Problem Solving Techniques

Topics

- Solving Problems by Searching: Problem Solving Agents, Searching for Solutions.
- Uninformed Search Strategies- Breadth first search, depth first Search
- Informed (Heuristic) Search Strategies- Hill climbing, A* Algorithm, Alpha-Beta Pruning, Constraint Satisfaction Problem.

Problem-solving agents

- **Problem solving agents are goal-directed agents:**

1. Goal Formulation: Set of one or more (desirable) world states (e.g. checkmate in chess).
2. **Problem formulation: What actions and states to consider given a goal and an initial state.**
3. Search for solution: Given the problem, search for a solution --- *a sequence of actions to achieve the goal starting from the initial state.*
4. Execution of the solution

Terminology

- **Search Space:** Search space represents a set of possible solutions, which a system may have.
- **Start State:** It is a state from where agent begins **the search**.
- **Goal test:** It is a function which observe the current state and returns whether the goal state is achieved or not.
- **Search tree:** A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- **Actions:** It gives the description of all the available actions to the agent.
- **Path Cost:** It is a function which assigns a numeric cost to each path.
- **Solution:** It is an action sequence which leads from the start node to the goal node.
- **Optimal Solution:** If a solution has the lowest cost among all solutions.

Water Jug Problem

Rule No	Left of Rule OR CONDITION	Right of Rule OR OPERATION	Description
1	(X,Y | X<5)	(5,Y)	Fill 5-L Jug
2	(X,Y | X>0)	(0,Y)	Empty 5-L Jug
3	(X,Y | Y<3)	(X,3)	Fill 3-L Jug
4	(X,Y | Y>0)	(X,0)	Empty 3-L Jug
5	(X,Y | X+Y<=5 & Y>0)	(X+Y,0)	Empty 3-L Jug into 5-L Jug
6	(X,Y | X+Y<=3 & X>0)	(0,X+Y)	Empty 5-L Jug into 3-L Jug
7	(X,Y | X+Y>=5 & Y>0)	(5,Y-(5-X))	Pour water from 3-L Jug into 5-L jug until 5-L Jug is full
8	(X,Y | X+Y>=3 & X>0)	(X-(3-Y),3)	Pour water from 5-L Jug into 3-L jug until 3-L Jug is full

Water Jug Problem (X-m, Y-n), m>n

Rule No	Left of Rule OR CONDITION	Right of Rule OR OPERATION	Description
1	(X,Y | X<m)	(m, Y)	Fill m-L Jug
2	(X,Y | X>0)	(0, Y)	Empty m-L Jug
3	(X,Y | Y<n)	(X, n)	Fill n-L Jug
4	(X,Y | Y>0)	(X, 0)	Empty n-L Jug
5	(X,Y | X+Y<=m & Y>0)	(X+Y, 0)	Empty n-L Jug into m-L Jug
6	(X,Y | X+Y<=n & X>0)	(0, X+Y)	Empty m-L Jug into n-L Jug
7	(X,Y | X+Y>=m & Y>0)	(m, Y-(m-X))	Pour water from n-L Jug into m-L jug until m-L Jug is full
8	(X,Y | X+Y>=n & X>0)	(X-(n-Y), n)	Pour water from m-L Jug into n-L jug until n-L Jug is full

Table 2.4 Production Rules for Missionaries and Cannibals Problem

RN	Left side of rule	→	Right side of rule
Rules for boat going from left bank to right bank of the river			
L1	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	→	$([(n_1-2)M, m_1C, 0B], [(n_2+2)M, m_2C, 1B])$
L2	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	→	$([(n_1-1)M,(m_1-1)C,0B],[(n_2+1)M,(m_2+1)C, 1B])$
L3	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	→	$([n_1M, (m_1-2)C, 0B], [n_2M, (m_2+2)C, 1B])$
L4	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	→	$([(n_1-1)M, m_1C, 0B],[(n_2+1)M, m_2C, 1B])$
L5	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	→	$([n_1M, (m_1-1)C, 0B], [n_2M, (m_2+1)C, 1B])$
Rules for boat coming from right bank to left bank of the river			
R1	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	→	$([(n_1+2)M, m_1C, 1B], [(n_2-2)M, m_2C, 0B])$
R2	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	→	$([(n_1+1)M,(m_1+1)C,1B],[(n_2-1)M,(m_2-1)C, 0B])$
R3	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	→	$([n_1M, (m_1+2)C, 1B], [n_2M, (m_2-2)C, 0B])$
R4	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	→	$([(n_1+1)M, m_1C,1B],[(n_2-1)M, m_2C, 0B])$
R5	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	→	$([n_1M, (m_1+1)C, 1B], [n_2M, (m_2-1)C, 0B])$

Table 2.5 Solution Path

Rule number	**([3M, 3C, 1B], [0M, 0C, 0B]] ← Start State**
L2:	([2M, 2C, 0B], [1M, 1C, 1B])
R4:	([3M, 2C, 1B], [0M, 1C, 0B])
L3:	([3M, 0C, 0B], [0M, 3C, 1B])
R5:	([3M, 1C, 1B], [0M, 2C, 0B])
L1:	([1M, 1C, 0B], [2M, 2C, 1B])
R2:	([2M, 2C, 1B], [1M, 1C, 0B])
L1:	([0M, 2C, 0B], [3M, 1C, 1B])
R5:	([0M, 3C, 1B], [3M, 0C, 0B])
L3:	([0M, 1C, 0B], [3M, 2C, 1B])
R5:	([0M, 2C, 1B], [3M, 1C, 0B])
L3:	**([0M, 0C, 0B], [3M, 3C, 1B]) → Goal state**

SEARCHING FOR SOLUTIONS

- The possible action sequences starting at the initial state form a **search tree** with the **initial state at the root**; the **branches are actions** and the **nodes correspond to states** in the state space of the problem.
- We do this by expanding the current state; that is, applying each legal action to the current state, thereby generating a new set of states.
- The set of all leaf nodes available for expansion at any given point is called the **frontier.**

Infrastructure for search algorithms

◉Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four components:

• n.STATE: the state in the state space to which the node corresponds;

• n.PARENT: the node in the search tree that generated this node;

• n.ACTION: the action that was applied to the parent to generate the node;

• n.PATH-COST: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

Measuring problem-solving performance

Completeness: Is the algorithm guaranteed to find a solution when there is one?

Optimality: Does the strategy find the optimal solution?

Time complexity: How long does it take to find a solution?

Space complexity: How much memory is needed to perform the search?

Uninformed Search Algorithms

Uninformed search is a kind of general-purpose search method that uses brute force to get results. Because uninformed search algorithms have no extra knowledge about the state or searching region **beyond how to traverse the tree**, it is sometimes referred to as blind search.

- **Breadth-first Search**
- **Depth-first Search**

Breadth-first search (BFS)

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.

BFS Searching Technique

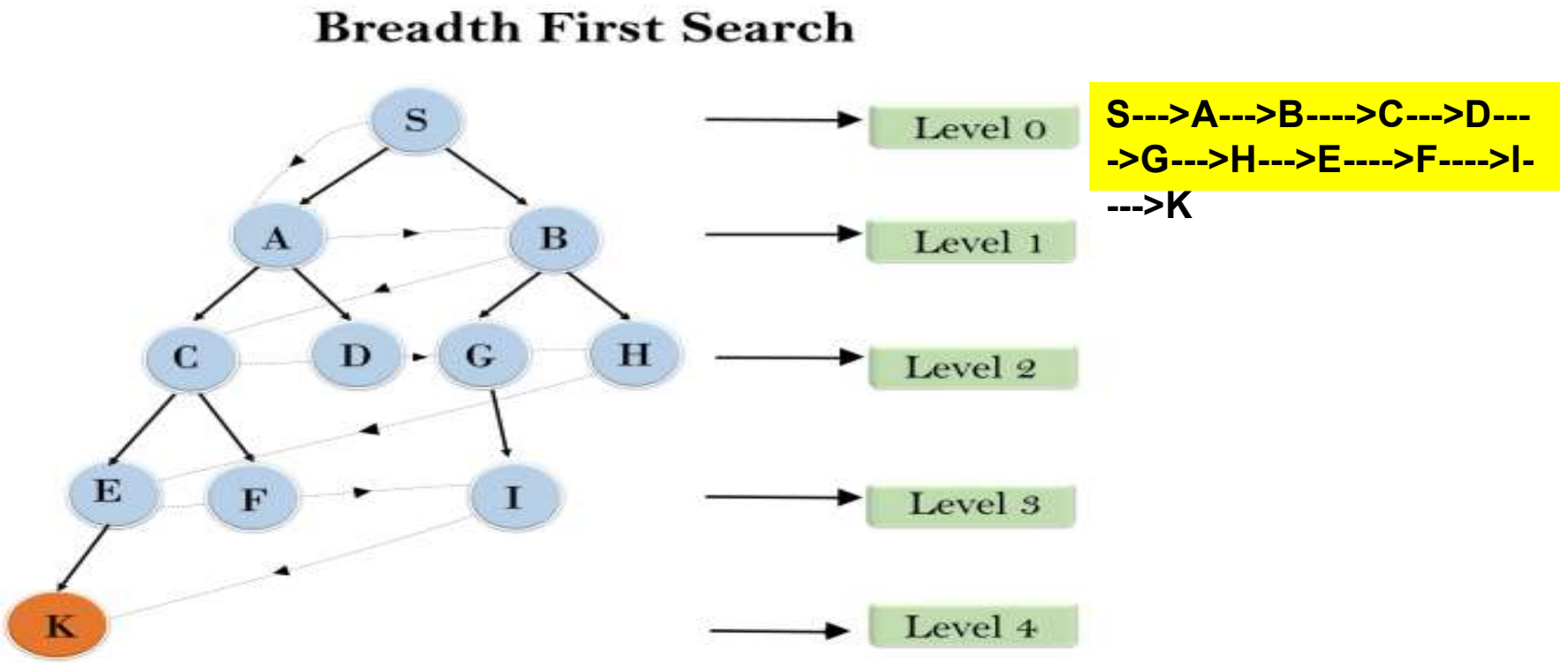
Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

Disadvantages:

- It **requires lots of memory** since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

BFS- Working


Breadth First Search
S
A
B
C
D
G
H
E
F
I
K
Level 0
Level 1
Level 2
Level 3
Level 4
S--->A--->B---->C--->D--->G--->H--->E---->F---->I--->K

Water Jug Problem

Rule No	Left of Rule OR CONDITION	Right of Rule OR OPERATION	Description
1	(X,Y | X<5)	(5,Y)	Fill 5-L Jug
2	(X,Y | X>0)	(0,Y)	Empty 5-L Jug
3	(X,Y | Y<3)	(X,3)	Fill 3-L Jug
4	(X,Y | Y>0)	(X,0)	Empty 3-L Jug
5	(X,Y | X+Y<=5 & Y>0)	(X+Y,0)	Empty 3-L Jug into 5-L Jug
6	(X,Y | X+Y<=3 & X>0)	(0,X+Y)	Empty 5-L Jug into 3-L Jug
7	(X,Y | X+Y>=5 & Y>0)	(5,Y-(5-X))	Pour water from 3-L Jug into 5-L jug until 5-L Jug is full
8	(X,Y | X+Y>=3 & X>0)	(X-(3-Y),3)	Pour water from 5-L Jug into 3-L jug until 3-L Jug is full

