed from fringe. The node is expanded, and its children B and C are generated. They are placed at the back of fringe.

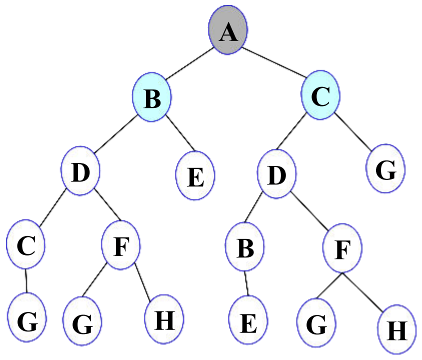


Figure 4

FRINGE: B C

Step 3: Node B is removed from fringe and is expanded. Its children D, E are generated and put at the back of fringe.

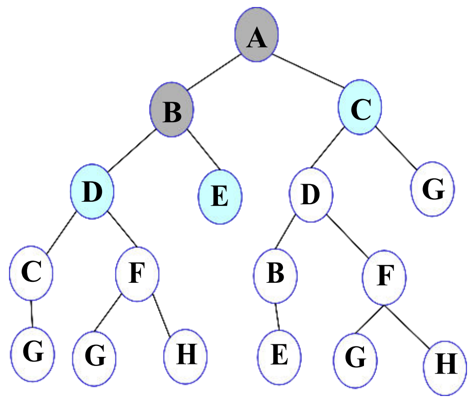


Figure 5

FRINGE: C D E

Step 4: Node C is removed from fringe and is expanded. Its children D and G are added to the back of fringe.

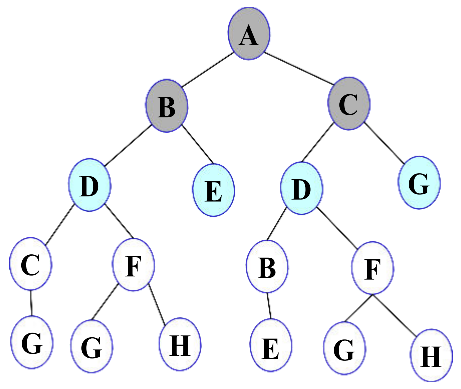


Figure 6

FRINGE: D E D G

Step 5: Node D is removed from fringe. Its children C and F are generated and added to the back of fringe.

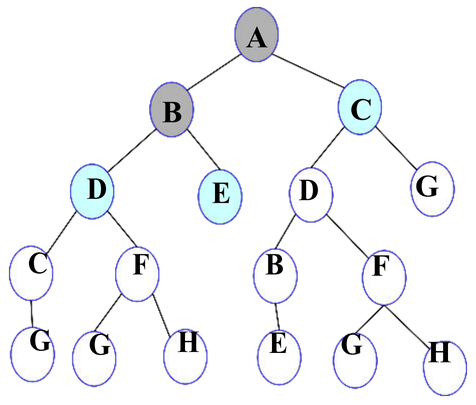
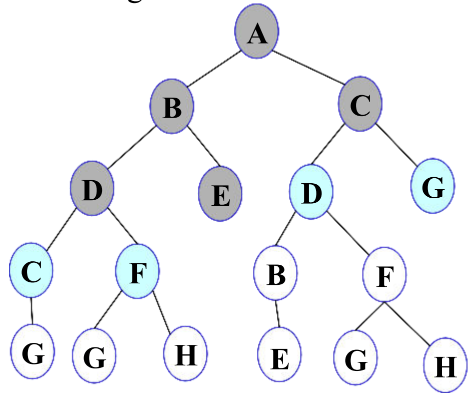


Figure 7

FRINGE: E D G C F

Step 6: Node E is removed from fringe. It has no children.



FRINGE: D G C F

Step 7: D is expanded, B and F are put in OPEN.

