

PROBLEM SOLVING METHODS, HEURISTIC SEARCH TECHNIQUES

Search techniques are the general problem solving methods. When there is a formulated search problem, a set of search states, a set of operators, an initial state and a goal criterion we can use search techniques to solve a problem.

3.1 Matching:

Problem solving can be done through search. Search involves choosing among the rules that can be applied at a particular point, the ones that are most likely to lead to a solution. This can be done by extracting rules from large number of collections.

How to extract from the entire collection of rules that can be applied at a given point?

□ This can be done by Matching between current state and the precondition of the rules. One

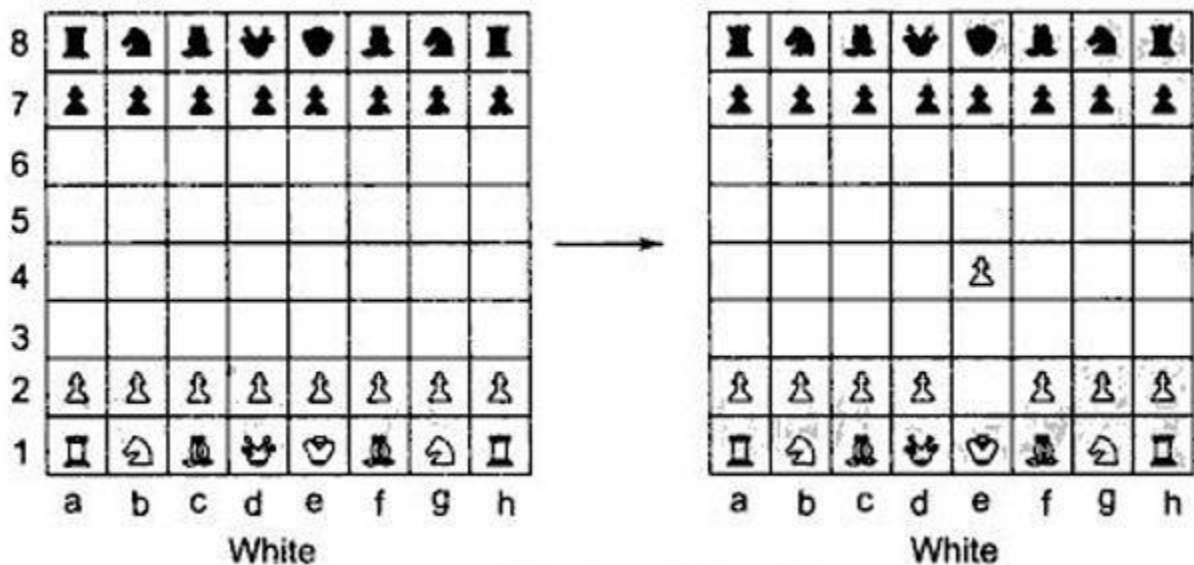
way to select applicable rules is to do simple search through all the rules comparing each one's preconditions to the current state and extracting the one that match. But there are two problems with this simple solution.

1. In big problems large number of rules are used. Scanning through all of this rules at every step of the search would be hopelessly inefficient.
2. It is not clearly visible to find out which condition will be satisfied.

Some of the matching techniques are described below:

3.1.1 Indexing: To overcome above problems indexing is used. In this instead of searching all the rules the current state is used as index into the rules, and select the matching rules immediately e.g consider the chess game playing. Here the set of valid moves is very large. To reduce the size of this set only useful moves are identified. At the time of playing the game, the next move will very much depend upon the current move. As the game is going on there will be only 'few' moves which are applicable in next move. Hence it will be wasteful effort to check applicability of all moves. Rather, the important and valid legal moves are directly stored as rules and through indexing the applicable rules are found. Here, the indexing will store the current board position. The indexing make the matching process easy, at the cost of lack of generality in the statement rules. Practically there is a tradeoff between the ease of writing rules and simplicity of matching process. The indexing technique is not very well suited for the rule base where rules are written in high level predicates. In PROLOG and many theorem proving systems, rules are indexed by predicates they contain. Hence all the applicable rules can be indexed quickly.

Example:



1.8 One Legal Chess Move

3.1.1.1 Matching with variable: In the rule base if the preconditions are not stated as exact description of particular situation, the indexing technique does not work well. In certain situations they describe properties that the situation must have. In situation, where single conditions is matched against a single element in state description, the unification procedure can be used. However in practical situation it is required to match complete set of rules that match the current state. In forward and backward chaining system, the depth first searching technique is used to select the individual rule. In the situation where multiple rules are applicable, conflict resolution technique is used to choose appropriate applicable rule. In the case of the situations requiring multiple match, the unification can be applied recursively, but more efficient method is used to use RETE matching algorithm.