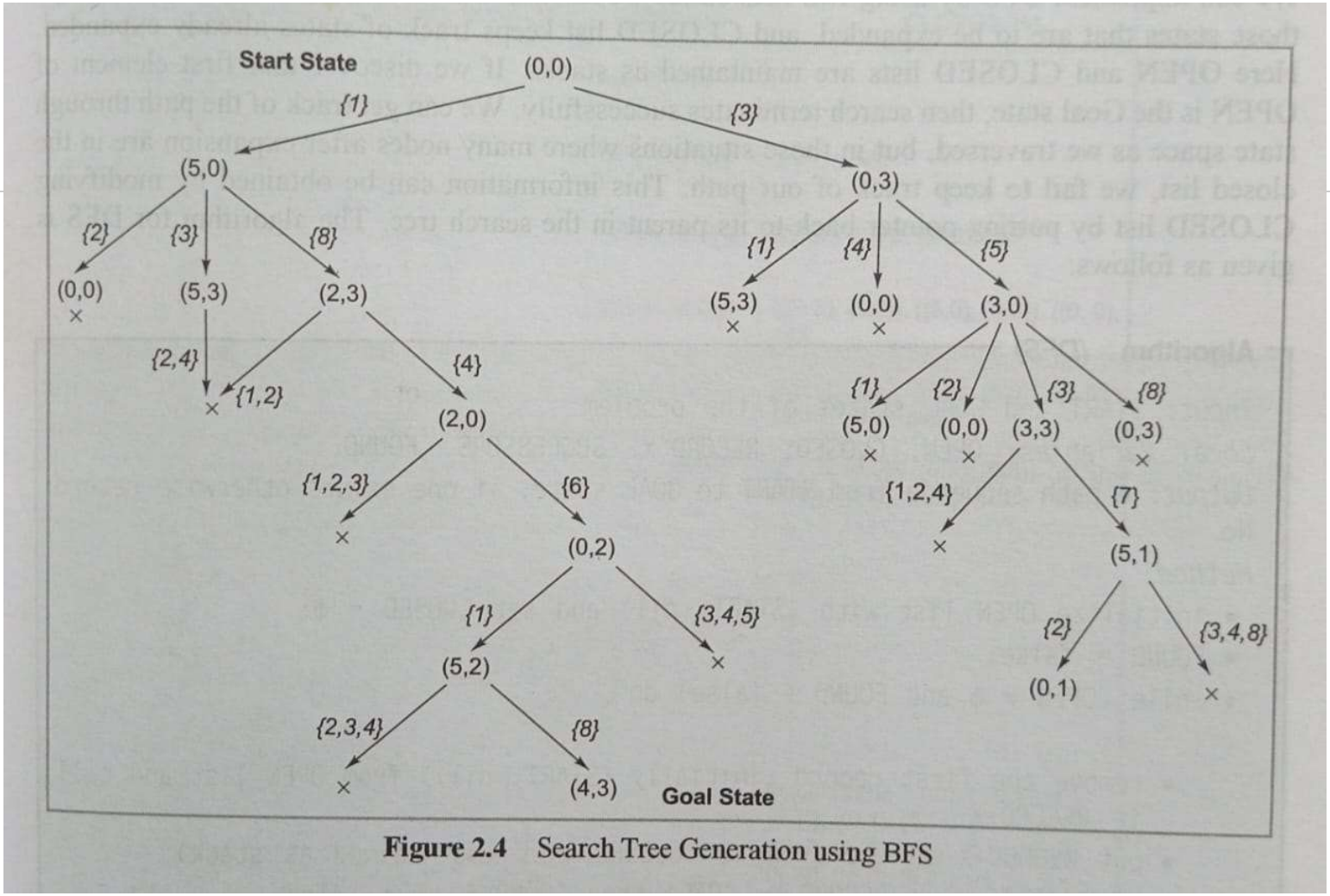
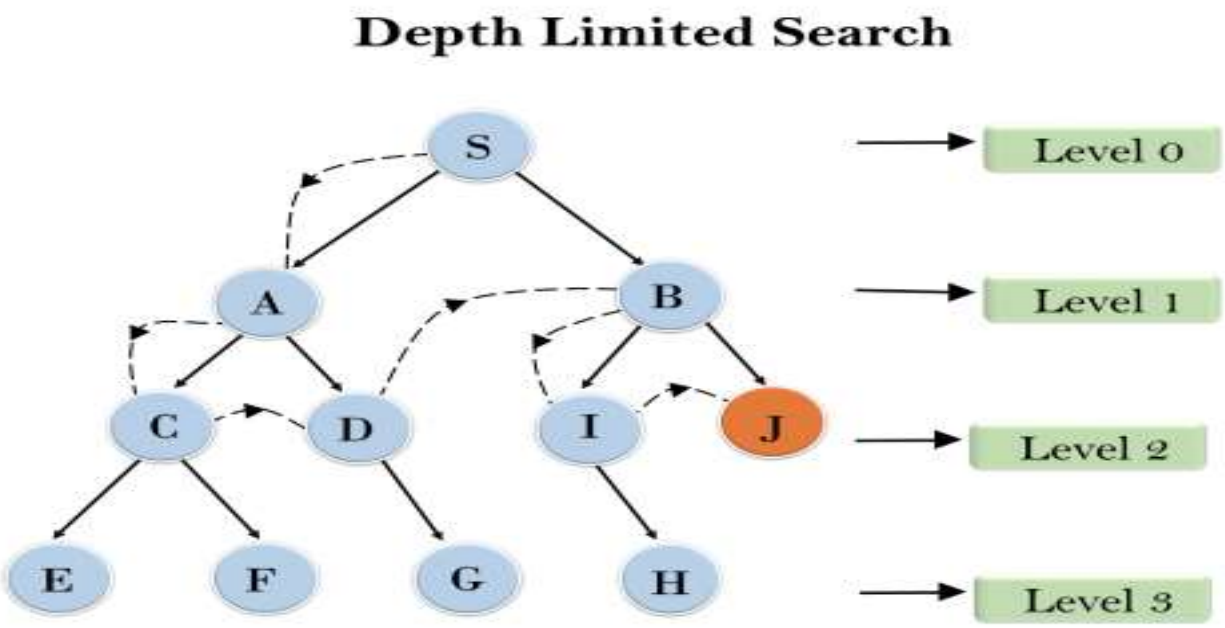


Figure 2.4 Search Tree Generation using BFS

Depth-first search (DFS)

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

DFS Working


Depth Limited Search
S
A
B
C
D
I
J
E
F
G
H
Level 0
Level 1
Level 2
Level 3

DFS Searching Technique

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantages:

- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

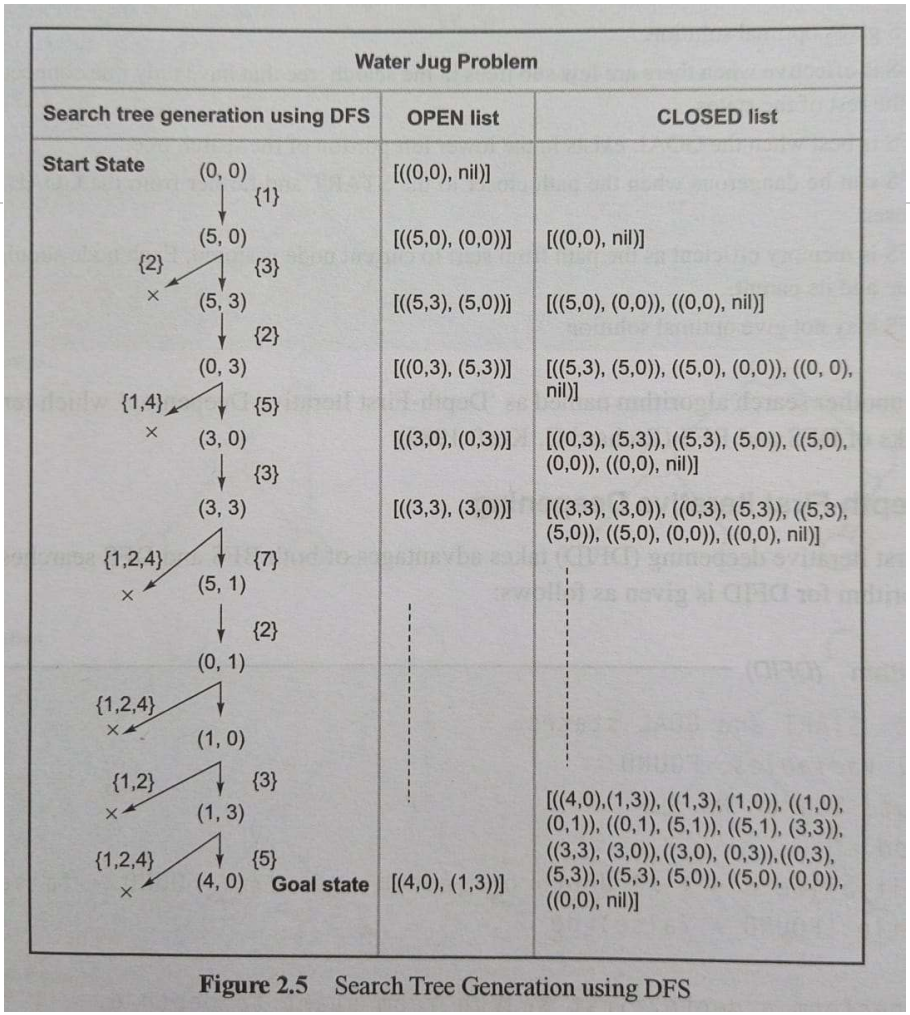


Figure 2.5 Search Tree Generation using DFS

Informed Search Algorithms

The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal.

Informed Search

- Hill climbing
- A* Algorithm
- Alpha-Beta Pruning

Hill Climbing

- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbour has a higher value.
- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance travelled by the salesman.
- It is also called **greedy local search as it only looks to its good immediate neighbour state and not beyond that.**

Features of Hill Climbing

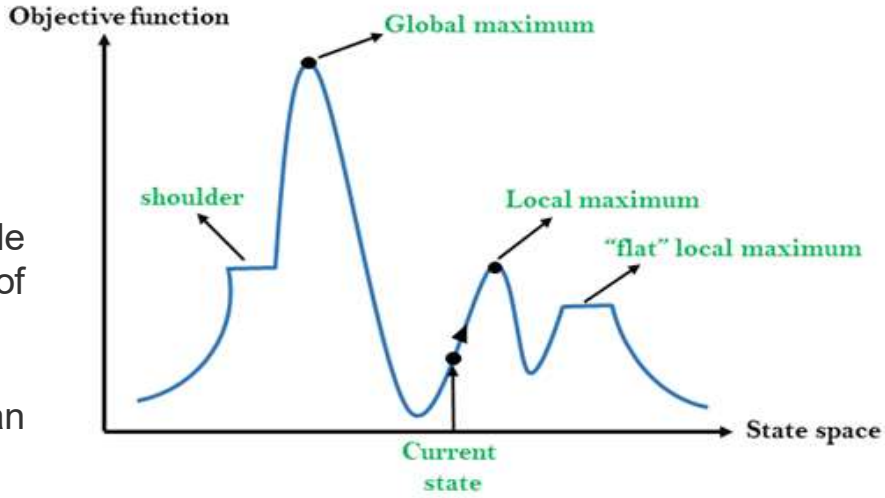
- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

Hill Climbing Different regions in the State Space Diagram:

Local Maximum: Local maximum is a state which is better than
its neighbour states, but there is also another state which is higher than it.

Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.

Current state: It is a state in a landscape diagram where an agent is currently present.



Flat local maximum: It is a flat space in the landscape where all the neighbour agents of current states have the same value.

Ridge: It is a region that is higher than its neighbors but itself has a slope. It is a special kind of local maximum.

Shoulder: It is a plateau that has an uphill edge.

Problems in different regions in Hill climbing

Hill climbing cannot reach the optimal/best state(global maximum) if it enters any of the following regions :

- **Local maximum:** At a local maximum all neighboring states have a value that is 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: Utilize 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.

Problems in different regions in Hill climbing

- **Plateau:** On the plateau, all neighbors 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 in a non-plateau region.
- **Ridge:** Any point on a ridge can look like a peak because movement in all possible directions is downward. Hence the algorithm stops when it reaches this state.

  To overcome Ridge: In this kind of obstacle, use two or more rules before testing. It implies moving in several directions at once.

Applications of Hill Climbing Algorithm

- **Machine Learning:** Hill climbing can be used for hyperparameter tuning in machine learning algorithms, finding the best combination of hyperparameters for a model.
- **Robotics:** In robotics, hill climbing can help robots navigate through physical environments, adjusting their paths to reach a destination.
- **Network Design:** It can be used to optimize network topologies and configurations in telecommunications and computer networks.
- **Game Playing:** In game playing AI, hill climbing can be employed to develop strategies that maximize game scores.
- **Natural Language Processing:** It can optimize algorithms for tasks like text summarization, machine translation, and speech recognition.

Algorithm for Simple Hill climbing :

- Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make the initial state as the current state.
- Loop until the solution state is found or there are no new operators present which can be applied to the current state.

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

  Perform these to evaluate the new state.

  If the current state is a goal state, then stop and return success.

  If it is better than the current state, then make it the current state and proceed further.

  If it is not better than the current state, then continue in the loop until a solution is found.
- Exit from the function.

Hill Climbing Algorithm

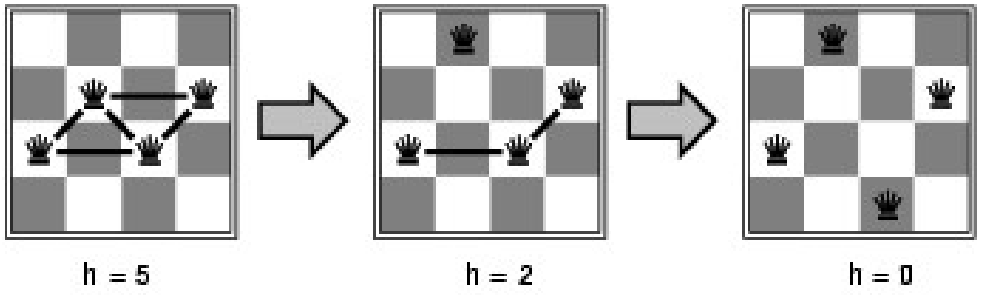
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

Problem: https://www.youtube.com/watch?v=wM4n12FHelM

4-Queens

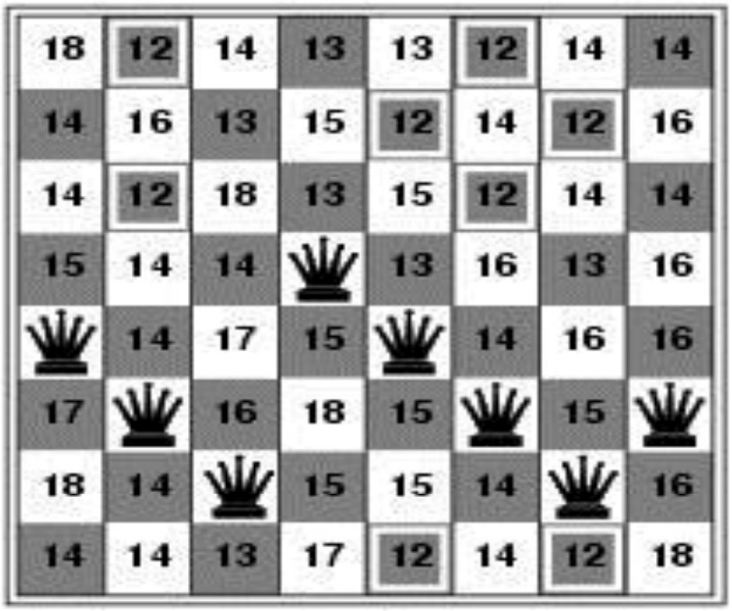
- **States:** 4 queens in 4 columns (256 states)
- **Neighborhood Operators:** move queen in column
- **Evaluation / Optimization function:** h(n) = number of attacks
- **Goal test:** no attacks, i.e., h(G) = 0

Initial state (guess).



Local search: Because we only consider local changes to the state at each step. We generally make sure that series of local changes can reach all possible states.

Hill-climbing search: 8-queens problem



- Need to convert to an optimization problem
- *h* = number of pairs of queens that are attacking each other
- *h = 17* for the above state

Types of Hill Climbing techniques

- Simple Hill Climbing
- Steepest-Ascent hill-climbing
- Stochastic hill Climbing

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**.
- It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state.
  - Less time consuming
  - Less optimal solution and the solution is not guaranteed

Algorithm

- **Step 1:** Evaluate the initial state, if it is goal state then return success and Stop.
- **Step 2:** Loop Until a solution is found or there is no new operator left to apply.
- **Step 3:** Select and apply an operator to the current state.
- **Step 4:** Check new state:
  - If it is goal state, then return success and quit.
  - Else if it is better than the current state then assign new state as a current state.
  - Else if not better than the current state, then return to step2.
- **Step 5:** Exit.

Steepest-Ascent hill-climbing

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

Algorithm

- **Step 1:** Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.
- **Step 2:** Loop until a solution is found or the current state does not change.
  - Let SUCC be a state such that any successor of the current state will be better than it.
  - For each operator that applies to the current state:
    - Apply the new operator and generate a new state.
    - Evaluate the new state.
    - If it is goal state, then return it and quit, else compare it to the SUCC.
    - If it is better than SUCC, then set new state as SUCC.
    - If the SUCC is better than the current state, then set current state to SUCC.
- **Step 5:** Exit.

Algorithm

Stochastic hill climbing does not examine for all its neighbour before moving. Rather, this search algorithm selects one neighbour node at random and decides whether to choose it as a current state or examine another state.

A* Searching Algorithm

- It is a searching algorithm that is used to find the shortest path between an initial and a final point.
- It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.
- Another aspect that makes A* so powerful is the use of weighted graphs in its implementation. A weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost, and find the best route in terms of distance and time.

Working of A*

◉A* Algorithm works as-

•It maintains a tree of paths originating at the start node.

•It extends those paths one edge at a time.

•It continues until its termination criterion is satisfied.

◉A* Algorithm extends the path that minimizes the following function-

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

•'n' is the last node on the path

•g(n) is the cost of the path from start node to node 'n'

•h(n) is a heuristic function that estimates cost of the cheapest path from node 'n' to the goal node

Algorithm for A*

- The implementation of A* Algorithm involves maintaining two lists- OPEN and CLOSED.
- OPEN contains those nodes that have been evaluated by the heuristic function but have not been expanded into successors yet.
- CLOSED contains those nodes that have already been visited.

Algorithm continued…

◉**Step-01:**

✓ Define a list OPEN.

✓ Initially, OPEN consists solely of a single node, the start node S.

◉**Step-02:**

✓ If the list is empty, return failure and exit.

◉**Step-03:**

✓ Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED.

✓ If node n is a goal state, return success and exit.

◉**Step-04:**

✓ Expand node n.

Algorithm Continued…

Step-05:

- ✓ If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S.
- ✓ Otherwise, go to Step-06.

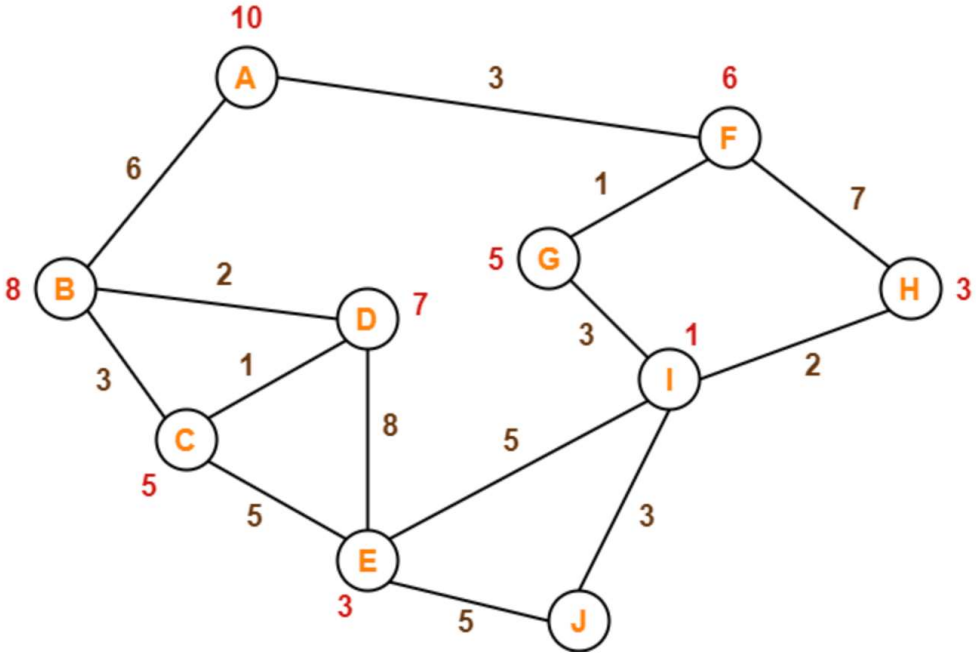
Step-06:

- ➢ For each successor node,
- ✓ Apply the evaluation function f to the node.
- ✓ If the node has not been in either list, add it to OPEN.

Step-07:

- ✓ Go back to Step-02.

Graph that we will work on… A is Initial and J is final state



Problem Example

◉**Step-01:**

✓ We start with node A.

