UNIT 2

INT426:GENERATIVE ARTIFICIAL INTELLIGENCE

•**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
```

Some Issues

- Search process constructs a search tree, where
  - –**root** is the initial state S, and
  - –**leaf nodes** are nodes
    - not yet been expanded (i.e., they are in OPEN list) or
    - having no successors (i.e., they're "deadends")
- Some important issue that arises
  - The direction in which conduct the search(forward vs. backward reasoning)
  - How to select applicable rules(matching).
  - How to represent each node of search process(the knowledge representation problem)
  - Search tree vs. search graph

Evaluating Search Strategies

- **Completeness**
  - –Guarantees finding a solution whenever one exists (or guarantees to return a solution if at least any solution exists for any random input.)
- **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**
  - –If a solution is found, is it guaranteed to be an optimal one? For example, is it the one with minimum cost?

Search Techniques

1. Blind(unguided or uninformed Search):
    - The uninformed is the search methodology having no additional information about states beyond that provided in the problem definitions.
    - In this search total search space is looked for solution.
    - It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes.
    - It examines each node of the tree until it achieves the goal node.

It can be divided into five main types:

- Breadth-first search
- Depth-first search
- Uniform cost search
- Iterative deepening depth-first search
- Bidirectional Search

2. Heuristic(or guided or informed Search):

- In this search methodology having additional information about the problem is provided in order to guide the search in a specific direction.
- A heuristic is a way which might not always be guaranteed for best solutions but guaranteed to find a good solution in reasonable time.
- Informed search can solve much complex problem which could not be solved in another way.

Example: a traveling salesman problem.

- Best-first search
- Hill climbing search
- A* search
- Generate and test
- Constraint satisfaction

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.

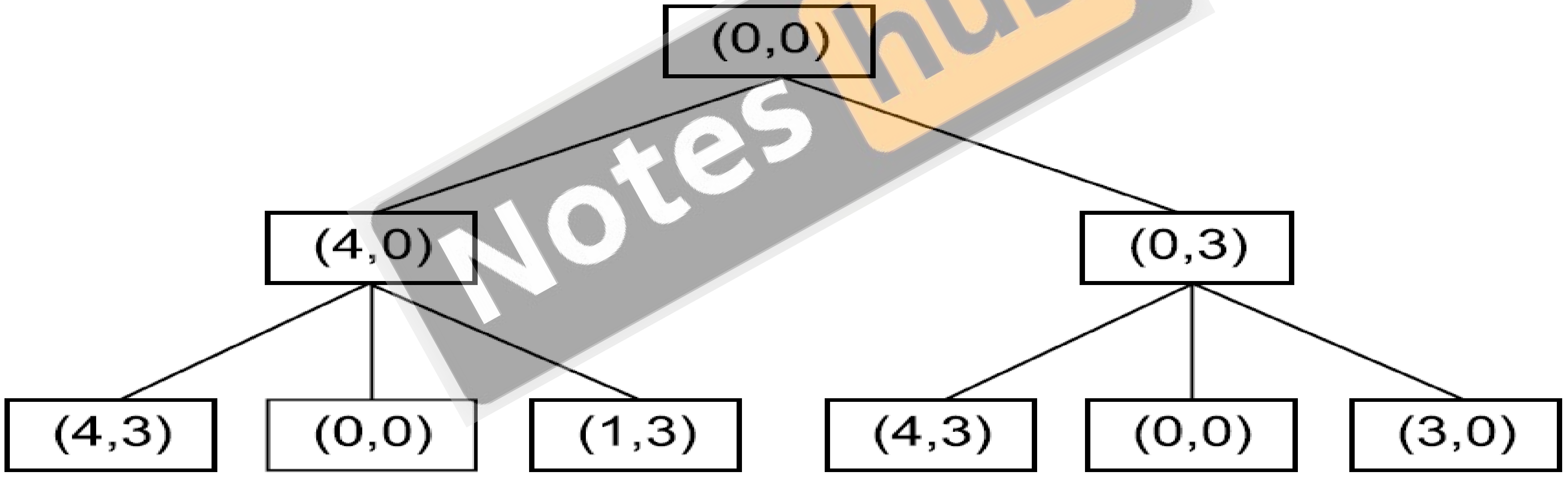
(b) For each way that each rule can match the state described in *E* do:

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

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

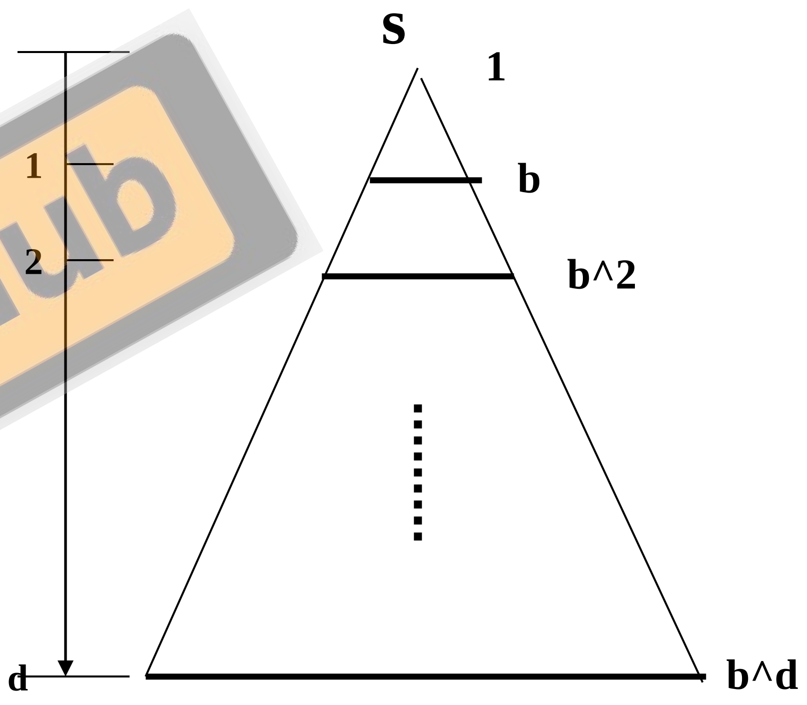
(iii) Otherwise, add the new state to the end of NODE-LIST.