3.1.1.2 Complex matching variable: A more complex matching process is required when preconditions of a rule specify required properties that are not stated explicitly in the description of current state. However the real world is full of uncertainties and sometimes practically it is not possible to define the rule in exact fashion. The matching process become more complicated in the situation where preconditions approximately match the current situation e.g a speech understanding program must contain the rules that map from a description of a physical wave form to phones. Because of the presence of noise the signal becomes more variable that there will be only approximate match between the rules that describe an ideal sound and the input that describes that unideal world. Approximate matching is particularly difficult to deal with, because as we increase the tolerance allowed in the match the new rules need to be written and it will increase number of rules. It will increase the size of main search process. But approximate matching is nevertheless superior to exact matching in situation such as speech understanding, where exact matching may result in no rule being matched and the search process coming to a grinding halt.

3.2 HEURISTIC SEARCH TECHNIQUES

3.2.1 Hill Climbing

- Hill climbing is the optimization technique which belongs to a family of local search. It is relatively simple to implement, making it a popular first choice. Although more advanced algorithms may give better results in some situations hill climbing works well.
- Hill climbing can be used to solve problems that have many solutions, some of which are better than others. It starts with a random (potentially poor) solution, and iteratively makes small changes to the solution, each time improving it a little. When the algorithm cannot see any improvement anymore, it terminates. Ideally, at that point the current solution is close to optimal, but it is not guaranteed that hill climbing will ever come close to the optimal solution.
- For example, hill climbing can be applied to the traveling salesman problem. It is easy to find a solution that visits all the cities but is be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much better route is obtained.
- Hill climbing is used widely in artificial intelligence, for reaching a goal state from a starting node. Choice of next node and starting node can be varied to give a list of related algorithms.
- Hill climbing attempts to maximize (or minimize) a function $f(x)$, where x are discrete states. These states are typically represented by vertices in a graph, where edges in the graph encode nearness or similarity of a graph. Hill climbing will follow the graph from vertex to vertex, always locally increasing (or decreasing) the value of f , until a local maximum (or local minimum) x_m is reached. Hill climbing can also operate on a continuous space: in that case, the algorithm is called gradient ascent (or gradient descent if the function is minimized).*
- Problems with hill climbing: local maxima (we've climbed to the top of the hill, and missed the mountain), plateau (everything around is about as good as where we are), ridges (we're on a ridge leading up, but we can't directly apply an operator to improve our situation, so we have to apply more than one operator to get there). Solutions include: backtracking, making big jumps (to handle plateaus or poor local maxima), applying multiple rules before testing (helps with ridges). Hill climbing is best suited to problems where the heuristic gradually improves the closer it gets to the solution; it works poorly where there are sharp drop-offs. It assumes that local improvement will lead to global improvement.
- Search methods based on hill climbing get their names from the way the nodes are selected for expansion. At each point in the search path a successor node that appears to lead most quickly to the top of the hill (goal) selected for exploration. This method requires that some information be available with which to evaluate and order the most

promising choices. Hill climbing is like depth first searching where the most promising child is selected for expansion.

- Hill climbing is a variant of generate and test in which feedback from the test procedure is used to help the generator decide which direction to move in the search space. Hill climbing is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available. For example, suppose you are in an unfamiliar city without a map and you want to get downtown. You simply aim for the tall buildings. The heuristic function is just distance between the current location and the location of the tall buildings and the desirable states are those in which this distance is minimized.

3.2.1 Simple Hill Climbing

The simplest way to implement hill climbing is the simple hill climbing whose algorithm is as given below:

Algorithm: Simple Hill Climbing:

Step 1: Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise continue with the initial state as the current state.

Step 2: Loop until a solution is found or until there are no new operators left to be applied in the current state:

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

(b) Evaluate the new state.

(i) If it is a goal state, then return it and quit.

(ii) If it is not a goal state, but it is better than the current state, then make it the current state.

(iii) If it is not better than the current state, then continue in the loop.

