Mini-Max Algorithm in Artificial Intelligence

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game, one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Pseudo-code for MinMax Algorithm:

```
function minimax(node, depth, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
maxEva= -infinity
 for each child of node do
 eva= minimax(child, depth-1, false)
maxEva= max(maxEva,eva)        //gives Maximum of the values
return maxEva

else                        // for Minimizer player
 minEva= +infinity
 for each child of node do
```

```
15. eva= minimax(child, depth-1, true)
16. minEva= min(minEva, eva)        //gives minimum of the values
17. return minEva
```

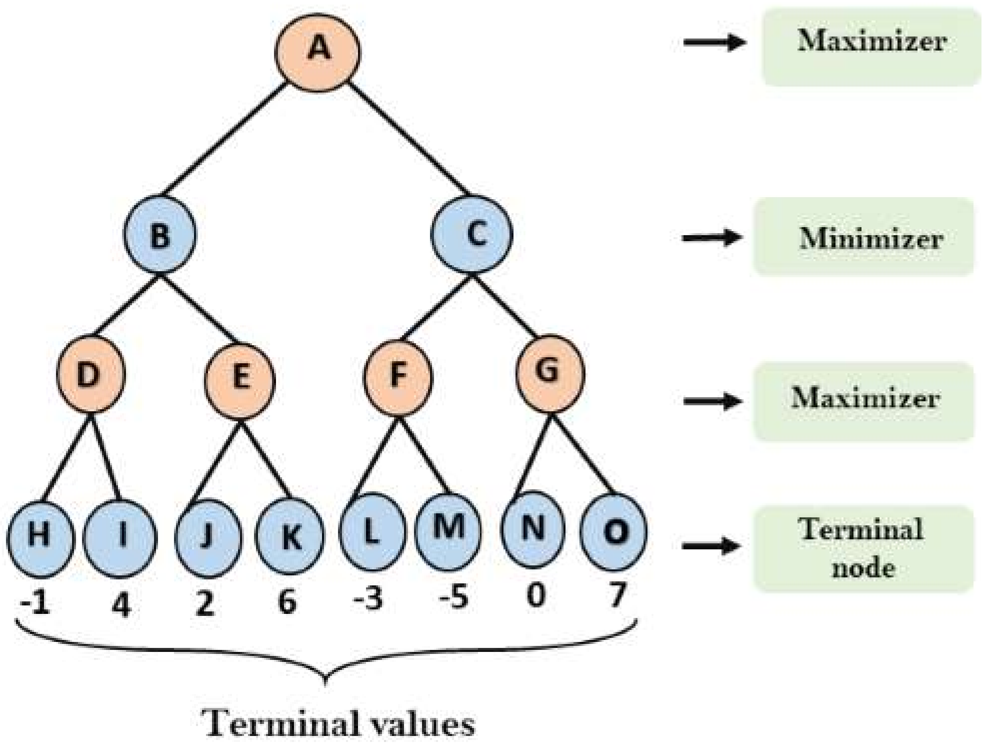
Initial call:

Minimax(node, 3, true)

Working of Min-Max Algorithm:

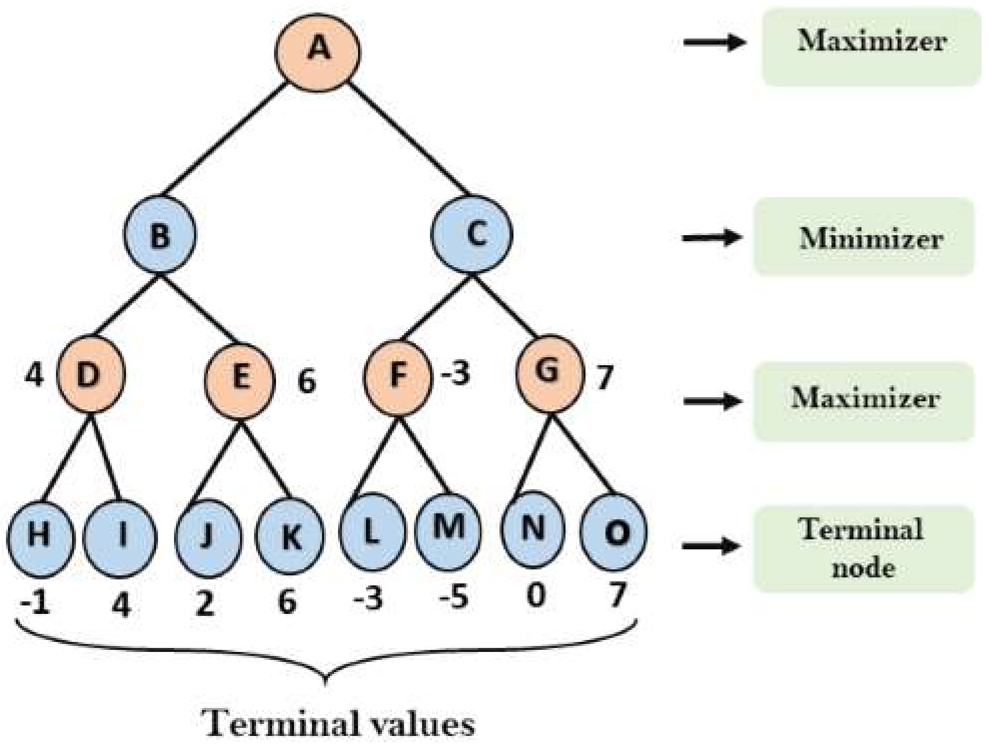
- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

Step-1: In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.



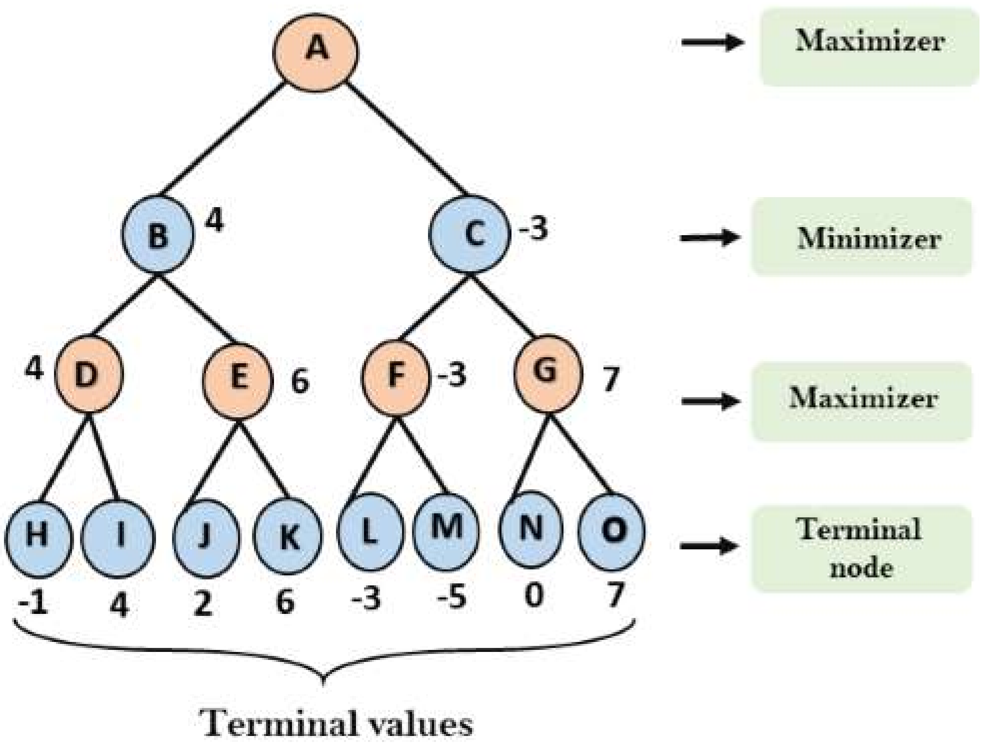
Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

- For node D max(-1,- -∞) => max(-1,4)= 4
- For Node E max(2, -∞) => max(2, 6)= 6
- For Node F max(-3, -∞) => max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7



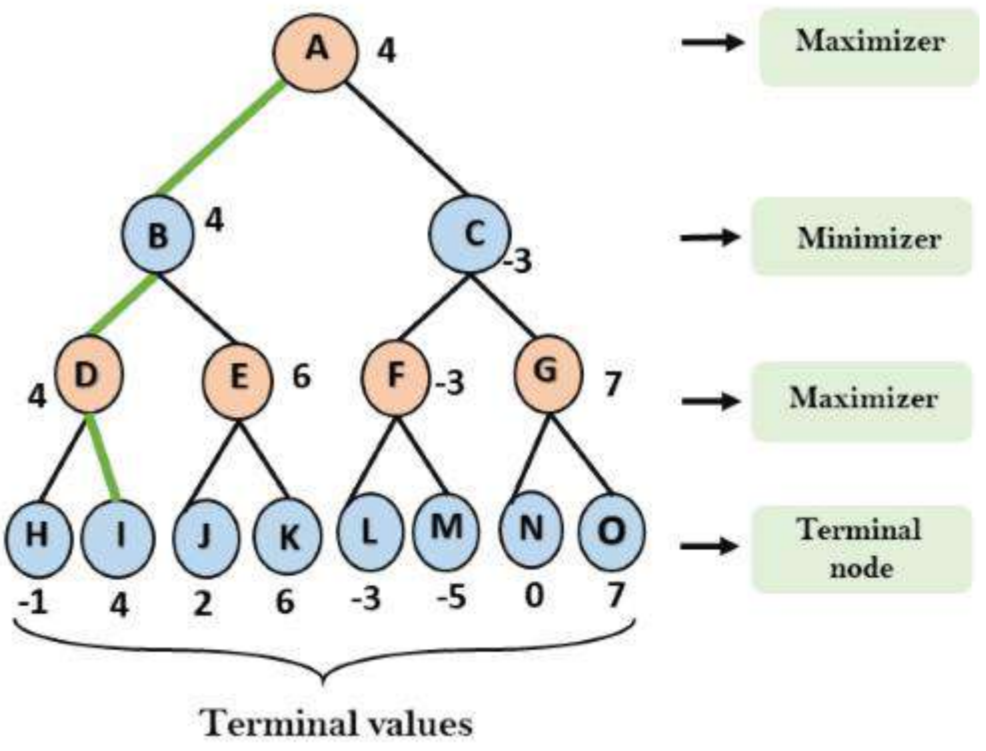
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3



Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

- For node A max(4, -3)= 4



That was the complete workflow of the minimax two player game.

Properties of Mini-Max algorithm:

- **Complete-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal-** Min-Max algorithm is optimal if both opponents are playing optimally.
- **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is **$O(b^m)$**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- **Space Complexity-** Space complexity of Mini-max algorithm is also similar to DFS which is **O(bm)**.

Limitation of the minimax Algorithm:

The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from **alpha-beta pruning** which we have discussed in the next topic.

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 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:
  a. **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 **-∞**.
  b. **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 **+∞**.
- The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Note: To better understand this topic, kindly study the minimax algorithm.

Condition for Alpha-beta pruning:

The main condition which required for alpha-beta pruning is:

1. $\alpha >= \beta$

Key points about alpha-beta pruning:

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.

- We will only pass the alpha, beta values to the child nodes.

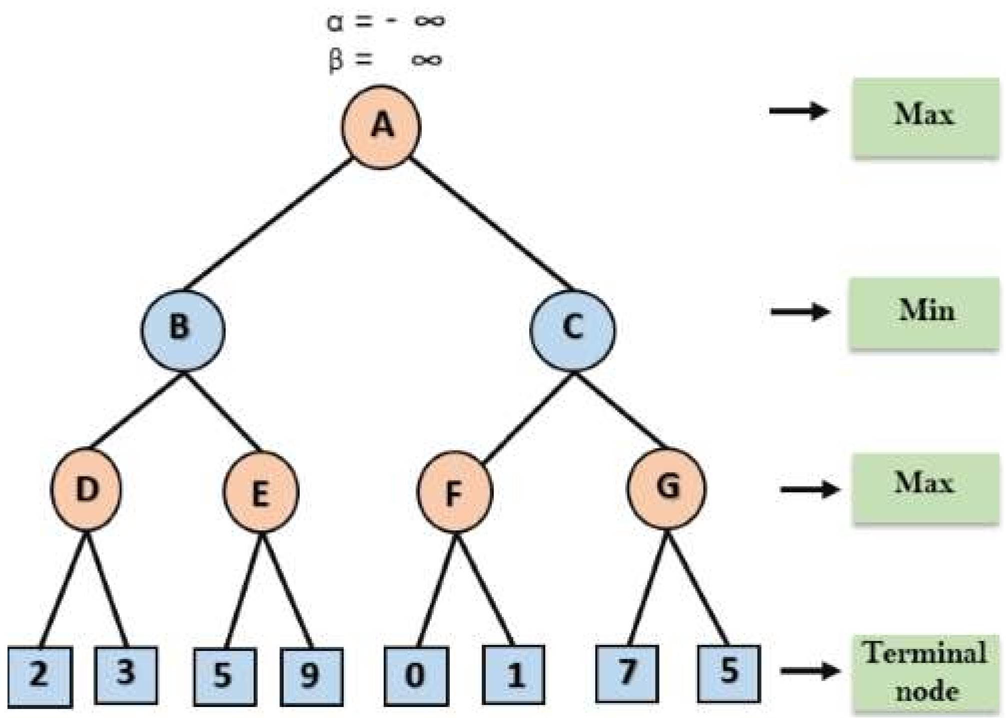
Pseudo-code for Alpha-beta Pruning:

```
function minimax(node, depth, alpha, beta, maximizingPlayer) is
if depth ==0 or node is a terminal node then
    return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
    maxEva= -infinity
    for each child of node do
        eva= minimax(child, depth-1, alpha, beta, False)
        maxEva= max(maxEva, eva)
        alpha= max(alpha, maxEva)
        if beta<=alpha
            break
    return maxEva

else                         // for Minimizer player
    minEva= +infinity
    for each child of node do
        eva= minimax(child, depth-1, alpha, beta, true)
        minEva= min(minEva, eva)
        beta= min(beta, eva)
        if beta<=alpha
            break
    return minEva
```

Working of Alpha-Beta Pruning:

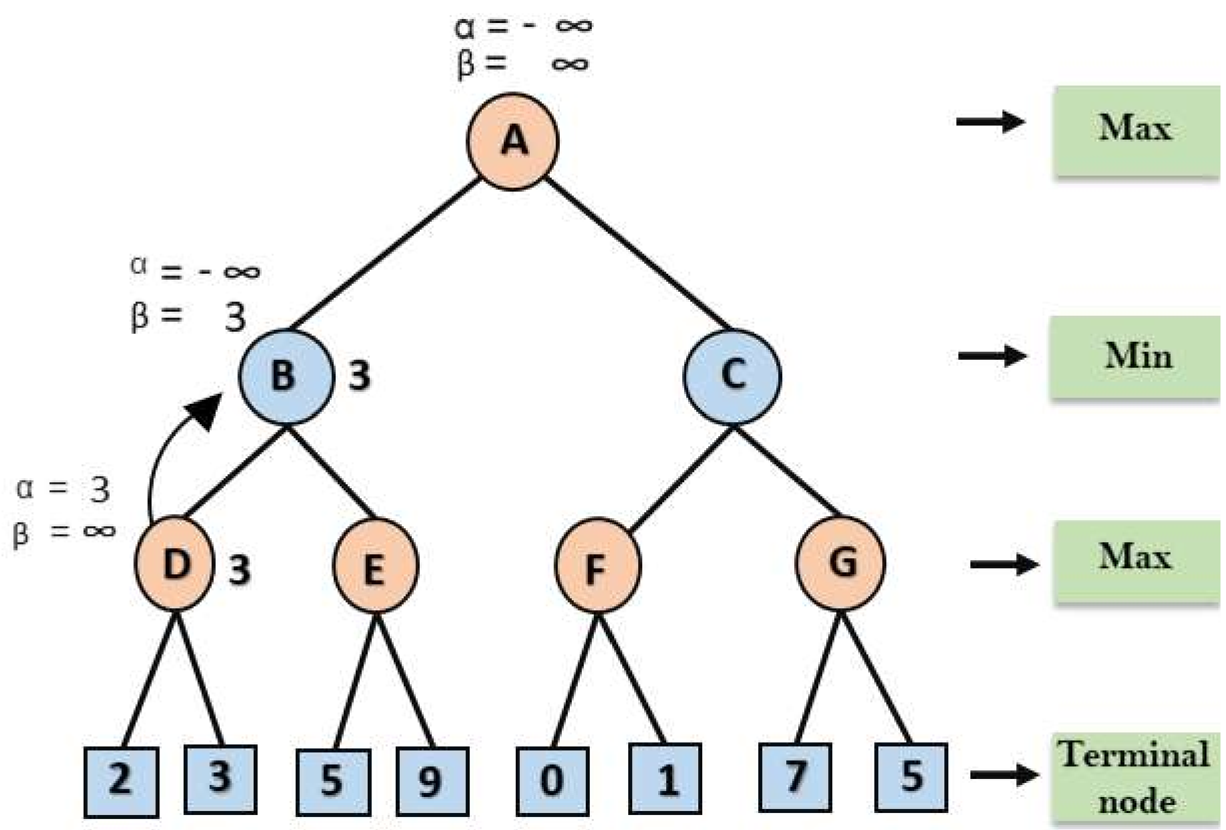
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

Step 1: At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.