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.

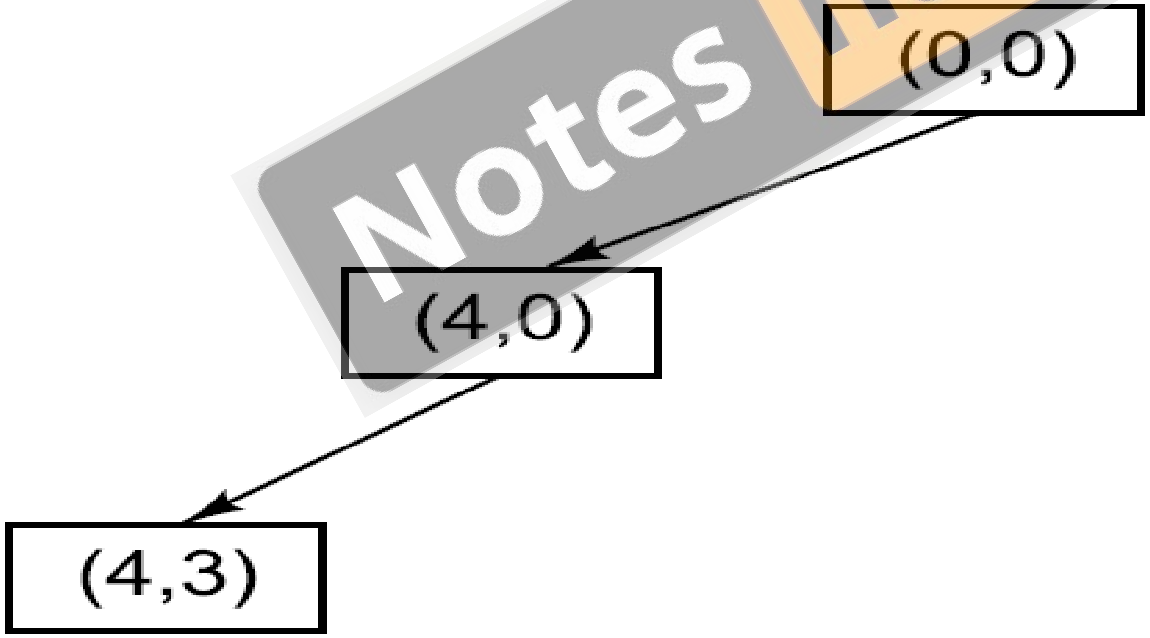
2. Otherwise, do the following until success or failure is signaled:

    (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



- For a state space with branching factor *b* and maximum depth m, depth-first search requires storage of only *bm* nodes
- The time complexity for depth-first search is $O(b^m)$. For problems that have very many solutions, depth-first may actually be faster than breadth-first, because it has a good chance of finding a solution after exploring only a small portion of the whole space.
- Breadth-first search would still have to look at all the paths of length ***d* - 1** before considering any of length *d*.

- Depth-first search is still $O(b^m)$. in the worst case.

- The drawback of depth-first search is that it can get stuck going down the wrong path.
- Many problems have very deep or even infinite search trees, so depth-first search will never be able to recover from an unlucky choice at one of the nodes near the top of the tree. That means depth-first search is neither complete nor optimal.
- *depth-first search should be avoided for search trees with large or infinite maximum depths.*

Thank You !!!