The key difference between this algorithm and the one we gave for generate and test is the use of an evaluation function as a way to inject task specific knowledge into the control process. It is the use of such knowledge that makes this heuristic search method. It is the same knowledge that gives these methods their power to solve some otherwise intractable problems

To see how hill climbing works, let's take the puzzle of the four colored blocks. To solve the problem we first need to define a heuristic function that describes how close a particular configuration is to being a solution. One such function is simply the sum of the number of different colors on each of the four sides. A solution to the puzzle will have a value of 16. Next we need to define a set of rules that describe ways of transforming one configuration to another. Actually one rule will suffice. It says simply pick a block and rotate it 90 degrees in any direction. Having provided these definitions the next step is to

generate a starting configuration. This can either be done at random or with the aid of the heuristic function. Now by using hill climbing, first we generate a new state by selecting a block and rotating it. If the resulting state is better than we keep it. If not we return to the previous state and try a different perturbation.

3.2.2 Problems in Hill Climbing

3.2.2.1 Steepest – Ascent Hill Climbing:

An useful variation on simple hill climbing considers all the moves from the current state and selects the best one as the next state. This method is called steepest – ascent hill climbing or gradient search. Steepest Ascent hill climbing contrasts with the basic method in which the first state that is better than the current state is selected. The algorithm works as follows.

Algorithm: Steepest – Ascent Hill Climbing

Step 1: Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise, continue with the initial state as the current state.

Step 2: Loop until a solution is found or until a complete iteration produces no change to current state:

(a) Let SUCC be a state such that any possible successor of the current state will be better than SUCC.

(b) For each operator that applies to the current state do:

i. Apply the operator and generate a new state.

ii. Evaluate the new state. If it is a goal state, then return it and quit. If not, compare it to SUCC. If it is better then set SUCC to this state. If it is not better, leave SUCC alone.

(c) If the SUCC is better than current state, then set current state to SUCC.

To apply steepest- ascent hill climbing to the colored blocks problem, we must consider all perturbations of the initial state and choose the best. For this problem this is difficult since there are so many possible moves. There is a trade-off between the time required to select a move and the number of moves required to get a solution that must be considered when deciding which method will work better for a particular problem. Usually the time required to select a move is longer for steepest – ascent hill climbing and the number of moves required to get to a solution is longer for basic hill climbing.

Both basic and steepest ascent hill climbing may fail to find a solution. Either algorithm may terminate not by finding a goal state but by getting to a state from which no better states can be generated. This will happen if the program has reached either a local maximum, a plateau, or a ridge.

Hill climbing Disadvantages

Local Maximum: A local maximum is a state that is better than all its neighbors but is not better than some other states farther away.

Plateau: A Plateau is a flat area of the search space in which a whole set of neighboring states have the same value.

Ridge: A ridge is a special kind of local maximum. It is an area of the search space that is higher than surrounding areas and that itself has a slope.

Ways out

- ☐ Backtrack to some earlier node and try going in a different direction.
- ☐ Make a big jump to try to get in a new section.
- ☐ Moving in several directions at once.

Hill climbing is a **local method**: Decides what to do next by looking only at the immediate consequence of its choice.

Global information might be encoded in heuristic functions.