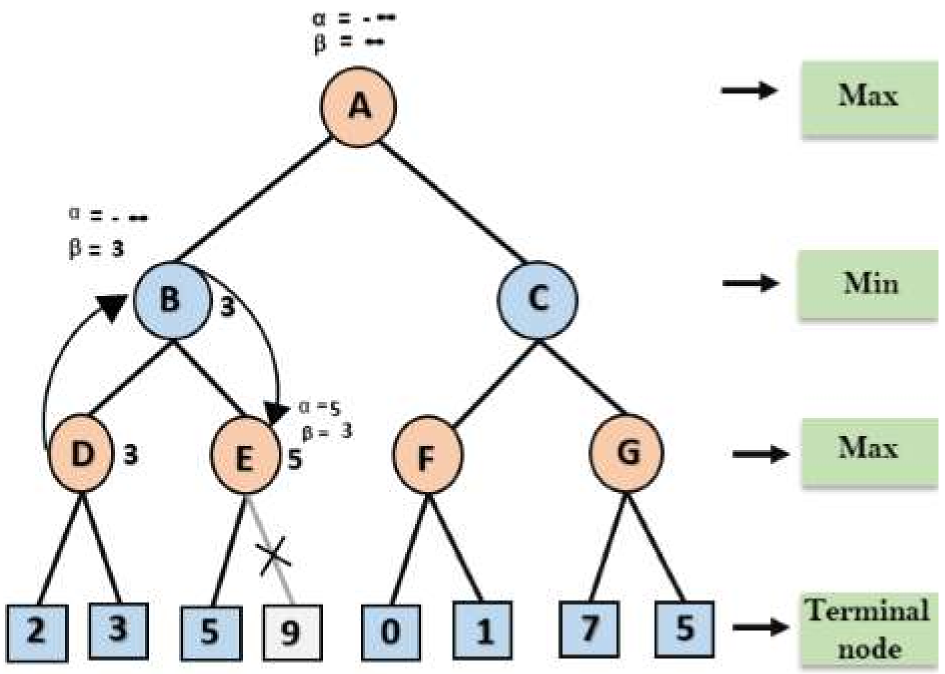
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.



In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

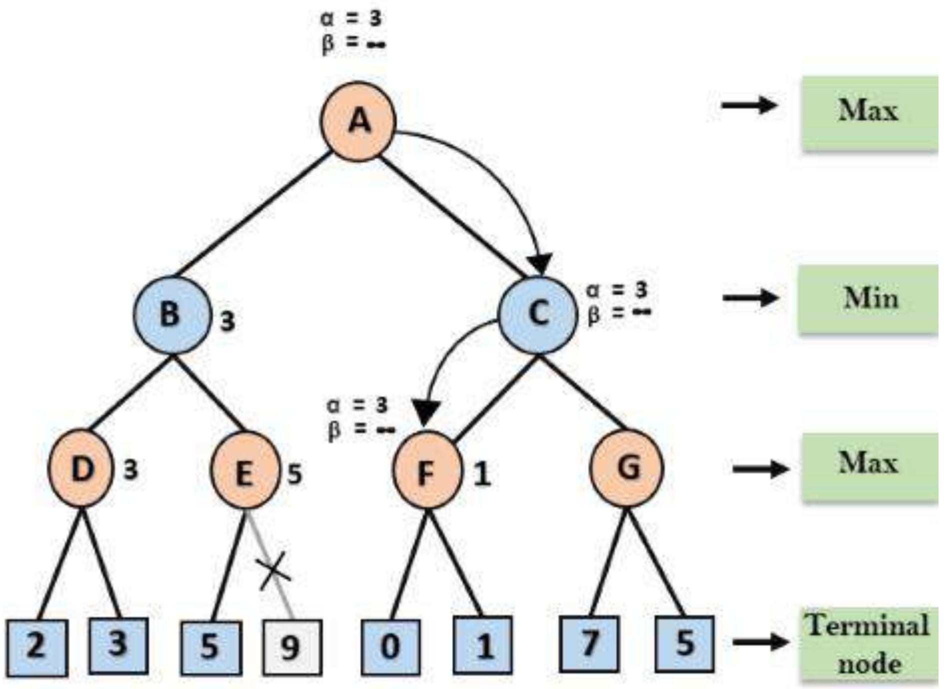
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



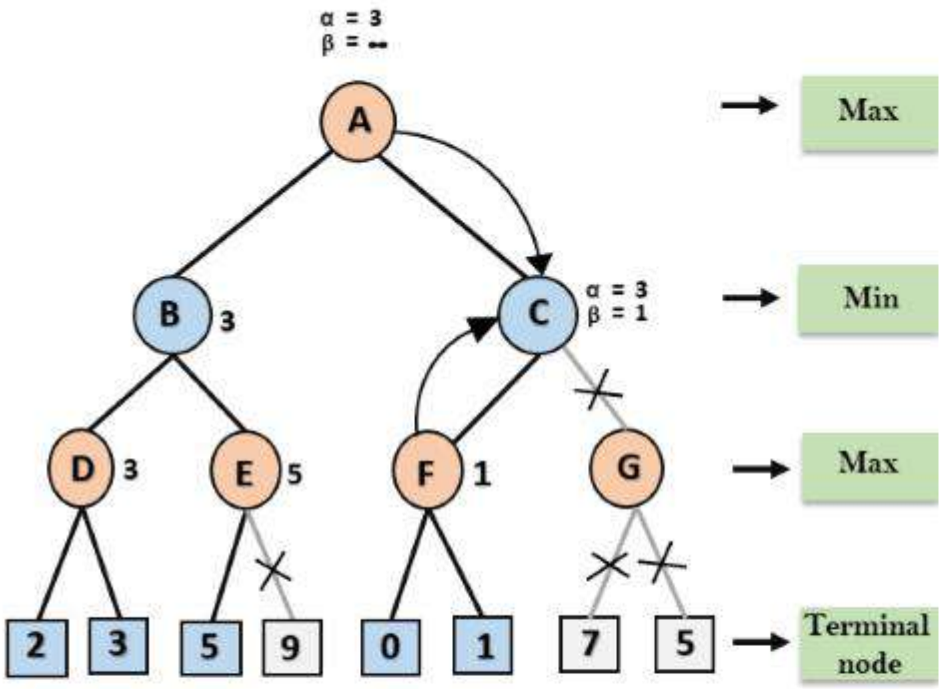
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

At node C, α=3 and β= +∞, and the same values will be passed on to node F.

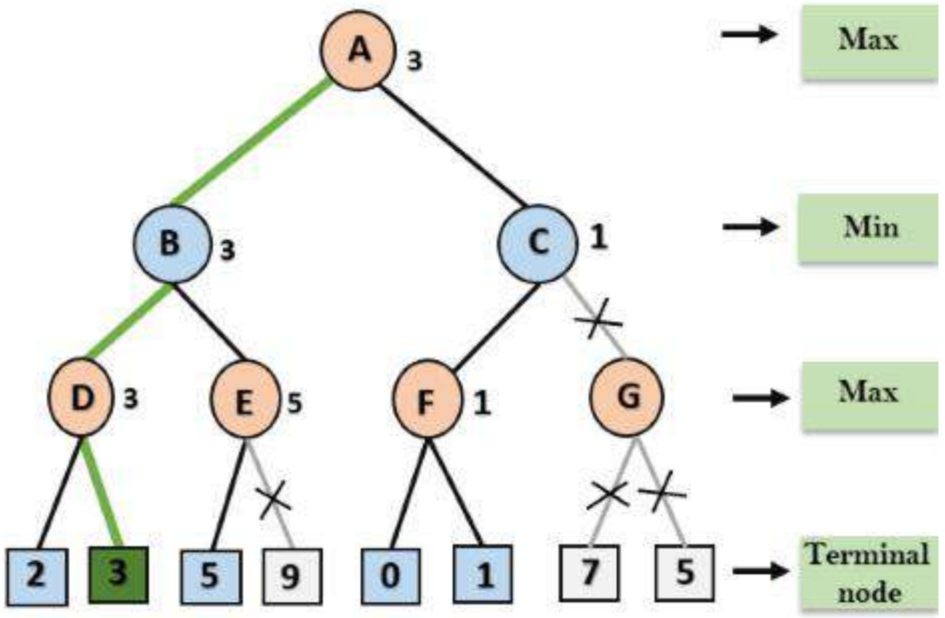
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



Step 7: Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



Step 8: C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



Move Ordering in Alpha-Beta pruning:

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.

It can be of two types:

- **Worst ordering:** In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is $O(b^m)$.
- **Ideal ordering:** The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is $O(b^{m/2})$.

Rules to find good ordering:

Following are some rules to find good ordering in alpha-beta pruning:

- Occur the best move from the shallowest node.
- Order the nodes in the tree such that the best nodes are checked first.
- Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.
- We can bookkeep the states, as there is a possibility that states may repeat.

Constraint Satisfaction Problems in Artificial Intelligence

We have encountered a wide variety of methods, including adversarial search and instant search, to address various issues. Every method for issue has a single purpose in mind: to locate a remedy that will enable that achievement of the objective. However there were no restrictions just on bots' capability to resolve issues as well as arrive at responses in adversarial search and local search, respectively.

These section examines the constraint optimization methodology, another form or real concern method. By its name, constraints fulfilment implies that such an issue must be solved while adhering to a set of restrictions or guidelines.

Whenever a problem is actually variables comply with stringent conditions of principles, it is said to have been addressed using the solving multi - objective method. Wow what a method results in a study sought to achieve of the intricacy and organization of both the issue.

Three factors affect restriction compliance, particularly regarding:

- It refers to a group of parameters, or X.
- D: The variables are contained within a collection several domain. Every variables has a distinct scope.
- C: It is a set of restrictions that the collection of parameters must abide by.

In constraint satisfaction, domains are the areas wherein parameters were located after the restrictions that are particular to the task. Those three components make up a constraint satisfaction technique in its entirety. The pair "scope, rel" makes up the number of something like the requirement. The scope is a tuple of variables that contribute to the restriction, as well as rel is indeed a relationship that contains a list of possible solutions for the parameters should assume in order to meet the restrictions of something like the issue.

Issues with Contains A certain amount Solved

For a constraint satisfaction problem (CSP), the following conditions must be met:

- States area
- fundamental idea while behind remedy.

The definition of a state in phase space involves giving values to any or all of the parameters, like as

X1 = v1, X2 = v2, etc.

There are 3 methods to economically beneficial to something like a parameter:

1. Consistent or Legal Assignment: A task is referred to as consistent or legal if it complies with all laws and regulations.
2. Complete Assignment: An assignment in which each variable has a number associated to it and that the CSP solution is continuous. One such task is referred to as a completed task.
3. A partial assignment is one that just gives some of the variables values. Projects of this nature are referred to as incomplete assignment.

Domain Categories within CSP

The parameters utilize one of the two types of domains listed below:

- Discrete Domain: This limitless area allows for the existence of a single state with numerous variables. For instance, every parameter may receive a endless number of beginning states.
- It is a finite domain with continous phases that really can describe just one area for just one particular variable. Another name for it is constant area.

Types of Constraints in CSP

Basically, there are three different categories of limitations in regard towards the parameters:

- Unary restrictions are the easiest kind of restrictions because they only limit the value of one variable.
- Binary resource limits: These restrictions connect two parameters. A value between x1 and x3 can be found in a variable named x2.
- Global Resource limits: This kind of restriction includes a unrestricted amount of variables.

The main kinds of restrictions are resolved using certain kinds of resolution methodologies:

- In linear programming, when every parameter carrying an integer value only occurs in linear equation, linear constraints are frequently utilised.
- Non-linear Constraints: With non-linear programming, when each variable (an integer value) exists in a non-linear form, several types of restrictions were utilised.

Note: The preferences restriction is a unique restriction that operates in the actual world.

Think of a Sudoku puzzle where some of the squares have initial fills of certain integers.

You must complete the empty squares with numbers between 1 and 9, making sure that no rows, columns, or blocks contains a recurring integer of any kind. This solving multi - objective issue is pretty elementary. A problem must be solved while taking certain limitations into consideration.

The integer range (1-9) that really can occupy the other spaces is referred to as a domain, while the empty spaces themselves were referred as variables. The values of the variables are drawn first from realm. Constraints are the rules that determine how a variable will select the scope.

Constraint Propagation

We have already seen constraint propagation in action, but let's define it bit more precisely.

The basic step in constraint propagation is called **filtering a constraint**. As input, filtering takes a constraint and the current domains for its variables. It then tries to remove all the values in the domains that do not appear in any solution to that constraint. That is, it tries to reduce the domains so that *all the solutions are preserved*. Thus "constraint filtering" in CSPs corresponds to a "unit clause propagation" step in CNF SAT solvers, but it is usually more complex because the domains are larger and the constraints are more complex. Filtering on the global constraint level can also be more effective than unit clause propagation in pruning the search space.

Constraint propagation then means the process of applying filtering to the constraints in the CSP instance at hand. As filtering one constraint can reduce the domains of its variables, it can trigger further reduction when filtering another constraint that also involves the reduced-domain variables. Thus a constraint can be filtered multiple times during the constraint propagation process. Usually the process is run to the end, meaning until no more reduction happens in filtering any of the constraints. Or if this takes too long, then until some resource usage limit is exceeded.

Let's define **constraint filtering** more mathematically. As input, it takes

- a constraint $C \subseteq D(y_1) \times \ldots \times D(y_k)$, and
- the current domains $D_{\text{before}}(y_i) \subseteq D(y_i)$ for $i \in [1..k]$.

It then produces new domains $D_{\text{after}}(y_i) \subseteq D_{\text{before}}(y_i)$ for all $i \in [1..k]$ such that

$$C \cap (D_{\text{before}}(y_1) \times \ldots \times D_{\text{before}}(y_k)) = C \cap (D_{\text{after}}(y_1) \times \ldots \times D_{\text{after}}(y_k)).$$

Example

When $D_{\text{before}}(x) = \{1,2,4\}$ and $D_{\text{before}}(y) = \{1,2,5\}$, filtering the constraint $x = y$ optimally gives to domains $D_{\text{after}}(x) = \{1,2\}$ and $D_{\text{after}}(y) = \{1,2\}$. This is because the solutions of the constraint are $x = 1, y = 1$ and $x = 2, y = 2$.

Whenever feasible, our goal is to reduce the domains so that the constraint in question becomes arc-consistent.

Definition: Arc consistency of a constraint

A constraint C is **generalized arc consistent** in the domains $D(y_1), \ldots, D(y_k)$ if, for every $i \in [1..k]$ and every $v \in D(y_i)$, there exists a tuple $(v_1, \ldots, v_k) \in C$ such that $v_i = v$.

We usually omit “generalized” and just use the term **arc consistent** (the term “arc consistency” is sometimes used for “generalized arc consistency” when the constraint is binary (i.e., over two variables)).

Example

Assume two variables, x and y, and the domains $D(x) = \{1, 2, 4\}$ and $D(y) = \{1, 2, 5\}$. In this case:

- The constraint $x = y$, corresponding to the set $\{(1, 1), (2, 2)\}$, is not arc consistent as for $4 \in D(x)$ there is no tuple of the form $(4, v_y)$ in the set because $4 \notin D(y)$. Similarly, $5 \in D(y)$ but $5 \notin D(x)$.
- The constraint $x \neq y$, corresponding to the set $\{(1, 2), (1, 5), (2, 1), (2, 5), (4, 1), (4, 2), (4, 5)\}$, is arc consistent as
  - for every $v_x \in D(x)$, there is a $v_y \in D(y)$ such that $v_x \neq v_y$, and
  - for every $v_y \in D(y)$, there is a $v_x \in D(x)$ such that $v_x \neq v_y$.

Definition: Arc-consistency of a CSP instance

A CSP instance is generalized arc consistent if every constraint in it is.

Note that if a CSP is arc consistent and the domains of the variables are non-empty, then the individual constraints in the CSP have solutions. But this does not mean that the CSP as a whole has solutions, as illustrated by the next example.

Example

Consider the CSP consisting of the constraints $x \neq y$, $x \neq z$ and $y \neq z$. When $D(x) = \{1, 2\}$, $D(y) = \{1, 2\}$ and $D(z) = \{1, 2\}$, the CSP is arc consistent as all the constraints in it are arc consistent, but the CSP as a whole does not have any solutions.

Usually, global constraints filter better than a (possibly large) corresponding set of primitive binary constraints, as illustrated in the next example.

Example

Assume that $D(x) = \{1, 2\}$, $D(y) = \{1, 2\}$ and $D(z) = \{1, 2, 3\}$.

- Filtering the disequality constraints $x \neq y$, $x \neq z$ and $y \neq z$ to be arc-consistent results in the domains $D'(x) = \{1, 2\}$, $D'(y) = \{1, 2\}$ and $D'(z) = \{1, 2, 3\}$.

- Filtering the constraint **alldifferent**(x, y, z) to arc-consistency yields the domains $D'(x) = \{1,2\}$, $D'(y) = \{1,2\}$ and $D'(z) = \{3\}$.

Of course, filtering complex global constraints is algoritmically more difficult than filtering simple constraints such as binary equality and disequality.

In addition to filtering, one can also be interested in consistency checking of constraints. A constraint C over the variables $y_1, \ldots, y_k$ is **consistent** in the domains $D(y_1)$,...,$D(y_k)$ if $C \cap (D(y_1) \times \ldots \times D(y_k)) \neq \emptyset$. Otherwise the constraint is **inconsistent**.

Observe that, by definition, filtering an inconsistent constraint to arc-consistency results in empty domains. However, for some constraints, just checking the consistency can be easier. Mathematically, the **consistency checking** task can be defined as: given a constraint over $y_1, \ldots, y_k$ and some domains $D(y_1)$,...,$D(y_k)$, does it hold that

$$C \cap (D(y_1) \times \ldots \times D(y_k)) \neq \emptyset?$$

Testing consistency of binary equality and disequality constraints is easy:

- For equality, check whether the domains contain at least one common element
- For disequality, check whether the domains are not empty and not equal to a common singleton set.

Testing consistency of more complex global constraints is algoritmically more involved. In the following, we introduce some basic algorithms for consistency checking and filtering some of the global constraints we saw earlier.

The sum constraint

Recall that a “sum” global constraint is of the form

$$c_1 x_1 + \ldots + c_n x_n = z$$

where the c_is are integer constants and $x_1, \ldots, x_n, z$ are integer-valued variables. Furthermore, recall that the domains of the variables $x_1, \ldots., x_n, z$ are, in general and especially during the backtracking search, arbitrary subsets of all integers. Thus, for example, solving the consistency of $c_1 x_1 + \ldots + c_n x_n = z$ in this setting is not the same as solving whether the equation $c_1 x_1 + \ldots + c_n x_n = z$ has solutions when $x_1, .., x_n, z \in \mathbb{Z}$ (see texts on linear diophantine equations for solving such problems in unrestricted integer domains).