✓ Node B and Node F can be reached from node A.

◉A* Algorithm calculates f(B) and f(F).

✓ f(B) = 6 + 8 = 14

✓ f(F) = 3 + 6 = 9

◉**Since f(F) < f(B), so it decides to go to node F.**

 **Path- A → F** 

Problem Example

- Node G and Node H can be reached from node F.
- A* Algorithm calculates f(G) and f(H).
  - f(G) = (3+1) + 5 = 9
  - f(H) = (3+7) + 3 = 13

- Since f(G) < f(H), so it decides to go to node G.

Path- A → F → G

Mini-Max algorithm

- The Mini-Max algorithm is a decision-making algorithm used in artificial intelligence, particularly in game theory and computer games. It is designed to **minimize the possible loss in a worst-case scenario (hence “min”) and maximize the potential gain (therefore “max”).**
- In a two-player game, one player is the maximizer, aiming to maximize their score, while the other is the minimizer, aiming to minimize the maximizer’s score. The algorithm operates by evaluating all possible moves for both players, predicting the opponent’s responses, and choosing the optimal move to ensure the best possible outcome

Working of Min-Max Process in AI

- The Min-Max algorithm is a decision-making process used in artificial intelligence for two-player games. It involves two players: the maximizer and the minimizer, each aiming to optimize their own outcomes.
- **Players Involved**
- **Maximizing Player (Max):**
- Aims to maximize their score or utility value.
- Chooses the move that leads to the highest possible utility value, assuming the opponent will play optimally.

Working of Min-Max Process in AI

- **Minimizing Player (Min):**
- Aims to minimize the maximizer's score or utility value.
- Selects the move that results in the lowest possible utility value for the maximizer, assuming the opponent will play optimally.
- The interplay between these two players is central to the Min-Max algorithm, as each player attempts to outthink and counter the other's strategies.

Step-by-Step involved in the Mini-Max Algorithm

- The Min-Max algorithm involves several key steps, executed recursively until the optimal move is determined. Here is a step-by-step breakdown:
- **Step 1: Generate the Game Tree**
- **Objective**: Create a tree structure representing all possible moves from the current game state.
- **Details**: Each node represents a game state, and each edge represents a possible move.

Step-by-Step involved in the Mini-Max Algorithm

- **Step 2: Evaluate Terminal States**
- **Objective**: Assign utility values to the terminal nodes of the game tree.
- **Details**: These values represent the outcome of the game (win, lose, or draw).

Step-by-Step involved in the Mini-Max Algorithm

- **Step 3: Propagate Utility Values Upwards**
- **Objective**: Starting from the terminal nodes, propagate the utility values upwards through the tree.
- **Details**: For each non-terminal node:

If it's the maximizing player's turn, select the maximum value from the child nodes.

If it's the minimizing player's turn, select the minimum value from the child nodes.

Step-by-Step involved in the Mini-Max Algorithm

- **Step 4: Select Optimal Move**
- **Objective**: At the root of the game tree, the maximizing player selects the move that leads to the highest utility value.

Min-Max Formula

- **Maximizing Player's Turn:**

$Max(s)=max_{a \in A(s)} Min(Result(s,a))$

Here:

Max(s)is the maximum value the maximizing player can achieve from state s.

A(s) is the set of all possible actions from state s.

Result(*s*,*a*) is the resulting state from taking action a in state s.

Min(Result(*s*,*a*)) is the value for the minimizing player from the resulting state.

- **Minimizing Player's Turn:**

$Min(s)=min_{a \in A(s)} Max(Result(s,a))$

Here:

Min(s)is the minimum value the minimizing player can achieve from state s.

Min-Max Formula

- **Terminal States**
- For terminal states, the utility value is directly assigned:

$$\text{Utility}(s) = \begin{cases} 1 & \text{if the maximizing player wins from state } s \\ 0 & \text{if the game is a draw from state } s \\ -1 & \text{if the minimizing player wins from state } s \end{cases}$$

Example of Min-Max in Action

Consider a simplified version of a game where each player can choose between two moves at each turn. Here's a basic game tree:

```
        Max
       /   \
     Min   Min
    / \    /  \
  +1  -1  0   +1
```

- At the leaf nodes, the utility values are +1, -1, 0, and +1.
- The minimizing player will choose the minimum values from the child nodes: -1 (left subtree) and 0 (right subtree).
- The maximizing player will then choose the maximum value between -1 and 0, which is 0.

Thus, the optimal move for the maximizing player, considering optimal play by the minimizer, leads to a utility value of 0.

Alpha-Beta Pruning

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.

Alpha-Beta Pruning

- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:
  - ✓ **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is **-∞**.
  - ✓ **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is **+∞**.

Alpha-Beta Pruning

$$\min_G \max_D V(D,G)$$

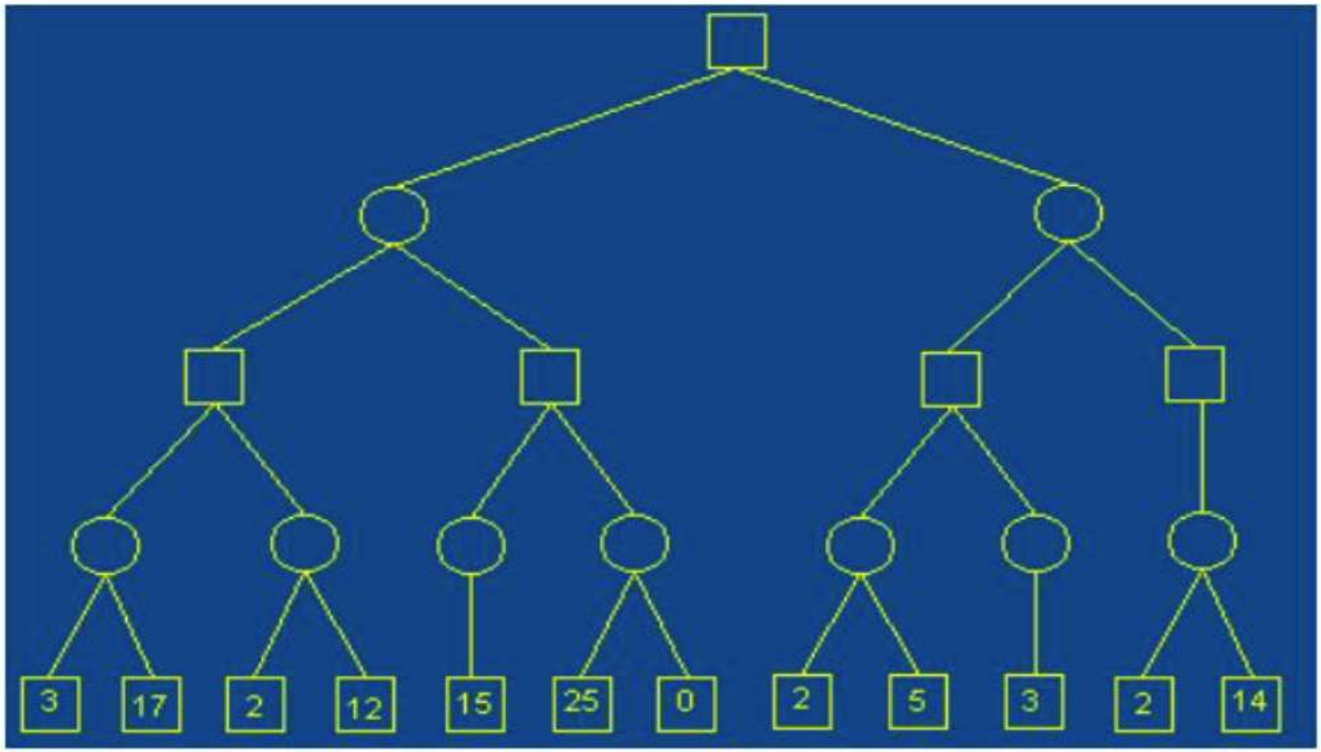
- It is formulated as a **minimax game**, where:
  - The Discriminator is trying to maximize its reward $\boldsymbol{V(D,G)}$
  - The Generator is trying to minimize Discriminator's reward (or maximize its loss)

$$V(D,G) = \mathbb{E}_{x\sim p(x)}[\log D(x)] + \mathbb{E}_{z\sim q(z)}[\log(1 - D(G(z)))]$$

- The Nash equilibrium of this particular game is achieved at:
  - $P_{data}(x) = P_{gen}(x) \ \forall x$
  - $\mathrm{D}(x) = \frac{1}{2} \ \forall x$

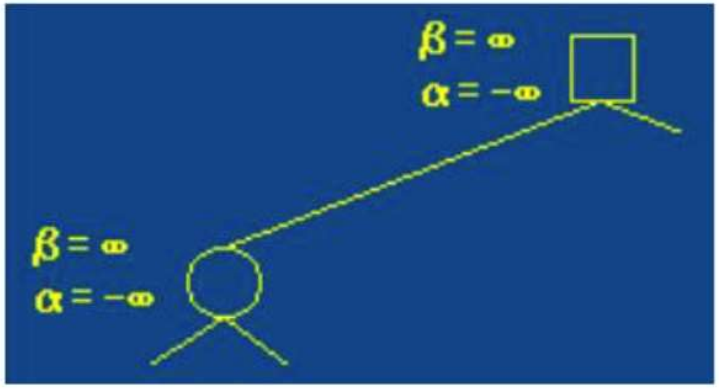
https://www.youtube.com/watch?v=OP-lkZo6fDY

Working of Alpha-Beta Pruning Example

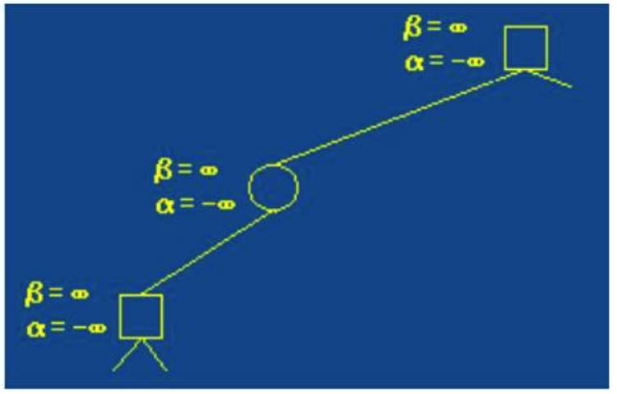


Continued...

