- **State-space search** is the process of searching through a state space for a solution by making explicit a sufficient portion of an implicit state-space graph to include a goal node.
  - Hence, initially V={S}, where S is the start node;
  - when S is expanded, its successors are generated and those nodes are added to V and the associated arcs are added to E.
  - This process continues until a goal node is generated (included in V) and identified (by goal test)
- During search, a node can be in one of the three categories:
  - Not generated yet (has not been made explicit yet)
  - **OPEN**: generated but not expanded
  - **CLOSED**: expanded
  - Search strategies differ mainly on how to select an OPEN node for expansion at each step of search

A General State-Space Search Algorithm

```
open := {S}; closed :={};
repeat
    n := select(open);              /* select one node from open for expansion */
            if n is a goal
                  then exit with success;  /* delayed goal testing */
            expand(n)
                          /* generate all children of n
                              put these newly generated nodes in open (check duplicates)
                              put n in closed (check duplicates) */
until open = {};
exit with failure
```

Evaluating Search Strategies

- **Completeness**
  - Guarantees finding a solution whenever one exists
- **Time Complexity**
  - How long (worst or average case) does it take to find a solution? Usually measured in terms of the **number of nodes expanded**
- **Space Complexity**
  - How much space is used by the algorithm? Usually measured in terms of the **maximum size that the “OPEN" list** becomes during the search
- **Optimality/Admissibility**
  - If a solution is found, is it guaranteed to be an optimal one? For example, is it the one with minimum cost?

Algorithm : Breadth-First Search

1. Create a variable called *NODE-LIST* and set it to the initial state.
2. Until a goal state is found or *NODE-LIST* is empty:

(a) Remove the first element from *NODE-LIST* and call it *E.* If *NODE-LIST* was empty, quit.

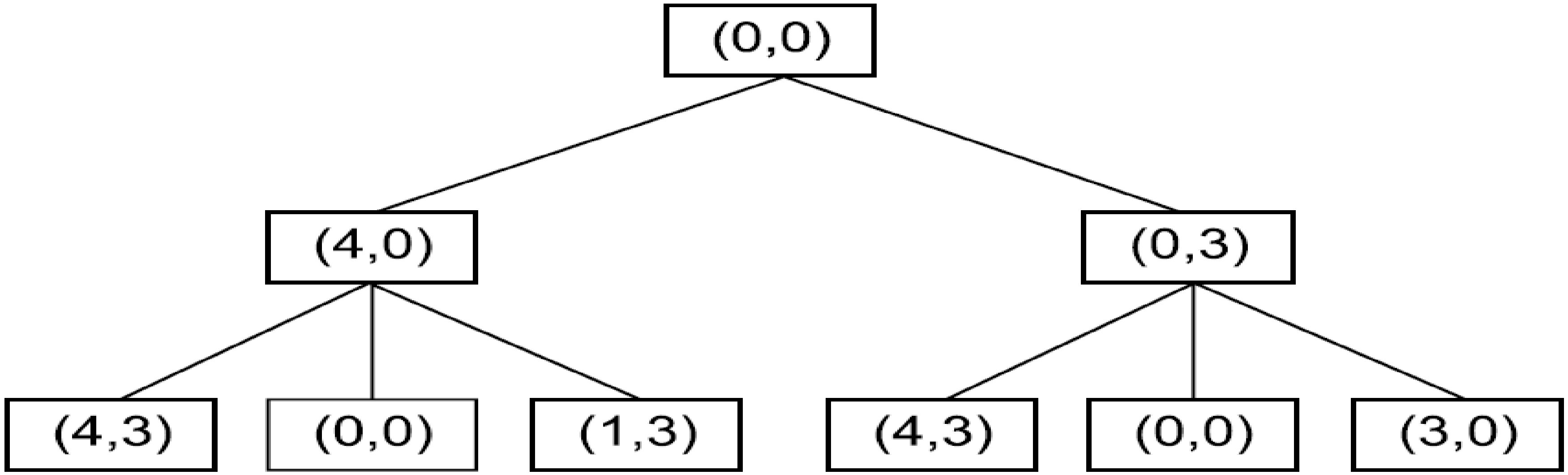
(b) For each way that each rule can match the state described in *E* do:

  (i) Apply the rule to generate a new state,

  (ii) If the new state is a goal state, quit and return this state.