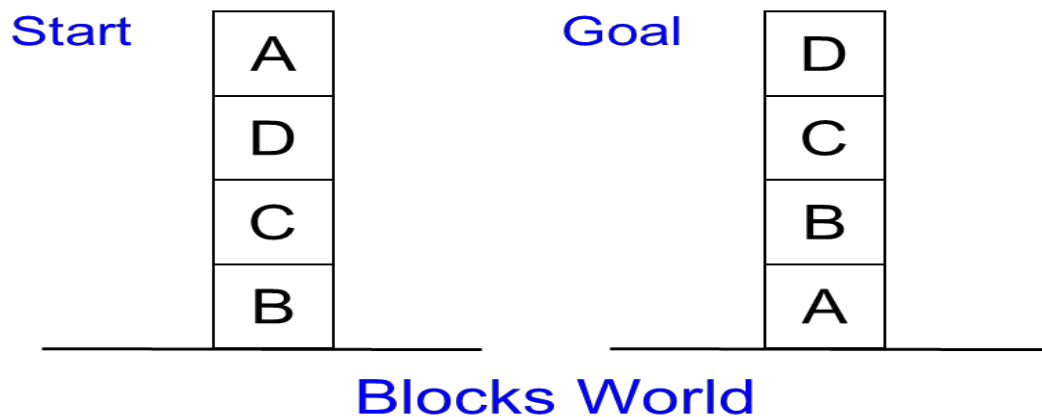
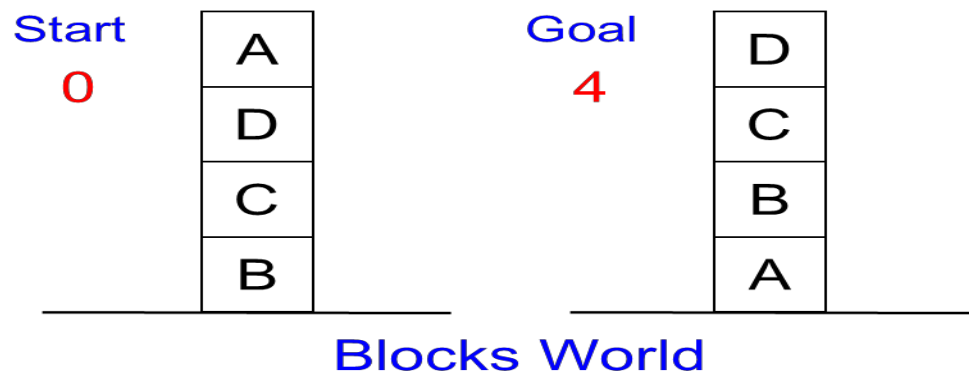
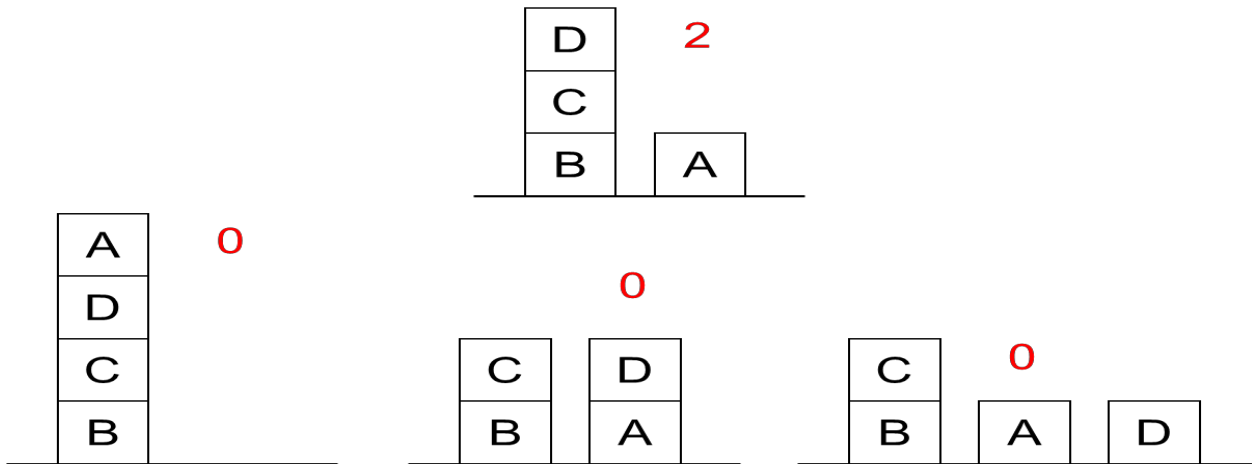
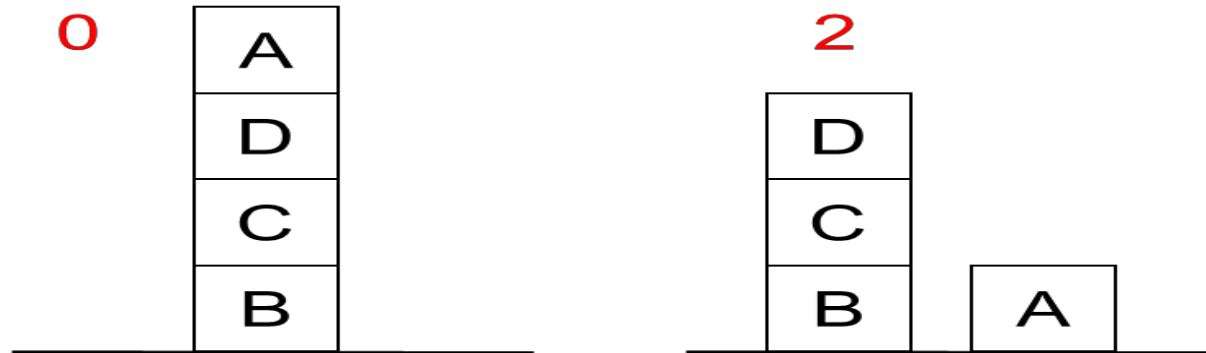


