INT404 ARTIFICIAL INTELLIGENCE

Best first search
OR graph
A*

Lecture 9

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.

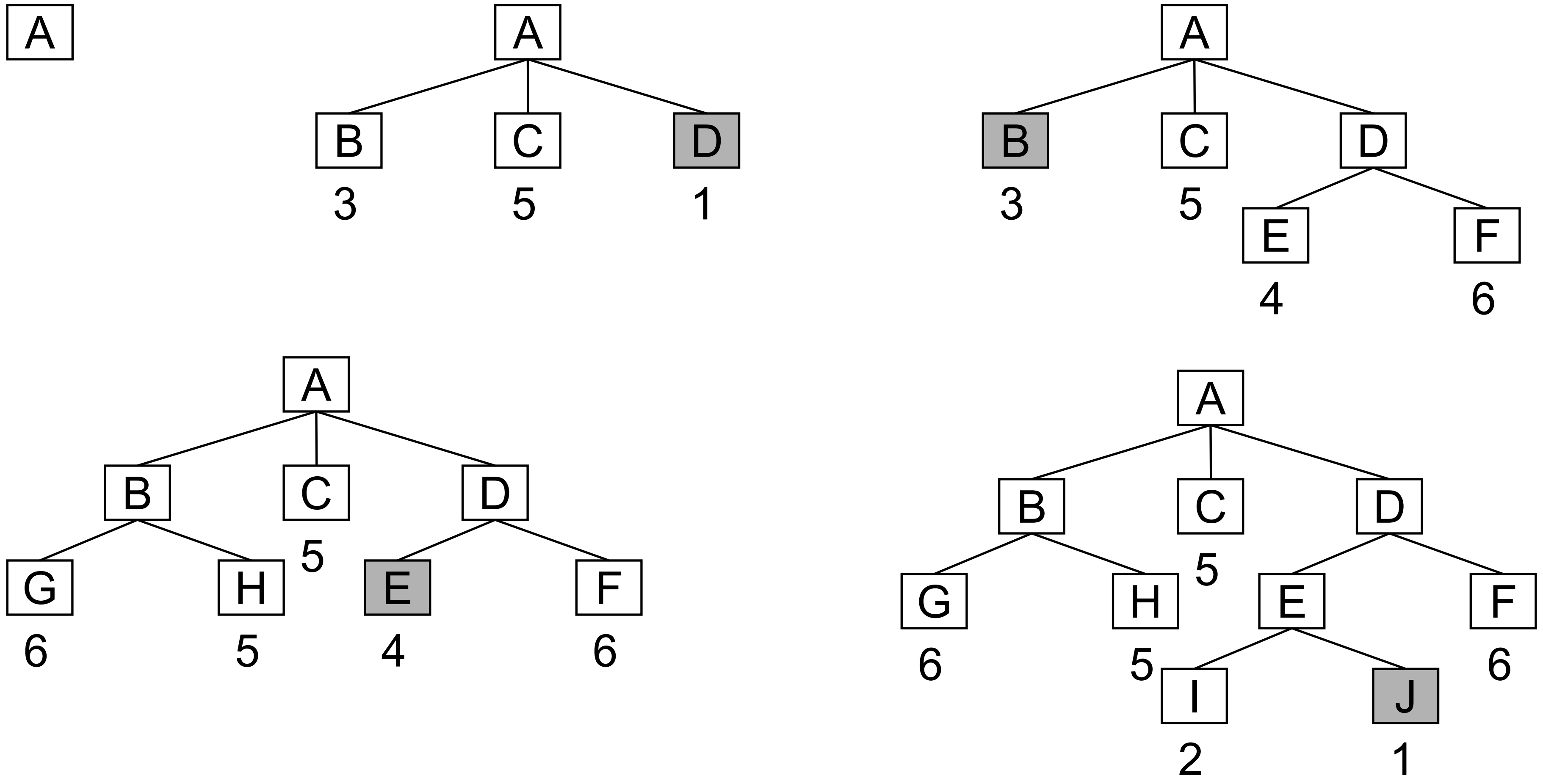
- **Breadth-first search:** not getting trapped on dead-end paths.

  $\Rightarrow$ 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 VS HILL CLIMBING

- Similar to Steepest ascent /Gradient search 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 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.