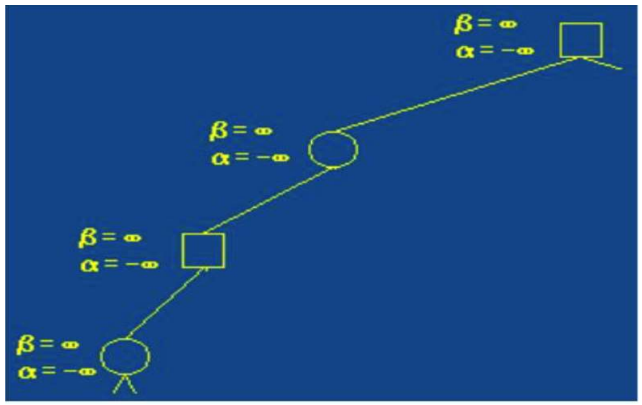
Step-1



Step-2

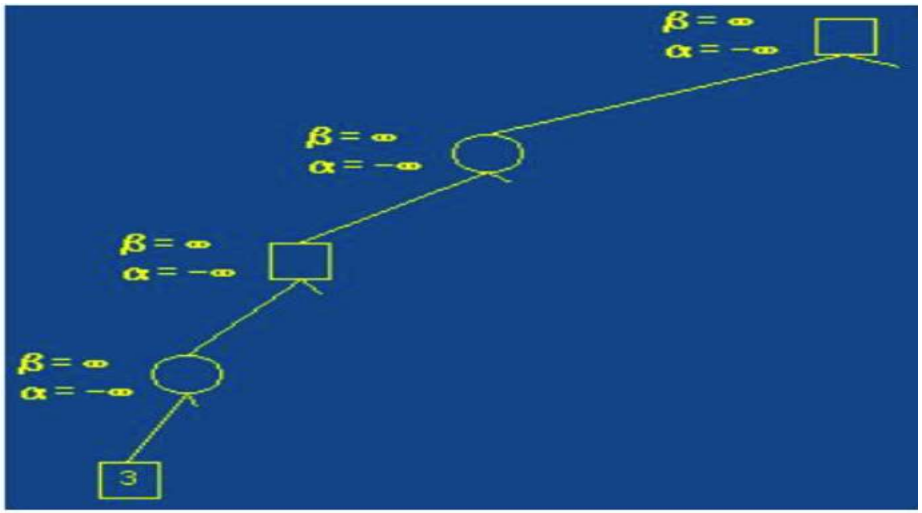


Step-3

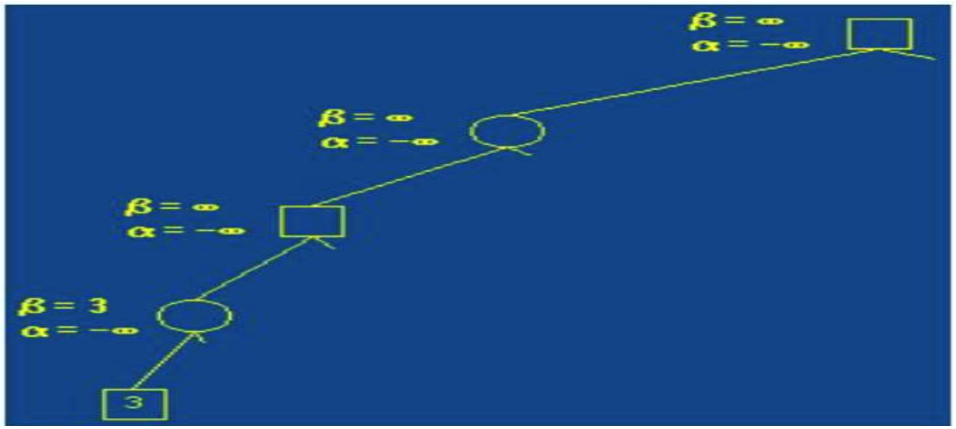


Continued...

Step-4

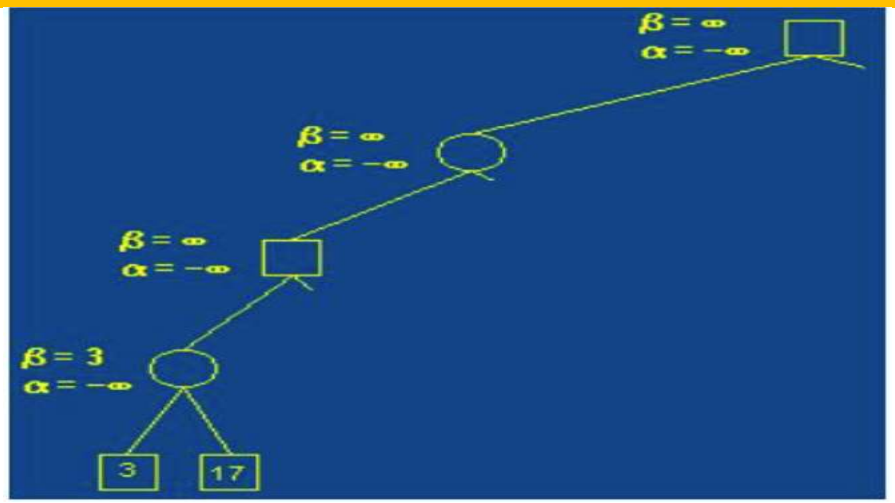


Step-5

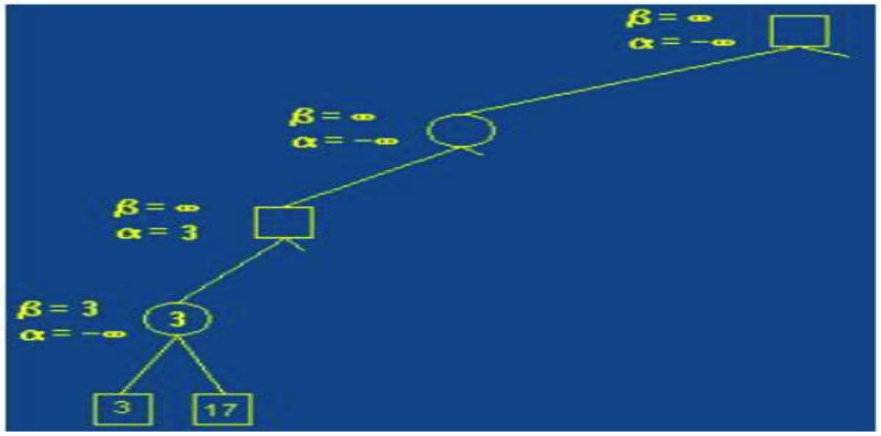


Continued...

Step-6

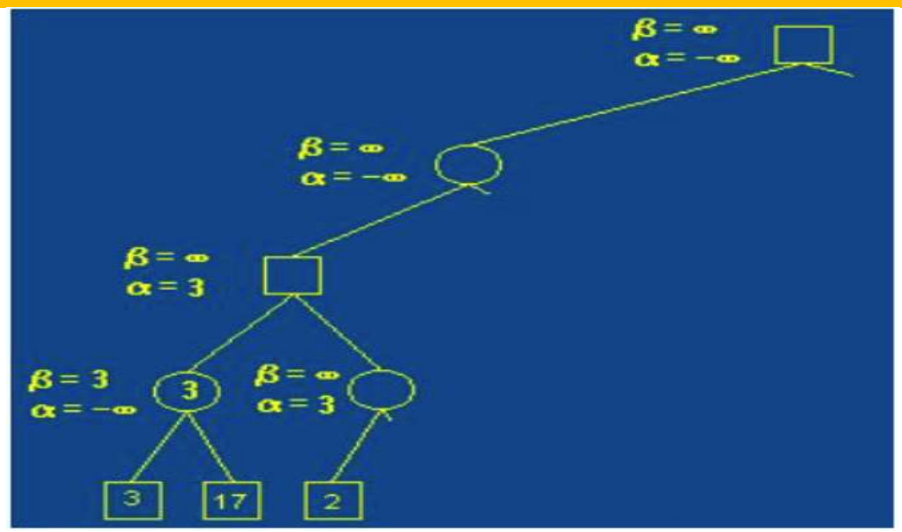


Step-7

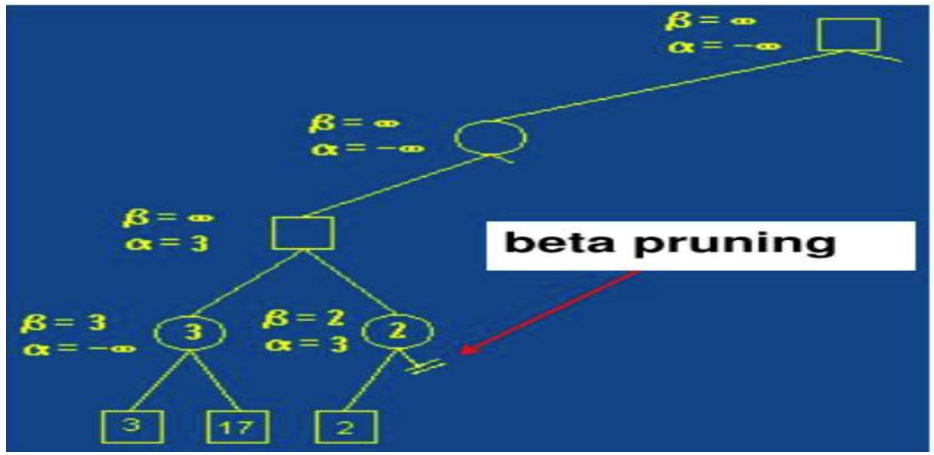


Continued...

Step-8

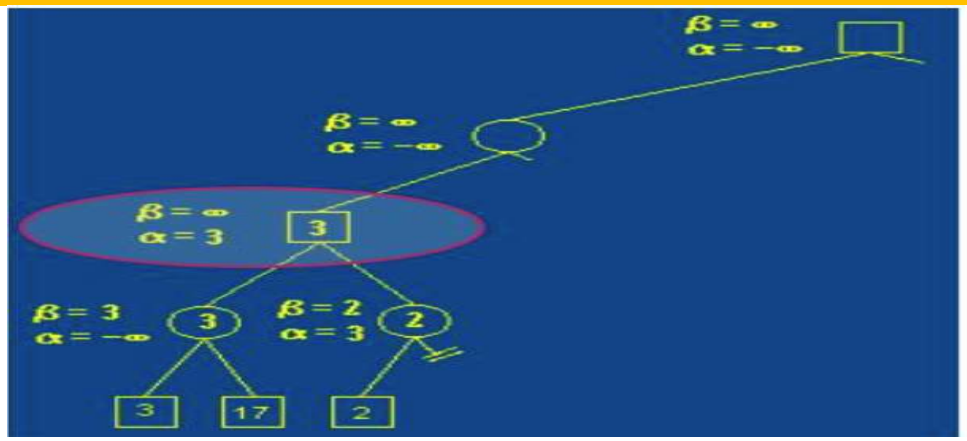


Step-9

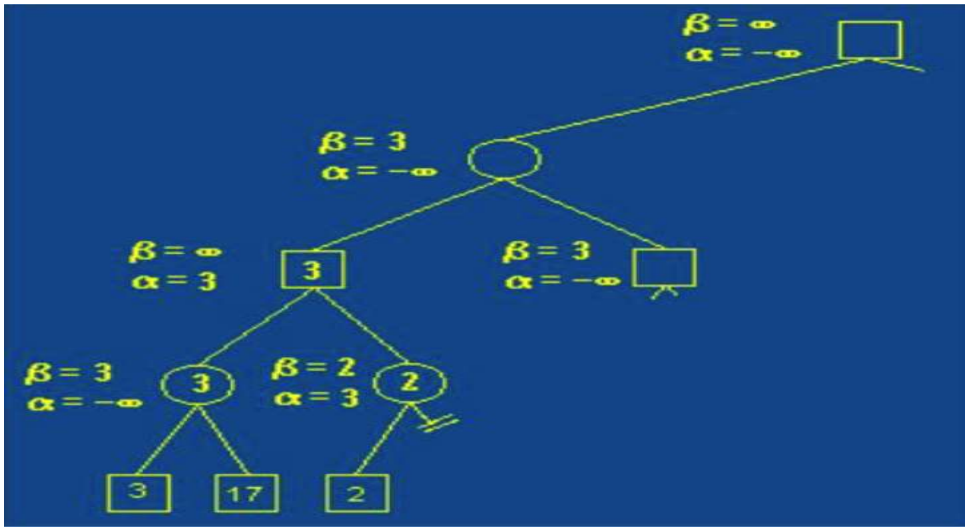


Continued...