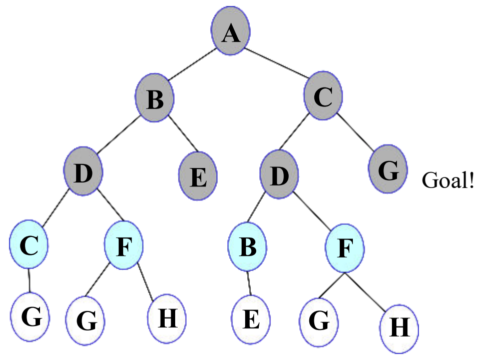


Figure 8

FRINGE: G C F B F

Step 8: G is selected for expansion. It is found to be a goal node. So the algorithm returns the path A C G by following the parent pointers of the node corresponding to G. The algorithm terminates.

2.5.3 Properties of Breadth-First Search

We will now explore certain properties of breadth first search. Let us consider a model of the search tree as shown in Figure 3. We assume that every non-leaf node has b children. Suppose that d is the depth o the shallowest goal node, and m is the depth of the node found first.

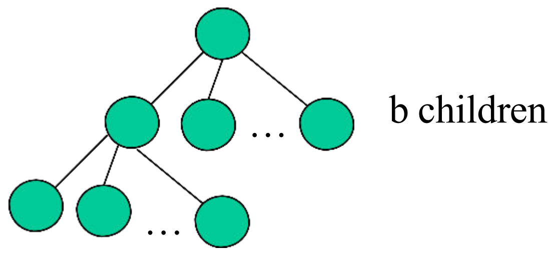


Figure 9: Model of a search tree with uniform branching factor b

Breadth first search is:

- Complete.
- The algorithm is optimal (i.e., admissible) if all operators have the same cost. Otherwise, breadth first search finds a solution with the shortest path length.
- The algorithm has exponential time and space complexity. Suppose the search tree can be modeled as a b-ary tree as shown in Figure 3. Then the time and space complexity of the algorithm is O(bd) where d is the depth of the solution and b is the branching factor (i.e., number of children) at each node.

A complete search tree of depth d where each non-leaf node has b children, has a total of

$$\mathbf{1 + b + b^2 + ... + b^d = (b^{(d+1)} - 1)/(b-1)\ nodes}$$

Consider a complete search tree of depth 15, where every node at depths 0 to14 has 10 children and every node at depth 15 is a leaf node. The complete search tree in this case will have $O(10^{15})$ nodes. If BFS expands 10000 nodes per second and each node uses 100 bytes of storage, then BFS will take 3500 years to run in the worst case, and it will use 11100 terabytes of memory. So you can see that the breadth first search algorithm cannot be effectively used unless the search space is quite small. You may also observe that even if you have all the time at your disposal, the search algorithm cannot run because it will run out of memory very soon.

Advantages of Breadth First Search

Finds the path of minimal length to the goal.

Disadvantages of Breadth First Search

Requires the generation and storage of a tree whose size is exponential the the depth of the shallowest goal node

2.6 Uniform-cost search

This algorithm is by Dijkstra [1959]. The algorithm expands nodes in the order of their cost from the source.

We have discussed that operators are associated with costs. The path cost is usually taken to be the sum of the step costs.

In uniform cost search the newly generated nodes are put in OPEN according to their path costs. This ensures that when a node is selected for expansion it is a node with the cheapest cost among the nodes in OPEN.
Let g(n) = cost of the path from the start node to the current node n. Sort nodes by increasing value of g.
Some properties of this search algorithm are:

- Complete
- Optimal/Admissible
- Exponential time and space complexity, $O(b^d)$

2.7 Depth first Search

2.7.1 Algorithm

```
Depth First Search
Let *fringe* be a list containing the initial state
Loop
    if fringe is empty return failure
    Node ← remove-first (fringe)
     if Node is a goal
        then return the path from initial state to Node
    else generate all successors of Node, and
        merge the newly generated nodes into fringe
        add generated nodes to the front of fringe
End Loop
```

The depth first search algorithm puts newly generated nodes in the front of OPEN. This results in expanding the deepest node first. Thus the nodes in OPEN follow a LIFO order (Last In First Out). OPEN is thus implemented using a stack data structure.