Fig 1.9 Three Possible Moves

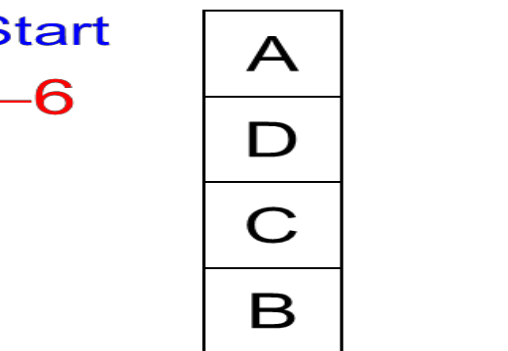


Local heuristic:

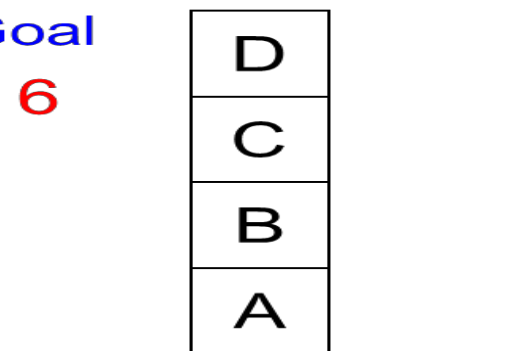
- +1 for each block that is resting on the thing it is supposed to be resting on.
- -1 for each block that is resting on a wrong thing.



Start
-6



Goal
6

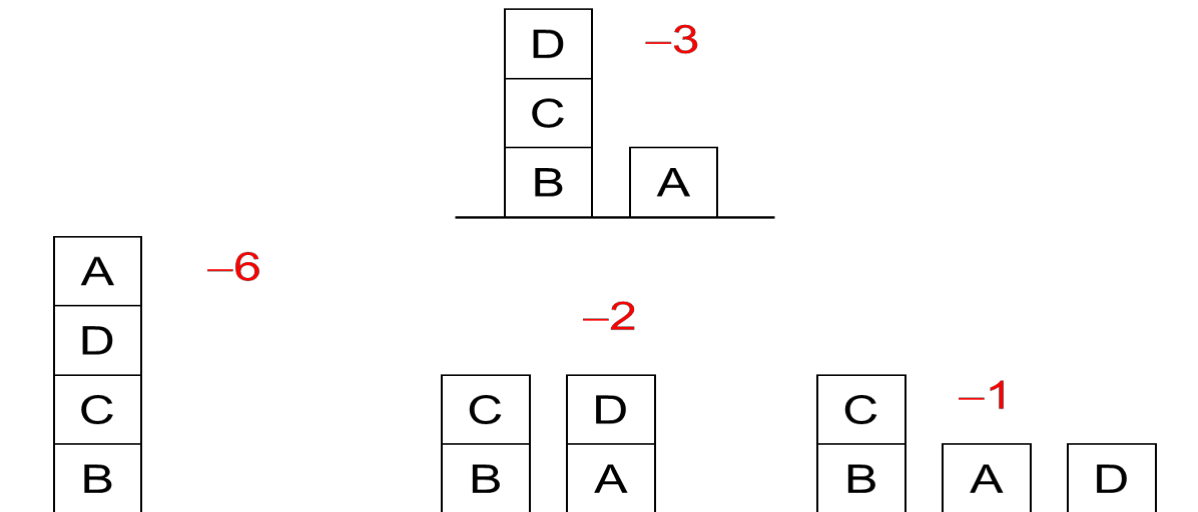


Blocks World

Global heuristic:

For each block that has the correct support structure: +1 to every block in the support structure.

For each block that has a wrong support structure: -1 to every block in the support structure.



Hill climbing conclusion

- ❑ Can be very inefficient in a large, rough problem space.
- ❑ Global heuristic may have to pay for computational complexity.
- ❑ Often useful when combined with other methods, getting it started right in the right general neighbourhood.

3.2.3 Simulated Annealing

The problem of local maxima has been overcome in simulated annealing search. In normal hill climbing search, the movements towards downhill are never made. In such algorithms the search may stuck up to local maximum. Thus this search cannot guarantee complete solutions. In contrast, a random search(or movement) towards successor chosen randomly from the set of successor will be complete, but it will be extremely inefficient. The combination of hill climbing and random search, which yields both efficiency and completeness is called simulated annealing.