Consistency checking

Consistency checking and filtering of sum constraints are NP-hard in general. Thisis because the NP-complete subset sum problem reduces to it in a rather straighforward way. However, one obtains pseudo-polynomial time algorithms by using dynamic programming [Trick2003] as follows. We define the predicate $p(i, v)$, where $i \in [0..n]$, that evaluates to true if and only if the sum $\sum_{j=1}^{i} c_j x_j$ can evaluate to v, recursively as follows:

- $p(0, v) =$ T for $v = 0$ and F otherwise. That is, summing nothing always gives zero.
- $p(i, v) = \bigvee_{d \in D(x_i)} p(i-1, v - c_i d)$ for $1 \in [1..n]$. That is, $\sum_{j=1}^{i} c_j x_j$ can evaluate to v if and only if $\sum_{j=1}^{i-1} c_j x_j$ can evaluate to $v - c_i d$ for some value d in the domain of x_i.

If $p(n, v) =$ T for some $v \in D(z)$, then $x_1, \ldots, x_n$ can have values so that $c_1 x_1 + \ldots + c_n x_n = v$ and the constraint is consistent.

Example

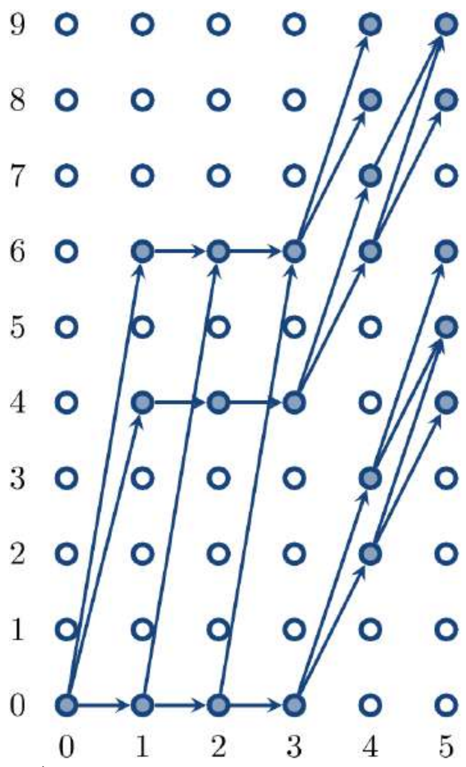
Consider the sum constraint

$$2x_1 + 2x_2 + 2x_3 + x_4 + x_5 = z$$

under the domains $D(x_1) = \{0, 2, 3\}$, $D(x_2) = \{0, 3\}$, $D(x_3) = \{0, 3\}$, $D(x_4) = \{2, 3\}$, $D(x_5) = \{2, 3\}$, and $D(z) = \{4, 7, 9\}$.

The figure below illustrates the computation of p:

- the dot at (i, v) is filled iff $p(i, v) =$ T, and
- there is an edge from (i, v_i) to $(i + 1, v_{i+1})$ iff $p([x_1, \ldots, x_i], v_i) =$ T and $v_{i+1} = v_i + c_{i+1} d$ for some $d \in D(x_{i+1})$.



We see that $p(5, 4)$ and $p(5, 9)$ are true, therefore the constraint is consistent.

Filtering

The graph representation shown above allows us to find all the solutions as well: they correspond to the paths from the node $(0,0)$ to the nodes (n,v) with $v \in D(z)$. Namely, if $(0,0),(1,v_1),\ldots,(n,v_n)$ is such a path, then the solution is

- $x_1 = (v_1 - 0)/c_1$,
- $x_2 = (v_2 - v_1)/c_2$,
-
- $x_n = (v_n - v_{n-1})/c_n$

Filtering is thus also easy:

1. $v \in D'(z)$ iff $v \in D(z)$ and there is a path from the node $(0,0)$ to (n,v), and
2. $v \in D'(x_i)$ iff there is a path $(0,0),(1,v_1),\ldots,(n,v_n)$ such that $v_n \in D(z)$ and $x_i = (v_i - v_{i-1})/c_i$.

Finding all the edges that take part in some of the paths above is easy: just follow the edges backwards from the nodes (n,v) with $v \in D(z)$.

Example

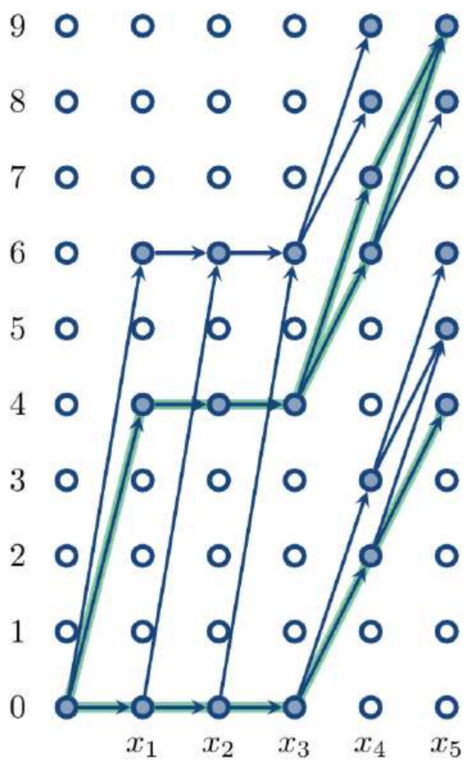
Consider again the sum constraint

$$2x_1 + 2x_2 + 2x_3 + x_4 + x_5 = z$$

under the domains $D(x_1) = \{0,2,3\}$, $D(x_2) = \{0,3\}$, $D(x_3) = \{0,3\}$, $D(x_4) = \{2,3\}$, $D(x_5) = \{2,3\}$, and $D(z) = \{4,7,9\}$. From the highlighted edges we see that the constraint has three solutions:

- $x_1 = 0, x_2 = 0, x_3 = 0, x_4 = 2, x_5 = 2$
- $x_1 = 2, x_2 = 0, x_3 = 0, x_4 = 2, x_5 = 2$, and
- $x_1 = 2, x_2 = 0, x_3 = 0, x_4 = 3, x_5 = 3$.

We can thus reduce the domains to $D(x_1) = \{0,2\}$, $D(x_2) = \{0\}$, $D(x_3) = \{0\}$, $D(x_4) = \{2,3\}$, $D(x_5) = \{2,3\}$, and $D(z) = \{4,9\}$.



When implemented with arrays, the dynamic programming approach only works well if the constants and domain values are reasonably small. When implemented with dynamic sets, the approach works well if the number of possible values of $\sum_{j=1}^{i} c_j x_j$ for each i is manageable. For the other cases (large domains etc), one can use weaker filtering that does not enforce arc-consistency. For instance, for each i one can compute the sum $m_i = \sum_{j \in \{1,\ldots,i-1,i+1,\ldots,n\}} c_j \min D(x_j)$ and remove the values v from $D(x_i)$ for which $m_i + c_i v > \max D(z)$.

The all-different constraint

Recall that an all-different global constraint

$$\textsf{alldifferent}(y_1, \ldots, y_n)$$

requires that the values assigned to the variables $y_1, \ldots, y_n$ are pairwise distinct. We now show a graph theoretical way for consistency checking and filtering of all-different constraints.

Consistency checking

We can *reduce* the consistency problem of **alldifferent** constraints to the matching problem in unweighted bipartite graphs [vanHoeveKatriel2006]. As there are efficient algorithms for solving the matching problem, the consistency problem can be solved efficiently as well.

Mathematically, an **unweighted bipartite graph** is a triple (U, W, E), where

- U and W are two disjoint, finite and non-empty sets of **vertices**, and
- $E \subseteq U \times W$ is a set of **edges**.

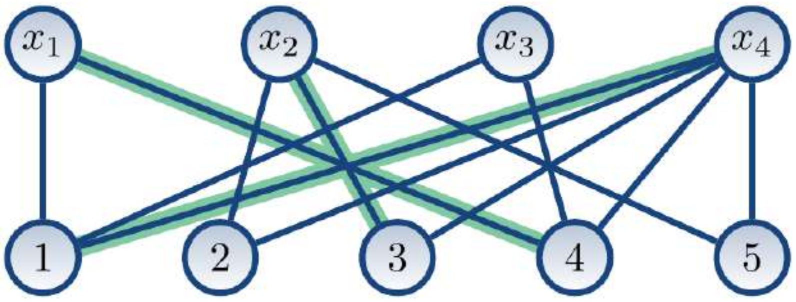
A **matching** in such a graph is a subset $M \subseteq E$ of the edges such that for all disjoint edges $(u_1, w_1), (u_2, w_2) \in M$ it holds that $u_1 \neq u_2$ and $w_1 \neq w_2$. In other words, a matching associates each vertex with at most one edge. A matching M **covers** a vertex $u \in U$ if $(u, w) \in M$ for some $w \in W$.

Example

The bipartite graph (U, W, E) with

$$\begin{aligned} U &= \{x_1, x_2, x_3, x_4\} \\ W &= \{1, 2, 3, 4, 5\} \\ E &= \{(x_1, 1), (x_1, 4), (x_2, 2), (x_2, 3), (x_2, 5), (x_3, 1), (x_3, 4), \\ &\quad (x_4, 1), (x_4, 2), (x_4, 3), (x_4, 4), (x_4, 5)\} \end{aligned}$$

is shown below. The matching $\{(x_1, 4), (x_2, 3), (x_4, 1)\}$ is shown with the highlighted edges. It covers the vertices x_1, x_2 and x_4 but not the vertex x_3.



The reduction from **alldifferent** constraints to undirected bipartite graphs is rather simple:

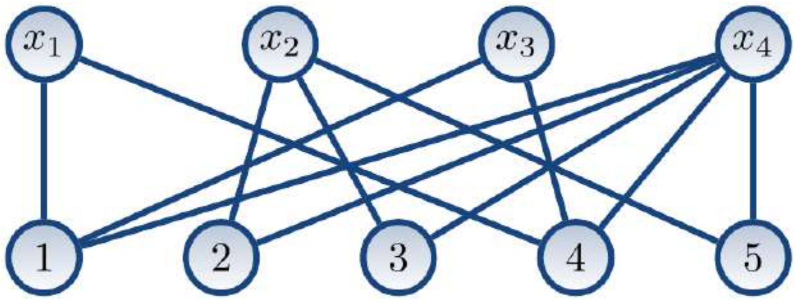
Definition: Value graphs

Assume an all-different constraint **alldifferent**$(y_1, \ldots, y_k)$ and some domains $D(y_1)$,..., $D(y_k)$. The **value graph** of the constraint under the domains is the undirected bipartite graph $(\{y_1, \ldots, y_k\}, \bigcup_{i=1}^{k} D(y_i), E)$, where $E = \{(y_i, v) \mid 1 \leq i \leq k \wedge v \in D(y_i)\}$.

The vertices in $\{y_1, \ldots, y_k\}$ are called **variable vertices** and the ones in $\bigcup_{i=1}^{k} D(y_i)$ **value vertices**.

Example

The value graph of the constraint **alldifferent**(x_1, x_2, x_3, x_4) under the domains $D(x_1) = \{1, 4\}$, $D(x_2) = \{2, 3, 5\}$, $D(x_3) = \{1, 4\}$, and $D(x_4) = \{1, 2, 3, 4, 5\}$ is shown below.

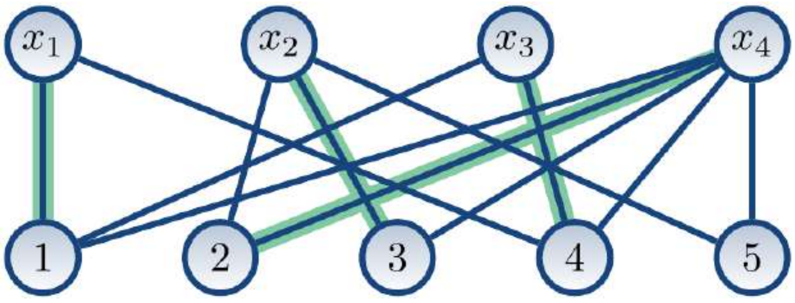


By construction, each solution to the constraint corresponds to a matching that covers all the variable vertices of the value graph. Conversely, each matching that covers all the variable vertices of the value graph corresponds to a solution to the constraint. Therefore, *the constraint is consistent if the value graph has a matching that covers all the variable vertices.*

Example

Consider again the constraint **alldifferent**(x_1, x_2, x_3, x_4) and the domains $D(x_1) = \{1, 4\}$, $D(x_2) = \{2, 3, 5\}$, $D(x_3) = \{1, 4\}$, and $D(x_4) = \{1, 2, 3, 4, 5\}$.

The matching corresponding to the solution $\{x_1 \mapsto 1, x_2 \mapsto 3, x_3 \mapsto 4, x_4 \mapsto 2\}$ is shown below with highlighted edges.



Whether a value graph has a matching covering all variable vertices can be found by computing a **maximum** matching, a matching having largest possible number of edges, for the graph and checking whether it covers all the variable vertices. A maximum matching can be found, e.g., with the Hopcroft-Karp algorithm. A simplified version of this algorithm is presented next.

Assume a bipartite graph $G = (U, W, E)$ and a matching $M \subseteq E$. We say that a vertex is M-**free** if it does not occur in any edge in M. An **augmenting path** for M is a path in G that starts and ends in an M-free vertex and alternates between edges not in M and in M. The following two facts on augmenting paths allow us to find a maximum matching:

1. A matching is maximum if there are no augmenting paths for it.
2. If M has an augmenting path p, a larger matching can be obtained by taking the symmetric difference of M and the edges in p.

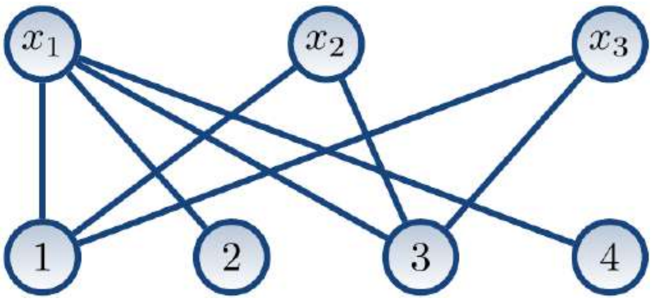
Based on this, we obtain an algorithm for finding a maximum matching:

1. Start with an empty matching M
2. Search for an augmenting path p
   - If none is found, return M
   - Otherwise, assign M to the symmetric difference of M and the edges in p and repeat

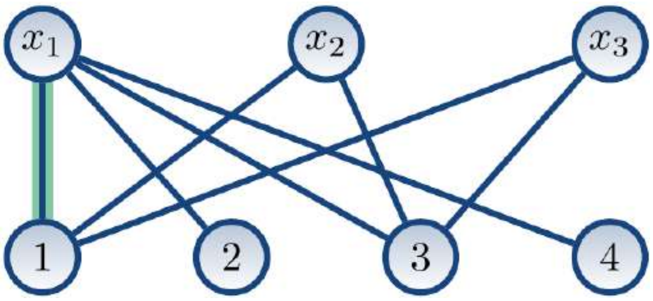
Augmenting paths can be easily found with breadth-first search.

Example: Finding a maximum matching

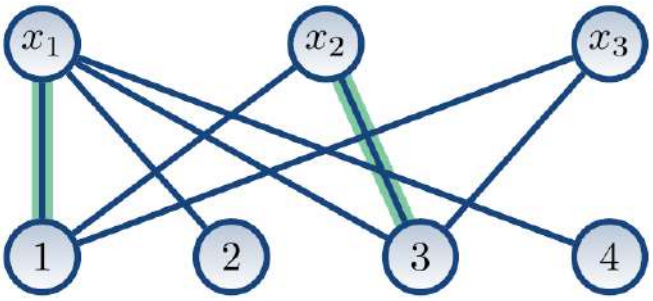
The value graph for the constraint **alldifferent**(x_1, x_2, x_3) with the domains $D(x_1) = \{1, 2, 3, 4\}$, $D(x_2) = \{1, 3\}$, and $D(x_3) = \{1, 3\}$ is shown below.



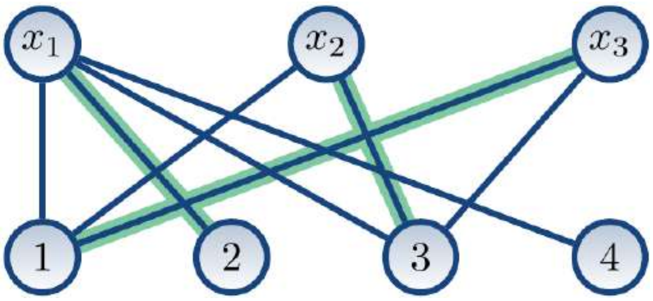
We start with the empty matching $M = \emptyset$. We may first find the augmenting path $[x_1, 1]$. After this, the matching is $M = \{(x_1, 1)\}$, shown below with highlighted edges.



The next the augmenting path could be $[x_2, 3]$. After this, the matching is $M = \{(x_1, 1), (x_2, 3)\}$:



The next the augmenting path could be $[x_3, 1, x_1, 2]$, giving the matching $M = \{(x_1, 2), (x_2, 3), (x_3, 1)\}$:



The are no more augmenting paths. All variable vertices are covered and a solution is $\{x_1 \mapsto 2, x_2 \mapsto 3, x_3 \mapsto 1\}$.

Filtering

By using value graphs, all-different constraints can be filtered to arc-consistency in polynomial time. As stated earlier,

- each solution to the constraint corresponds to a value vertex covering matching of the value graph, and
- each value vertex covering matching of the value graph corresponds to a solution to the constraint.

Thus if an edge (x, v) in the value graph does not appear in any value vertex covering matching, the element v can be removed from the domain $D(x)$.