2.7.2 DFS illustrated

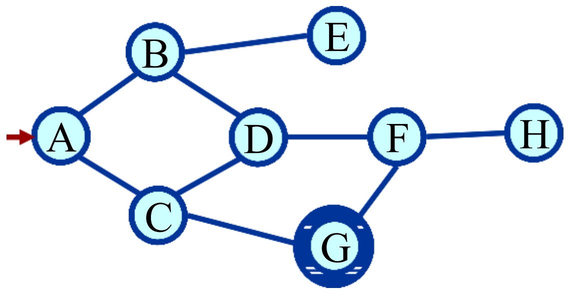


Figure 10

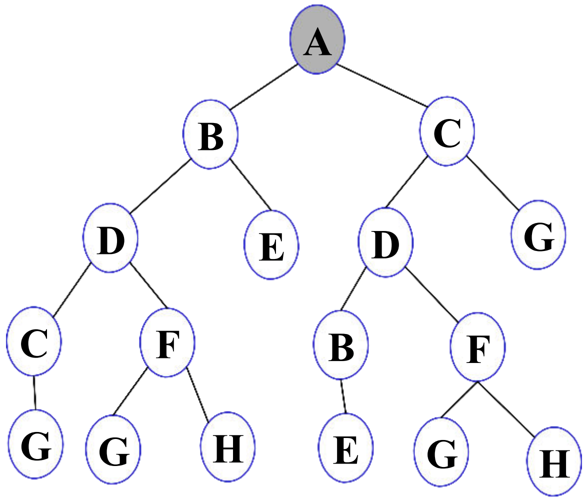


Figure 11: Search tree for the state space graph in Figure 34

Let us now run Depth First Search on the search space given in Figure 34, and trace its progress.

Step 1: Initially fringe contains only the node for A.

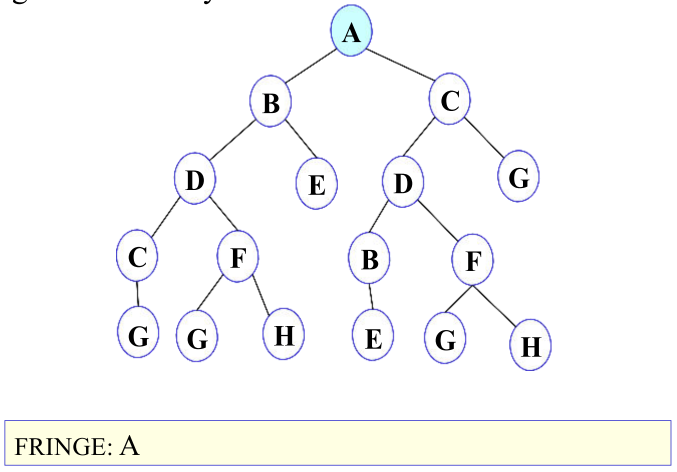
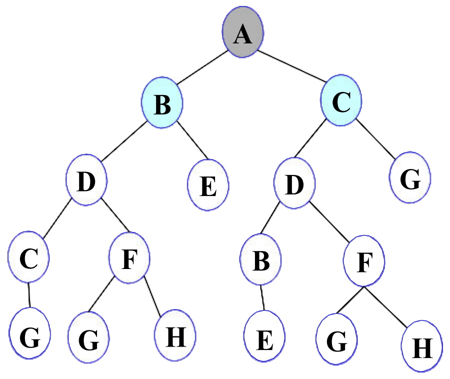


Figure 12

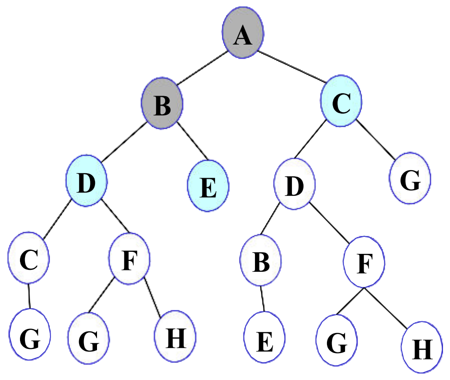
Step 2: A is removed from fringe. A is expanded and its children B and C are put in front of fringe.



