Water jug problem

- **Problem statement**:
  - Given two jugs, a 4-gallon and 3-gallon having no measuring markers on them. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into 4-gallon jug?
- **Defining the problem as a state space search :**
  - **State space**: State for this problem can be described as the set of ordered pairs of integers (X, Y) such that
    - X represents the number of gallons of water in 4-gallon jug and
    - Y for 3-gallon jug.
  - **Initial state**: Start state is (0,0)
  - **Goal state:** is (2, N) for any value of N
  - **Action/ Rules:** Following are the production rules for this problem

R1 :(X, Y | X < 4) → (4, Y) {Fill 4-gallon jug}

R2:(X, Y | Y < 3) → (X, 3) {Fill 3-gallon jug}

R3:(X, Y | X > 0) → (0, Y) {Empty 4-gallon jug}

R4:(X, Y | Y > 0) → (X, 0) {Empty 3-gallon jug}

R5:(X, Y | X+Y >= 4 Λ Y > 0) →(4, Y – (4 – X)) {Pour water from 3- gallon jug into 4-gallon jug until 4-gallon jug is full}

R6:(X, Y | X+Y >= 3 Λ X > 0) →(X – (3 – Y), 3) {Pour water from 4-gallon jug into 3-gallon jug until 3-gallon jug is full}

R7:(X, Y | X+Y <= 4 Λ Y > 0) →(X+Y, 0) {Pour all water from 3-gallon jug into 4-gallon jug }

R8:(X, Y | X+Y <= 3 Λ X > 0) →(0, X+Y) {Pour all water from 4-gallon jug into 3-gallon jug }

R9:(X, Y | X > 0) →(X – D, Y) {Pour some water D out from 4-gallon jug}

R10:(X, Y | Y > 0) →(X, Y - D) {Pour some water D out from 3- gallon jug}

- Using appropriate control strategy production rules can be applied to get solution for this problem as:

Number of Steps	Rules applied	4-g jug	3-g jug	
1		0	0	**Initial State**
2	R2 {Fill 3-g jug}	0	3	
3	R7{Pour all water from 3 to 4-g jug }	3	0	
4	R2 {Fill 3-g jug}	3	3	
5	R5 {Pour from 3 to 4-g jug until it is full}	4	2	
6	R3 {Empty 4-gallon jug}	0	2	
7	R7 {Pour all water from 3 to 4-g jug}	2	0	**Goal State**

BFS vs DFS

BFS	DFS
BFS traverse tree level wise	DFS traverse tree depth wise
No backtracking is required	DFS uses backtracking
Data structure used is queue(FIFO)	Data structure used is stack(LIFO)
BFS never gets trapped into infinite loop	DFS generally gets trapped into infinite loop
When succeeds, the goal node found is minimum depth	When succeeds, the goal node found is not necessarily minimum depth
Guarantees complete, optimal solution	Does not guarantee complete, optimal solution
Large tree may require excessive memory	Large tree may take excessive long time to find even a nearby goal node

Algorithm:

Step 1: Place the root node inside the queue.

Step 2: If the queue is empty then stops and return failure.

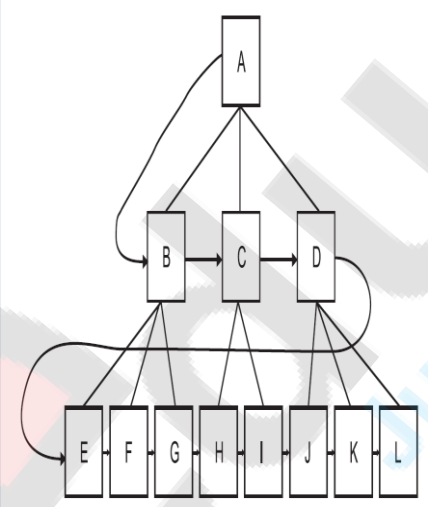
Step 3: If the FRONT node of the queue is a goal node then stop and return success.

Step 4: Remove the FRONT node from the queue. Process it and find all its neighbors that are in ready state then place them inside the queue in any order.

Step 5: Go to Step 3.

Step 6: Exit

Example



A, B, C, D, E, F, G, H, I, J, K, L

Algorithm:

Step 1: PUSH the starting node into the stack.

Step 2: If the stack is empty then stops and return failure.

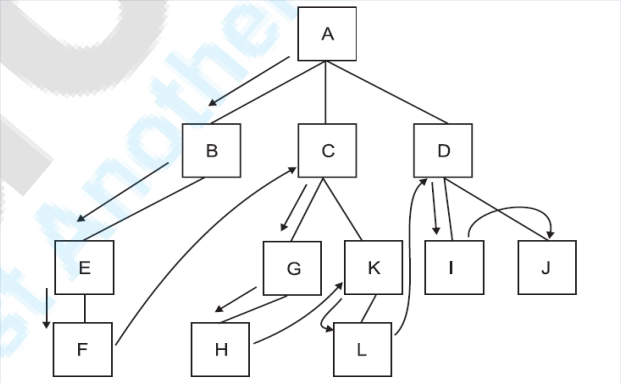
Step 3: If the top node of the stack is the goal node, then stop and return success.

Step 4: Else POP the top node from the stack and process it. Find all its neighbors that are in ready state and PUSH them into the stack in any order

Step 5: Go to step 3.

Step 6: Exit

Example:



A, B, E, F, C, G, H, K, L, D, I, J

- **Hill climbing**

Refer ppt