

# UNIT 2

## SEARCH STRATEGIES

# UNINFORMED SEARCH

- BFS

- DFS

- **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

- **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

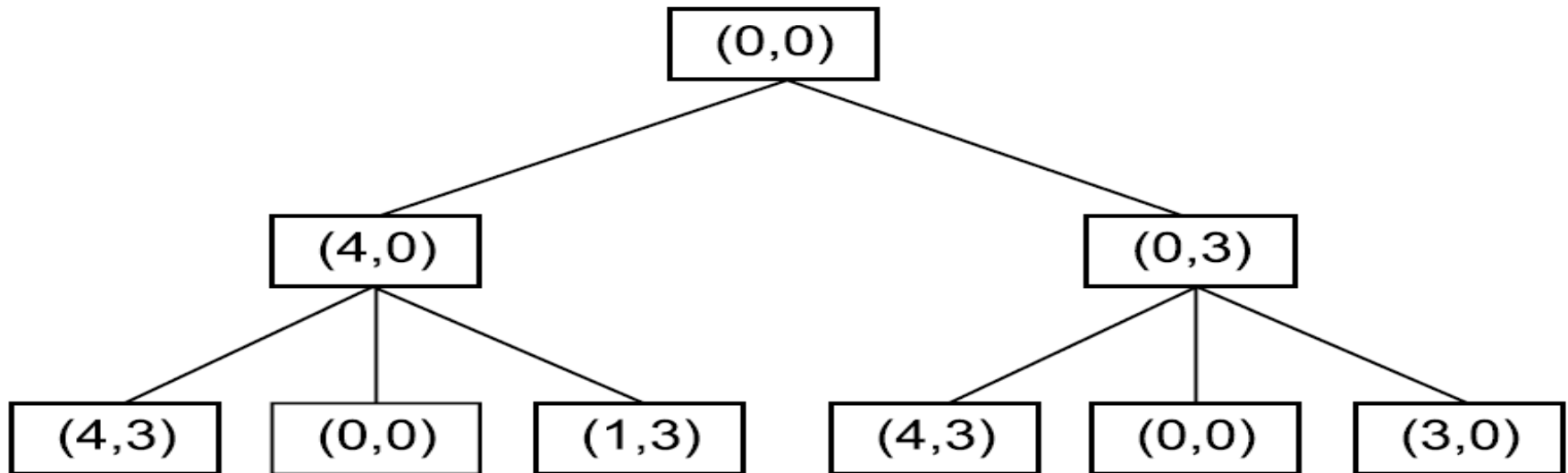
- **Optimality/Admissibility**

- 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.**

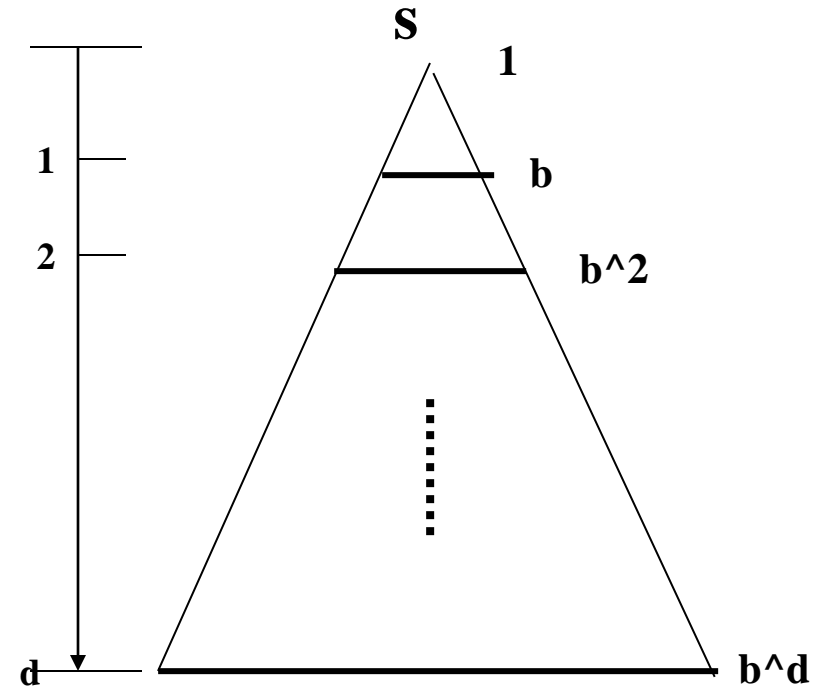
# Two Levels of a Breadth-First Search Tree





# Breadth-First

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

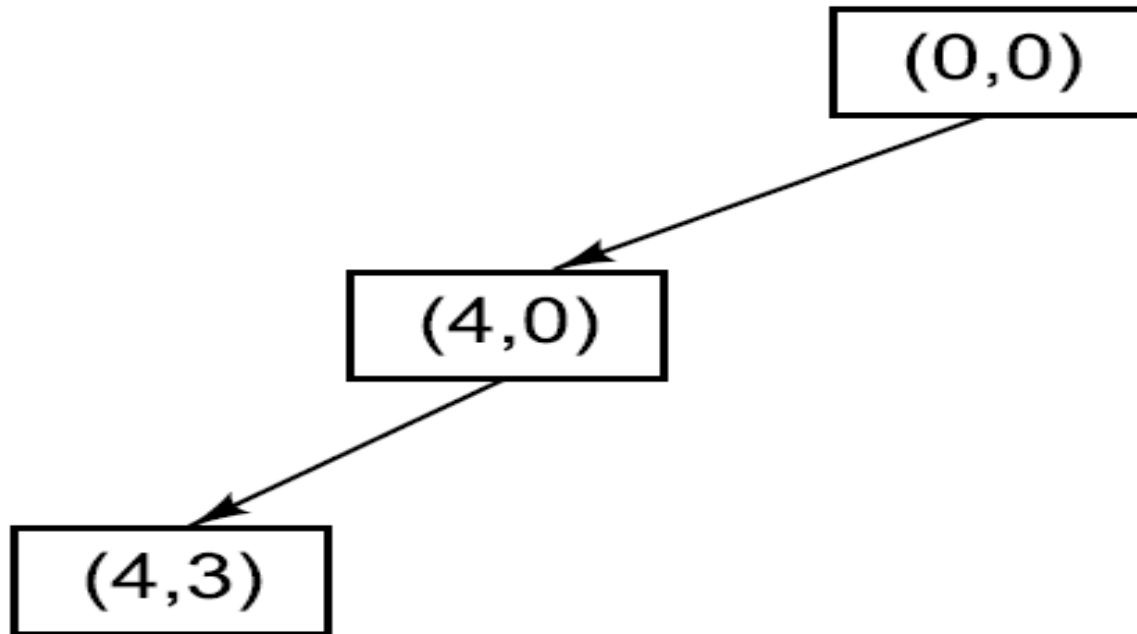


- For a complete search tree of depth 12, where every node at depths 0, ..., 11 has 10 children and every node at depth 12 has 0 children, there are  $1 + 10 + 100 + 1000 + \dots + 10^{12} = (10^{13} - 1)/9 = O(10^{12})$  nodes in the complete search tree.
  - BFS is suitable for problems with shallow solutions