Step-10

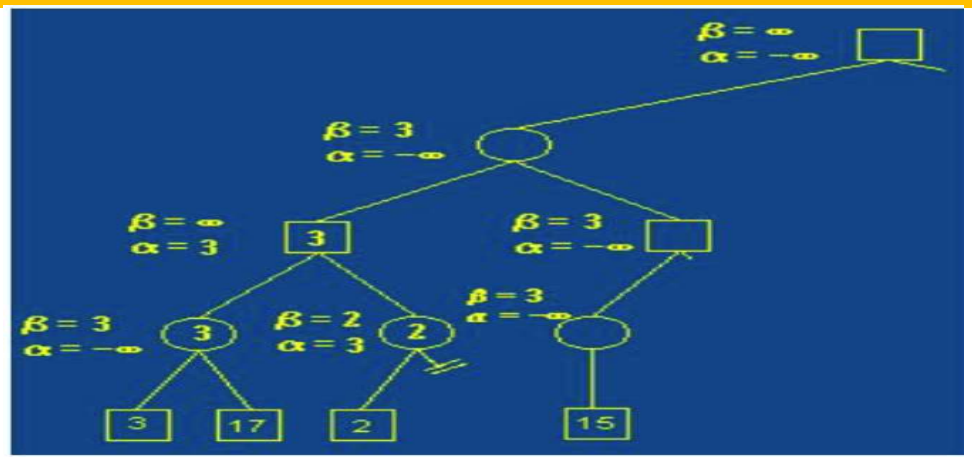


Step-12

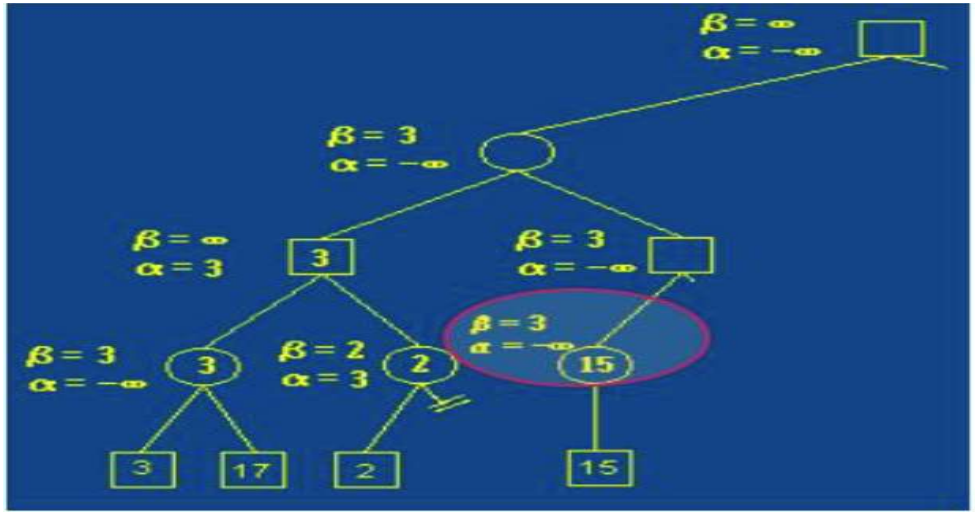


Continued...

Step-13

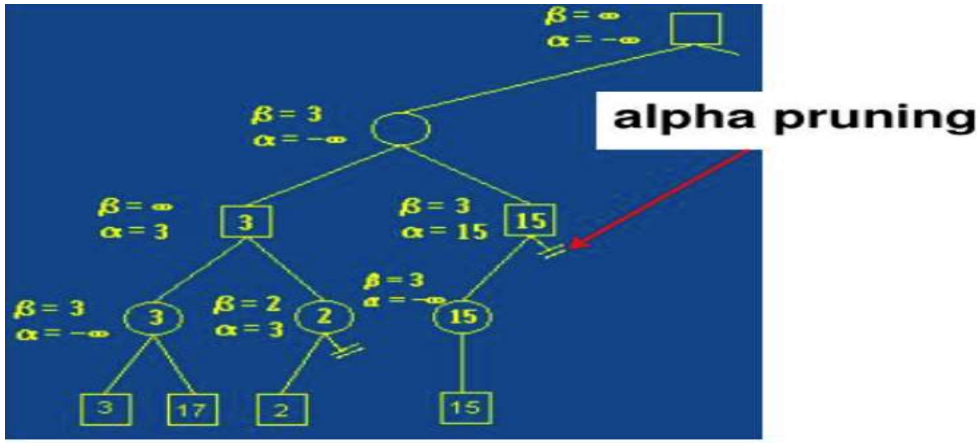


Step-14

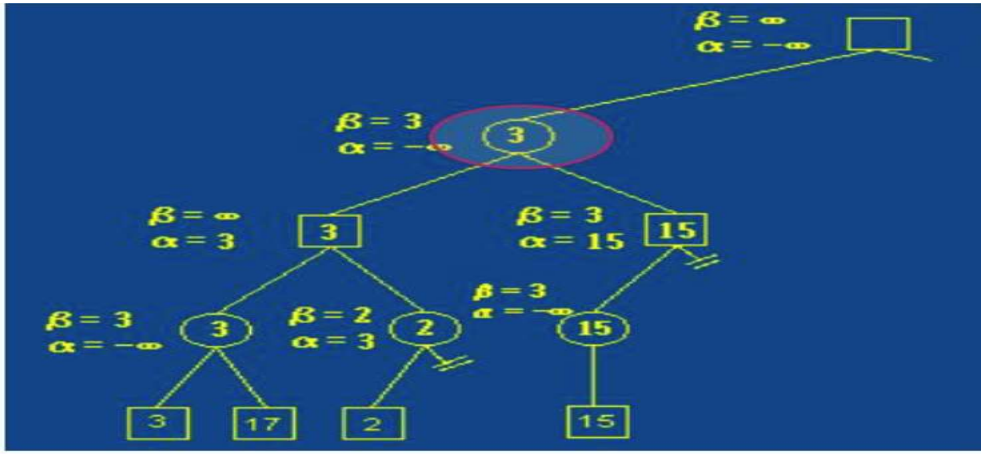


Continued...

Step-15

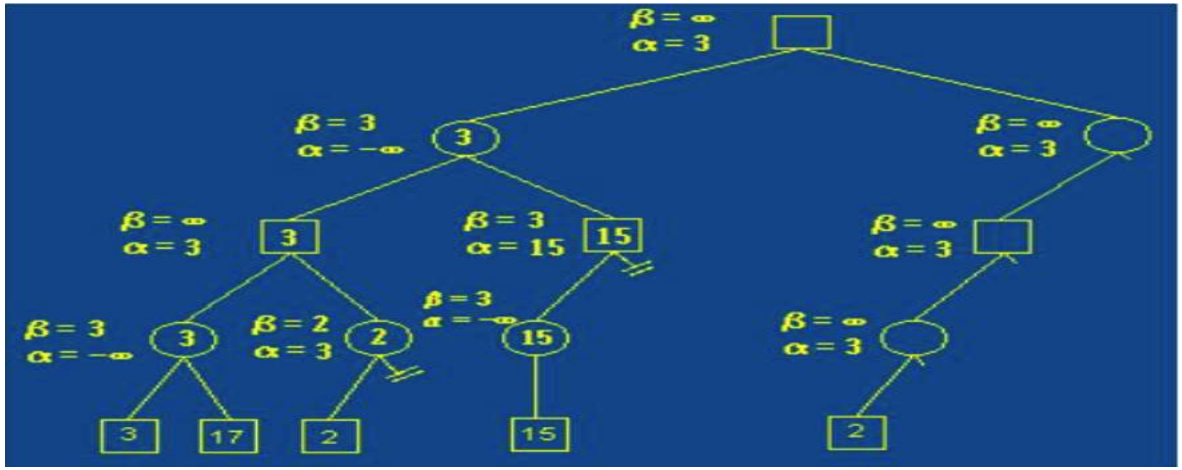


Step-16

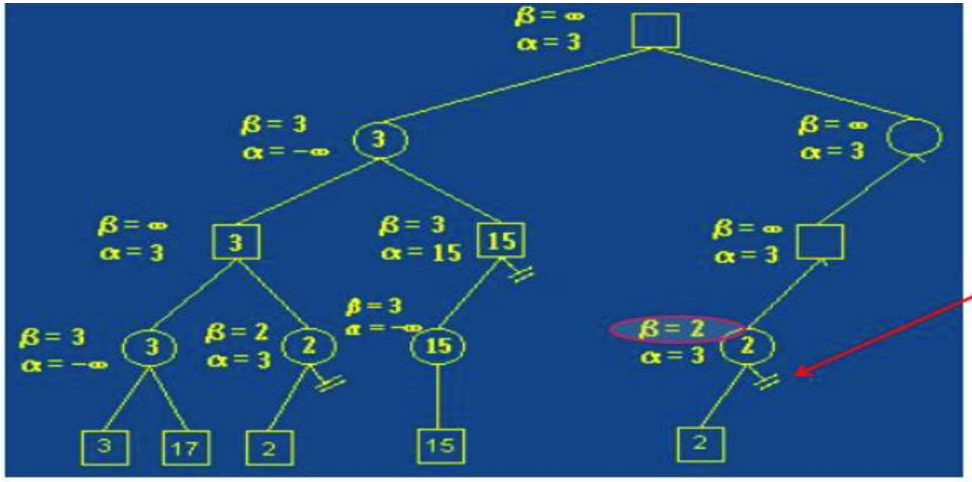


Continued...