Constraint Satisfaction Problems

Constraint Satisfaction Problems

A CSP consists of:

- *Finite set of variables X1, X2, ..., Xn*
- *Nonempty **domain of possible values***

➢ **for each variable *D1, D2, … Dn where Di = {v1, …, vk}***

- ***Finite set of constraints C1, C2, …, Cm***

➢ **Each *constraint Ci* limits the values that variables can take, e.g., *X1 ≠ X2***

➢ **A *state* is defined as an *assignment* of values to some or all variables.**

➢ **A *consistent assignment* does not violate the constraints.**

➢ **Example: Sudoku**

Types of Constraints:

- Unary constraints:
  - Which is applicable on one variable.
- Binary constraints:
  - Which is related to two variables.
- Higher order constraints:
  - Which involves three or more variables.

Heuristic search

INT 404

Introduction:

- Heuristic search is also called guided search because in this, the search is guided in a specific direction.
- In heuristic search, besides normal production rules, additional information or knowledge about the problems is given in the form of clue.
- The additional information or clue is called ***heuristics***.
- This additional information or clue restricts the expansion of only promising nodes in search tree and guides the search in a specific direction towards the goal. Thus, it reduces the search space and the solution of the problem is obtained efficiently.
- The additional information is given for search in terms of ***heuristic function***.

Heuristic Function:

- A Heuristic function maps the desirability of a problem state from descriptive to quantitative numbers.
- The heuristic function attach a numerical value with each state.
- The heuristic function considers important aspects of problem state, evaluates those aspects and gives weights to individual aspects in such a way that heuristic function at a given node in a search process gives an estimate of whether that node is on the desired path to a solution.
- *The purpose of heuristic function is to guide the search process in the most profitable search direction by suggesting which path to follow first, when more than one search paths are available.*

Designing Heuristic Function:

- The designing of heuristic function is a critical task. *In the problems where start and goal states are known*, the heuristic function somehow relates current search state with the desired goal state.
- The designing of heuristic function is done in such a way that it provides best mathematical value on a move.

3	2	1
8	5	6
0	7	4

1	2	3
8	0	4
7	6	5

- The branching factor of 8-puzzle problem is 3. The middle tile is 4, for the corner tiles it is 2 , otherwise it is 3.
  - **h= 2+0+2+1+2+2+1+0 = 10**

- **There exist multiple heuristic function for the same problem. in such situations , function which brings the solution in minimum number of steps is considered best.**

- Generate-and-test
- Best-first search
- Hill climbing
- Problem reduction
- Constraint satisfaction
- Means-ends analysis

Algorithm : Generate-and-Test

Do the following until satisfactory solution is obtained or no more solutions can be generated.

1. Generate a possible solution.(for some problems, this means generating a particular point in the problem space and for others, it means generating a path from a start state).
2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.
3. If a solution has been found, quit. Otherwise, return to step 1.

GENERATE-AND-TEST

- Acceptable for simple problems.
  - Eg : 1. finding key of a 3 digit lock.
    2. 8-puzzle problem
- Inefficient for problems with large space.
- Use DFS as all possible solution generated, before they can be tested.

Generate-and-Test

There can be two methods to complete generate-and-test process.

1. Generate a random solution and test it. If found wrong create another solution and test it again. This method might gives you a solution on the first trial. However there is no guarantee that the solution will ever be found.

2. Systematic approach of generating a solution is applied and we are sure of ultimately getting the right solution, if exists.
   But the difficulty with this method could be that it might take a lot of time if possible space is large.

GENERATE-AND-TEST

- Generate solution randomly: British museum algorithm-- wandering randomly.
- Exhaustive generate-and-test. : consider each case in depth
- Heuristic generate-and-test: not consider paths that seem to lead to a solution. unlikely
- Plan generate-test:

– Create a list of candidates.

– Apply generate-and-test to that list on the basis of constraint-satisfaction.

Ex – DENDRAL, which infers the structure of organic compounds using mass spectrogram and nuclear magnetic resonance (NMR) data.

- It uses plan-generate-test strategy , in which planning part uses constraint satisfaction techniques to create lists of recommended substructures and generate-test-method then uses these list to explore only limited set of structures.

BEST-FIRST SEARCH

- Combines the advantages of both DFS and BFS into a single method.

- Depth-first search: not all competing branches having to be expanded.