# Algorithm : Depth-First Search

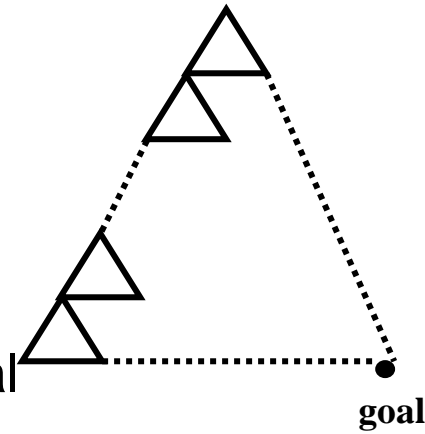
1. If the initial state is a goal state, quit and return success.
2. Otherwise, do the following until success or failure is signaled:
  - (a) **Generate a successor,  $E$ , of the initial state. If there are no more successors, signal failure.**
  - (b) **Call Depth-First Search with  $E$  as the initial 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"

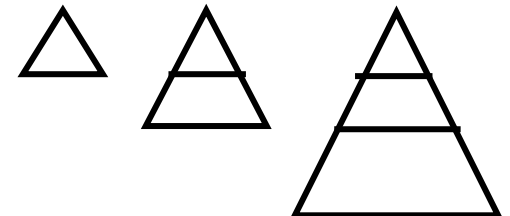


## Depth-First Iterative Deepening (DFID)

- 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

$$\text{total}(d) = b^d + 2b^{(d-1)} + \dots + db$$

$$\leq b^d / (1 - 1/b)^2 = O(b^d).$$

- 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.



# Heuristic search

# Heuristic search

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

# Algorithm : Generate-and-Test

1. Generate a possible solution.
2. Test to see if this is actually a solution by comparing the chose 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

- **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 unlikely to lead to a solution.
  - **Plan** generate-test:
    - Create a list of candidates.
    - Apply generate-and-test to that list on the basis of constraint-satisfaction.
- Ex – DENDERAL, which infers the structure of organic compounds using mass spectrogram and nuclear magnetic resonance (NMR) data.

# HILL CLIMBING

- Generate-and-test + **direction to move** (feedback from test procedure).
- Test function + heuristic function = Hill Climbing
- **Heuristic function (objective function)** to estimate how close a given state is to a goal state.
- Hill climbing is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available.

# SIMPLE HILL CLIMBING

- Evaluation function as a way to inject **task-specific knowledge** into the control process.
- Key difference between Simple Hill climbing and Generate-and-test is the use of evaluation function as a way to inject task specific knowledge into the control process.
- Better : higher value of heuristic function  
Lower value

# Algorithm : Simple Hill-Climbing

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 better 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.



# EX

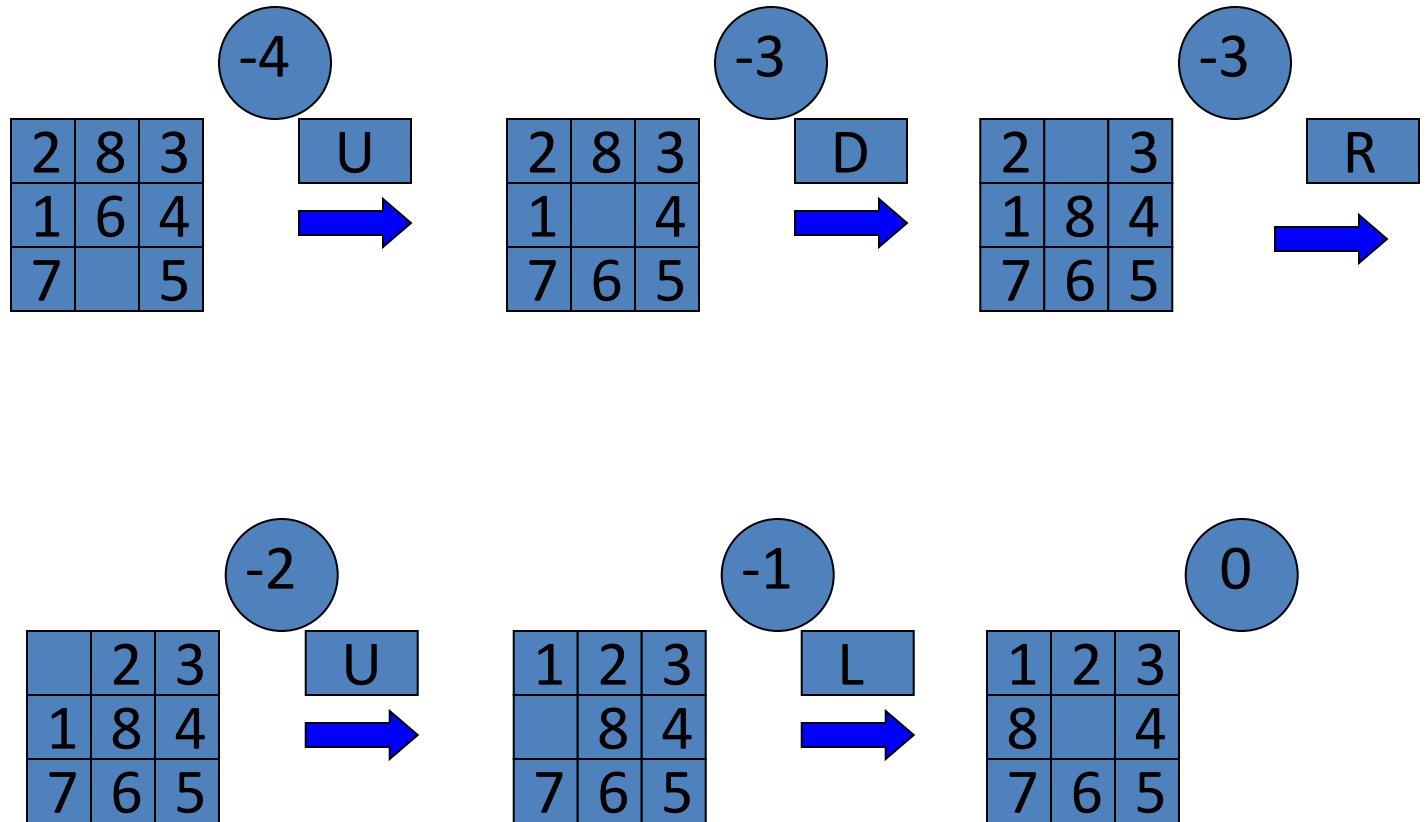
- 1) Use heuristic function as measure of how far off the number of tiles out of place.
- 2) Choose rule giving best increase in function.

2	8	3
1	6	4
7		5



1	2	3
8		4
7	6	5

# Example



# STEEPEST-ASCENT HILL CLIMBING

- Considers **all the moves** from the current state.
- Selects **the best one** as the next state.
- Also known as **Gradient Search**.