Step-17



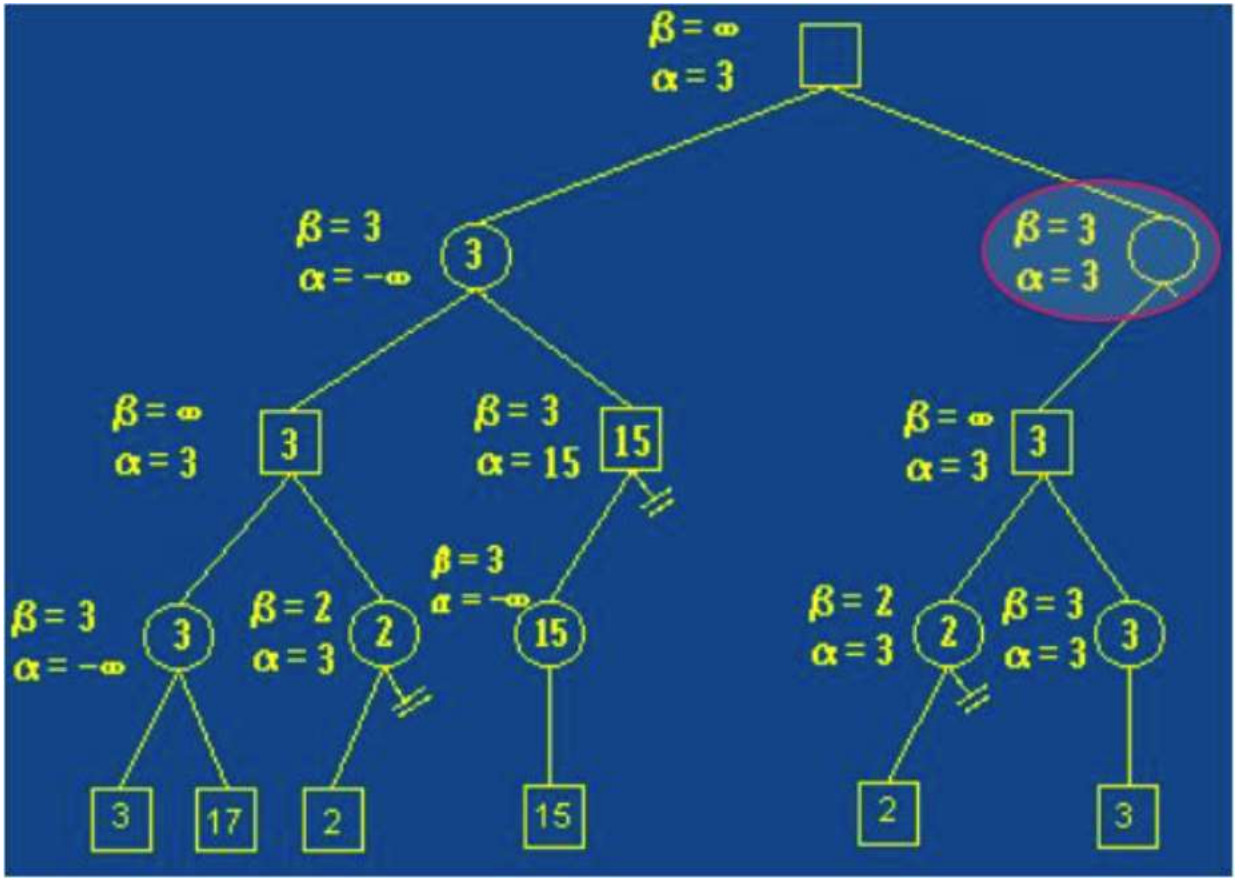
Step-18



beta pruning

Continued...

Final Step



Constraint Satisfaction Problem

Outline

- CSP examples
- Backtracking search for CSPs
- Problem structure and problem decomposition
- Local search for CSPs

Constraint Satisfaction Problem

A **constraint satisfaction problem (CSP)** is a problem that requires its solution within some limitations or conditions also known as constraints. It consists of the following:

- A finite set of **variables** which stores the solution **(V = {V1, V2, V3,....., Vn})**
- It is a set of **domains** where the variables reside. There is a specific domain for each variable.(D = {D1, D2, D3,.....,Dn})
- A finite set of **constraints** (C = {C1, C2, C3,....., Cn})

These are the three main elements of a constraint satisfaction technique. The constraint value consists of a pair of **{scope, rel}**. The **scope** is a tuple of variables which participate in the constraint and **rel** is a relation which includes a list of values which the variables can take to satisfy the constraints of the problem.

Constraint Satisfaction Problem

- Start State: the empty assignment i.e all variables are unassigned.
- Goal State: all the variables are assigned values which satisfy constraints.
- Operator: assigns value to any unassigned variable, provided that it does not conflict with previously assigned variables.

Ref: Prof Mausam, IIT Delhi, NPTEL Lectures, https://www.youtube.com/watch?v=RD9tUlJRLjA

Constraint satisfaction problems (CSPs)

Standard search problem:

state is a "black box"—any old data structure
that supports goal test, eval, successor

CSP:

state is defined by variables X_i with values from domain D_i

goal test is a set of constraints specifying
allowable combinations of values for subsets of variables

Simple example of a **formal representation language**

Allows useful **general-purpose** algorithms with more power
than standard search algorithms

W.A. N.T. Q. NSW. V. SA. T

Example: Map-Coloring

Variables WA, NT, Q, NSW, V, SA, T

Domains $D_i = \{red, green, blue\}$

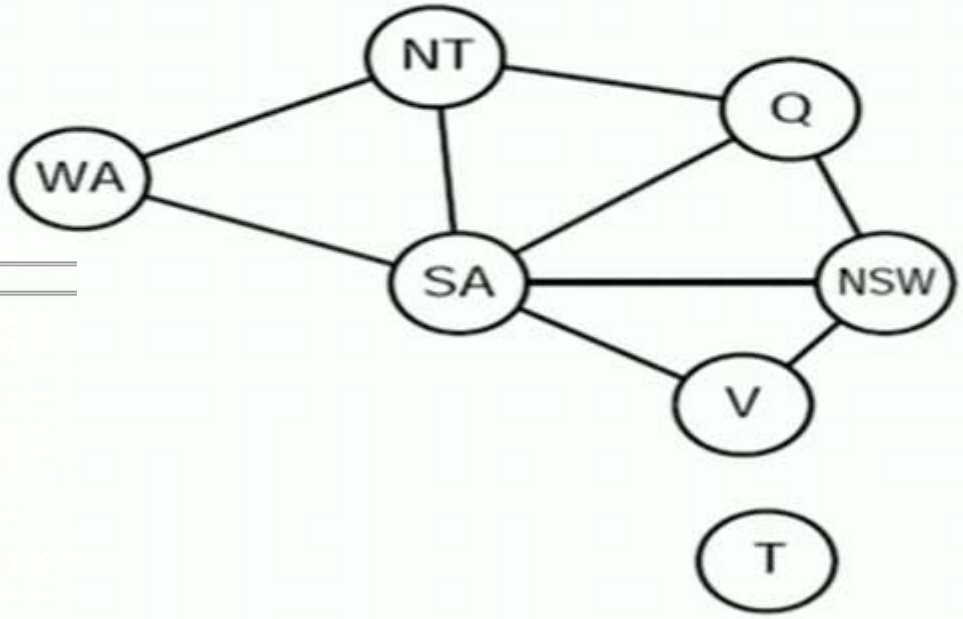
Constraints: adjacent regions must have different colors

Example: Map-Coloring contd.

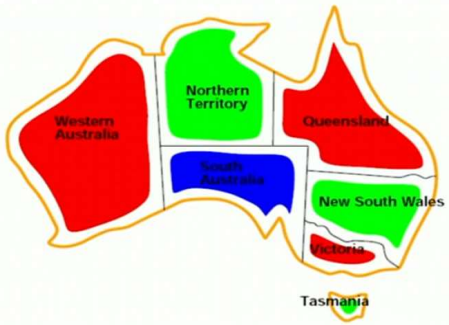
Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints



Example: Map-Coloring contd.



Varieties of CSPs

Discrete variables

- finite domains; size $d \Rightarrow O(d^n)$ complete assignments
  - ◇ e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)
- infinite domains (integers, strings, etc.)
  - ◇ e.g., job scheduling, variables are start/end days for each job
  - ◇ need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
  - ◇ linear constraints solvable, nonlinear undecidable

Continuous variables

- ◇ e.g., start/end times for Hubble Telescope observations
- ◇ linear constraints solvable in poly time by LP methods

Varieties of constraints

Unary constraints involve a single variable,
e.g., $SA \neq green$

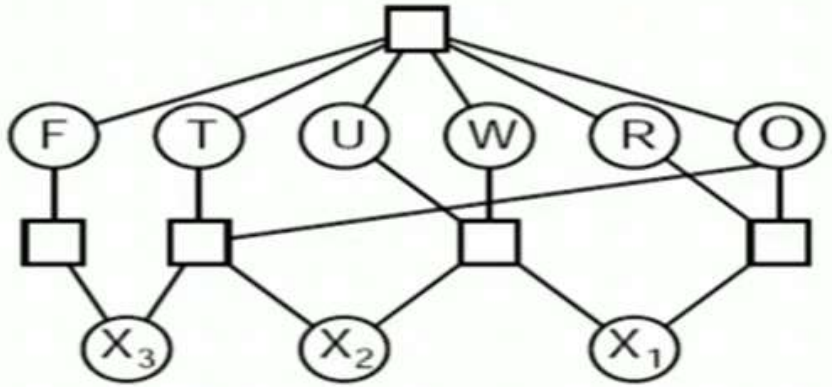
Binary constraints involve pairs of variables,
e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables,
e.g., cryptarithmetic column constraints

Preferences (soft constraints), e.g., red is better than $green$
often representable by a cost for each variable assignment
$\rightarrow$ constrained optimization problems

Example: Cryptarithmetic

$$\begin{array}{cccc} & T & W & O \\ + & T & W & O \\ \hline F & O & U & R \end{array}$$



Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
Domains: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Constraints
alldiff(F, T, U, W, R, O)
$O + O = R + 10 \cdot X_1$, etc.

Real-world CSPs

Assignment problems
    e.g., who teaches what class

Timetabling problems
    e.g., which class is offered when and where?

Hardware configuration

Spreadsheets

Transportation scheduling

Factory scheduling

Floorplanning

Standard search formulation (incremental)

Let's start with the straightforward, dumb approach, then fix it

States are defined by the values assigned so far

- Initial state: the empty assignment, $\{\}$
- Successor function: assign a value to an unassigned variable that does not conflict with current assignment.
  $\Rightarrow$ fail if no legal assignments (not fixable!)
- Goal test: the current assignment is complete

1) This is the same for all CSPs!
2) Every solution appears at depth n with n variables
   $\Rightarrow$ use depth-first search
3) Path is irrelevant, so can also use complete-state formulation
4) $b = (n - \ell)d$ at depth ℓ, hence $n!d^n$ leaves!!!!