The simulated annealing method was originally developed for the physical process of annealing. That is how the name simulated annealing was found and restored. In simulated annealing searching algorithm, instead of picking the best move, a random move is picked. The standard simulated annealing uses term objective function instead of heuristic function. If the move improves the situation it is accepted otherwise the algorithm accepts the move with some probability less than

This probability is

$$P = e^{-\Delta E/kT}$$

Where $-\Delta E$ is positive change in energy level, t is temperature and k is Boltzman constant. As indicated by the equation the probability decreases with badness of the move (evaluation gets worsened by amount $-\Delta E$). The rate at which $-\Delta E$ is cooled is called annealing schedule. The proper annealing schedule is maintained to monitor T .

This process has following differences from hill climbing search:

- ☐ The annealing schedule is maintained.
- ☐ Moves to worse states are also accepted.
- ☐ In addition to current state, the best state record is maintained.

The algorithm of simulated annealing is presented as follows:

Algorithm: “simulated annealing”

1. Evaluate the initial state. Mark it as current state. Till the current state is not a goal state, initialize best state to current state. If the initial state is the best state, return it and quit.
2. Initialize T according to annealing schedule.
3. Repeat the following until a solution is obtained or operators are not left:
 - a. Apply yet unapplied operator to produce a new state
 - b. For new state compute $-\Delta E = \text{value of current state} - \text{value of new state}$. If the new state is the goal state then stop, or if it is better than current state, make it as current state and record as best state.
 - c. If it is not better than the current state, then make it current state with probability P .
 - d. Revise T according to annealing schedule
4. Return best state as answer.

3.3 Best-First Search:

It is a general heuristic based search technique. In best first search, in the graph of problem representation, one evaluation function (which corresponds to heuristic function) is attached with every node. The value of evaluation function may depend upon cost or distance of current node from goal node. The decision of which node to be expanded depends on the value of this evaluation function. The best first can understood from following tree. In the tree, the attached value with nodes indicates utility value. The expansion of nodes according to best first search is illustrated in the following figure.