# Algorithm : Steepest-Ascent Hill Climbing or gradient search

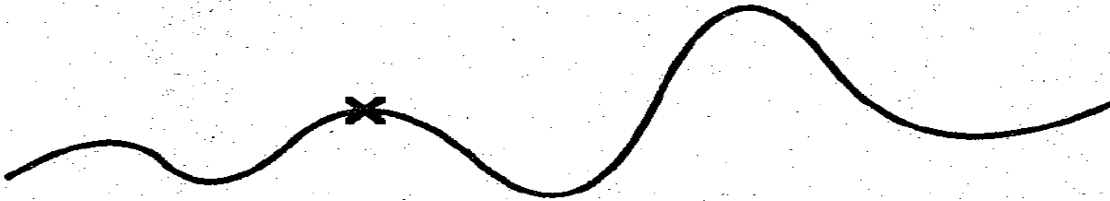
1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.
2. Loop until a solution is found or until a complete iteration produces no change to current state:
  - (a) Let *SUCC* be a state such that any possible successor of the current state will be better than *SUCC*.
  - (b) For each operator that applies to the current state do:
    - (i) Apply the operator and generate a new state.
    - (ii) Evaluate the new state. If it is a goal state, then return it and quit. If not, compare it to *SUCC*. If it is better, then set *SUCC* to this state. If it is not better, leave *SUCC* alone.
  - (c) If the *SUCC* is better than current state, then set current state to *SUCC*.

# HILL CLIMBING: DISADVANTAGES

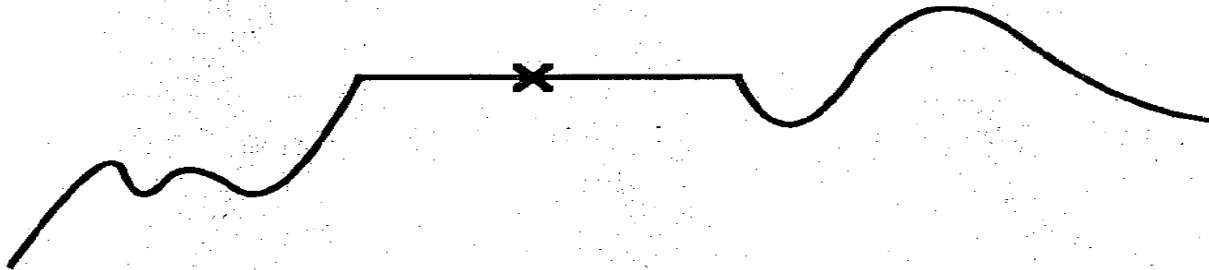
- Fail to find a solution
- Either Algorithm may terminate not by finding a goal state but by getting to a state from which no better state can be generated.
- This happen if program reached
  - Local maximum: A state that is better than all of its neighbours, but not better than some other states far away.
  - Plateau: A flat area of the search space in which all neighbouring states have the same value.
  - Ridge: Special kind of local maximum.  
The orientation of the high region, compared to the set of available moves, makes it impossible to climb up.

# Hill-Climbing Dangers

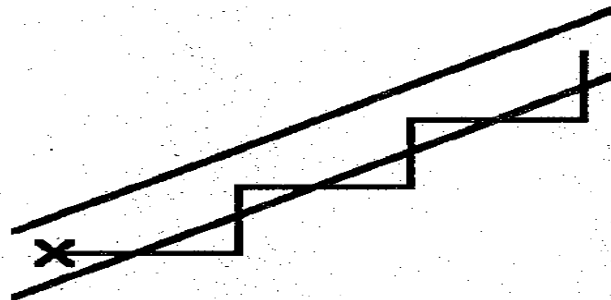
## ◎ Local maximum



## ◎ Plateau



## ◎ Ridge



# HILL CLIMBING: DISADVANTAGES

## Ways Out

- **Backtrack** to some earlier node and try going in a different direction. (good way in dealing with local maxima)
- Make a **big jump** to try to get in a new section. (good way in dealing with plateaus)
- Moving in **several directions** at once. (good strategy for dealing with ridges)

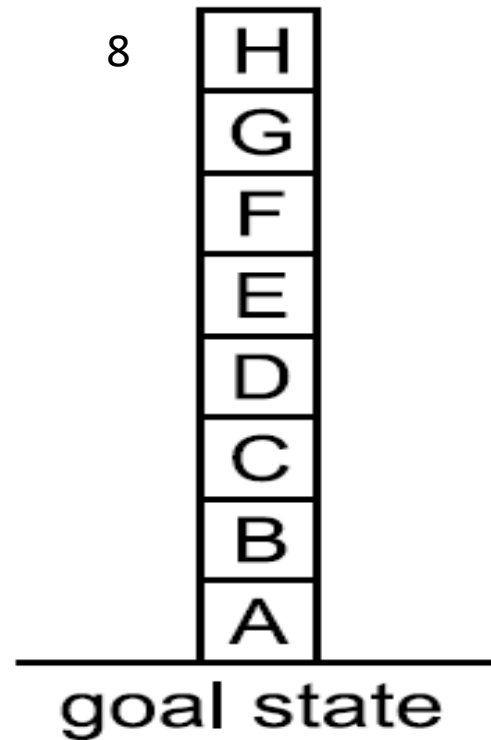
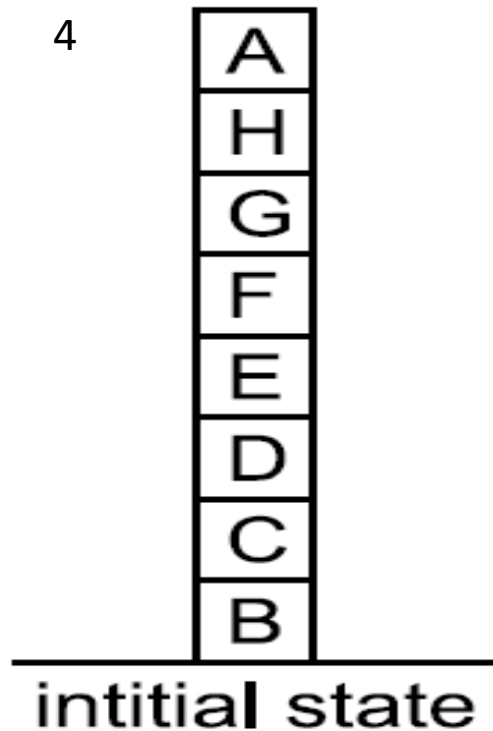
# HILL CLIMBING: DISADVANTAGES

- Hill climbing is a **local method**:  
Decides what to do next by looking only at the “immediate” consequences of its choices rather than by exhaustively exploring all the consequences.
- **Global information** might be encoded in heuristic functions.