CSP Algorithm

Until a complete solution is found or all paths have a lead to dead ends

{

- Select an unexpanded node of the search graph.
- Apply the constraint inference rules to the selected node to generate all possible new constraints.
- If the set of constraints contain a contradiction then report that this path is a dead end.
- If the set of constraints describe a complete solution, then report success.
- If neither a contradiction nor a complete solution has been found, then apply the problem space rules to generate new partial solutions that are consistent with the current set of constraints. Insert these partial solutions into the search graph

}

STOP

Backtracking search

Variable assignments are commutative, i.e.,

$[WA = red$ then $NT = green]$ same as $[NT = green$ then $WA = red]$

Only need to consider assignments to a single variable at each node

$\Rightarrow \quad b = d$ and there are d^n leaves

Depth-first search for CSPs with single-variable assignments is called backtracking search

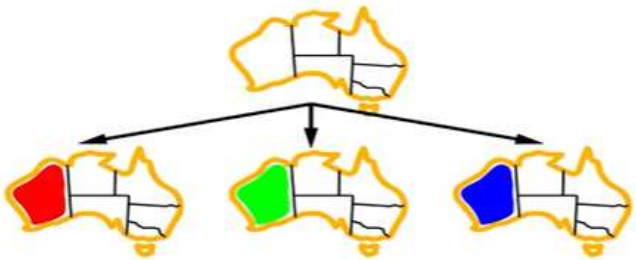
Backtracking search is the basic uninformed algorithm for CSPs

Can solve n-queens for $n \approx 25$

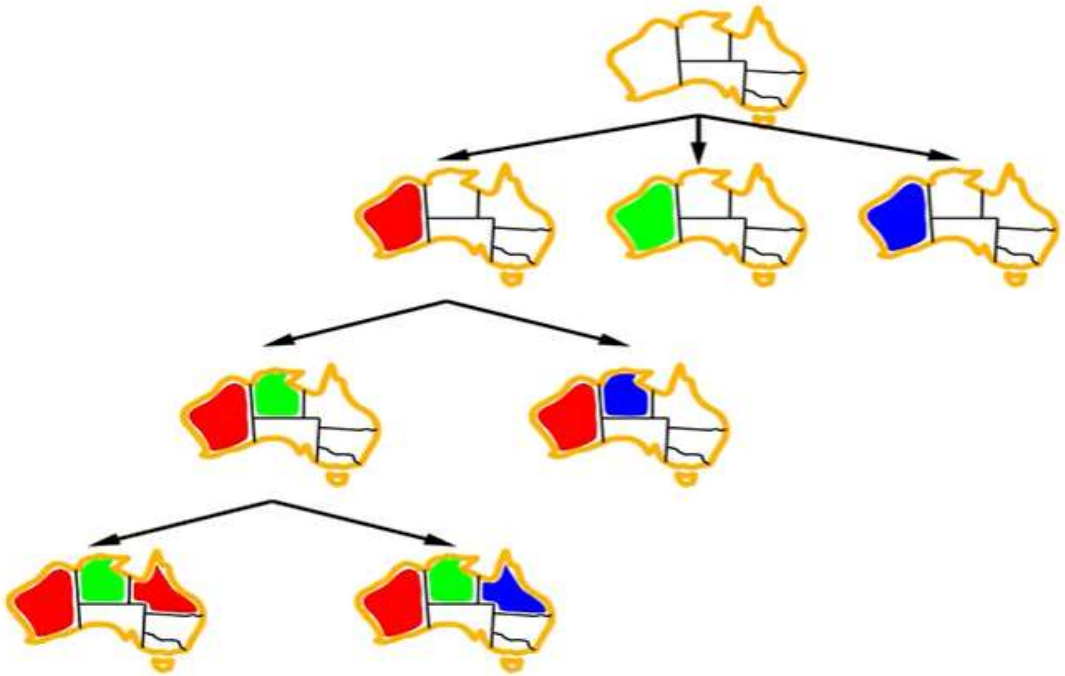
```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

Backtracking example

Backtracking example

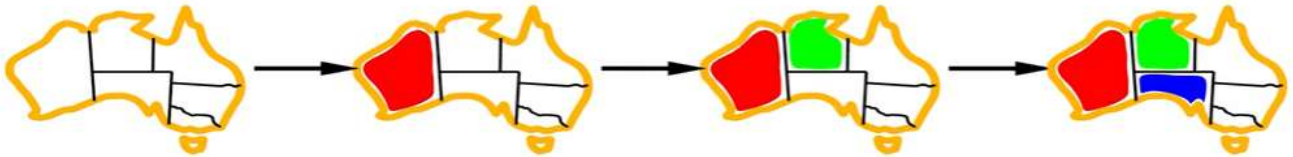


Backtracking example



Minimum remaining values

Minimum remaining values (MRV):
choose the variable with the fewest legal values

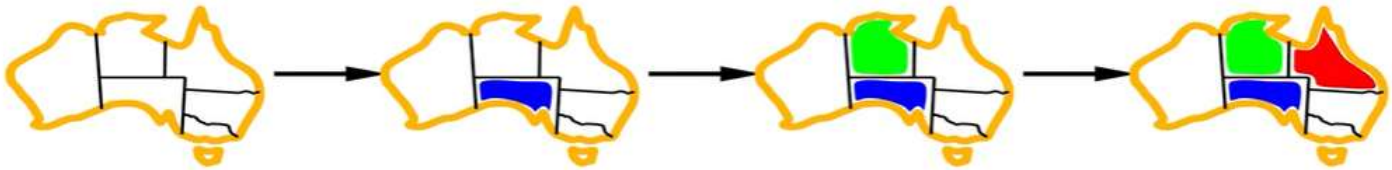


Degree heuristic

Tie-breaker among MRV variables

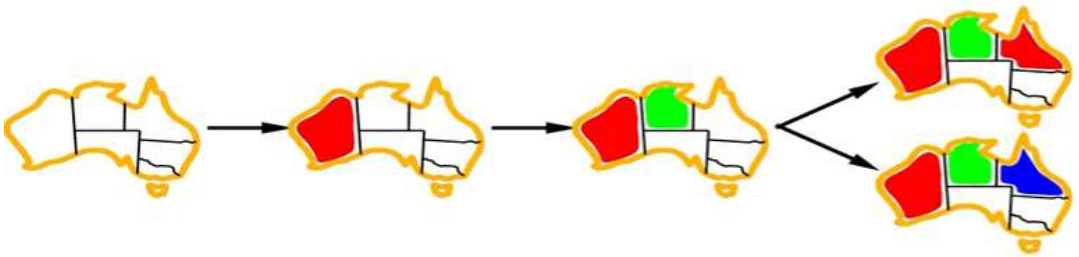
Degree heuristic:

choose the variable with the most constraints on remaining variables



Least constraining value

Given a variable, choose the least constraining value:
the one that rules out the fewest values in the remaining variables



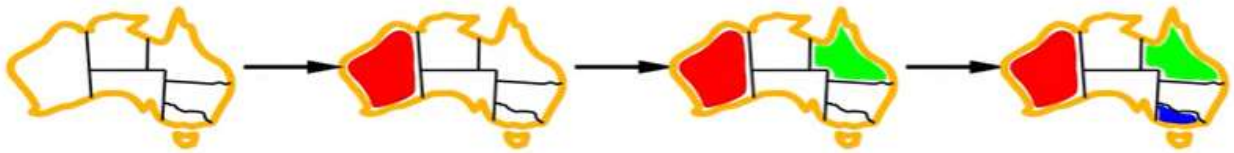
Allows 1 value for SA

Allows 0 values for SA

Combining these heuristics makes 1000 queens feasible

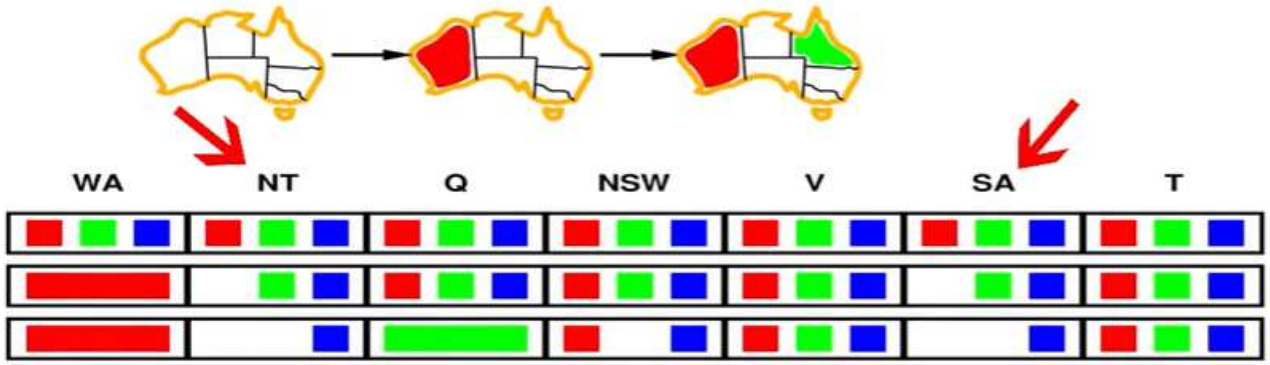
Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



Constraint propagation

Forward checking propagates information from assigned to unassigned v
ables, but doesn't provide early detection for all failures:



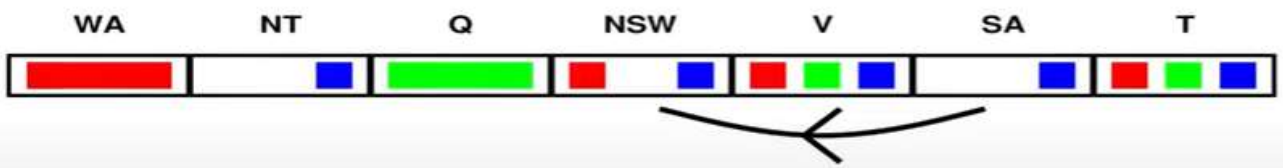
NT and SA cannot both be blue!

Constraint propagation repeatedly enforces constraints locally

Arc consistency

Simplest form of propagation makes each arc consistent

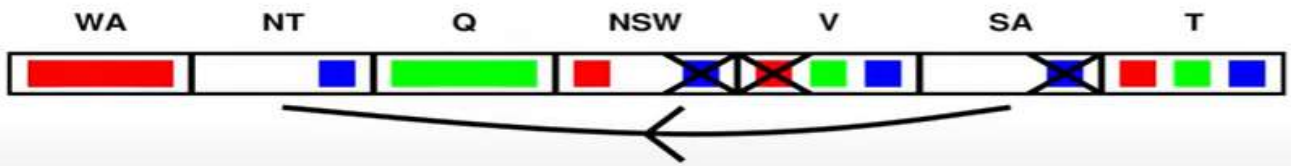
$X \rightarrow Y$ is consistent iff
for **every** value x of X there is **some** allowed y



Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff
for **every** value x of X there is **some** allowed y



If X loses a value, neighbors of X need to be rechecked

Arc consistency detects failure earlier than forward checking

Arc consistency algorithm

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
  local variables: queue, a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← REMOVE-FIRST(queue)
    if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
      for each X_k in NEIGHBORS[X_i] do
        add (X_k, X_i) to queue
```

```
function REMOVE-INCONSISTENT-VALUES( X_i, X_j) returns true iff succeeds
  removed ← false
  for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j
      then delete x from DOMAIN[X_i]; removed ← true
  return removed
```