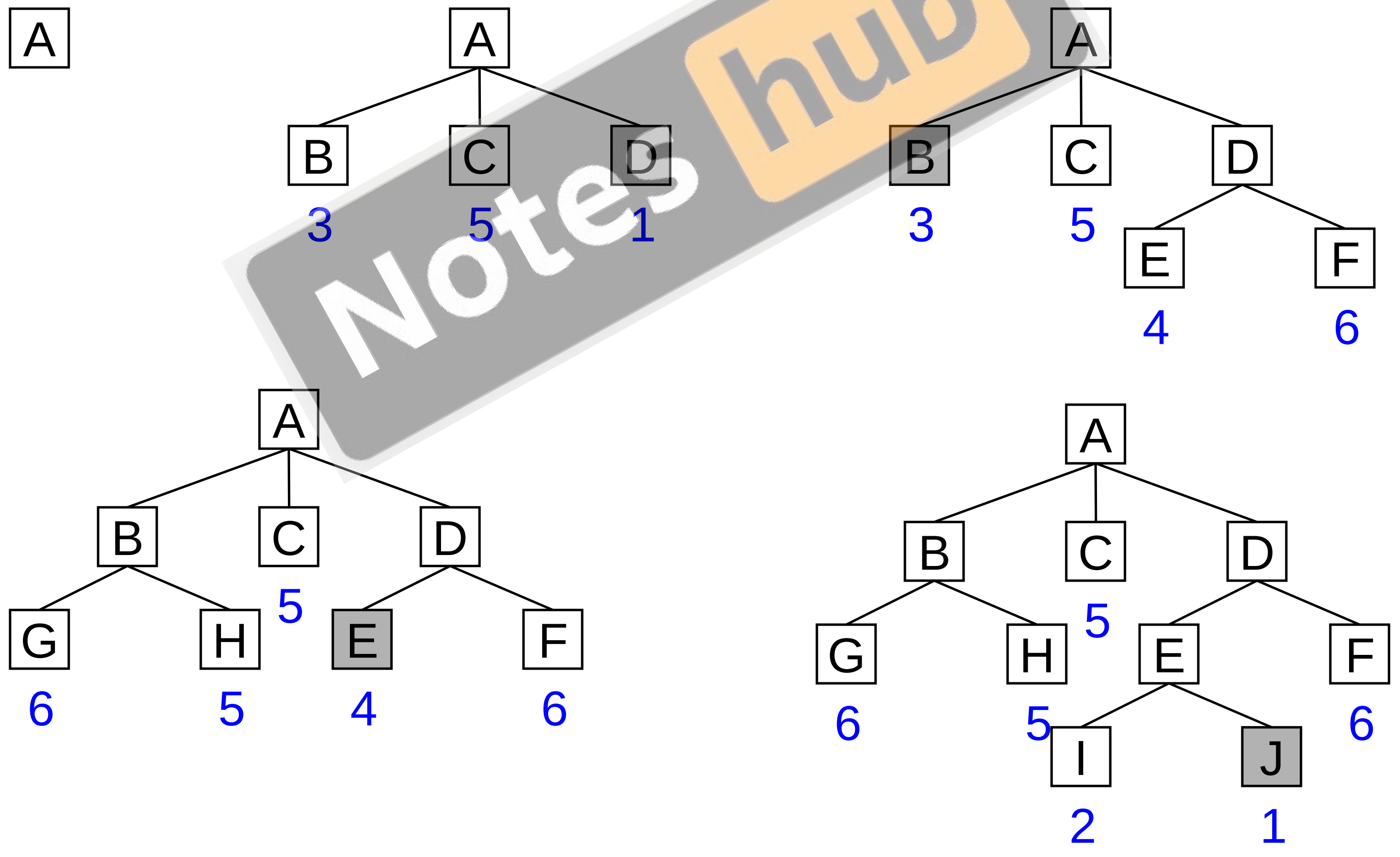
- Breadth-first search: not getting trapped on dead-end paths.

$\Rightarrow$ Combining the two is to follow a single path at a time, but switch paths whenever some competing path look more <u>promising</u> than the current one.

BEST-FIRST SEARCH

- At each step of the BFS search process, we select the most promising of the nodes we have generated so far.
- This is done by applying an appropriate heuristic function to each of them.
- We then expand the chosen node by using the rules to generate its successors
- This is called OR-graph, since each of its branches represents an alternative problem solving path

BEST-FIRST SEARCH



BEST-FIRST SEARCH

- OPEN: nodes that have been generated, but have not examined.

  This is organized as a priority queue.

- CLOSED: nodes that have already been examined.

  Whenever a new node is generated, check whether it has been generated before.

Algorithm : Best-First Search