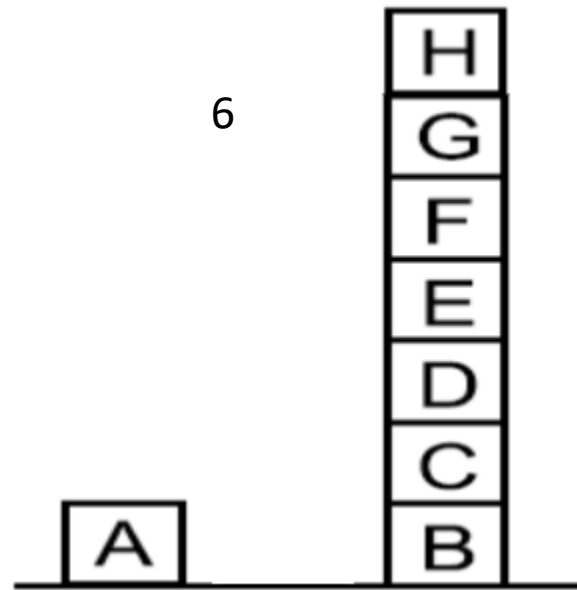


# A Hill-Climbing Problem

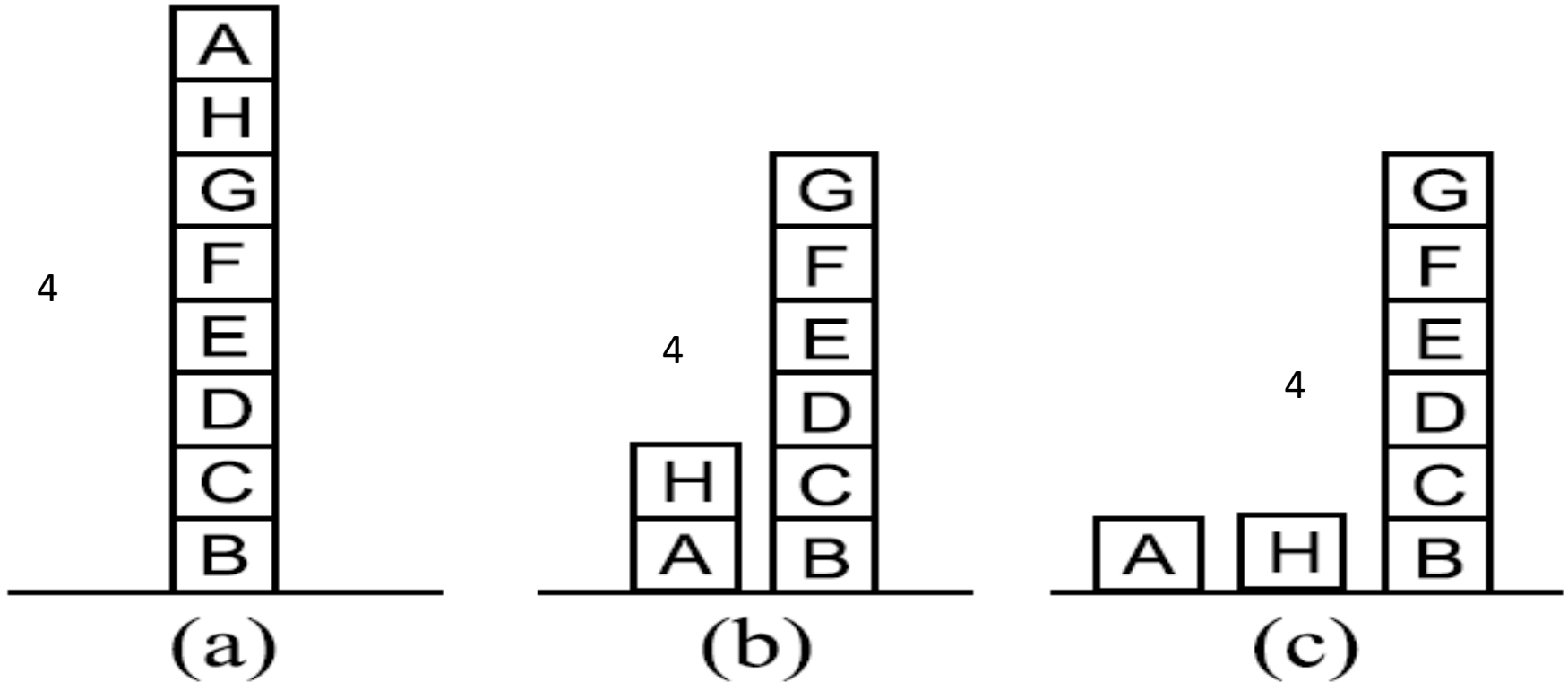
Local: Add one point for every block that is resting on thing it is supposed to be resting on.  
Subtract one point from every block that is sitting on wrong thing.