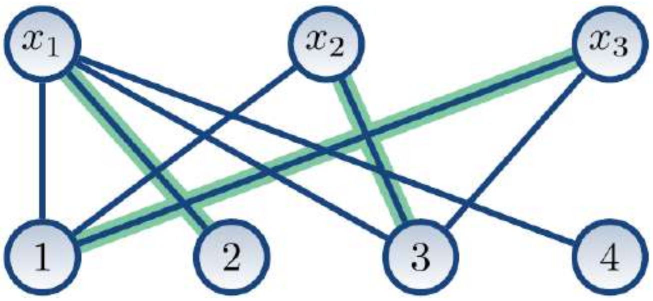
Once one maximum matching is found, we can find all edges that appear in some maximum matching [Tassa2012] [vanHoeveKatriel2006]. Assume a bipartite graph $G = (U, W, E)$ and a maximum matching $M \subseteq E$. Build the directed graph $G_M = (U \cup W, \{(u', w') \mid (u', w') \in M\} \cup \{(w', u') \mid (u', w') \in E \setminus M\})$. Now an edge (u, w) belongs to some maximum matching if

1. $(u, w) \in M$,
2. (u, w) is an edge inside a strongly connected component of G_M, or
3. (u, w) appears in a path of G_M that starts from an M-free vertex

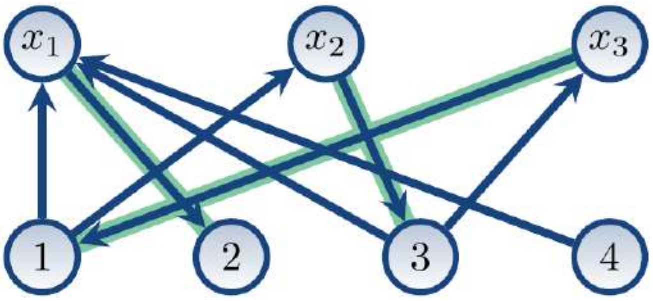
Strongly connected components can be computed efficiently with Tarjan’s algorithm or with Kosaraju’s algorithm.

Example

Consider again our running example having the value graph G and a maximum matching M shown below.



The directed graph G_M is the following.



Now:

- The edges $(x_1, 2)$, $(x_2, 3)$ and $(x_3, 1)$ in M belong to some maximum matching.
- The directed graph has only one non-singleton strongly connected component, $\{x_2, 3, x_3, 1\}$. Thus the edges $(x_2, 1)$ and $(x_3, 3)$ belong to some maximum matching.
- As $[4, x_1]$ is a path of the form required by the third case, $(x_1, 4)$ belongs to some maximum matching.

The edge $(x_1, 1)$ cannot be included with any one three cases above. Therefore, the value 1 can be removed from the domain of x_1. The same applies to edge $(x_1, 3)$ and thus the value 3 can be removed from the domain of x_1.

Indeed, the constraint has the four solutions $\{x_1 \mapsto 2, x_2 \mapsto 1, x_3 \mapsto 3\}$, $\{x_1 \mapsto 4, x_2 \mapsto 1, x_3 \mapsto 3\}$, $\{x_1 \mapsto 2, x_2 \mapsto 3, x_3 \mapsto 1\}$, and $\{x_1 \mapsto 4, x_2 \mapsto 3, x_3 \mapsto 1\}$.

Backtracking Search

How can we solve CSP instances? That is, how can we deduce whether a set of constraints is satisfiable or not, or to find an optimal solution for them? When the domains are finite, CSP instances could be solved by

- reducing the CSP instance to a SAT instance, and then
- solving the SAT instance with a SAT solver.

Indeed, some CSP solvers do this. However, performing a backtracking search and constraint propagation on the global constraints level can result in better performance as

1. the expansion of non-binary domains and global constraints to the Boolean level can be quite large, and
2. one can use efficient custom algorithms for propagating global constraints.

Below is a *very simple* backtracking search procedure that

- first performs constraint propagation, and
- if the solution is not yet found, selects a variable and then recursively searches for a solution in which the variable has a fixed value from its domain.

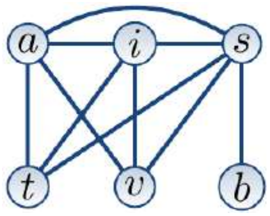
```
def $\mathit{solve}(\mathcal{C}, D)$:
    $D' \leftarrow \mathit{propagate}(\mathcal{C}, D)$
    if $\mathcal{C}$ has a constraint $C$ that is inconsistent under $D'$: return $\mathbf{unsat}$
    if $|D'(x)| = 1$ for all the variables $x$: return $\mathbf{sat}$
    choose a variable $x$ with $D'(x) = \{v_1, \ldots, v_k\}$ and $k \geq 2$
    for each $v \in \{v_1, \ldots, v_k\}$:
        $r \leftarrow \mathit{solve}(\mathcal{C}, D'_{x \mapsto \{v\}})$
        if $r = \mathbf{sat}$: return $\mathbf{sat}$
    return $\mathbf{unsat}$
```

Example

Consider a simple ARM architecture assembly program with symbolic register names below.

```
  LDR a, a_addr  // a_addr is the address of the input array
  MOV i, #0      // i = 0
  MOV s, #0      // s = 0
loop:
  CMP i, #10            // if i >= 10, goto end
  BGE end
  LDR t, [a, i, LSL#2]  // t = a[i]
  MUL v, t, t           // v = t * t
  ADD s, s, v           // s = s + v
  ADD i, i, #1          // i = i + 1
  B   loop
end:
  LDR b, b_addr  // save s to the memory location b_addr
  STR s, [b]
```

If we perform a liveness analysis for the registers, we get a register-inference graph (see e.g. Aho, Sethi, and Ullman: "Compilers: Principles, Techniques, and Tools") shown below. It has an edge between register nodes if and only if there is a line in which one register is written to and the other can be read later without being written to in between. The graph is k-colorable iff the program can be implemented by using only k hardware registers and no spills to the memory.

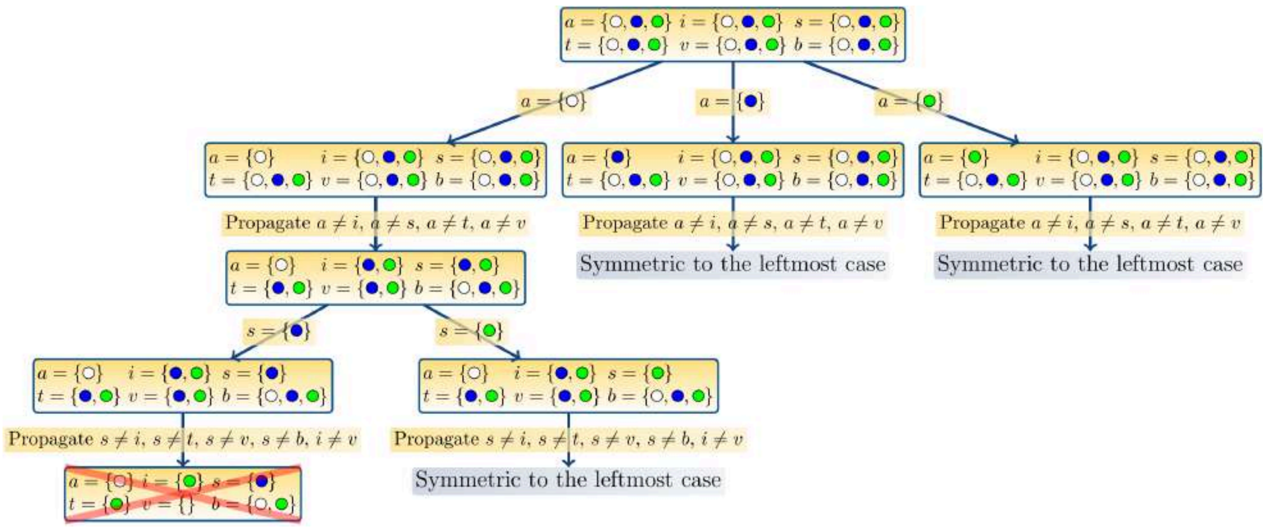


The graph k-coloring problem can be easily expressed as a CSP problem by taking a variable for each vertex. The domain of all the variables is a set of k colors. The constraints are $x \neq y$ for each edge between x and y in the graph. In our simple example, the constraints are

$$a \neq i \quad a \neq s \quad a \neq t \quad a \neq v \quad i \neq s$$

$$i \neq t \quad i \neq v \quad s \neq t \quad s \neq v \quad s \neq b$$

A search tree for our instance when three colors are allowed is shown below. Each node describes the domains either when entering the `solve` call or after constraint propagation.



$a = \{○, ●, ●\}$ $i = \{○, ●, ●\}$ $s = \{○, ●, ●\}$ $t = \{○, ●, ●\}$ $v = \{○, ●, ●\}$ $b = \{○, ●, ●\}$
$a = \{○\}$
$a = \{●\}$
$a = \{●\}$
Propagate $a \neq i$, $a \neq s$, $a \neq t$, $a \neq v$
Symmetric to the leftmost case
Symmetric to the leftmost case
$s = \{●\}$
$s = \{●\}$
Propagate $s \neq i$, $s \neq t$, $s \neq v$, $s \neq b$, $i \neq v$
Propagate $s \neq i$, $s \neq t$, $s \neq v$, $s \neq b$, $i \neq v$
Symmetric to the leftmost case
$a = \{○\}$ $i = \{●\}$ $s = \{●\}$ $t = \{●\}$ $v = \{\}$ $b = \{○, ●\}$

The instance has no solutions as all the branches end in a situation where the domain of some variable is empty.

Our example graph is colorable with 4 colors and thus the program can be implemented with 4 registers:

```
  LDR R0, a_addr  // a_addr is the address of the input array
  MOV R1, #0      // i = 0
  MOV R2, #0      // s = 0
loop:
  CMP R1, #10              // if i >= 10, goto end
  BGE end
  LDR R3, [R0, R1, LSL#2]  // t = a[i]
  MUL R3, R3, R3           // v = t * t
  ADD R2, R2, R3           // s = s + v
  ADD R1, R1, #1           // i = i + 1
  B   loop
end:
  LDR R3, b_addr  // save s to the memory location b_addr
  STR R2, [R3]
```

The above description of the backtracking search scheme is simplified and omits many details. For instance, the variable selection heuristic, i.e. how one chooses the variable x whose domain $\{v_1, \ldots, v_k\}$ is split into $\{v_1\}$,...,$\{v_k\}$, is very important but not discussed further in this course. And one could also split the domain $\{v_1, \ldots, v_k\}$ of the branching variable x differently, for instance into $\{v_1, \ldots, v_{k/2}\}$ and $\{v_{k/2+1}, \ldots, v_k\}$ if k is large. Some solvers allow the user to control these aspects and thus tune the solver for specific problem types.

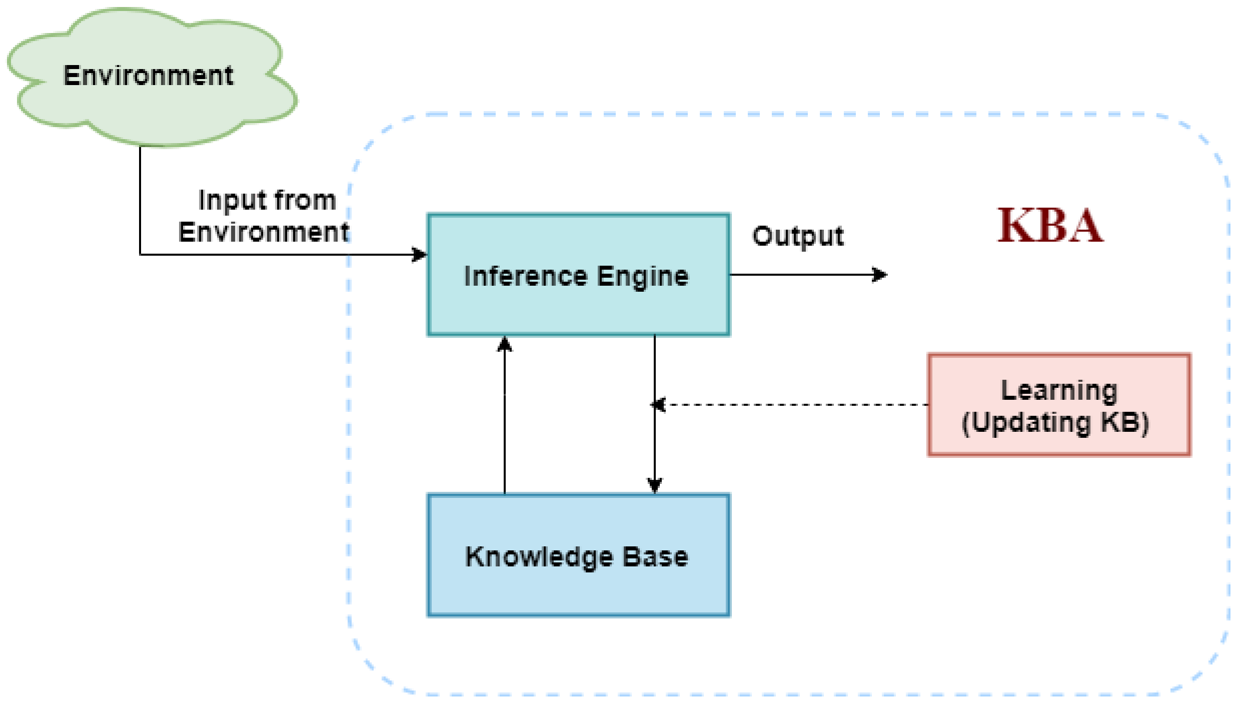
Knowledge-Based Agent in Artificial intelligence

- An intelligent agent needs **knowledge** about the real world for taking decisions and **reasoning** to act efficiently.
- Knowledge-based agents are those agents who have the capability of **maintaining an internal state of knowledge, reason over that knowledge, update their knowledge after observations and take actions. These agents can represent the world with some formal representation and act intelligently**.
- Knowledge-based agents are composed of two main parts:
  - **Knowledge-base and**
  - **Inference system**.

A knowledge-based agent must able to do the following:

- An agent should be able to represent states, actions, etc.
- An agent Should be able to incorporate new percepts
- An agent can update the internal representation of the world
- An agent can deduce the internal representation of the world
- An agent can deduce appropriate actions.

The architecture of knowledge-based agent:



The above diagram is representing a generalized architecture for a knowledge-based agent. The knowledge-based agent (KBA) take input from the environment by perceiving the environment. The input is taken by the inference engine of the agent and which also communicate with KB to decide as per the knowledge store in KB. The learning element of KBA regularly updates the KB by learning new knowledge.

Knowledge base: Knowledge-base is a central component of a knowledge-based agent, it is also known as KB. It is a collection of sentences (here 'sentence' is a technical term and it is not identical to sentence in English). These sentences are expressed in a language which is called a knowledge representation language. The Knowledge-base of KBA stores fact about the world.

Why use a knowledge base?

Knowledge-base is required for updating knowledge for an agent to learn with experiences and take action as per the knowledge.

Inference system

Inference means deriving new sentences from old. Inference system allows us to add a new sentence to the knowledge base. A sentence is a proposition about the world. Inference system applies logical rules to the KB to deduce new information.

Inference system generates new facts so that an agent can update the KB. An inference system works mainly in two rules which are given as:

- **Forward chaining**
- **Backward chaining**

Operations Performed by KBA

Following are three operations which are performed by KBA in order to show the intelligent behavior:

1. **TELL:** This operation tells the knowledge base what it perceives from the environment.
2. **ASK:** This operation asks the knowledge base what action it should perform.
3. **Perform:** It performs the selected action.

A generic knowledge-based agent:

Following is the structure outline of a generic knowledge-based agents program:

```
function KB-AGENT(percept):
persistent: KB, a knowledge base
          t, a counter, initially 0, indicating time
TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
Action = ASK(KB, MAKE-ACTION-QUERY(t))
TELL(KB, MAKE-ACTION-SENTENCE(action, t))
 t = t + 1
 return action
```

The knowledge-based agent takes percept as input and returns an action as output. The agent maintains the knowledge base, KB, and it initially has some background knowledge of the real world. It also has a counter to indicate the time for the whole process, and this counter is initialized with zero.

Each time when the function is called, it performs its three operations:

- Firstly it TELLs the KB what it perceives.
- Secondly, it asks KB what action it should take
- Third agent program TELLS the KB that which action was chosen.

The MAKE-PERCEPT-SENTENCE generates a sentence as setting that the agent perceived the given percept at the given time.

The MAKE-ACTION-QUERY generates a sentence to ask which action should be done at the current time.

MAKE-ACTION-SENTENCE generates a sentence which asserts that the chosen action was executed.

Various levels of knowledge-based agent:

A knowledge-based agent can be viewed at different levels which are given below:

1. Knowledge level

Knowledge level is the first level of knowledge-based agent, and in this level, we need to specify what the agent knows, and what the agent goals are. With these specifications, we can fix its behavior. For example, suppose an automated taxi agent needs to go from a station A to station B, and he knows the way from A to B, so this comes at the knowledge level.

2. Logical level:

At this level, we understand that how the knowledge representation of knowledge is stored. At this level, sentences are encoded into different logics. At the logical level, an encoding of knowledge into logical sentences occurs. At the logical level we can expect to the automated taxi agent to reach to the destination B.

3. Implementation level:

This is the physical representation of logic and knowledge. At the implementation level agent perform actions as per logical and knowledge level. At this level, an automated taxi agent actually implement his knowledge and logic so that he can reach to the destination.

Approaches to designing a knowledge-based agent:

There are mainly two approaches to build a knowledge-based agent:

1. **1. Declarative approach:** We can create a knowledge-based agent by initializing with an empty knowledge base and telling the agent all the sentences with which we want to start with. This approach is called Declarative approach.

2. **2. Procedural approach:** In the procedural approach, we directly encode desired behavior as a program code. Which means we just need to write a program that already encodes the desired behavior or agent.