1. Start with *OPEN* containing just the initial state.

2. Until a goal is found or there are no nodes left on *OPEN* do:

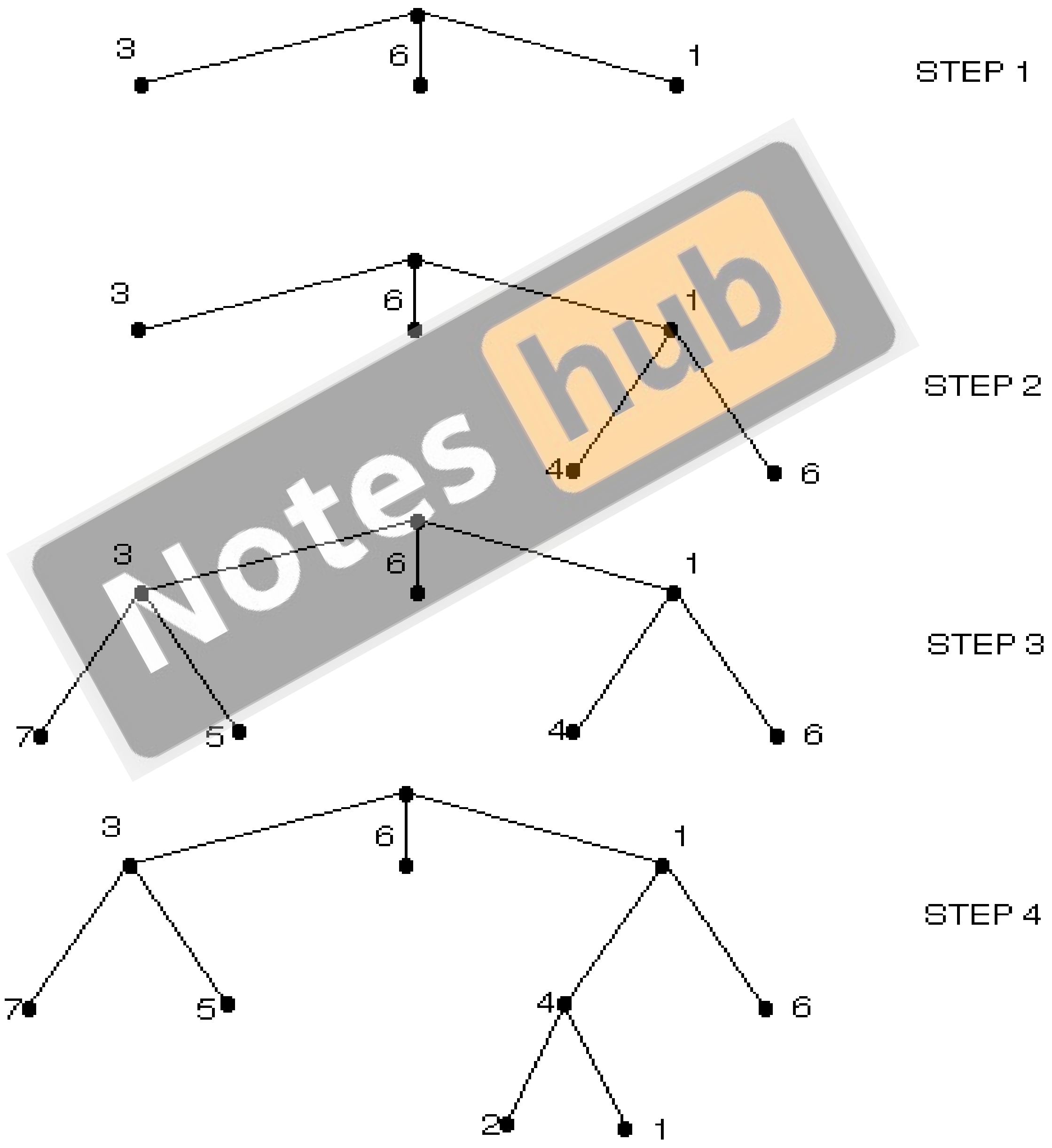
(a) Pick them best node on *OPEN.*

(b) Generate its successors.

(c) For each successor do:

(i) If it has not been generated before, evaluate it, add it to *OPEN,* and record its parent.

(ii) If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already, have.



All figures indicate "cost" of move

Hill Climbing

Hill Climbing is a heuristic search used for mathematical optimization problems in the field of Artificial Intelligence.

Given a large set of inputs and a good heuristic function, the algorithm tries to find the best possible solution to the problem in the most reasonable time period.

This solution may not be the absolute best (global optimal maximum) but it is sufficiently good considering the time allotted. Therefore, it falls in the category of Local search algorithm.

Example of this is the ***Travelling Salesman Problem,*** where we need to minimize the distance travelled by the salesman.

Features of Hill Climbing

It carries out a Heuristic search.

A **heuristic function** is one that ranks all the potential alternatives in a search algorithm based on the information available. It helps the algorithm to select the best route to its solution.

It is a variant of the generate-and-test algorithm.

Hill climbing takes the feedback from the test procedure and the generator uses it in deciding the next move in the search space. Hence, we call it as a *variant of the generate-and-test algorithm*.

It uses the Greedy approach.

At any point in state space, the search moves in that direction only which optimises the cost of function with the hope of finding the most optimum solution at the end.

Simple Hill-Climbing

Simple hill climbing is the simplest way to implement a hill-climbing algorithm.
It only evaluates the neighbour node state at a time and selects the first one which optimizes current cost and set it as a current state. Less time consuming, Less optimal solution, The solution is not guaranteed

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.

2. Loop until a solution is found or until there are no new operators left to be applied in the current state:

(a) Select an operator that has not yet been applied to the current state and apply it to produce a new state.

(b) Evaluate the new state.

(i) If it is a goal state, then return it and quit.