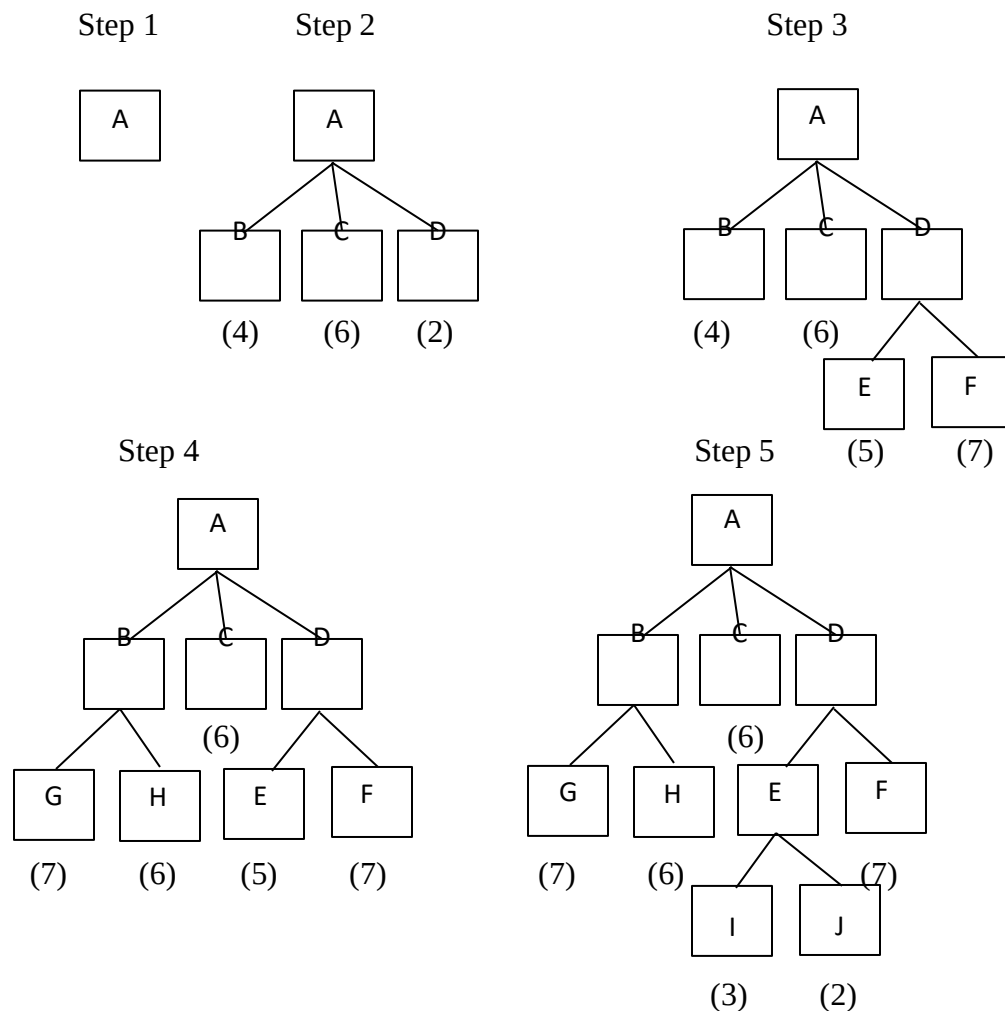


fig:1.10 Tree getting expansion according to best first search

Here, at any step, the most promising node having least value of utility function is chosen for expansion.

In the tree shown above, best first search technique is applied, however it is beneficial sometimes to search a graph instead of tree to avoid the searching of duplicate paths. In the process to do so, searching is done in a directed graph in which each node represents a point in the problem space. This graph is known as OR-graph. Each of the branches of an OR graph represents an alternative problem solving path.

Two lists of nodes are used to implement a graph search procedure discussed above. These are

- 1.OPEN: these are the nodes that have been generated and have had the heuristic function applied to them but not have been examined yet.
- 2.CLOSED: these are the nodes that have already been examined. These nodes are kept in the memory if we want to search a graph rather than a tree because whenever a node will be generated, we will have to check whether it has been generated earlier.

The best first search is a way of combining the advantage of both depth first and breadth first search. The depth first search is good because it allows a solution to be found without all competing branches have to be expanded. Breadth first search is good because it does not get trapped on dead ends of path. The way of combining this is to follow a single path at a time but switches between paths whenever some competing paths looks more promising than current one does. Hence at each step of best first search process, we select most promising node out of successor nodes that have been generated so far.

The functioning of best first search is summarized in the following steps:

1. It maintains a list open containing just the initial state.
2. Until a goal is found or there are no nodes left in open list do:
 - a. Pick the best node from open,
 - b. Generate its successor, and for each successor:
 - i. Check, and if it has not been generated before evaluate it and add it to open and record its parent.
 - ii. If it has been generated before, and new path is better than the previous parent then change the parent.

The algorithm for best first search is given as follows:

Algorithm: Best first search

1. Put the initial node on the list say `_OPEN`.
2. If (`OPEN` = empty or `OPEN` = goal) terminate search, else
3. Remove the first node from open(say node is `a`)
4. If (`a`=goal) terminate search with success else
5. Generate all the successor node of `a`. Send node `a` to a list called `_CLOSED`. Find out the value of heuristic function of all nodes. Sort all children generated so far on the basis of their utility value. Select the node of minimum heuristic value for further expansion.
6. Go back to step 2.

The best first search can be implemented using priority queue. There are variations of best first search. Example of these are greedy best first search, A* and recursive best first search.

3.3.2 The A* Algorithm:

The A* algorithm is a specialization of best first search. Its most widely known form of best first search. It provides general guidelines about how to estimate goal distance for general search graph. At each node along a path to the goal node, the A* algorithm generates all successor nodes and computes an estimate of distance (cost) from the start node to a goal node through each of

the successors. it then chooses the successor with shortest estimated distance from expansion. It calculates the heuristic function based on distance of current node from the start node and distance of current node to goal node.

The form of heuristic estimation function for A* is defined as follows:

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

where $f(n)$ = evaluation function

$g(n)$ = cost (or distance) of current node from start node.

$h(n)$ = cost of current node from goal node.