# One Possible Moves



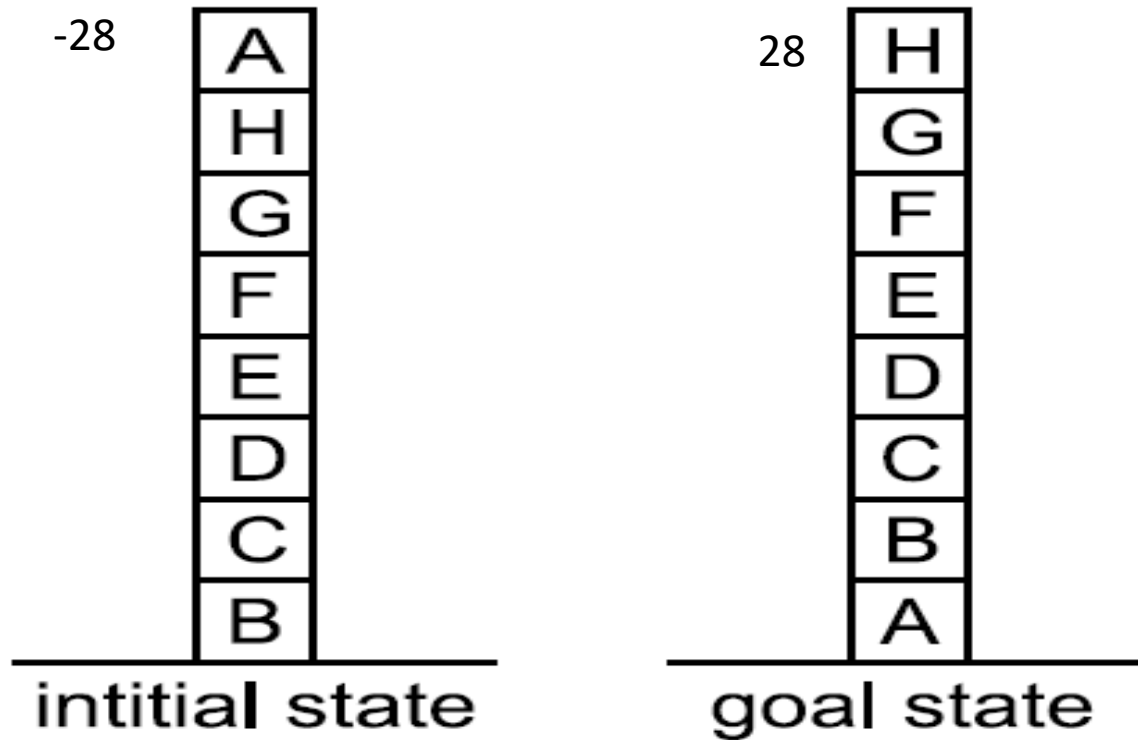
# Three Possible Moves



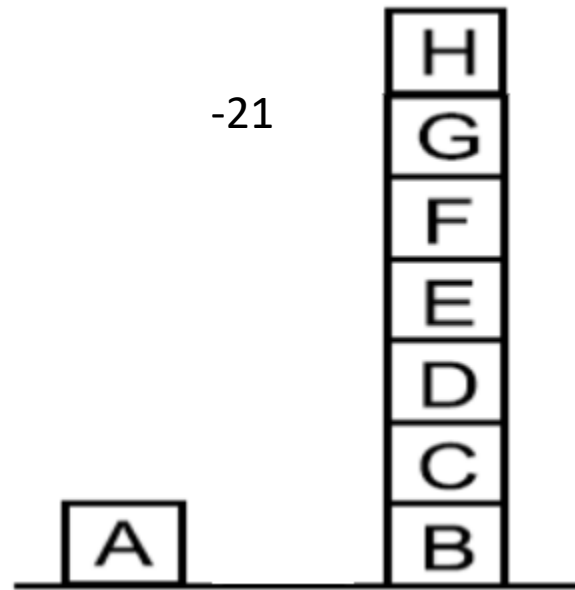
Hill Climbing will Halt because all states have lower score than the Current state.

# A Hill-Climbing Problem

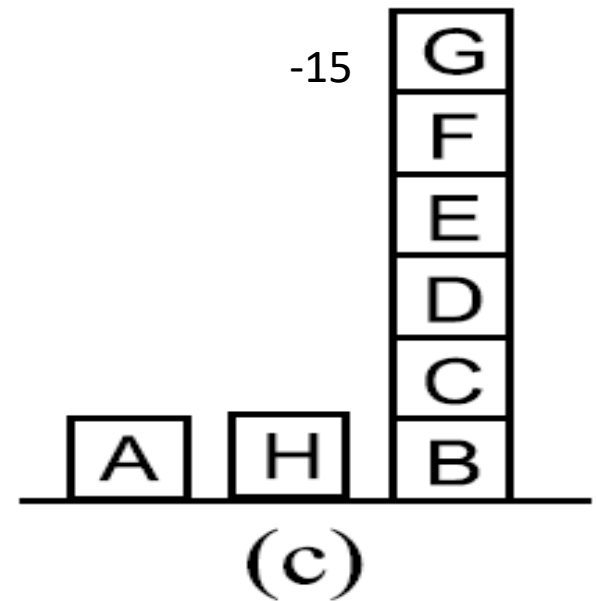
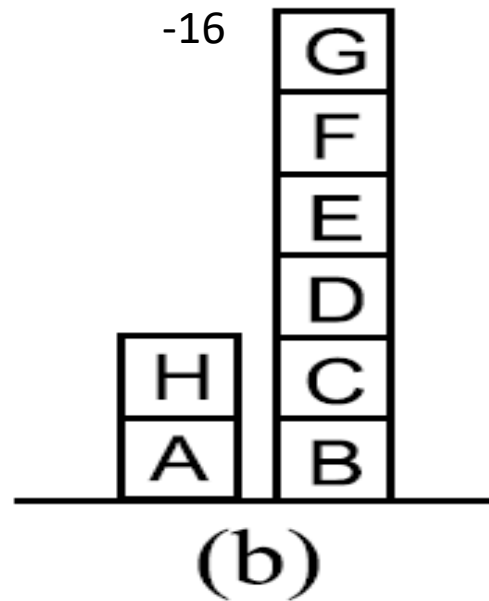
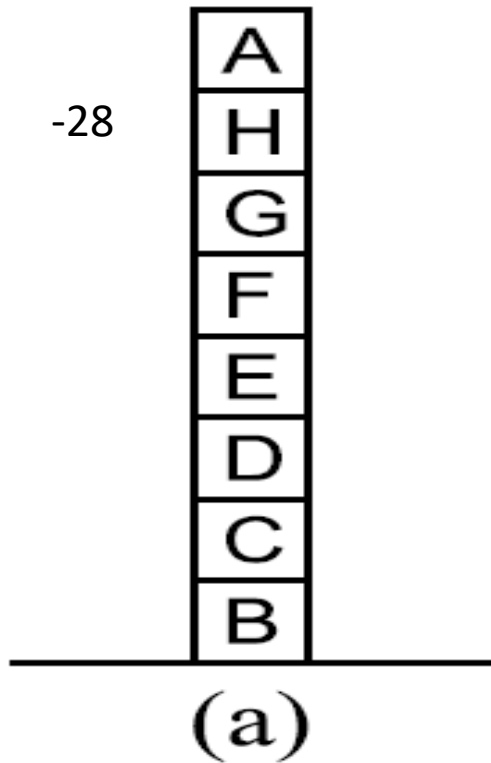
Global: Add one point for every block in correct support structure, subtract one point for every block in existing support structure.



# One Possible Moves



# Three Possible Moves



**Thank You!!!**

# BEST FIRST SEARCH

-OR Graph

-A\* Search

INT404



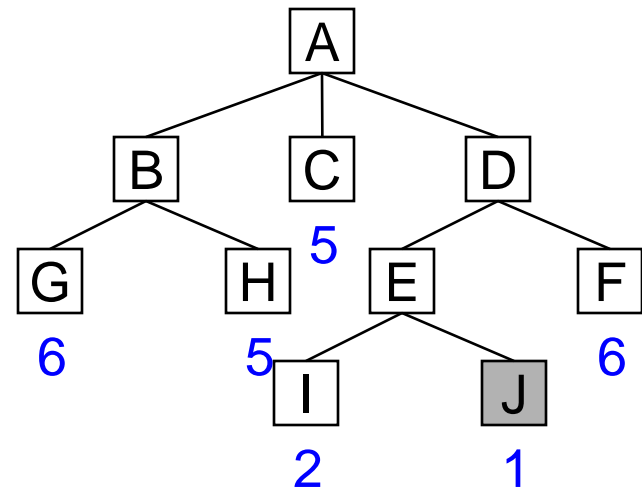
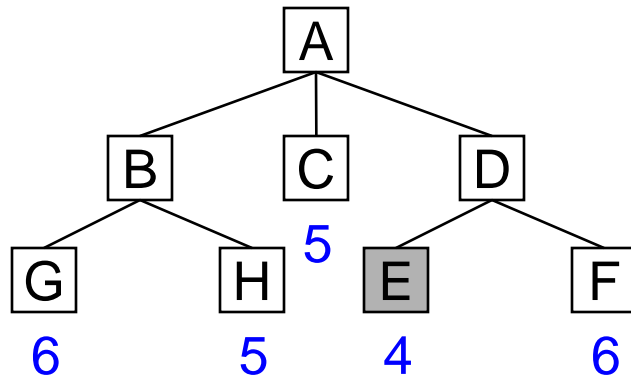
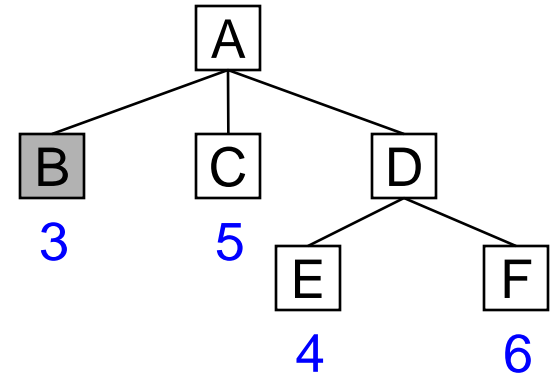
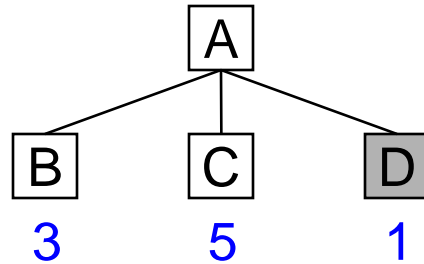
# BEST-FIRST SEARCH