However, in the real world, a successful agent can be built by combining both declarative and procedural approaches, and declarative knowledge can often be compiled into more efficient procedural code.

Propositional logic in Artificial intelligence

Propositional logic (PL) is the simplest form of logic where all the statements are made by propositions. A proposition is a declarative statement which is either true or false. It is a technique of knowledge representation in logical and mathematical form.

Example:

1. a) It is Sunday.
2. b) The Sun rises from West (False proposition)
3. c) 3+3= 7(False proposition)
4. d) 5 is a prime number.

Following are some basic facts about propositional logic:

ADVERTISEMENT

- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a representing a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.
- Propositional logic consists of an object, relations or function, and **logical connectives**.
- These connectives are also called logical operators.
- The propositions and connectives are the basic elements of the propositional logic.
- Connectives can be said as a logical operator which connects two sentences.
- A proposition formula which is always true is called **tautology**, and it is also called a valid sentence.
- A proposition formula which is always false is called **Contradiction**.
- A proposition formula which has both true and false values is called
- Statements which are questions, commands, or opinions are not propositions such as "**Where is Rohini**", "**How are you**", "**What is your name**", are not propositions.

Syntax of propositional logic:

The syntax of propositional logic defines the allowable sentences for the knowledge representation. There are two types of Propositions:

a. **Atomic Propositions**
b. **Compound propositions**

- **Atomic Proposition:** Atomic propositions are the simple propositions. It consists of a single proposition symbol. These are the sentences which must be either true or false.

Example:

1. a) 2+2 is 4, it is an atomic proposition as it is a **true** fact.
2. b) "The Sun is cold" is also a proposition as it is a **false** fact.

- **Compound proposition:** Compound propositions are constructed by combining simpler or atomic propositions, using parenthesis and logical connectives.

Example:

1. a) "It is raining today, and street is wet."
2. b) "Ankit is a doctor, and his clinic is in Mumbai."

Logical Connectives:

Logical connectives are used to connect two simpler propositions or representing a sentence logically. We can create compound propositions with the help of logical connectives. There are mainly five connectives, which are given as follows:

1. **Negation:** A sentence such as ¬ P is called negation of P. A literal can be either Positive literal or negative literal.
2. **Conjunction:** A sentence which has ∧ connective such as, **P ∧ Q** is called a conjunction.
   Example: Rohan is intelligent and hardworking. It can be written as,
   P= Rohan is intelligent,
   Q= Rohan is hardworking. → P∧ Q.
3. **Disjunction:** A sentence which has ∨ connective, such as **P ∨ Q**. is called disjunction, where P and Q are the propositions.

Example: "Ritika is a doctor or Engineer",
Here P= Ritika is Doctor. Q= Ritika is Doctor, so we can write it as **P ∨ Q**.

4. **Implication:** A sentence such as P → Q, is called an implication. Implications are also known as if-then rules. It can be represented as **If** it is raining, then the street is wet.
   Let P= It is raining, and Q= Street is wet, so it is represented as P → Q

5. **Biconditional:** A sentence such as **P⇔ Q is a Biconditional sentence, example If I am breathing, then I am alive**
   P= I am breathing, Q= I am alive, it can be represented as P ⇔ Q.

Following is the summarized table for Propositional Logic Connectives:

Connective symbols	Word	Technical term	Example
∧	AND	Conjunction	A ∧ B
∨	OR	Disjunction	A ∨ B
→	Implies	Implication	A → B
⇔	If and only if	Biconditional	A⇔ B
¬ or ~	Not	Negation	¬ A or ¬ B

Truth Table:

In propositional logic, we need to know the truth values of propositions in all possible scenarios. We can combine all the possible combination with logical connectives, and the representation of these combinations in a tabular format is called **Truth table**. Following are the truth table for all logical connectives:

For Negation:

P	¬ P
True	**False**
False	**True**

For Conjunction:

P	Q	P∧ Q
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**False**

For disjunction:

P	Q	P V Q.
True	True	**True**
False	True	**True**
True	False	**True**
False	False	**False**

For Implication:

P	Q	P→ Q
True	True	**True**
True	False	**False**
False	True	**True**
False	False	**True**

For Biconditional:

P	Q	P⇔ Q
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**True**

Truth table with three propositions:

We can build a proposition composing three propositions P, Q, and R. This truth table is made-up of 8n Tuples as we have taken three proposition symbols.

P	Q	R	¬R	P∨ Q	P∨Q→¬R
True	True	True	False	True	False
True	True	False	True	True	True
True	False	True	False	True	False
True	False	False	True	True	True
False	True	True	False	True	False
False	True	False	True	True	True
False	False	True	False	False	True
False	False	False	True	False	True

Precedence of connectives:

Just like arithmetic operators, there is a precedence order for propositional connectors or logical operators. This order should be followed while evaluating a propositional problem. Following is the list of the precedence order for operators:

Precedence	**Operators**
First Precedence	Parenthesis
Second Precedence	Negation
Third Precedence	Conjunction(AND)
Fourth Precedence	Disjunction(OR)
Fifth Precedence	Implication
Six Precedence	Biconditional

Note: For better understanding use parenthesis to make sure of the correct interpretations. Such as ¬R∨ Q, It can be interpreted as (¬R) ∨ Q.

Logical equivalence:

Logical equivalence is one of the features of propositional logic. Two propositions are said to be logically equivalent if and only if the columns in the truth table are identical to each other.

Let's take two propositions A and B, so for logical equivalence, we can write it as A⇔B. In below truth table we can see that column for ¬A∨ B and A→B, are identical hence A is Equivalent to B

A	B	¬A	¬A∨ B	A→B
T	T	F	**T**	**T**
T	F	F	**F**	**F**
F	T	T	**T**	**T**
F	F	T	**T**	**T**

Properties of Operators:

- **Commutativity:**
  - P∧ Q= Q ∧ P, or
  - P ∨ Q = Q ∨ P.
- **Associativity:**
  - (P ∧ Q) ∧ R= P ∧ (Q ∧ R),
  - (P ∨ Q) ∨ R= P ∨ (Q ∨ R)
- **Identity element:**
  - P ∧ True = P,
  - P ∨ True= True.
- **Distributive:**
  - P∧ (Q ∨ R) = (P ∧ Q) ∨ (P ∧ R).
  - P ∨ (Q ∧ R) = (P ∨ Q) ∧ (P ∨ R).
- **DE Morgan's Law:**
  - ¬ (P ∧ Q) = (¬P) ∨ (¬Q)
  - ¬ (P ∨ Q) = (¬ P) ∧ (¬Q).
- **Double-negation elimination:**
  - ¬ (¬P) = P.

Limitations of Propositional logic:

- We cannot represent relations like ALL, some, or none with propositional logic. Example:
  a. **All the girls are intelligent.**
  b. **Some apples are sweet.**
- Propositional logic has limited expressive power.
- In propositional logic, we cannot describe statements in terms of their properties or logical relationships.

Rules of Inference in Artificial intelligence

Inference:

In artificial intelligence, we need intelligent computers which can create new logic from old logic or by evidence, **so generating the conclusions from evidence and facts is termed as Inference**.

Inference rules:

Inference rules are the templates for generating valid arguments. Inference rules are applied to derive proofs in artificial intelligence, and the proof is a sequence of the conclusion that leads to the desired goal.

In inference rules, the implication among all the connectives plays an important role. Following are some terminologies related to inference rules:

- **Implication:** It is one of the logical connectives which can be represented as P → Q. It is a Boolean expression.
- **Converse:** The converse of implication, which means the right-hand side proposition goes to the left-hand side and vice-versa. It can be written as Q → P.
- **Contrapositive:** The negation of converse is termed as contrapositive, and it can be represented as ¬ Q → ¬ P.
- **Inverse:** The negation of implication is called inverse. It can be represented as ¬ P → ¬ Q.

From the above term some of the compound statements are equivalent to each other, which we can prove using truth table:

P	Q	P → Q	Q→ P	¬ Q → ¬ P	¬ P → ¬ Q.
T	T	T	T	T	T
T	F	F	T	F	T
F	T	T	F	T	F
F	F	T	T	T	T

Hence from the above truth table, we can prove that P → Q is equivalent to ¬ Q → ¬ P, and Q→ P is equivalent to ¬ P → ¬ Q.

Types of Inference rules:

1. Modus Ponens:

The Modus Ponens rule is one of the most important rules of inference, and it states that if P and P → Q is true, then we can infer that Q will be true. It can be represented as:

Notation for Modus ponens: $\dfrac{P \to Q, \quad P}{\therefore Q}$

Example:

Statement-1: "If I am sleepy then I go to bed" ==> P→ Q
Statement-2: "I am sleepy" ==> P
Conclusion: "I go to bed." ==> Q.
Hence, we can say that, if P→ Q is true and P is true then Q will be true.

Proof by Truth table:

P	Q	$P \to Q$
0	0	0
0	1	1
1	0	0
1	1	1

2. Modus Tollens:

The Modus Tollens rule state that if P→ Q is true and **¬ Q is true, then ¬ P** will also true. It can be represented as:

Notation for Modus Tollens: $\dfrac{P \to Q, \quad \sim Q}{\sim P}$

Statement-1: "If I am sleepy then I go to bed" ==> P→ Q
Statement-2: "I do not go to the bed."==> ~Q
Statement-3: Which infers that "**I am not sleepy**" => ~P

Proof by Truth table:

P	Q	$\sim P$	$\sim Q$	$P \rightarrow Q$
0	0	1	1	1
0	1	1	0	1
1	0	0	1	0
1	1	0	0	1

3. Hypothetical Syllogism:

The Hypothetical Syllogism rule state that if P→R is true whenever P→Q is true, and Q→R is true. It can be represented as the following notation:

Example:

Statement-1: If you have my home key then you can unlock my home. **P→Q**
Statement-2: If you can unlock my home then you can take my money. **Q→R**
Conclusion: If you have my home key then you can take my money. **P→R**

Proof by truth table:

P	Q	R	$P \rightarrow Q$	$Q \rightarrow R$	$P \rightarrow R$
0	0	0	1	1	1
0	0	1	1	1	1
0	1	0	1	0	1
0	1	1	1	1	1
1	0	0	0	1	1
1	0	1	0	1	1
1	1	0	1	0	0
1	1	1	1	1	1

4. Disjunctive Syllogism:

The Disjunctive syllogism rule state that if PvQ is true, and ¬P is true, then Q will be true. It can be represented as:

Notation of Disjunctive syllogism: $\dfrac{P \vee Q,\ \neg P}{Q}$

Example:

Statement-1: Today is Sunday or Monday. ==>PvQ
Statement-2: Today is not Sunday. ==> ¬P
Conclusion: Today is Monday. ==> Q

Proof by truth-table:

P	Q	$\neg P$	$P \lor Q$
0	0	1	0
0	1	1	1 ←
1	0	0	1
1	1	0	1

5. Addition:

The Addition rule is one the common inference rule, and it states that If P is true, then PvQ will be true.

Notation of Addition: $\frac{P}{P \lor Q}$

Example:

Statement: I have a vanilla ice-cream. ==> P
Statement-2: I have Chocolate ice-cream.
Conclusion: I have vanilla or chocolate ice-cream. ==> (PvQ)

Proof by Truth-Table:

P	Q	$P \lor Q$
0	0	0
1	0	1 ←
0	1	1
1	1	1 ←

6. Simplification:

The simplification rule state that if **P∧ Q** is true, then **Q or P** will also be true. It can be represented as:

Notation of Simplification rule: $\frac{P \land Q}{Q}$ Or $\frac{P \land Q}{P}$

Proof by Truth-Table:

P	Q	$P \wedge Q$
0	0	0
1	0	0
0	1	0
1	1	1 ←

7. Resolution:

The Resolution rule state that if P∨Q and ¬ P∧R is true, then Q∨R will also be true. **It can be represented as**

Notation of Resolution $\frac{P\vee Q,\ \neg P\wedge R}{Q\vee R}$

Proof by Truth-Table:

P	¬ P	Q	R	$P \vee Q$	¬ P∧R	$Q \vee R$
0	1	0	0	0	0	0
0	1	0	1	0	0	1
0	1	1	0	1	1	1 ←
0	1	1	1	1	1	1 ←
1	0	0	0	1	0	0
1	0	0	1	1	0	1
1	0	1	0	1	0	1
1	0	1	1	1	0	1 ←

First-Order Logic in Artificial intelligence

In the topic of Propositional logic, we have seen that how to represent statements using propositional logic. But unfortunately, in propositional logic, we can only represent the facts, which are either true or false. PL is not sufficient to represent the complex sentences or natural language statements. The propositional logic has very limited expressive power. Consider the following sentence, which we cannot represent using PL logic.

- **"Some humans are intelligent", or**
- **"Sachin likes cricket."**

To represent the above statements, PL logic is not sufficient, so we required some more powerful logic, such as first-order logic.

First-Order logic:

- First-order logic is another way of knowledge representation in artificial intelligence. It is an extension to propositional logic.
- FOL is sufficiently expressive to represent the natural language statements in a concise way.
- First-order logic is also known as **Predicate logic or First-order predicate logic**. First-order logic is a powerful language that develops information about the objects in a more easy way and can also express the relationship between those objects.
- First-order logic (like natural language) does not only assume that the world contains facts like propositional logic but also assumes the following things in the world:
  - **Objects:** A, B, people, numbers, colors, wars, theories, squares, pits, wumpus, ...
  - **Relations: It can be unary relation such as:** red, round, is adjacent, **or n-any relation such as:** the sister of, brother of, has color, comes between
  - **Function:** Father of, best friend, third inning of, end of, ...
- As a natural language, first-order logic also has two main parts:
  a. **Syntax**
  b. **Semantics**

Syntax of First-Order logic:

The syntax of FOL determines which collection of symbols is a logical expression in first-order logic. The basic syntactic elements of first-order logic are symbols. We write statements in short-hand notation in FOL.

Basic Elements of First-order logic:

Following are the basic elements of FOL syntax:

Constant	1, 2, A, John, Mumbai, cat,....
Variables	x, y, z, a, b,....
Predicates	Brother, Father, >,....
Function	sqrt, LeftLegOf, ...
Connectives	∧, ∨, ¬, ⇒, ⇔
Equality	==
Quantifier	∀, ∃

Atomic sentences:

- Atomic sentences are the most basic sentences of first-order logic. These sentences are formed from a predicate symbol followed by a parenthesis with a sequence of terms.
- We can represent atomic sentences as **Predicate (term1, term2, ..., term n)**.

Example: Ravi and Ajay are brothers: => Brothers(Ravi, Ajay).
Chinky is a cat: => cat (Chinky).

Complex Sentences:

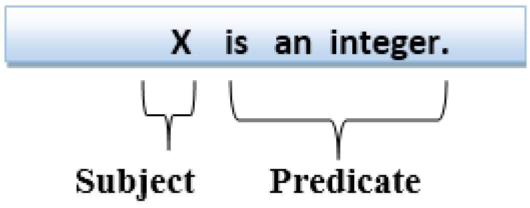
- Complex sentences are made by combining atomic sentences using connectives.

First-order logic statements can be divided into two parts:

- **Subject:** Subject is the main part of the statement.

- **Predicate:** A predicate can be defined as a relation, which binds two atoms together in a statement.

Consider the statement: "x is an integer.", it consists of two parts, the first part x is the subject of the statement and second part "is an integer," is known as a predicate.



Quantifiers in First-order logic:

- A quantifier is a language element which generates quantification, and quantification specifies the quantity of specimen in the universe of discourse.
- These are the symbols that permit to determine or identify the range and scope of the variable in the logical expression. There are two types of quantifier:

a. **Universal Quantifier, (for all, everyone, everything)**

b. **Existential quantifier, (for some, at least one).**

Universal Quantifier:

Universal quantifier is a symbol of logical representation, which specifies that the statement within its range is true for everything or every instance of a particular thing.

The Universal quantifier is represented by a symbol ∀, which resembles an inverted A.

Note: In universal quantifier we use implication "→".

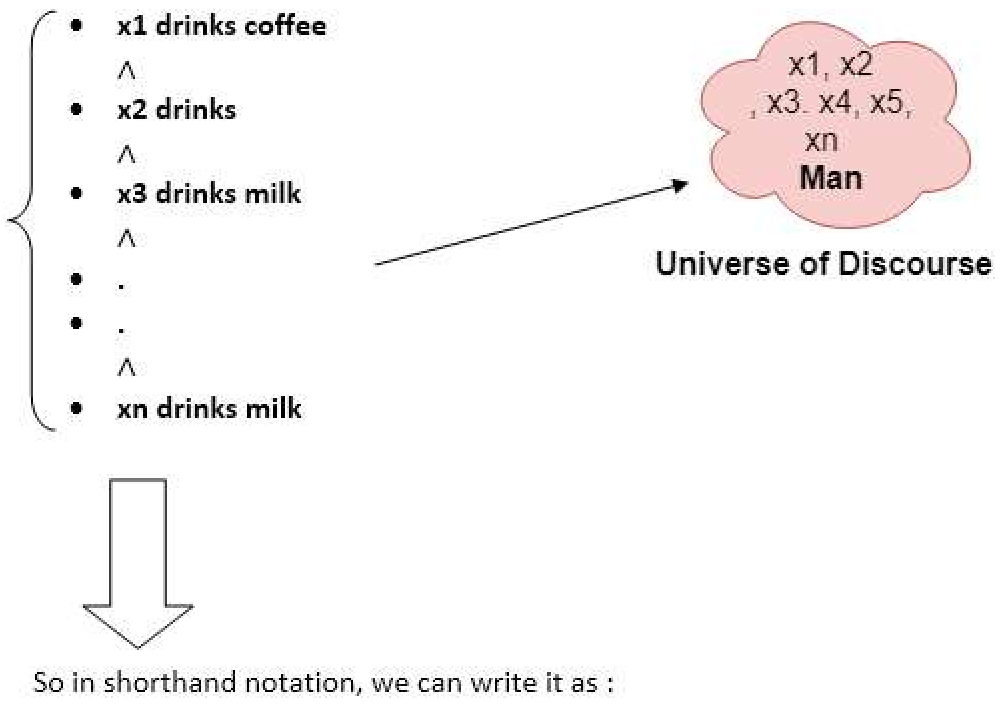
If x is a variable, then ∀x is read as:

- **For all x**
- **For each x**
- **For every x.**

Example:

All man drink coffee.