Analysis :

The worst case time complexity for Best First Search is O(n * Log n) where n is number of nodes. In worst case, we may have to visit all nodes before we reach goal. Note that priority queue is implemented using Min(or Max) Heap, and insert and remove operations take O(log n) time.
Performance of the algorithm depends on how well the cost or evaluation function is designed.

Best first search heuristic **$f(n)=h(n)$**

Manhattan distance:

The Manhattan distance heuristic is used for its simplicity and also because it is actually a pretty good underestimate (aka a lower bound) on the number of moves required to bring a given board to the solution board. We simply compute the sum of the distances of each tile from where it belongs, completely ignoring all the other tiles. For example, the Manhattan distance between "213540678" and "123456780" is 9:

```
2 1 3     1 2 3          1 2 3 4 5 6 7 8
5 4 0     4 5 6          ---------------
6 7 8     7 8 0          1+1+0+1+1+3+1+1  = 9
```

This is so, because the tile "1" is 1 move away, the tile "2" is 1 move away, the tile "3" is 0 moves away, the tile "4" is 1 move away, the tile "5" is 1 move away, the "6" tile is 3 moves away, the "7" is 1 move away, and the "8" is 1 move away.

Note that we do not consider the empty space, as this would cause the Manhattan distance heuristic to *not* be an underestimate.

Another example:

```
6 4 7     1 2 3          1 2 3 4 5 6 7 8
8 5 0     4 5 6          ---------------
3 2 1     7 8 0          4+2+4+2+0+3+4+2  = 21
```

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.

A* ALGORITHM (CONTD….)

- For each of the SUCCESSOR, do the following:
  - Set SUCCESSOR to point back to BESTNODE. These backwards links will make it possible to recover the path once a solution is found.
  - Compute g(SUCCESSOR) = g(BESTNODE) + the cost of getting from BESTNODE to SUCCESSOR
  - See if SUCCESSOR is the same as any node on OPEN. If so call the node OLD.
  - 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.
  - 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)

Example

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.

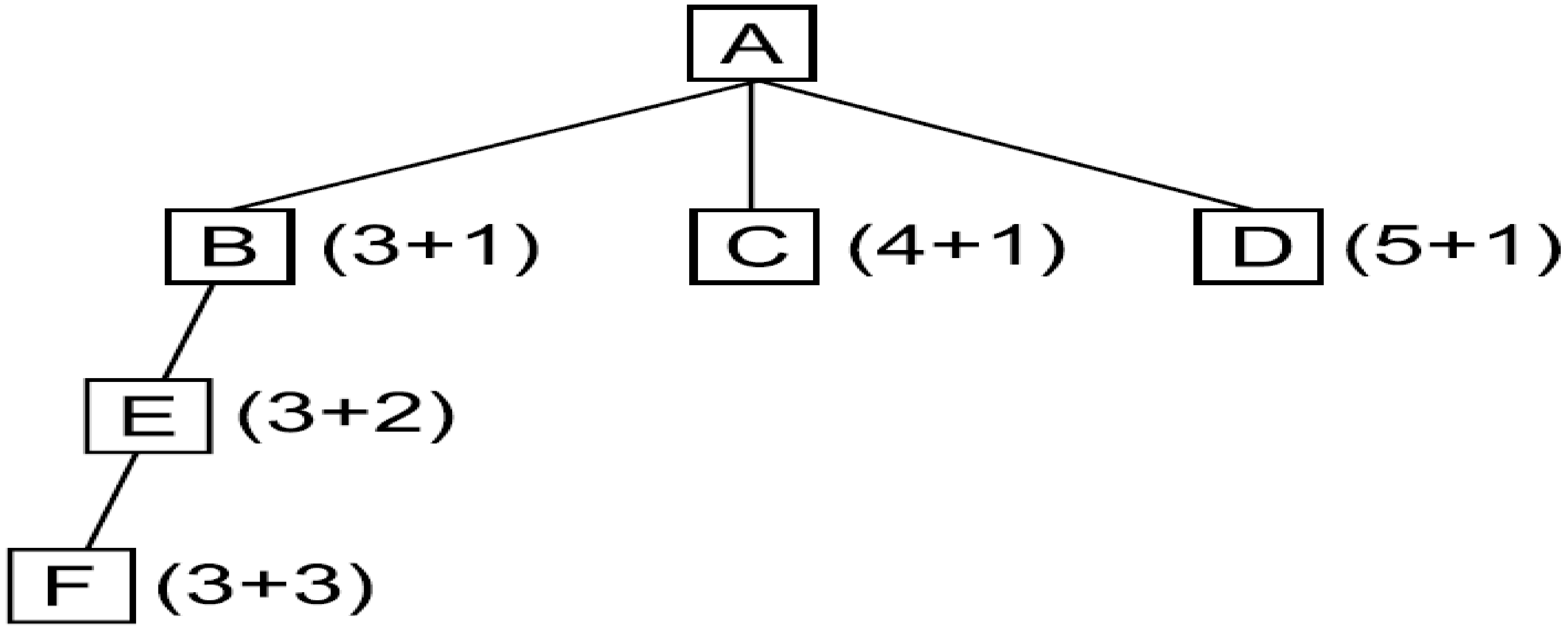
Time Complexity
Considering a graph, it may take us to travel all the edge to reach the destination cell from the source cell [For example, consider a graph where source and destination nodes are connected by a series of edges, like – 0(source) –>1 –> 2 –> 3 (target) So the worse case time complexity is **O(E)**, where E is the number of edges in the graph