- 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.
- ⇒ Combining the two is to **follow a single path at a time**, but **switch paths** whenever some competing path look more promising 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 VS HILL CLIMBING

- Similar to Steepest ascent hill climbing with two exceptions:
  - In hill climbing, one move is selected and all the others are rejected, never to be reconsidered. In BFS, one move is selected, but the others are kept around so that they can be revisited later if the selected path becomes less promising
  - The best available state is selected in the BFS, even if that state has a value that is lower than the value of the state that was just explored. Whereas in hill climbing the progress stop if there are no better successor nodes.

# 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.

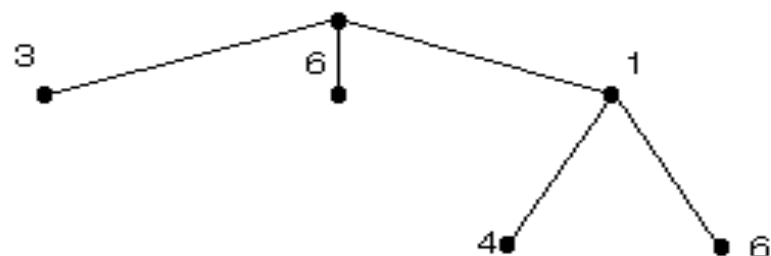
Whenever a new node is generated, **check** whether it has been **generated before**.

# Algorithm : Best-First Search

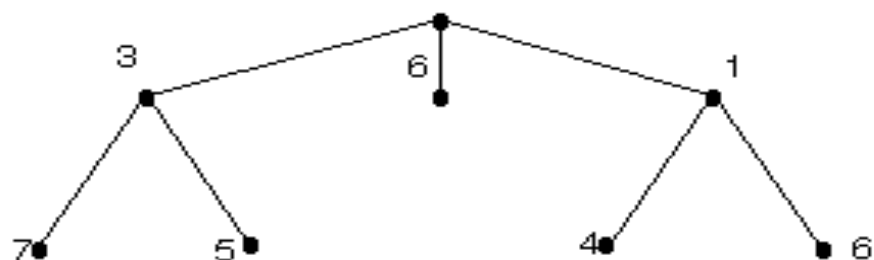
1. Start with *OPEN* containing just the initial state.
2. Until a goal is found or there are no nodes left on *OPEN* do:
  - (a) Pick the best node on *OPEN*.
  - (b) Generate its successors.
  - (c) For each successor do:
    - (i) If it has not been generated before, evaluate it, add it to *OPEN*, 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.



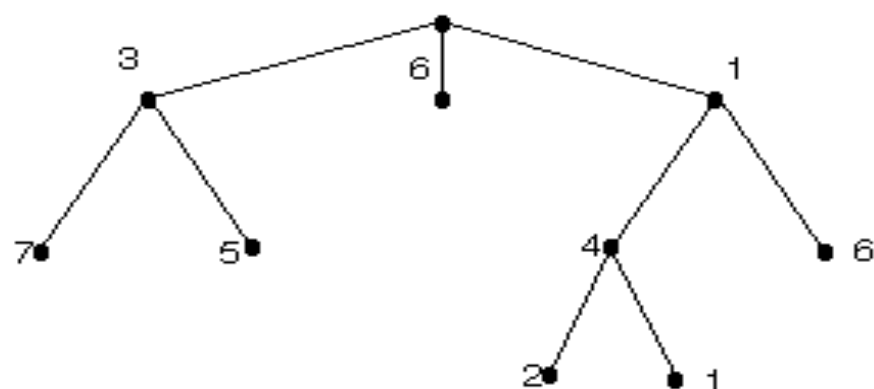
STEP 1



STEP 2



STEP 3



STEP 4

*All figures indicate "cost" of move*

# A\* ALGORITHM

- Best First Search is a simplification of A\* Algorithm
- Algorithm uses:
  - $f'$  : Heuristic function that estimates the merits of each node we generate. This is sum of two components,  $g$  and  $h'$ 
    - $f'$  represents an estimate of the cost of getting from the initial state to a goal state along with the path that generated the current node.
  - $g$  : The function  $g$  is a measure of the cost of getting from initial state to the current node.
  - $h'$  : The function  $h'$  is an estimate of the additional cost of getting from the current node to a goal state.
  - OPEN
  - CLOSED



# ALGORITHM A\*

○ Algorithm A\* (Hart et al., 1968):

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

$h(n)$  = cost of the cheapest path from node  $n$  to a goal state.

$g(n)$  = cost of the cheapest path from the initial state to node  $n$ .

○ Algorithm A\*:

$$f^*(n) = g^*(n) + h^*(n)$$

$h^*(n)$  (heuristic factor) = estimate of  $h(n)$ .

$g^*(n)$  (depth factor) = approximation of  $g(n)$  found by A\* so far.