Let a variable x which refers to a cat so all x can be represented in UOD as below:



∀x man(x) → drink (x, coffee).

It will be read as: There are all x where x is a man who drink coffee.

Existential Quantifier:

Existential quantifiers are the type of quantifiers, which express that the statement within its scope is true for at least one instance of something.

It is denoted by the logical operator ∃, which resembles as inverted E. When it is used with a predicate variable then it is called as an existential quantifier.

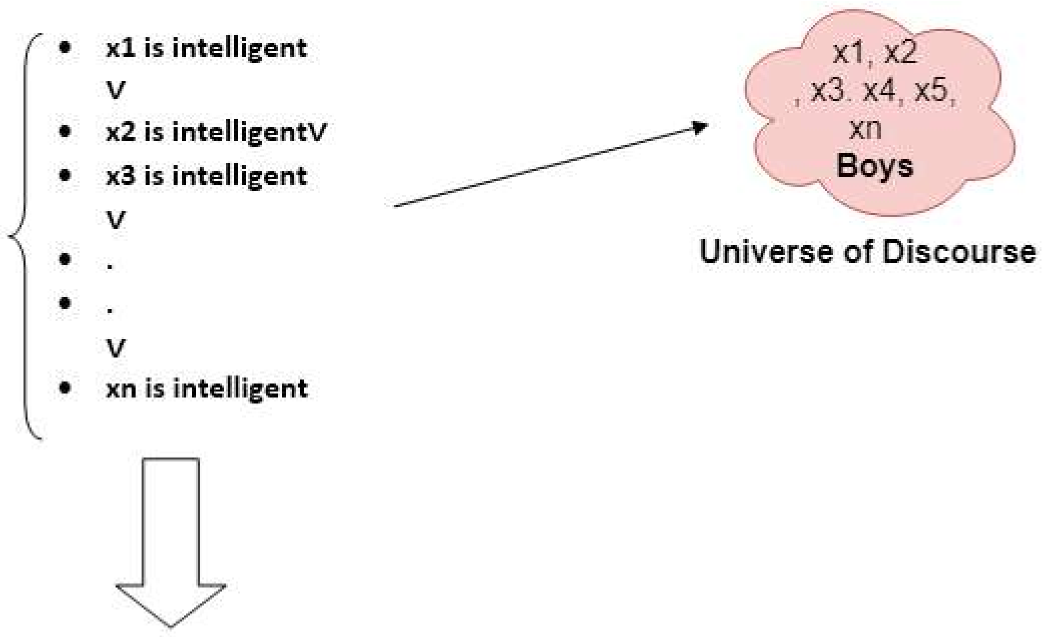
Note: In Existential quantifier we always use AND or Conjunction symbol (∧).

If x is a variable, then existential quantifier will be ∃x or ∃(x). And it will be read as:

- **There exists a 'x.'**
- **For some 'x.'**
- **For at least one 'x.'**

Example:

Some boys are intelligent.



So in short-hand notation, we can write it as:

∃x: boys(x) ∧ intelligent(x)

It will be read as: There are some x where x is a boy who is intelligent.

Points to remember:

- The main connective for universal quantifier **∀** is implication **→**.
- The main connective for existential quantifier **∃** is and **∧**.

Properties of Quantifiers:

- In universal quantifier, ∀x∀y is similar to ∀y∀x.
- In Existential quantifier, ∃x∃y is similar to ∃y∃x.
- ∃x∀y is not similar to ∀y∃x.

Some Examples of FOL using quantifier:

1. All birds fly.
In this question the predicate is "**fly(bird)**."
And since there are all birds who fly so it will be represented as follows.
∀x bird(x) →fly(x).

2. Every man respects his parent.
In this question, the predicate is "**respect(x, y)," where x=man, and y= parent**.

Since there is every man so will use ∀, and it will be represented as follows:

∀x man(x) → respects (x, parent).

3. Some boys play cricket.

In this question, the predicate is "**play(x, y)**," where x= boys, and y= game. Since there are some boys so we will use **∃, and it will be represented as**:

∃x boys(x) → play(x, cricket).

4. Not all students like both Mathematics and Science.

In this question, the predicate is "**like(x, y)," where x= student, and y= subject**. Since there are not all students, so we will use **∀ with negation, so** following representation for this:

¬∀ (x) [student(x) → like(x, Mathematics) ∧ like(x, Science)].

5. Only one student failed in Mathematics.

In this question, the predicate is "**failed(x, y)," where x= student, and y= subject**. Since there is only one student who failed in Mathematics, so we will use following representation for this:

∃(x) [student(x) → failed (x, Mathematics) ∧∀ (y) [¬(x==y) ∧ student(y) → ¬failed (x, Mathematics)].

Free and Bound Variables:

The quantifiers interact with variables which appear in a suitable way. There are two types of variables in First-order logic which are given below:

Free Variable: A variable is said to be a free variable in a formula if it occurs outside the scope of the quantifier.

Example: ∀x ∃(y)[P (x, y, z)], where z is a free variable.

Bound Variable: A variable is said to be a bound variable in a formula if it occurs within the scope of the quantifier.

Example: ∀x [A (x) B(y)], here x and y are the bound variables.

Knowledge Engineering in First-order logic

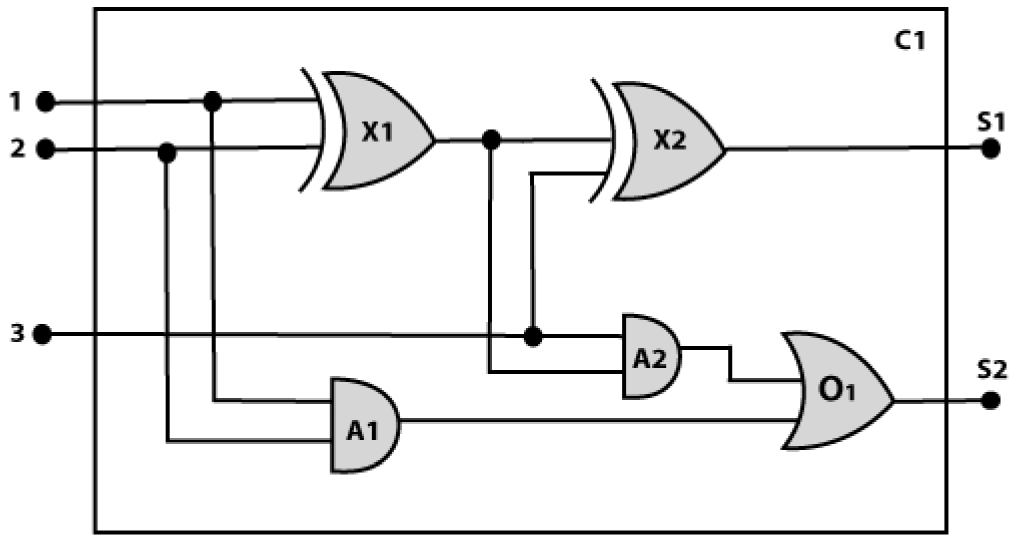
What is knowledge-engineering?

The process of constructing a knowledge-base in first-order logic is called as knowledge- engineering. In **knowledge-engineering**, someone who investigates a particular domain, learns important concept of that domain, and generates a formal representation of the objects, is known as **knowledge engineer**.

In this topic, we will understand the Knowledge engineering process in an electronic circuit domain, which is already familiar. This approach is mainly suitable for creating **special-purpose knowledge base**.

The knowledge-engineering process:

Following are some main steps of the knowledge-engineering process. Using these steps, we will develop a knowledge base which will allow us to reason about digital circuit (**One-bit full adder**) which is given below



1. Identify the task:

The first step of the process is to identify the task, and for the digital circuit, there are various reasoning tasks.

At the first level or highest level, we will examine the functionality of the circuit:

- **Does the circuit add properly?**
- **What will be the output of gate A2, if all the inputs are high?**

At the second level, we will examine the circuit structure details such as:

- **Which gate is connected to the first input terminal?**
- **Does the circuit have feedback loops?**

2. Assemble the relevant knowledge:

In the second step, we will assemble the relevant knowledge which is required for digital circuits. So for digital circuits, we have the following required knowledge:

- Logic circuits are made up of wires and gates.
- Signal flows through wires to the input terminal of the gate, and each gate produces the corresponding output which flows further.
- In this logic circuit, there are four types of gates used: **AND, OR, XOR, and NOT**.
- All these gates have one output terminal and two input terminals (except NOT gate, it has one input terminal).

3. Decide on vocabulary:

The next step of the process is to select functions, predicate, and constants to represent the circuits, terminals, signals, and gates. Firstly we will distinguish the gates from each other and from other objects. Each gate is represented as an object which is named by a constant, such as, **Gate(X1)**. The functionality of each gate is determined by its type, which is taken as constants such as **AND, OR, XOR, or NOT**. Circuits will be identified by a predicate: **Circuit (C1)**.

For the terminal, we will use predicate: **Terminal(x)**.

For gate input, we will use the function **In(1, X1)** for denoting the first input terminal of the gate, and for output terminal we will use **Out (1, X1)**.

The function **Arity(c, i, j)** is used to denote that circuit c has i input, j output.

The connectivity between gates can be represented by predicate **Connect(Out(1, X1), In(1, X1))**.

We use a unary predicate **On (t)**, which is true if the signal at a terminal is on.

4. Encode general knowledge about the domain:

To encode the general knowledge about the logic circuit, we need some following rules:

- If two terminals are connected then they have the same input signal, it can be represented as:

1. ∀ t1, t2 Terminal (t1) ∧ Terminal (t2) ∧ Connect (t1, t2) → Signal (t1) = Signal (2).

- Signal at every terminal will have either value 0 or 1, it will be represented as:

1. ∀ t Terminal (t) →Signal (t) = 1 ∨Signal (t) = 0.

- Connect predicates are commutative:

1. ∀ t1, t2 Connect(t1, t2) → Connect (t2, t1).

- Representation of types of gates:

1. ∀ g Gate(g) ∧ r = Type(g) → r = OR ∨r = AND ∨r = XOR ∨r = NOT.

- Output of AND gate will be zero if and only if any of its input is zero.

1. ∀ g Gate(g) ∧ Type(g) = AND →Signal (Out(1, g))= 0 ⇔ ∃n Signal (In(n, g))= 0.

- Output of OR gate is 1 if and only if any of its input is 1:

1. ∀ g Gate(g) ∧ Type(g) = OR → Signal (Out(1, g))= 1 ⇔ ∃n Signal (In(n, g))= 1

- Output of XOR gate is 1 if and only if its inputs are different:

1. ∀ g Gate(g) ∧ Type(g) = XOR → Signal (Out(1, g)) = 1 ⇔ Signal (In(1, g)) ≠ Signal (In (2, g)).

- Output of NOT gate is invert of its input:

1. ∀ g Gate(g) ∧ Type(g) = NOT → Signal (In(1, g)) ≠ Signal (Out(1, g)).

- All the gates in the above circuit have two inputs and one output (except NOT gate).

1. ∀ g Gate(g) ∧ Type(g) = NOT → Arity(g, 1, 1)
2. ∀ g Gate(g) ∧ r =Type(g) ∧ (r= AND ∨r= OR ∨r= XOR) → Arity (g, 2, 1).

- All gates are logic circuits:

1. ∀ g Gate(g) → Circuit (g).

5. Encode a description of the problem instance:

Now we encode problem of circuit C1, firstly we categorize the circuit and its gate components. This step is easy if ontology about the problem is already thought. This step involves the writing simple atomics sentences of instances of concepts, which is known as ontology.

For the given circuit C1, we can encode the problem instance in atomic sentences as below:

Since in the circuit there are two XOR, two AND, and one OR gate so atomic sentences for these gates will be:

1. For XOR gate: Type(x1)= XOR, Type(X2) = XOR
2. For AND gate: Type(A1) = AND, Type(A2)= AND
3. For OR gate: Type (O1) = OR.

And then represent the connections between all the gates.

Note: Ontology defines a particular theory of the nature of existence.

6. Pose queries to the inference procedure and get answers:

In this step, we will find all the possible set of values of all the terminal for the adder circuit. The first query will be:

What should be the combination of input which would generate the first output of circuit C1, as 0 and a second output to be 1?

1. ∃ i1, i2, i3 Signal (In(1, C1))=i1 ∧ Signal (In(2, C1))=i2 ∧ Signal (In(3, C1))= i3
2. ∧ Signal (Out(1, C1)) =0 ∧ Signal (Out(2, C1))=1

7. Debug the knowledge base:

Now we will debug the knowledge base, and this is the last step of the complete process. In this step, we will try to debug the issues of knowledge base.

In the knowledge base, we may have omitted assertions like 1 ≠ 0.

Inference in First-Order Logic

Inference in First-Order Logic is used to deduce new facts or sentences from existing sentences. Before understanding the FOL inference rule, let's understand some basic terminologies used in FOL.

Substitution:

Substitution is a fundamental operation performed on terms and formulas. It occurs in all inference systems in first-order logic. The substitution is complex in the presence of quantifiers in FOL. If we write **F[a/x]**, so it refers to substitute a constant "**a**" in place of variable "**x**".

> Note: First-order logic is capable of expressing facts about some or all objects in the universe.

Equality:

First-Order logic does not only use predicate and terms for making atomic sentences but also uses another way, which is equality in FOL. For this, we can use **equality symbols** which specify that the two terms refer to the same object.

Example: Brother (John) = Smith.

As in the above example, the object referred by the **Brother (John)** is similar to the object referred by **Smith**. The equality symbol can also be used with negation to represent that two terms are not the same objects.

Example: ¬(x=y) which is equivalent to x ≠y.

FOL inference rules for quantifier:

As propositional logic we also have inference rules in first-order logic, so following are some basic inference rules in FOL:

- **Universal Generalization**
- **Universal Instantiation**
- **Existential Instantiation**
- **Existential introduction**

1. Universal Generalization:

- Universal generalization is a valid inference rule which states that if premise P(c) is true for any arbitrary element c in the universe of discourse, then we can have a conclusion as ∀ x P(x).
- It can be represented as: $\frac{P(c)}{\forall x\, P(x)}$.
- This rule can be used if we want to show that every element has a similar property.
- In this rule, x must not appear as a free variable.

Example: Let's represent, P(c): "**A byte contains 8 bits**", so for ∀ **x P(x)** "**All bytes contain 8 bits**.", it will also be true.

2. Universal Instantiation:

- Universal instantiation is also called as universal elimination or UI is a valid inference rule. It can be applied multiple times to add new sentences.
- The new KB is logically equivalent to the previous KB.
- As per UI, **we can infer any sentence obtained by substituting a ground term for the variable**.
- The UI rule state that we can infer any sentence P(c) by substituting a ground term c (a constant within domain x) from ∀ **x P(x) for any object in the universe of discourse**.
- It can be represented as: $\frac{\forall x\, P(x)}{P(c)}$.

Example:1.

IF "Every person like ice-cream"=> ∀x P(x) so we can infer that "John likes ice-cream" => P(c)

Example: 2.

Let's take a famous example,

"All kings who are greedy are Evil." So let our knowledge base contains this detail as in the form of FOL:

∀x king(x) ∧ greedy (x) → Evil (x),

So from this information, we can infer any of the following statements using Universal Instantiation:

- **King(John) ∧ Greedy (John) → Evil (John),**
- **King(Richard) ∧ Greedy (Richard) → Evil (Richard),**
- **King(Father(John)) ∧ Greedy (Father(John)) → Evil (Father(John)),**

3. Existential Instantiation:

- Existential instantiation is also called as Existential Elimination, which is a valid inference rule in first-order logic.
- It can be applied only once to replace the existential sentence.
- The new KB is not logically equivalent to old KB, but it will be satisfiable if old KB was satisfiable.
- This rule states that one can infer P(c) from the formula given in the form of ∃x P(x) for a new constant symbol c.
- The restriction with this rule is that c used in the rule must be a new term for which P(c) is true.
- It can be represented as: $\frac{\exists x\, P(x)}{P(c)}$

Example:

From the given sentence: **∃x Crown(x) ∧ OnHead(x, John),**

So we can infer: **Crown(K) ∧ OnHead(K, John),** as long as K does not appear in the knowledge base.

- The above used K is a constant symbol, which is called **Skolem constant**.
- The Existential instantiation is a special case of **Skolemization process**.

4. Existential introduction

- An existential introduction is also known as an existential generalization, which is a valid inference rule in first-order logic.
- This rule states that if there is some element c in the universe of discourse which has a property P, then we can infer that there exists something in the universe which has the property P.
- It can be represented as: $\frac{P(c)}{\exists x P(x)}$

- **Example: Let's say that,**
  "Priyanka got good marks in English."
  "Therefore, someone got good marks in English."

Generalized Modus Ponens Rule:

For the inference process in FOL, we have a single inference rule which is called Generalized Modus Ponens. It is lifted version of Modus ponens.

Generalized Modus Ponens can be summarized as, " P implies Q and P is asserted to be true, therefore Q must be True."

According to Modus Ponens, for atomic sentences **pi, pi', q**. Where there is a substitution θ such that SUBST **(θ, pi',) = SUBST(θ, pi)**, it can be represented as:

$$\frac{p1', p2', \ldots, pn', (p1 \land p2 \land \ldots \land pn \Rightarrow q)}{\text{SUBST}(\theta, q)}$$

Example:

We will use this rule for Kings are evil, so we will find some x such that x is king, and x is greedy so we can infer that x is evil.

1. Here let say, p1' is king(John)   p1 is king(x)
2. p2' is Greedy(y)   p2 is Greedy(x)
3. θ is {x/John, y/John}   q is evil(x)
4. SUBST(θ,q).

What is Unification?