Auxiliary Space In the worse case we can have all the edges inside the open list, so required auxiliary space in worst case is O(V), where V is the total number of vertices.

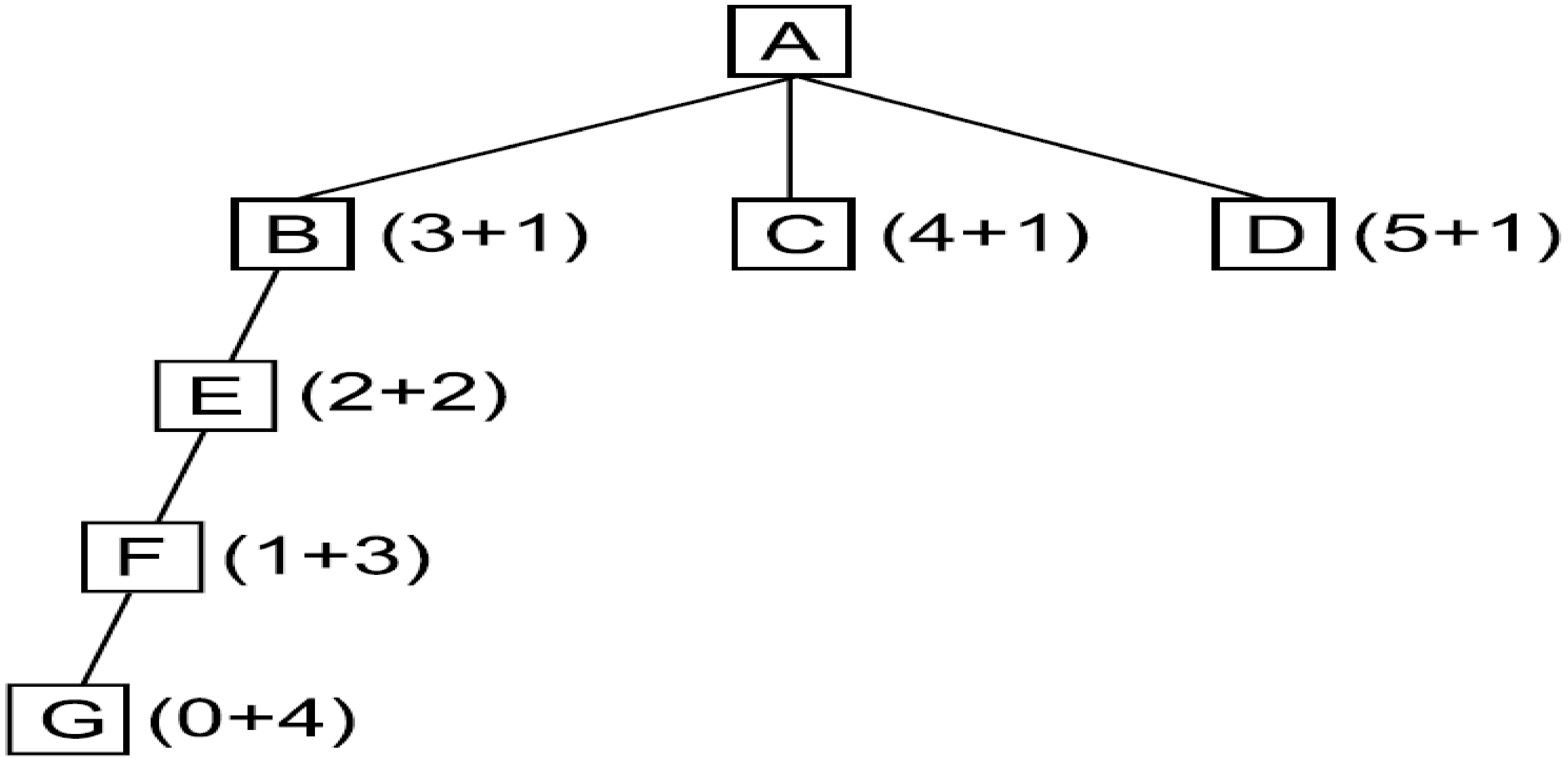
Admissible heuristics

- A heuristic *h(n)* is admissible if for every node *n*,
  $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from *n*.
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Theorem: If *h(n)* is admissible, A^* using `TREE-SEARCH` is optimal

h´ Underestimates h



h´ Overestimates h



Heuristic Function for Different Problems are as follow:

- WATER JUG= |XC-XG|+|YC-YG|, WHERE (XC,YC) REPRESENT CURRENT STATE AND (XG,YG) REPRESENT GOAL STATE

- TOWER OF HANOI=(CA-GA)+(CB-GB)+(CC-GB), WHERE (CA,CB,CC) REPRESENT CURRENT NUMBER OF TILES IN POLE A, B AND C AND, (GA,GB,GC) REPRESENTS GOAL TILES IN POLE A, B AND C