# A\* ALGORITHM

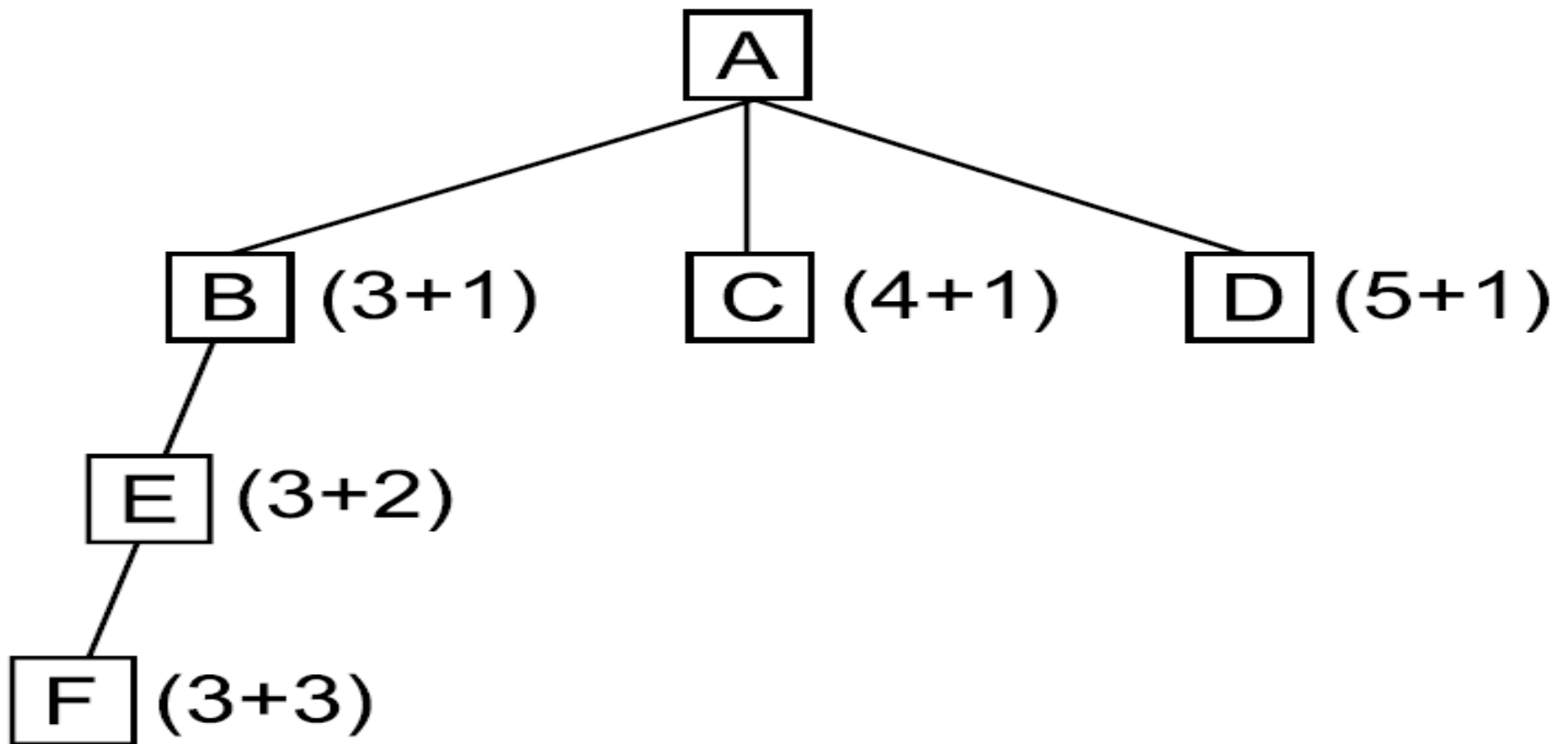
1. Start with OPEN containing only initial node. Set that node's  $g$  value to 0, its  $h'$  value to whatever it is, and its  $f'$  value to  $h' + 0$  or  $h'$ . Set CLOSED to empty list.
2. Until a goal node is found, repeat the following procedure:
  1. If there are no nodes on OPEN, report failure.
  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.

# OBSERVATIONS ABOUT A\*

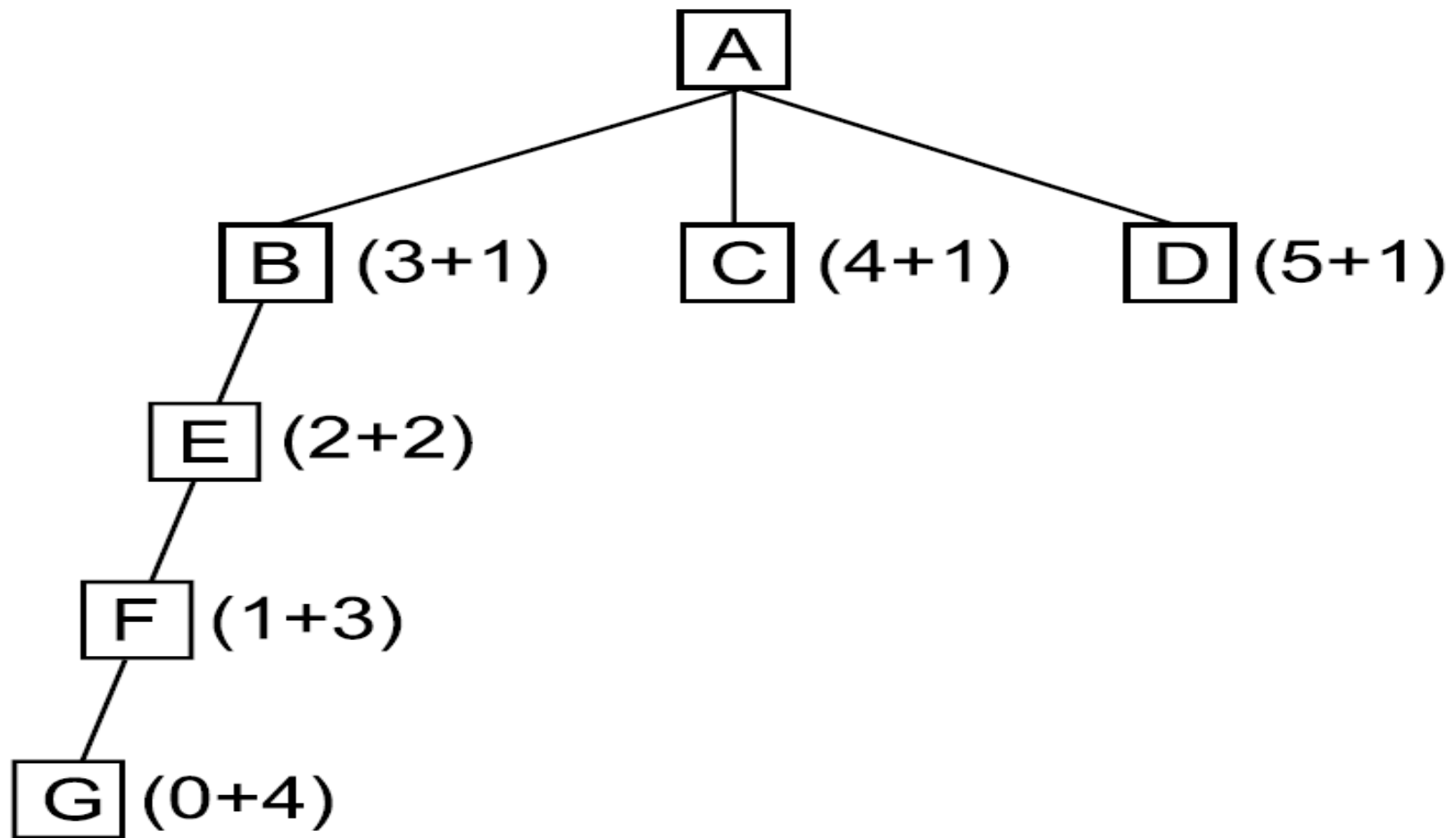
- **Role of g function:** This lets us choose which node to expand next on the basis of not only of how good the node itself looks, but also on the basis of how good the path to the node was.
- **h**, the distance of a node to the goal. If  $h'$  is a perfect estimator of  $h$ , then A\* will converge immediately to the goal with no search.