- Unification is a process of making two different logical atomic expressions identical by finding a substitution. Unification depends on the substitution process.
- It takes two literals as input and makes them identical using substitution.
- Let Ψ_1 and Ψ_2 be two atomic sentences and σ be a unifier such that, $\mathbf{\Psi_1\sigma = \Psi_2\sigma}$, then it can be expressed as **UNIFY(Ψ_1, Ψ_2)**.
- **Example: Find the MGU for Unify{King(x), King(John)}**

Let Ψ_1 = King(x), Ψ_2 = King(John),

Substitution θ = {John/x} is a unifier for these atoms and applying this substitution, and both expressions will be identical.

- The UNIFY algorithm is used for unification, which takes two atomic sentences and returns a unifier for those sentences (If any exist).
- Unification is a key component of all first-order inference algorithms.
- It returns fail if the expressions do not match with each other.
- The substitution variables are called Most General Unifier or MGU.

E.g. Let's say there are two different expressions, **P(x, y), and P(a, f(z))**.

In this example, we need to make both above statements identical to each other. For this, we will perform the substitution.

P(x, y)......... (i)
P(a, f(z))......... (ii)

- Substitute x with a, and y with f(z) in the first expression, and it will be represented as **a/x** and f(z)/y.
- With both the substitutions, the first expression will be identical to the second expression and the substitution set will be: **[a/x, f(z)/y]**.

Conditions for Unification:

Following are some basic conditions for unification:

- Predicate symbol must be same, atoms or expression with different predicate symbol can never be unified.
- Number of Arguments in both expressions must be identical.
- Unification will fail if there are two similar variables present in the same expression.

Unification Algorithm:

Algorithm: Unify(Ψ_1, Ψ_2)

```
Step. 1: If $\Psi_1$ or $\Psi_2$ is a variable or constant, then:
        a) If $\Psi_1$ or $\Psi_2$ are identical, then return NIL.
        b) Else if $\Psi_1$is a variable,
                a. then if $\Psi_1$ occurs in $\Psi_2$, then return FAILURE
                b. Else return { ($\Psi_2$/ $\Psi_1$)}.
        c) Else if $\Psi_2$ is a variable,
                a. If $\Psi_2$ occurs in $\Psi_1$ then return FAILURE,
                b. Else return {( $\Psi_1$/ $\Psi_2$)}.
        d) Else return FAILURE.
Step.2: If the initial Predicate symbol in $\Psi_1$ and $\Psi_2$ are not same, then
return FAILURE.
Step. 3: IF $\Psi_1$ and $\Psi_2$ have a different number of arguments, then return
FAILURE.
Step. 4: Set Substitution set(SUBST) to NIL.
Step. 5: For i=1 to the number of elements in $\Psi_1$.
        a) Call Unify function with the ith element of $\Psi_1$ and ith element of
$\Psi_2$, and put the result into S.
        b) If S = failure then returns Failure
        c) If S ≠ NIL then do,
                a. Apply S to the remainder of both L1 and L2.
                b. SUBST= APPEND(S, SUBST).
Step.6: Return SUBST.
```

Implementation of the Algorithm

Step.1: Initialize the substitution set to be empty.

Step.2: Recursively unify atomic sentences:

a. Check for Identical expression match.

b. If one expression is a variable v_i, and the other is a term t_i which does not contain variable v_i, then:

    a. Substitute t_i / v_i in the existing substitutions

b. Add t_i /v_i to the substitution setlist.

c. If both the expressions are functions, then function name must be similar, and the number of arguments must be the same in both the expression.

For each pair of the following atomic sentences find the most general unifier (If exist).

1. Find the MGU of {p(f(a), g(Y)) and p(X, X)}

Sol: S_0 => Here, Ψ_1 = p(f(a), g(Y)), and Ψ_2 = p(X, X)
SUBST θ= {f(a) / X}
S1 => Ψ_1 = p(f(a), g(Y)), and Ψ_2 = p(f(a), f(a))
SUBST θ= {f(a) / g(y)}, **Unification failed**.

Unification is not possible for these expressions.

2. Find the MGU of {p(b, X, f(g(Z))) and p(Z, f(Y), f(Y))}

Here, Ψ_1 = p(b, X, f(g(Z))) , and Ψ_2 = p(Z, f(Y), f(Y))
S_0 => { p(b, X, f(g(Z))); p(Z, f(Y), f(Y))}
SUBST θ={b/Z}

S_1 => { p(b, X, f(g(b))); p(b, f(Y), f(Y))}
SUBST θ={f(Y) /X}

S_2 => { p(b, f(Y), f(g(b))); p(b, f(Y), f(Y))}
SUBST θ= {g(b) /Y}

S_2 => { p(b, f(g(b)), f(g(b)); p(b, f(g(b)), f(g(b))} **Unified Successfully.**
And Unifier = { b/Z, f(Y) /X , g(b) /Y}.

3. Find the MGU of {p (X, X), and p (Z, f(Z))}

Here, Ψ_1 = {p (X, X), and Ψ_2 = p (Z, f(Z))
S_0 => {p (X, X), p (Z, f(Z))}
SUBST θ= {X/Z}
S1 => {p (Z, Z), p (Z, f(Z))}
SUBST θ= {f(Z) / Z}, **Unification Failed**.

Hence, unification is not possible for these expressions.

4. Find the MGU of UNIFY(prime (11), prime(y))

Here, Ψ_1 = {prime(11) , and Ψ_2 = prime(y)}
S_0 => {prime(11) , prime(y)}
SUBST θ= {11/y}

S_1 => {prime(11) , prime(11)} , **Successfully unified.**
Unifier: {11/y}.

5. Find the MGU of Q(a, g(x, a), f(y)), Q(a, g(f(b), a), x)}

Here, Ψ_1 = Q(a, g(x, a), f(y)), and Ψ_2 = Q(a, g(f(b), a), x)
S_0 => {Q(a, g(x, a), f(y)); Q(a, g(f(b), a), x)}
SUBST θ= {f(b)/x}
S_1 => {Q(a, g(f(b), a), f(y)); Q(a, g(f(b), a), f(b))}

SUBST θ= {b/y}
S_1 => {Q(a, g(f(b), a), f(b)); Q(a, g(f(b), a), f(b))}, **Successfully Unified.**

Unifier: [a/a, f(b)/x, b/y].

6. UNIFY(knows(Richard, x), knows(Richard, John))

Here, Ψ_1 = knows(Richard, x), and Ψ_2 = knows(Richard, John)
S_0 => { knows(Richard, x); knows(Richard, John)}
SUBST θ= {John/x}
S_1 => { knows(Richard, John); knows(Richard, John)}, **Successfully Unified.**
Unifier: {John/x}.

Resolution in FOL

Resolution

Resolution is a theorem proving technique that proceeds by building refutation proofs, i.e., proofs by contradictions. It was invented by a Mathematician John Alan Robinson in the year 1965.

Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements. Unification is a key concept in proofs by resolutions. Resolution is a single inference rule which can efficiently operate on the **conjunctive normal form or clausal form**.

Clause: Disjunction of literals (an atomic sentence) is called a **clause**. It is also known as a unit clause.

Conjunctive Normal Form: A sentence represented as a conjunction of clauses is said to be **conjunctive normal form** or **CNF**.

Note: To better understand this topic, firstly learns the FOL in AI.

The resolution inference rule:

The resolution rule for first-order logic is simply a lifted version of the propositional rule. Resolution can resolve two clauses if they contain complementary literals, which are assumed to be standardized apart so that they share no variables.

$$\frac{l_1 \lor \ldots\ldots \lor l_k, \qquad m_1 \lor \ldots\ldots \lor m_n}{\text{SUBST}(\theta, l_1 \lor \ldots\ldots \lor l_{i-1} \lor l_{i+1} \lor \ldots \lor l_k \lor m_1 \lor \ldots\ldots \lor m_{j-1} \lor m_{j+1} \lor \ldots \lor m_n)}$$

Where **l_i** and **m_j** are complementary literals.

This rule is also called the **binary resolution rule** because it only resolves exactly two literals.

Example:

We can resolve two clauses which are given below:

[Animal (g(x) V Loves (f(x), x)] **and** **[¬ Loves(a, b) V ¬Kills(a, b)]**

Where two complimentary literals are: **Loves (f(x), x) and ¬ Loves (a, b)**

These literals can be unified with unifier **θ= [a/f(x), and b/x]** , and it will generate a resolvent clause:

[Animal (g(x) V ¬ Kills(f(x), x)].

Steps for Resolution:

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF
3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph (unification).

To better understand all the above steps, we will take an example in which we will apply resolution.

Example:

a. **John likes all kind of food.**

b. **Apple and vegetable are food**

c. **Anything anyone eats and not killed is food.**

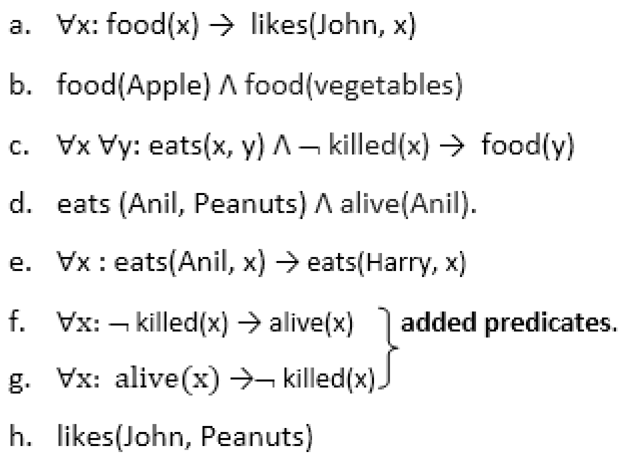
d. **Anil eats peanuts and still alive**

e. **Harry eats everything that Anil eats.**
   Prove by resolution that:

f. **John likes peanuts.**

Step-1: Conversion of Facts into FOL

In the first step we will convert all the given statements into its first order logic.



a. ∀x: food(x) → likes(John, x)

b. food(Apple) Λ food(vegetables)

c. ∀x ∀y: eats(x, y) Λ ¬ killed(x) → food(y)

d. eats (Anil, Peanuts) Λ alive(Anil).

e. ∀x : eats(Anil, x) → eats(Harry, x)

f. ∀x: ¬ killed(x) → alive(x)

g. ∀x: alive(x) →¬ killed(x)

added predicates.

h. likes(John, Peanuts)

Step-2: Conversion of FOL into CNF

In First order logic resolution, it is required to convert the FOL into CNF as CNF form makes easier for resolution proofs.

- **Eliminate all implication (→) and rewrite**
    a. ∀x ¬ food(x) V likes(John, x)
    b. food(Apple) Λ food(vegetables)
    c. ∀x ∀y ¬ [eats(x, y) Λ ¬ killed(x)] V food(y)
    d. eats (Anil, Peanuts) Λ alive(Anil)
    e. ∀x ¬ eats(Anil, x) V eats(Harry, x)
    f. ∀x¬ [¬ killed(x)] V alive(x)
    g. ∀x ¬ alive(x) V ¬ killed(x)
    h. likes(John, Peanuts).
- **Move negation (¬)inwards and rewrite**

. ∀x ¬ food(x) V likes(John, x)
    a. food(Apple) Λ food(vegetables)
    b. ∀x ∀y ¬ eats(x, y) V killed(x) V food(y)
    c. eats (Anil, Peanuts) Λ alive(Anil)
    d. ∀x ¬ eats(Anil, x) V eats(Harry, x)
    e. ∀x ¬killed(x)] V alive(x)
    f. ∀x ¬ alive(x) V ¬ killed(x)
    g. likes(John, Peanuts).
- **Rename variables or standardize variables**

. ∀x ¬ food(x) V likes(John, x)
    a. food(Apple) Λ food(vegetables)
    b. ∀y ∀z ¬ eats(y, z) V killed(y) V food(z)
    c. eats (Anil, Peanuts) Λ alive(Anil)
    d. ∀w¬ eats(Anil, w) V eats(Harry, w)
    e. ∀g ¬killed(g)] V alive(g)
    f. ∀k ¬ alive(k) V ¬ killed(k)
    g. likes(John, Peanuts).

- **Eliminate existential instantiation quantifier by elimination.** In this step, we will eliminate existential quantifier ∃, and this process is known as **Skolemization**. But in this example problem since there is no existential quantifier so all the statements will remain same in this step.
- **Drop Universal quantifiers.**
  In this step we will drop all universal quantifier since all the statements are not implicitly quantified so we don't need it.

  . ¬ food(x) V likes(John, x)

  a. food(Apple)

  b. food(vegetables)

  c. ¬ eats(y, z) V killed(y) V food(z)

  d. eats (Anil, Peanuts)

  e. alive(Anil)

  f. ¬ eats(Anil, w) V eats(Harry, w)

  g. killed(g) V alive(g)

  h. ¬ alive(k) V ¬ killed(k)

  i. likes(John, Peanuts).

Note: Statements "food(Apple) Λ food(vegetables)" and "eats (Anil, Peanuts) Λ alive(Anil)" can be written in two separate statements.

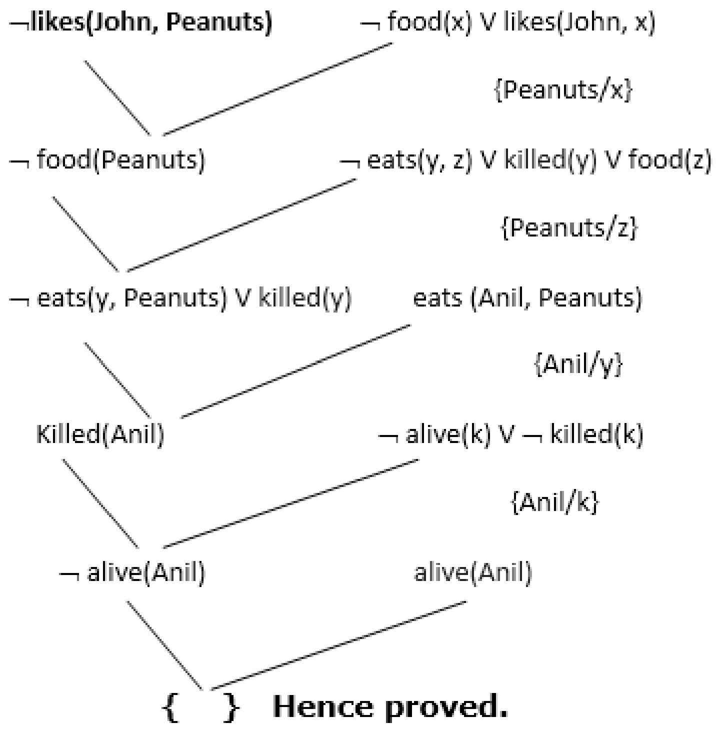
- **Distribute conjunction ∧ over disjunction ¬.**
  This step will not make any change in this problem.

Step-3: Negate the statement to be proved

In this statement, we will apply negation to the conclusion statements, which will be written as ¬likes(John, Peanuts)

Step-4: Draw Resolution graph:

Now in this step, we will solve the problem by resolution tree using substitution. For the above problem, it will be given as follows:



Hence the negation of the conclusion has been proved as a complete contradiction with the given set of statements.

Explanation of Resolution graph:

- In the first step of resolution graph, **¬likes(John, Peanuts)** , and **likes(John, x)** get resolved(canceled) by substitution of **{Peanuts/x}**, and we are left with **¬ food(Peanuts)**
- In the second step of the resolution graph, **¬ food(Peanuts)** , and **food(z)** get resolved (canceled) by substitution of **{ Peanuts/z}**, and we are left with **¬ eats(y, Peanuts) V killed(y)** .
- In the third step of the resolution graph, **¬ eats(y, Peanuts)** and **eats (Anil, Peanuts)** get resolved by substitution **{Anil/y}**, and we are left with **Killed(Anil)** .
- In the fourth step of the resolution graph, **Killed(Anil)** and **¬ killed(k)** get resolve by substitution **{Anil/k}**, and we are left with **¬ alive(Anil)** .
- In the last step of the resolution graph **¬ alive(Anil)** and **alive(Anil)** get resolved.

Forward Chaining and backward chaining in AI

In artificial intelligence, forward and backward chaining is one of the important topics, but before understanding forward and backward chaining lets first understand that from where these two terms came.

Inference engine:

The inference engine is the component of the intelligent system in artificial intelligence, which applies logical rules to the knowledge base to infer new information from known facts. The first inference engine was part of the expert system. Inference engine commonly proceeds in two modes, which are:

a. **Forward chaining**
b. **Backward chaining**

Horn Clause and Definite clause:

Horn clause and definite clause are the forms of sentences, which enables knowledge base to use a more restricted and efficient inference algorithm. Logical inference algorithms use forward and backward chaining approaches, which require KB in the form of the **first-order definite clause**.

Definite clause: A clause which is a disjunction of literals with **exactly one positive literal** is known as a definite clause or strict horn clause.

Horn clause: A clause which is a disjunction of literals with **at most one positive literal** is known as horn clause. Hence all the definite clauses are horn clauses.

Example: (¬ p V ¬ q V k). It has only one positive literal k.

It is equivalent to p ∧ q → k.

A. Forward Chaining

Forward chaining is also known as a forward deduction or forward reasoning method when using an inference engine. Forward chaining is a form of reasoning which start with atomic sentences in the knowledge base and applies inference rules (Modus Ponens) in the forward direction to extract more data until a goal is reached.

The Forward-chaining algorithm starts from known facts, triggers all rules whose premises are satisfied, and add their conclusion to the known facts. This process repeats until the problem is solved.

Properties of Forward-Chaining:

- It is a down-up approach, as it moves from bottom to top.
- It is a process of making a conclusion based on known facts or data, by starting from the initial state and reaches the goal state.
- Forward-chaining approach is also called as data-driven as we reach to the goal using available data.
- Forward -chaining approach is commonly used in the expert system, such as CLIPS, business, and production rule systems.

Consider the following famous example which we will use in both approaches:

Example:

"As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen."

Prove that **"Robert is criminal."**

To solve the above problem, first, we will convert all the above facts into first-order definite clauses, and then we will use a forward-chaining algorithm to reach the goal.