- 8 PUZZLE= TOTAL NUMBER OF MISPLACED TILES (EXCLUDING SPACE)

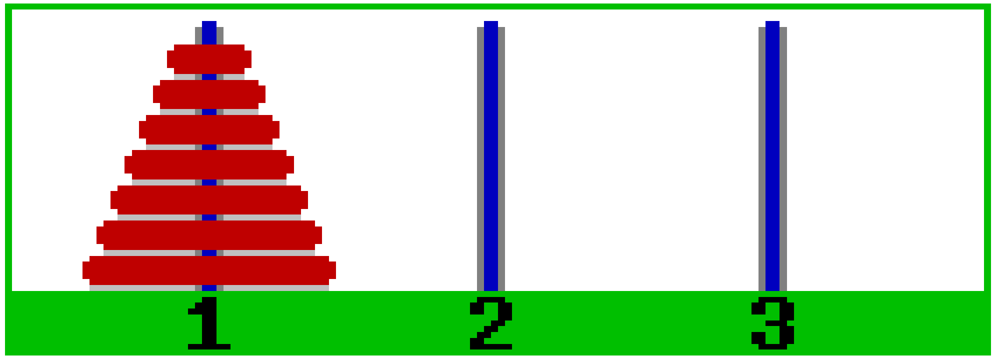
  OR

  TOTAL NUMBER OF CORRECTLY PLACED TILES(EXCLUDING SPACE)

- MISSIONARY CANNIBAL= TOTAL NUMBER OF MISSIONARY ON THE RHS+TOTAL NUMBER OF CANNIBAL ON THE RIGHT HAND SIDE

- BLOCK WORD= LOCAL AND GLOBAL HEURISTIC GIVEN IN PPTS

Example Problems – Towers of Hanoi



States: combinations of poles and disks

Operators: move disk x from pole y to pole z subject to constraints
- cannot move disk on top of smaller disk
- cannot move disk if other disks on top

Goal test: disks from largest (at bottom) to smallest on goal pole

Path cost: 1 per move

❖TOWER OF HANOI=(CA-GA)+(CB-GB)+(CC-GB), WHERE (CA,CB,CC) REPRESENT CURRENT NUMBER OF TILES IN POLE A, B AND C AND, (GA,GB,GC) REPRESENTS GOAL TILES IN POLE A, B AND C