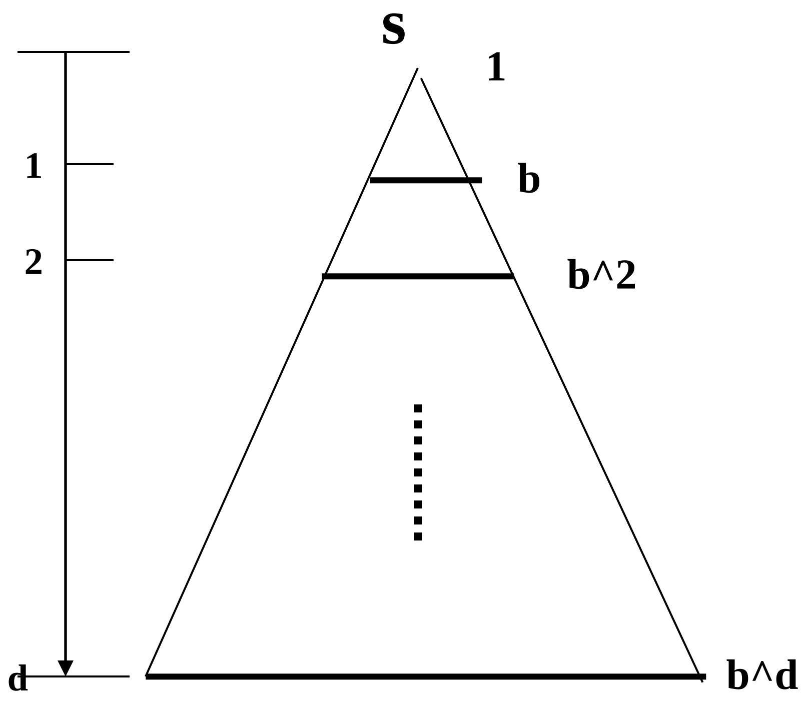
  (iii) Otherwise, add the new state to the end of NODE-LIST.

Two Levels of a Breadth-First Search Tree



Breadth-First

- A complete search tree of depth d where each non-leaf node has b children, has a total of 1 + b + b^2 + ... + b^d = (b^(d+1) - 1)/(b-1) nodes
- Time complexity (# of nodes generated): O(b^d)
- Space complexity (maximum length of OPEN): O(b^d)



– For a complete search tree of depth 12, where every node at depths 0, ..., 11 has 10 children and every node at depth 12 has 0 children, there are 1 + 10 + 100 + 1000 + ... + 10^12 = (10^13 - 1)/9 = O(10^12) nodes in the complete search tree.

- BFS is suitable for problems with shallow solutions

Algorithm : Depth-First Search

1. If the initial state is a goal state, quit and return success.

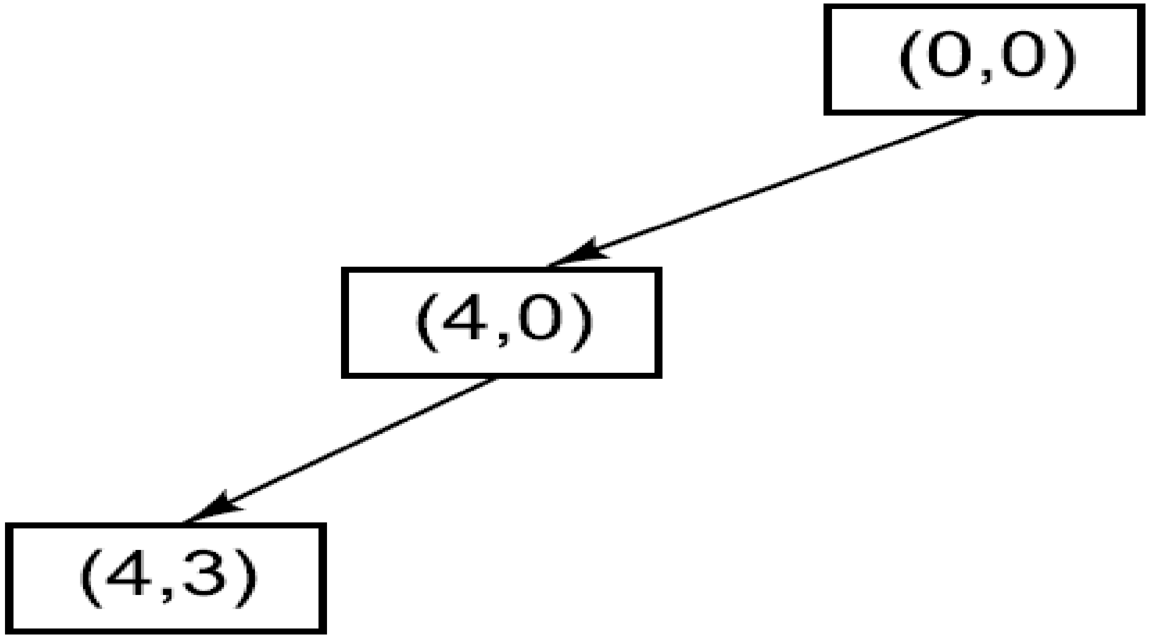
2. Otherwise, do the following until success or failure is signaled:

   (a) Generate a successor, *E,* of the initial state. If there are no more successors, signal failure.

   (b) Call Depth-First Search with *E* as the initial state.

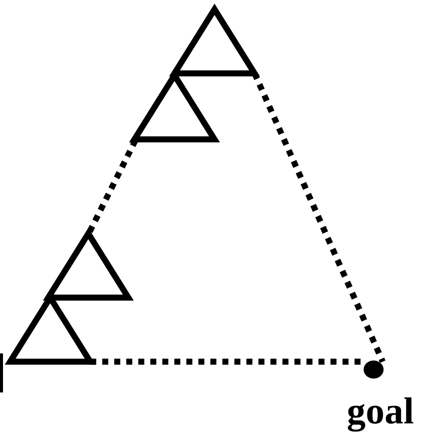
   (c) If success is returned, signal success. Otherwise continue in this loop.

A Depth-First Search Tree



Depth-First (DFS)

- Algorithm outline:
  - Always select from the OPEN the node with the greatest depth for expansion, and put all newly generated nodes into OPEN
  - OPEN is organized as **LIFO** (last-in, first-out) list.
  - Terminate if a node selected for expansion is a goal



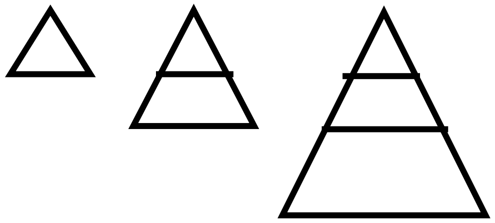
- **May not terminate** without a "depth bound," i.e., cutting off search below a fixed depth D (How to determine the depth bound?)
- **Not complete** (with or without cycle detection, and with or without a cutoff depth)
- **Exponential time**, O(b^d), but only **linear space**, O(bd), required
- Can find **deep solutions quickly** if lucky
- When search hits a deadend, can only back up one level at a time even if the "problem" occurs because of a bad operator choice near the top of the tree. Hence, only does "chronological backtracking"

Depth-First Iterative Deepening (DFID)

- BF and DF both have exponential time complexity O(b^d)
  BF is **complete** but has exponential space complexity
  DF has **linear space complexity** but is incomplete
- Space is often a **harder** resource constraint than time
- Can we have an algorithm that
  - Is complete
  - Has linear space complexity, and
  - Has time complexity of O(b^d)
- DFID by Korf in 1985 (17 years after A*)

  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.

```
until solution found do
    DFS with depth bound d
    d = d+1
```



Depth-First Iterative Deepening (DFID)

- **Complete** (iteratively generate all nodes up to depth d)
- **Optimal/Admissible** if all operators have the same cost. Otherwise, not optimal but does guarantee finding solution of shortest length (like BF).
- **Linear space complexity:** O(bd), (like DF)
- **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)

Depth-First Iterative Deepening

- 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., and node at depth 1 is generated d times.

  Hence

  total(d) = b^d + 2b^(d-1) + ... + db

  <= b^d / (1 - 1/b)^2 = O(b^d).

  – If b=4, then worst case is 1.78 * 4^d, I.e., 78% more nodes searched than exist at depth d (in the worst case).

Bidirectional Search

- **Bidirectional search** is a graph search algorithm that finds a shortest path from an initial vertex to a goal vertex in a directed graph.
- It runs two simultaneous searches: one forward from the initial state, and one backward from the goal, stopping when the two meet in the middle.
- problem complexity in which both searches expand a tree with branching factor b, and the distance from start to goal is d, each of the two searches has complexity $O(b^{d/2})$, and the sum of these two search times is much less than the $O(b^d)$ complexity that would result from a single search from the beginning to the goal.

Thank You !!!