FRINGE: B C

Figure 13

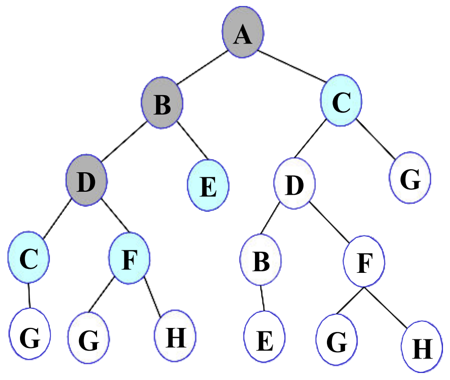
Step 3: Node B is removed from fringe, and its children D and E are pushed in front of fringe.



FRINGE: D E C

Figure 14

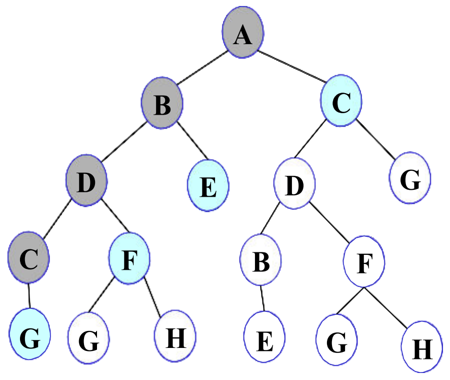
Step 4: Node D is removed from fringe. C and F are pushed in front of fringe.



FRINGE: C F E C

Figure 15

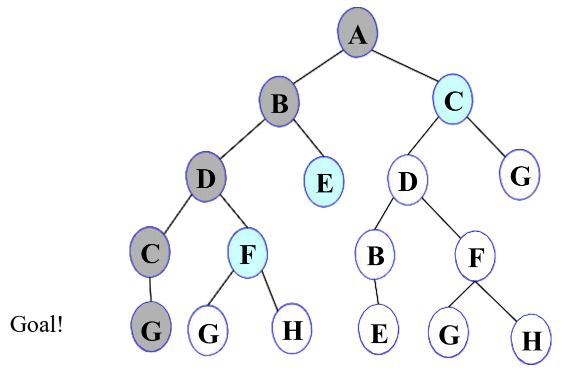
Step 5: Node C is removed from fringe. Its child G is pushed in front of fringe.



FRINGE: G F E C

Figure 16

Step 6: Node G is expanded and found to be a goal node. The solution path A-B-D-C-G is returned and the algorithm terminates.



FRINGE: **G** F E C

Figure 17

2.7.3 Properties of Depth First Search

Let us now examine some properties of the DFS algorithm. The algorithm takes exponential time. If N is the maximum depth of a node in the search space, in the worst case the algorithm will take time $O(b^d)$. However the space taken is linear in the depth of the search tree, O(bN).
Note that the time taken by the algorithm is related to the maximum depth of the search tree. If the search tree has infinite depth, the algorithm may not terminate. This can happen if the search space is infinite. It can also happen if the search space contains cycles. The latter case can be handled by checking for cycles in the algorithm. Thus Depth First Search is not complete.

2.7.4 Depth Limited Search

A variation of Depth First Search circumvents the above problem by keeping a depth bound. Nodes are only expanded if they have depth less than the bound. This algorithm is known as depth-limited search.

```
Depth limited search (limit)
Let fringe be a list containing the initial state
Loop
      if fringe is empty return failure
      Node ⇐ remove-first (fringe)
      if Node is a goal
         then return the path from initial state to Node
      else if depth of Node = limit return cutoff
      else add generated nodes to the front of fringe
End Loop
```

.

2.7.5 Depth-First Iterative Deepening (DFID)

First do DFS to depth 0 (i.e., treat start node as having no successors), then, if no solution found, do DFS to depth 1, etc.

```
DFID
until solution found do
  DFS with depth cutoff c
  c = c+1
```

Advantage

- Linear memory requirements of depth-first search
- Guarantee for goal node of minimal depth

Procedure

Successive depth-first searches are conducted – each with depth bounds increasing by 1