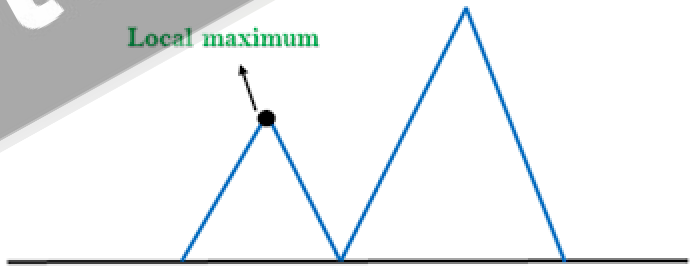
(ii) If it is not a goal state but it is <u>better</u> than the current state, then make it the current state.

(iii) If it is not better than the current state, then continue in the loop.

Problems in Hill climbing

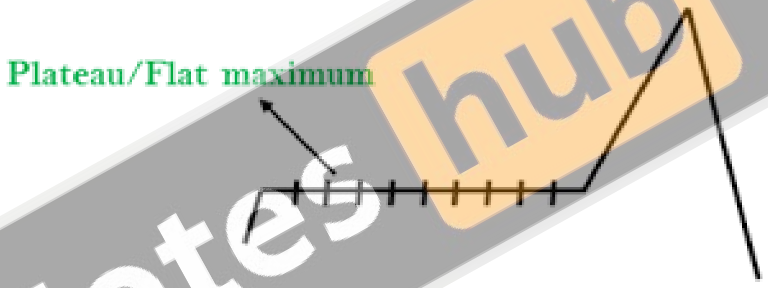
Hill climbing cannot reach the best possible state if it enters any of the following regions :

1. **Local maximum:** At a local maximum all neighbouring states have values which are worse than the current state. Since hill-climbing uses a greedy approach, it will not move to the worse state and terminate itself. The process will end even though a better solution may exist.



To overcome the local maximum problem: Utilise the ***backtracking technique***. Maintain a list of visited states. If the search reaches an undesirable state, it can backtrack to the previous configuration and explore a new path.

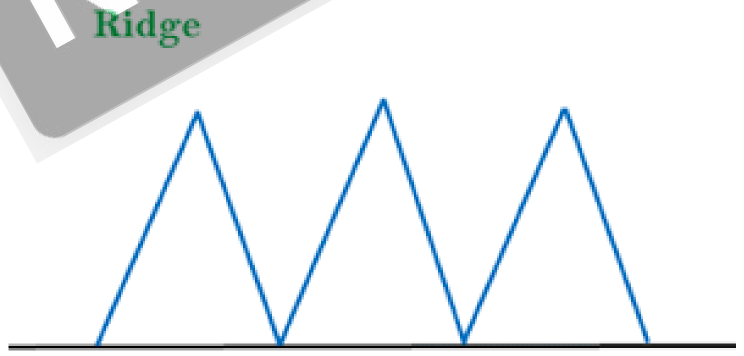
2. **Plateau:** On the plateau, all neighbours have the same value. Hence, it is not possible to select the best direction.



To overcome plateaus: Make a big jump. Randomly select a state far away from the current state. Chances are that we will land at a non-plateau region

3. **Ridge:** is a special kind of local maximum. It is a area of search space that is higher than surroundings areas and that itself has a slope.
But the orientation of the high region , compared to the set of available moves and the directions in which they move, make it impossible to traverse a ridge by single moves.

To overcome Ridge: You could use two or more rules before testing. It implies moving in several directions at once.



Steepest-Ascent hill climbing

The steepest-Ascent algorithm is a variation of the simple hill-climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state.
This algorithm consumes more time as it searches for multiple neighbors.

•**Step 1:** Evaluate the initial state, if it is goal state then return success and stop, else make the current state as your initial state.
•**Step 2:** Loop until a solution is found or the current state does not change.

1. Let **S** be a state such that any successor of the current state will be better than it.
2. For each operator that applies to the current state;
    1. Apply the new operator and generate a new state.
    2. Evaluate the new state.
    3. If it is goal state, then return it and quit, else compare it to the **S**.
    4. If it is better than **S**, then set new state as **S**.
    5. If the **S** is better than the current state, then set the current state to **S**.

•**Step 5:** Exit.

A* algorithm

- **The A * algorithm is a specialization of best-first search.**

- **A * algorithm** is a searching algorithm that searches for the shortest path between the *initial and the final state*. It is used in various applications, such as *maps*.
- In *maps* the A* algorithm is used to calculate the shortest distance between the source (initial state) and the destination (final state).

- It provides general guidelines about how to estimate goal distances for general search graphs.
- At each node along the path to the goal node, **A * algorithm** generate all successor nodes and computes an estimate of distance(cost) from the start node to a goal node through each of the successors.
- It then chooses the successor with the shortest estimated distance for expansion.
- It calculates the heuristic function based on distance of current node from start state and distance of current node to goal state.