*$h'$  Underestimates  $h$*



*$h'$  Overestimates  $h$*



**Thank You!!!**

# Algorithm : Constraint Satisfaction

1. Propagate available constraints. To do this, first set *OPEN* to the set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until *OPEN* is empty:
  - (a) Select an object *OB* from *OPEN*. Strengthen as much as possible the set of constraints that apply to *OB*.
  - (b) If this set is different from the set that was assigned the last time *OB* was examined or if this is the first time *OB* has been examined, then add to *OPEN* all objects that share any constraints with *OB*.
  - (c) Remove *OB* from *OPEN*.
2. If the union of the constraints discovered above defines a solution, then quit and report the solution.
3. If the union of the constraints discovered above defines a contradiction, then return failure.
4. If neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this, loop until a solution is found or all possible solutions have been eliminated:
  - (a) Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.
  - (b) Recursively invoke constraint satisfaction with the current set of constraints augmented by the strengthening constraint just selected.



# A Cryptarithmic Problem

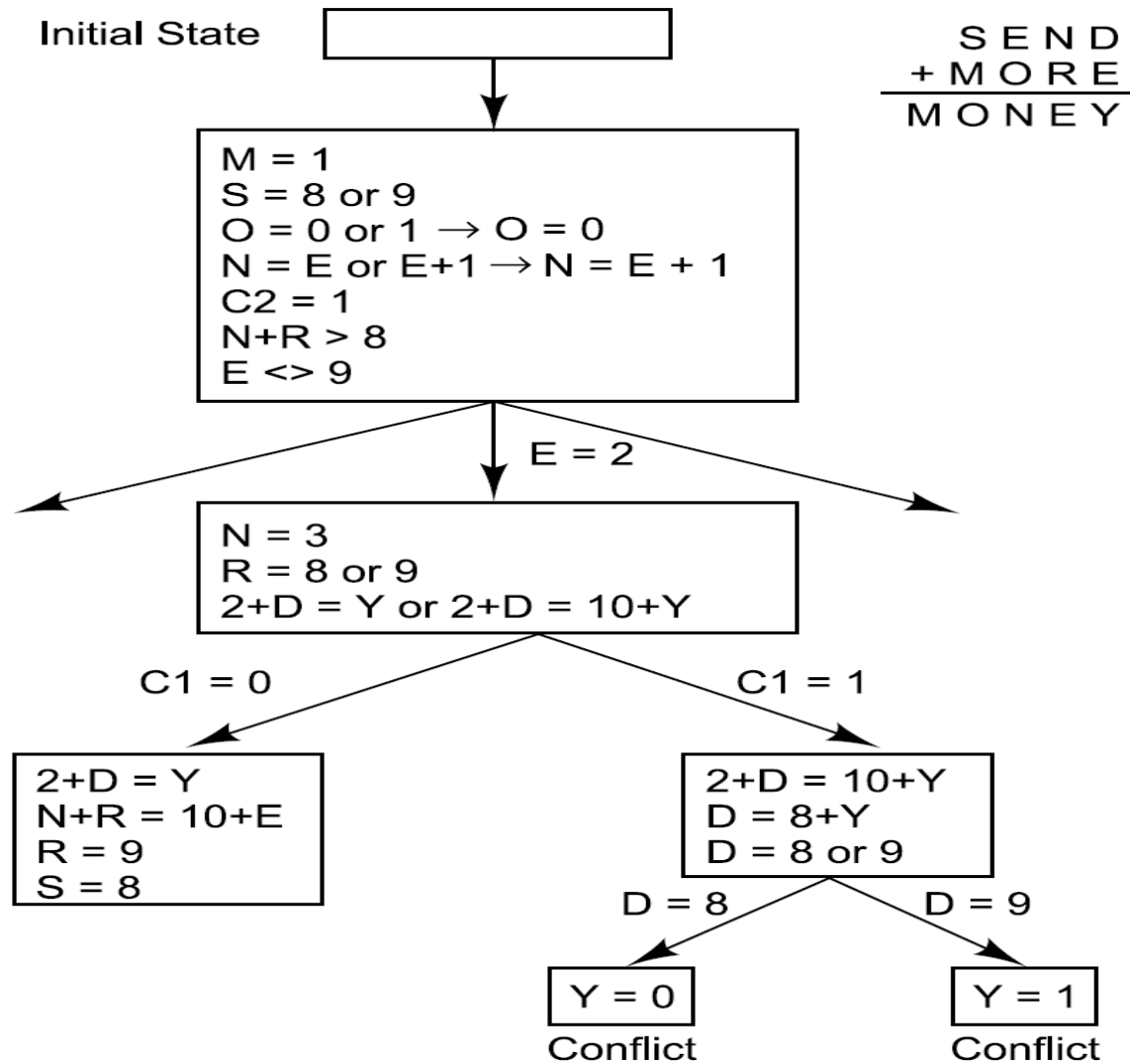
Problem:

$$\begin{array}{r} \text{SEND} \\ + \text{MORE} \\ \hline \text{MONEY} \end{array}$$

Initial State:

No two letters have the same value.  
The sums of the digits must be as shown in the problem.

# Solving a Cryptarithmic Problem



From column 5, **M = 1** since it is the only carry-over possible from the sum of two single digit numbers in column 4.

Since there is a carry in column 5, and  $M = 1$ , then **O = 0**

Since  $O = 0$  there cannot be a carry in column 4 (otherwise  $N$  would also be 0 in column 3) so **S = 9**.

If there were no carry in column 3 then  $E = N$ , which is impossible. Therefore there is a carry and  $N = E + 1$ .

If there were no carry in column 2, then  $(N + R) \bmod 10 = E$ , and  $N = E + 1$ , so  $(E + 1 + R) \bmod 10 = E$  which means  $(1 + R) \bmod 10 = 0$ , so  $R = 9$ . But  $S = 9$ , so there must be a carry in column 2 so **R = 8**.

To produce a carry in column 2, we must have  $D + E = 10 + Y$ .

$Y$  is at least 2 so  $D + E$  is at least 12.

The only two pairs of available numbers that sum to at least 12 are (5,7) and (6,7) so either  $E = 7$  or  $D = 7$ .

Since  $N = E + 1$ ,  $E$  can't be 7 because then  $N = 8 = R$  so **D = 7**.

$E$  can't be 6 because then  $N = 7 = D$  so **E = 5** and **N = 6**.

$D + E = 12$  so **Y = 2**.