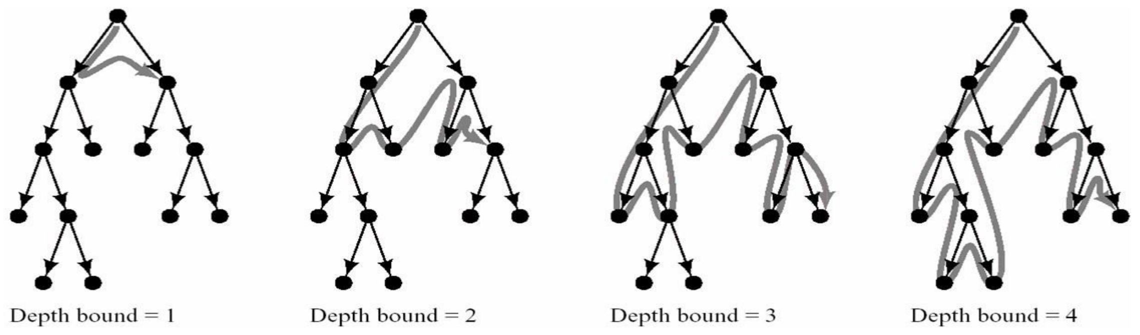


Figure 18: Depth First Iterative Deepening

Properties

For large *d* the ratio of the number of nodes expanded by DFID compared to that of DFS is given by *b*/(*b*-*1*).
For a branching factor of 10 and deep goals, 11% more nodes expansion in iterative-deepening search than breadth-first search

The algorithm is

- Complete
- Optimal/Admissible if all operators have the same cost. Otherwise, not optimal but guarantees finding solution of shortest length (like BFS).
- Time complexity is a little worse than BFS or DFS because nodes near the top of the search tree are generated multiple times, but because almost all of the nodes are near the bottom of a tree, the worst case time complexity is still exponential, $O(b^d)$

If branching factor is b and solution is at depth d, then nodes at depth d are generated once, nodes at depth d-1 are generated twice, etc.
Hence $b^d + 2b^{(d-1)} + ... + db <= b^d / (1 - 1/b)^2 = O(b^d)$.

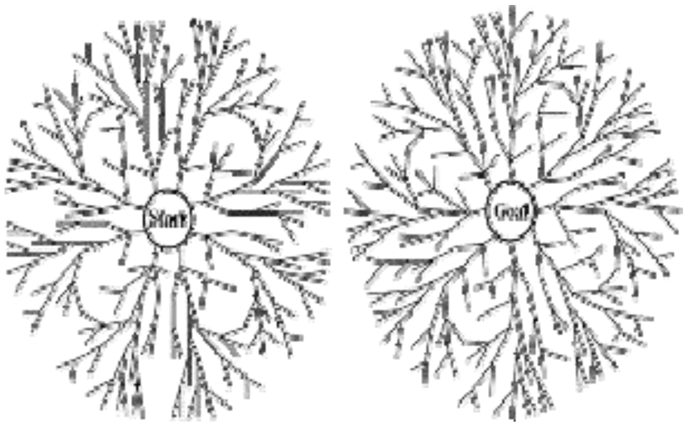
- **Linear space complexity**, O(bd), like DFS

Depth First Iterative Deepening combines the advantage of BFS (i.e., completeness) with the advantages of DFS (i.e., limited space and finds longer paths more quickly)
This algorithm is generally preferred for **large state spaces** where the **solution depth is unknown.**

There is a related technique called *iterative broadening* is useful when there are many goal nodes. This algorithm works by first constructing a search tree by expanding only one child per node. In the 2nd iteration, two children are expanded, and in the ith iteration I children are expanded.