f(n) = g(n) + h(n)

Where,

- ✓ *f(n)* = it is the sum of g and h. So, ***f(n) = g(n) + h(n)***
- ✓ *g(n)* = the cost of moving from the initial cell to the current cell. Basically, it is the sum of all the cells that have been visited since leaving the first cell.
- ✓ *h(n)* = also known as the *heuristic value,* it is the **estimated** cost of moving from the current cell to the final cell. The actual cost cannot be calculated until the final cell is reached. Hence, h is the estimated cost. We **must** make sure that there is **never** an over estimation of the cost.

A* is **optimal** as well as a **complete** algorithm. **Optimal** meaning that *A* is sure to find the least cost from the source to the destination* and **Complete** meaning that *it is going to find all the paths that are available to us from the source to the destination*.

A* ALGORITHM

1. Start with OPEN containing only initial node. Set that node's g value to 0, its h' value to whatever it is, and its f' value to h'+0 or h'. Set CLOSED to empty list.

2. Until a goal node is found, repeat the following procedure:
   1. If there are no nodes on OPEN, report failure.
   2. Otherwise pick the node on OPEN with the lowest f' value. Call it BESTNODE. Remove it from OPEN. Place it in CLOSED.
      1. See if the BESTNODE is a goal state. If so exit and report a solution.
      2. Otherwise, generate the successors of BESTNODE but do not set the BESTNODE to point to them yet.

- For each of the SUCCESSOR, do the following:

a) Set SUCCESSOR to point back to BESTNODE. These backwards links will make it possible to recover the path once a solution is found.

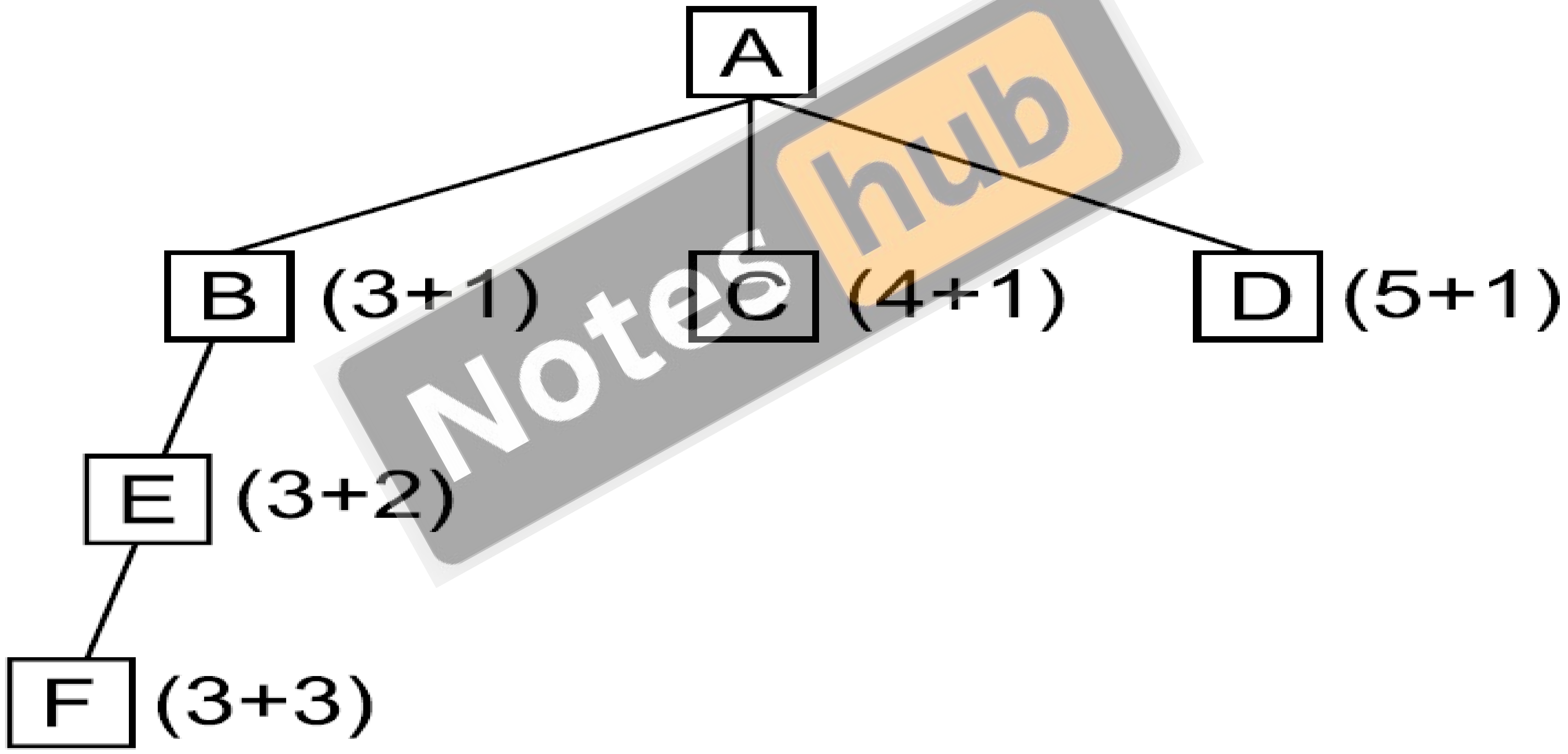
b) Compute g(SUCCESSOR) = g(BESTNODE) + the cost of getting from BESTNODE to SUCCESSOR

c) See if SUCCESSOR is the same as any node on OPEN. If so call the node OLD.