Facts Conversion into FOL:

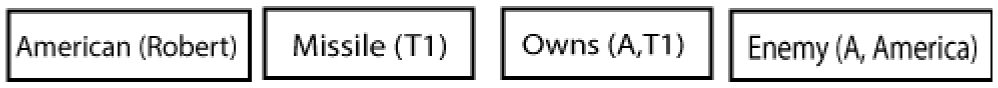
- It is a crime for an American to sell weapons to hostile nations. (Let's say p, q, and r are variables)
  American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)
- Country A has some missiles. **?p Owns(A, p) ∧ Missile(p)**. It can be written in two definite clauses by using Existential Instantiation, introducing new Constant T1.
  Owns(A, T1) ......(2)
  Missile(T1) ....(3)
- All of the missiles were sold to country A by Robert.
  ?p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) ......(4)
- Missiles are weapons.
  Missile(p) → Weapons (p) ....(5)

- Enemy of America is known as hostile.
  Enemy(p, America) →Hostile(p) **........(6)**
- Country A is an enemy of America.
  Enemy (A, America) **.........(7)**
- Robert is American
  American(Robert). **..........(8)**

Forward chaining proof:

Step-1:

In the first step we will start with the known facts and will choose the sentences which do not have implications, such as: **American(Robert), Enemy(A, America), Owns(A, T1), and Missile(T1)**. All these facts will be represented as below.



Step-2:

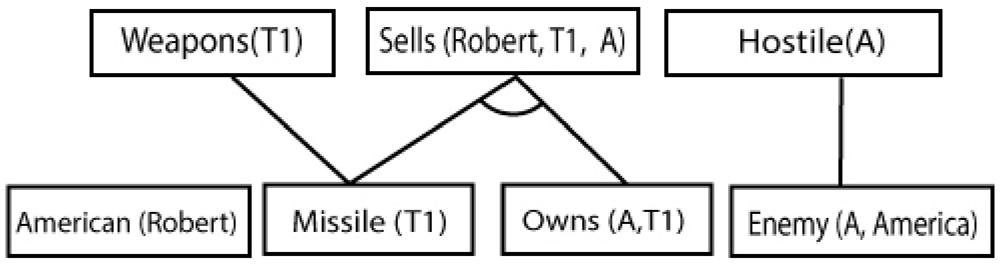
At the second step, we will see those facts which infer from available facts and with satisfied premises.

Rule-(1) does not satisfy premises, so it will not be added in the first iteration.

Rule-(2) and (3) are already added.

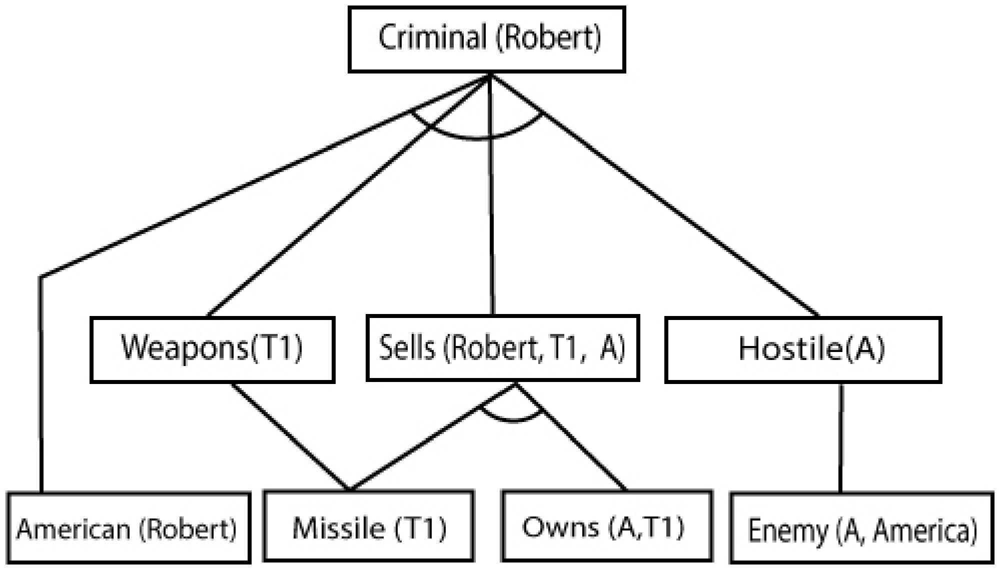
Rule-(4) satisfy with the substitution {p/T1}, **so Sells (Robert, T1, A)** is added, which infers from the conjunction of Rule (2) and (3).

Rule-(6) is satisfied with the substitution(p/A), so Hostile(A) is added and which infers from Rule-(7).



Step-3:

At step-3, as we can check Rule-(1) is satisfied with the substitution **{p/Robert, q/T1, r/A}, so we can add Criminal(Robert)** which infers all the available facts. And hence we reached our goal statement.



Hence it is proved that Robert is Criminal using forward chaining approach.

B. Backward Chaining:

Backward-chaining is also known as a backward deduction or backward reasoning method when using an inference engine. A backward chaining algorithm is a form of reasoning, which starts with the goal and works backward, chaining through rules to find known facts that support the goal.

Properties of backward chaining:

- It is known as a top-down approach.
- Backward-chaining is based on modus ponens inference rule.
- In backward chaining, the goal is broken into sub-goal or sub-goals to prove the facts true.
- It is called a goal-driven approach, as a list of goals decides which rules are selected and used.
- Backward -chaining algorithm is used in game theory, automated theorem proving tools, inference engines, proof assistants, and various AI applications.
- The backward-chaining method mostly used a **depth-first search** strategy for proof.

Example:

In backward-chaining, we will use the same above example, and will rewrite all the rules.

- **American (p) ∧ weapon(q) ∧ sells (p, q, r) ∧ hostile(r) → Criminal(p) ...(1)**
  Owns(A, T1) ....(2)
- **Missile(T1)**
- **?p Missiles(p) ∧ Owns (A, p) → Sells (Robert, p, A) ......(4)**
- **Missile(p) → Weapons (p) ....(5)**
- **Enemy(p, America) →Hostile(p) ....(6)**
- **Enemy (A, America) ....(7)**
- **American(Robert). ....(8)**

Backward-Chaining proof:

In Backward chaining, we will start with our goal predicate, which is **Criminal(Robert)**, and then infer further rules.

Step-1:

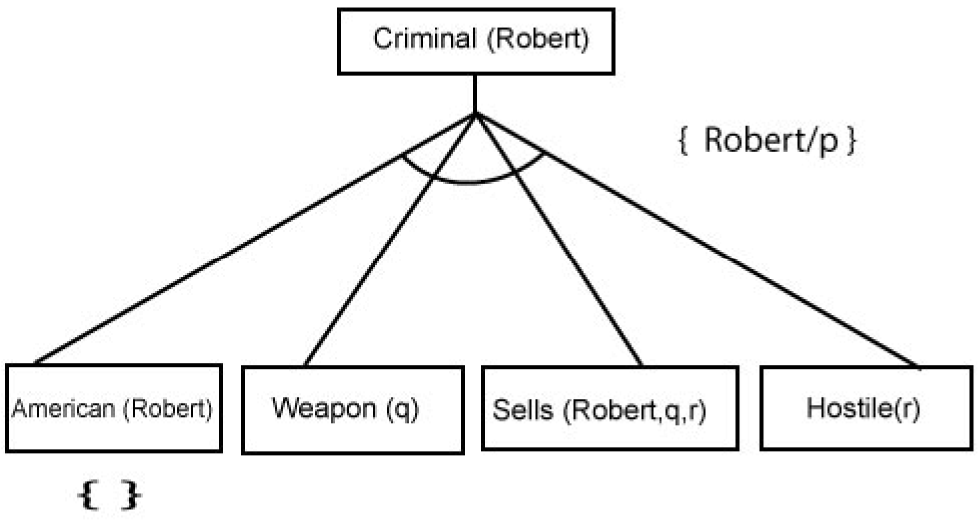
At the first step, we will take the goal fact. And from the goal fact, we will infer other facts, and at last, we will prove those facts true. So our goal fact is "Robert is Criminal," so following is the predicate of it.

Criminal (Robert)

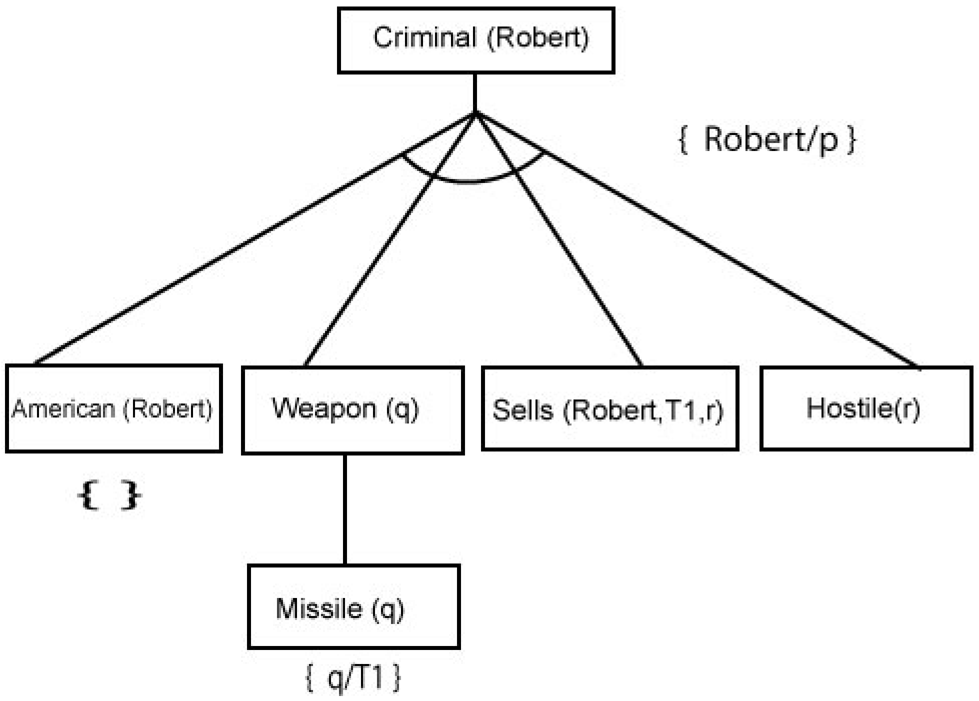
Step-2:

At the second step, we will infer other facts form goal fact which satisfies the rules. So as we can see in Rule-1, the goal predicate Criminal (Robert) is present with substitution {Robert/P}. So we will add all the conjunctive facts below the first level and will replace p with Robert.

Here we can see American (Robert) is a fact, so it is proved here.

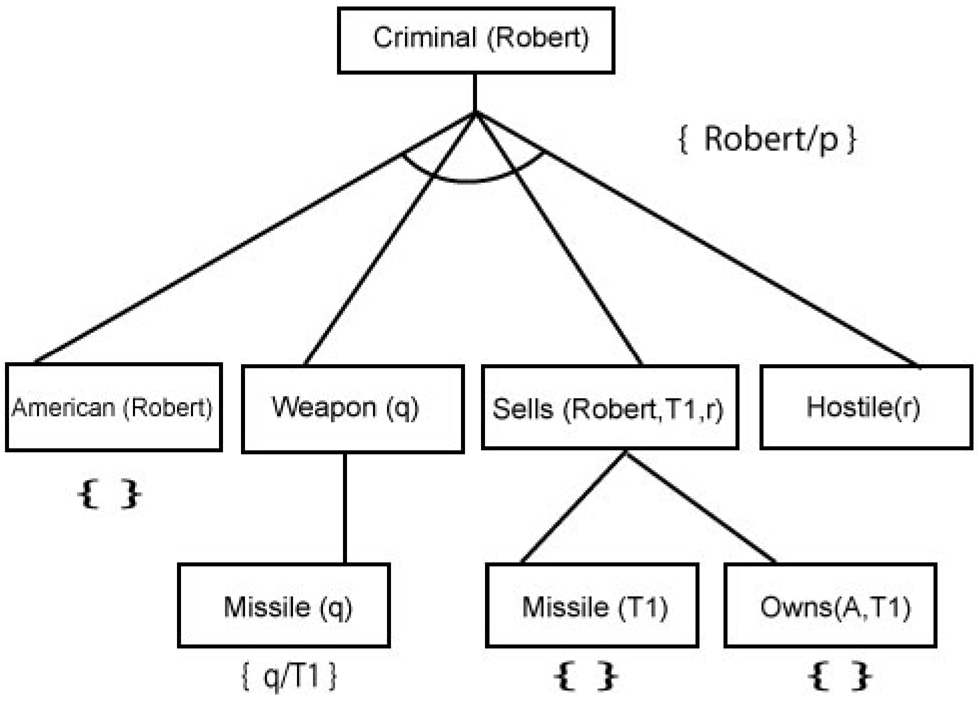


Step-3:t At step-3, we will extract further fact Missile(q) which infer from Weapon(q), as it satisfies Rule-(5). Weapon (q) is also true with the substitution of a constant T1 at q.



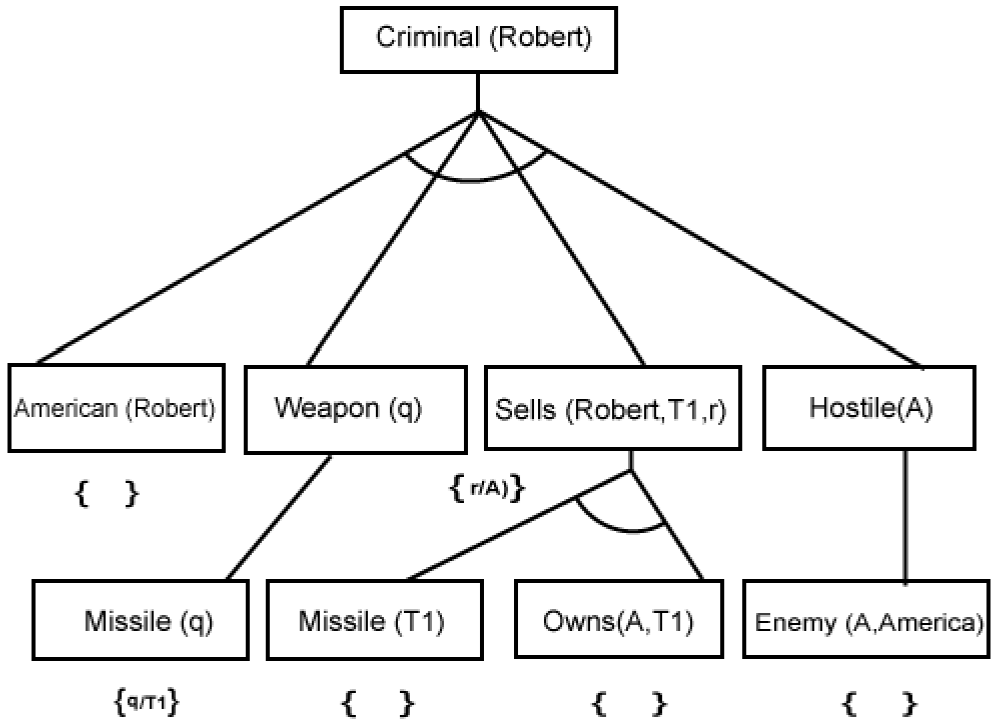
Step-4:

At step-4, we can infer facts Missile(T1) and Owns(A, T1) form Sells(Robert, T1, r) which satisfies the **Rule- 4**, with the substitution of A in place of r. So these two statements are proved here.



Step-5:

At step-5, we can infer the fact **Enemy(A, America)** from **Hostile(A)** which satisfies Rule- 6. And hence all the statements are proved true using backward chaining.



Difference between backward chaining and forward chaining

Following is the difference between the forward chaining and backward chaining:

- Forward chaining as the name suggests, start from the known facts and move forward by applying inference rules to extract more data, and it continues until it reaches to the goal, whereas backward chaining starts from the goal, move backward by using inference rules to determine the facts that satisfy the goal.
- Forward chaining is called a **data-driven** inference technique, whereas backward chaining is called a **goal-driven** inference technique.
- Forward chaining is known as the **down-up** approach, whereas backward chaining is known as a **top-down** approach.
- Forward chaining uses **breadth-first search** strategy, whereas backward chaining uses **depth-first search** strategy.
- Forward and backward chaining both applies **Modus ponens** inference rule.
- Forward chaining can be used for tasks such as **planning, design process monitoring, diagnosis, and classification**, whereas backward chaining can be used for **classification and diagnosis tasks**.
- Forward chaining can be like an exhaustive search, whereas backward chaining tries to avoid the unnecessary path of reasoning.
- In forward-chaining there can be various ASK questions from the knowledge base, whereas in backward chaining there can be fewer ASK questions.
- Forward chaining is slow as it checks for all the rules, whereas backward chaining is fast as it checks few required rules only.

S. No.	Forward Chaining	Backward Chaining
1.	Forward chaining starts from known facts and applies inference rule to extract more data unit it reaches to the goal.	Backward chaining starts from the goal and works backward through inference rules to find the required facts that support the goal.
2.	It is a bottom-up approach	It is a top-down approach

3.	Forward chaining is known as data-driven inference technique as we reach to the goal using the available data.	Backward chaining is known as goal-driven technique as we start from the goal and divide into sub-goal to extract the facts.
4.	Forward chaining reasoning applies a breadth-first search strategy.	Backward chaining reasoning applies a depth-first search strategy.
5.	Forward chaining tests for all the available rules	Backward chaining only tests for few required rules.
6.	Forward chaining is suitable for the planning, monitoring, control, and interpretation application.	Backward chaining is suitable for diagnostic, prescription, and debugging application.
7.	Forward chaining can generate an infinite number of possible conclusions.	Backward chaining generates a finite number of possible conclusions.
8.	It operates in the forward direction.	It operates in the backward direction.
9.	Forward chaining is aimed for any conclusion.	Backward chaining is only aimed for the required data.