Bi-directional search

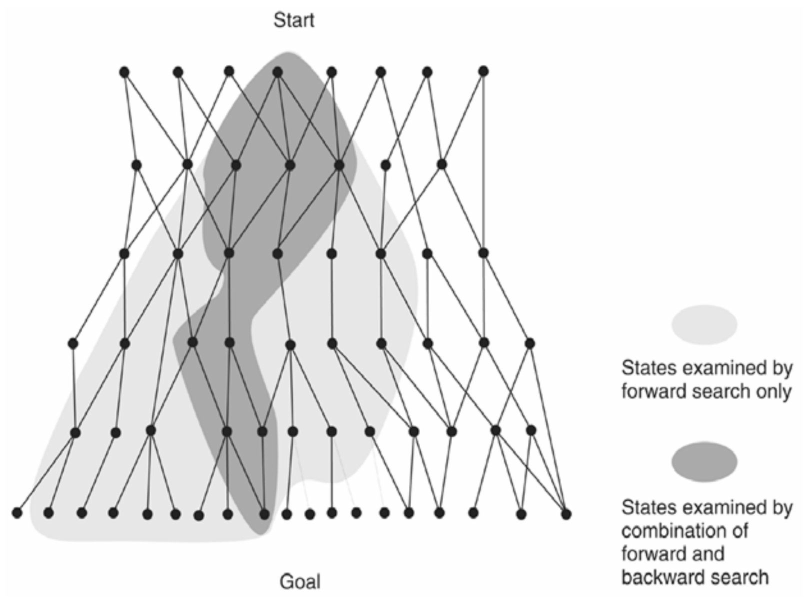
Suppose that the search problem is such that the arcs are bidirectional. That is, if there is an operator that maps from state A to state B, there is another operator that maps from state B to state A. Many search problems have reversible arcs. 8-puzzle, 15-puzzle, path planning etc are examples of search problems. However there are other state space search formulations which do not have this property. The water jug problem is a problem that does not have this property. But if the arcs are reversible, you can see that instead of starting from the start state and searching for the goal, one may start from a goal state and try reaching the start state. If there is a single state that satisfies the goal property, the search problems are identical.
How do we search backwards from goal? One should be able to generate predecessor states. Predecessors of node n are all the nodes that have n as successor. This is the motivation to consider bidirectional search.



Algorithm: Bidirectional search involves alternate searching from the start state toward the goal and from the goal state toward the start. The algorithm stops when the frontiers intersect.

A search algorithm has to be selected for each half. How does the algorithm know when the frontiers of the search tree intersect? For bidirectional search to work well, there must be an efficient way to check whether a given node belongs to the other search tree.

Bidirectional search can sometimes lead to finding a solution more quickly. The reason can be seen from inspecting the following figure.


Start
Goal
States examined by forward search only
States examined by combination of forward and backward search

Also note that the algorithm works well only when there are unique start and goal states. Question: How can you make bidirectional search work if you have 2 possible goal states?

Time and Space Complexities

Consider a search space with branching factor b. Suppose that the goal is d steps away from the start state. Breadth first search will expand O(b^d) nodes.
If we carry out bidirectional search, the frontiers may meet when both the forward and the backward search trees have depth = d/2. Suppose we have a good hash function to check for nodes in the fringe. IN this case the time for bidirectional search will be O(($b^{d/2}$). Also note that for at least one of the searches the frontier has to be stored. So the space complexity is also O(($b^{d/2}$).

Comparing Search Strategies

	Breadth first	Depth first	Iterative deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^d	$b^{d/2}$
Space	b^d	bm	bd	$b^{d/2}$
Optimum?	Yes	No	Yes	Yes
Complete?	Yes	No	Yes	Yes

Search Graphs

If the search space is not a tree, but a graph, the search tree may contain different nodes corresponding to the same state. It is easy to consider a pathological example to see that the search space size may be exponential in the total number of states.

In many cases we can modify the search algorithm to avoid repeated state expansions. The way to avoid generating the same state again when not required, the search algorithm can be modified to check a node when it is being generated. If another node corresponding to the state is already in OPEN, the new node should be discarded. But what if the state was in OPEN earlier but has been removed and expanded? To keep track of this phenomenon, we use another list called CLOSED, which records all the expanded nodes. The newly generated node is checked with the nodes in CLOSED too, and it is put in OPEN if the corresponding state cannot be found in CLOSED. This algorithm is outlined below:

```
Graph search algorithm
Let fringe be a list containing the initial state
Let closed be initially empty
Loop
        if fringe is empty return failure
        Node ← remove-first (fringe)
        if Node is a goal
            then return the path from initial state to Node
        else put Node in closed
              generate all successors of Node S
              for all nodes m in S
                  if m is not in fringe or closed
                      merge m into fringe
End Loop
```

But this algorithm is quite expensive. Apart from the fact that the CLOSED list has to be maintained, the algorithm is required to check every generated node to see if it is already there in OPEN or CLOSED. Unless there is a very efficient way to index nodes, this will require additional overhead for every node.