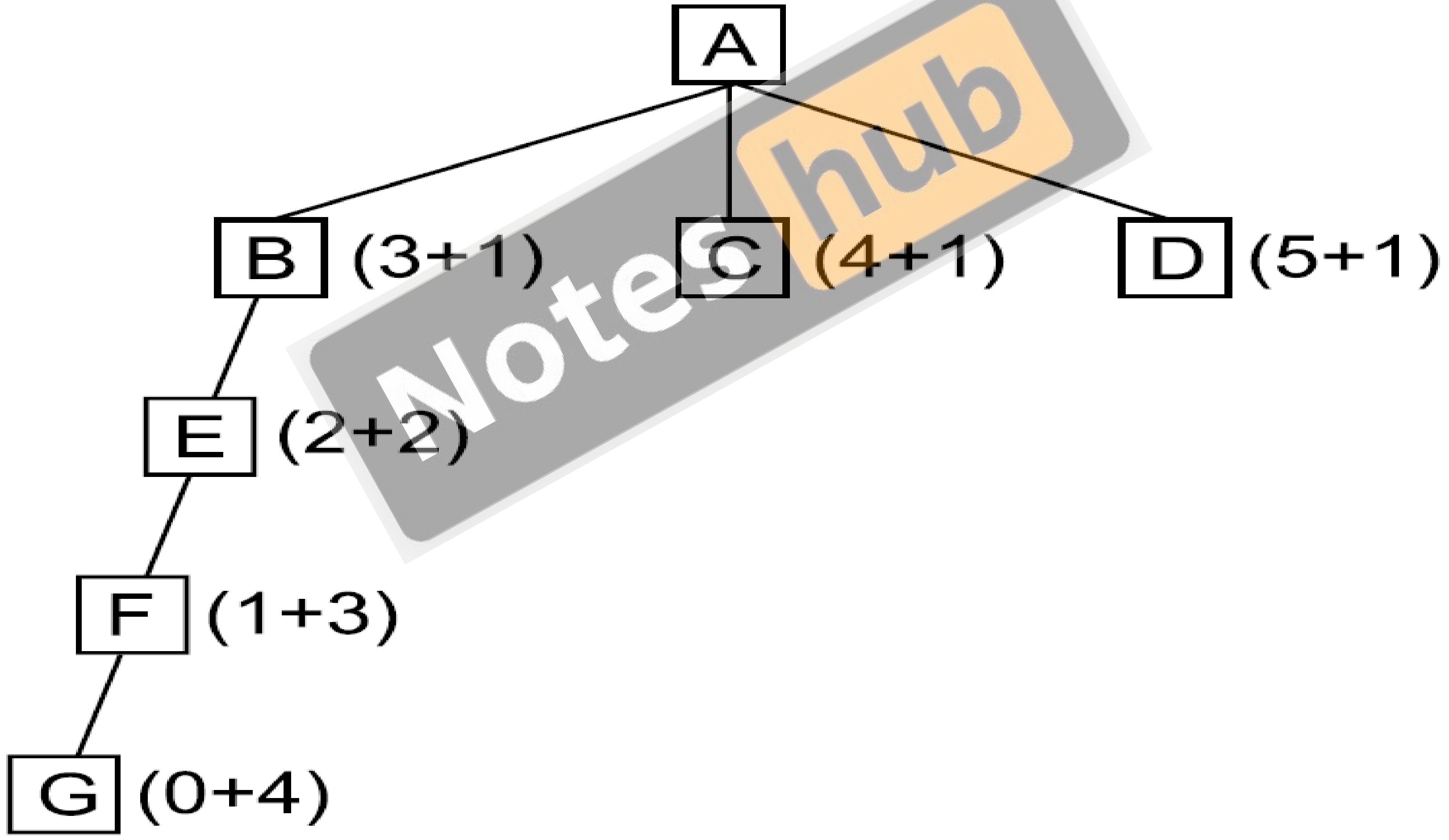
d) If SUCCESSOR was not on OPEN, see if it is on CLOSED. If so, call the node on CLOSED OLD and add OLD to the list of BESTNODE's successors.

e) If SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN and add it to the list of BESTNODE's successors. Compute f'(SUCCESSOR) = g(SUCCESSOR) + h'(SUCCESSOR)

h′ Underestimates h



h´ Overestimates h


A
B (3+1)
C (4+1)
D (5+1)
E (2+2)
F (1+3)
G (0+4)