In A* algorithm the most promising node is chosen from expansion. The promising node is decided based on the value of heuristic function. Normally the node having lowest value of $f(n)$ is chosen for expansion. We must note that the goodness of a move depends upon the nature of problem, in some problems the node having least value of heuristic function would be most promising node, where in some situation, the node having maximum value of heuristic function is chosen for expansion. A* algorithm maintains two lists. One store the list of open nodes and other maintain the list of already expanded nodes. A* algorithm is an example of optimal search algorithm. A search algorithm is optimal if it has admissible heuristic. An algorithm has admissible heuristic if its heuristic function $h(n)$ never overestimates the cost to reach the goal. Admissible heuristic are always optimistic because in them, the cost of solving the problem is less than what actually is. The A* algorithm works as follows:

A* algorithm:

- 1.Place the starting node s on $OPEN$ list.
- 2.If $OPEN$ is empty, stop and return failure.
- 3.Remove from $OPEN$ the node n that has the smallest value of $f^*(n)$. if node n is a goal node, return success and stop otherwise.
- 4.Expand n generating all of its successors n' and place n' on $CLOSED$. For every successor n' if n' is not already $OPEN$, attach a back pointer to n . compute $f^*(n')$ and place it on $CLOSED$.
- 5.Each n' that is already on $OPEN$ or $CLOSED$ should be attached to back pointers which reflect the lowest $f^*(n)$ path. If n' was on $CLOSED$ and its pointer was changed, remove it and place it on $OPEN$.
- 6.Return to step 2.

3.4 Constraint Satisfaction

The general problem is to find a solution that satisfies a set of constraints. The heuristics which are used to decide what node to expand next and not to estimate the distance to the goal. Examples of this technique are design problem, labeling graphs robot path planning and crypt arithmetic puzzles.

In constraint satisfaction problems a set of constraints are available. This is the search space. Initial State is the set of constraints given originally in the problem description. A goal state is any state that has been constrained enough. Constraint satisfaction is a two-step process.

1. First constraints are discovered and propagated throughout the system.
2. Then if there is not a solution search begins, a guess is made and added to this constraint. Propagation then occurs with this new constraint.

Algorithm

1. Propagate available constraints:

- ☐ Open all objects that must be assigned values in a complete solution.
- ☐ Repeat until inconsistency or all objects are assigned valid values:

Select an object and strengthen as much as possible the set of constraints that apply to object.

If set of constraints different from previous set then open all objects that share any of these constraints. Remove selected object.

2. If union of constraints discovered above defines a solution return solution.

3. If union of constraints discovered above defines a contradiction return failure.

4. Make a guess in order to proceed. Repeat until a solution is found or all possible solutions exhausted:

- ☐ Select an object with a no assigned value and try to strengthen its constraints.
- ☐ Recursively invoke constraint satisfaction with the current set of constraints plus the selected strengthening constraint.

Crypt arithmetic puzzles are examples of constraint satisfaction problems in which the goal to discover some problem state that satisfies a given set of constraints. some problems of crypt arithmetic are show below

$$\begin{array}{r} \text{S E N D} \\ + \text{M O R E} \\ \hline \text{M O N E Y} \end{array} \quad \begin{array}{r} \text{D O N A L D} \\ + \text{G E R A L D} \\ \hline \text{R O B E R T} \end{array} \quad \begin{array}{r} \text{C R O S S} \\ + \text{R O A D S} \\ \hline \text{D A N G E R} \end{array}$$

Here each decimal digit is to be assigned to each of the letters in such a way that the answer to the problem is correct. If the same letter occurs more than once it must be assigned the same digit each time. No two different letters may be assigned the same digit.

The puzzle $SEND + MORE = MONEY$, after solving, will appear like this

$$\begin{array}{r}
 \text{S E N D} \\
 9 \ 5 \ 6 \ 7 \\
 + \text{M O R E} \\
 1 \ 0 \ 8 \ 5 \\
 \hline
 \text{M O N E Y} \\
 1 \ 0 \ 6 \ 5 \ 2
 \end{array}$$

State production and heuristics for crypt arithmetic problem.

Ans.

The heuristics and production rules are specific to the following example:

$$\begin{array}{r}
 \text{S E N D} \\
 \text{M O R E} \\
 \hline
 \text{M O N E Y}
 \end{array}$$

Heuristics Rules

1. If sum of two $_{n-1}$ digit operands yields $_{n+1}$ digit result then the $_{n+1}$ th digit has to be one.
2. Sum of two digits may or may not generate carry.
3. Whatever might be the operands the carry can be either 0 or 1.
4. No two distinct alphabets can have same numeric code.
5. Whenever more than 1 solution appears to be existing, the choice is governed by the fact that no two alphabets can have same number code.

Production Rules:

1. $x + y = yz$ $\rightarrow y = 1, z = 0, x = 9$
2. $ax + 0y = cz$ $\rightarrow c = a + 1, x + y = z + 10$
3. $y = x + 1, y + z = x + 10$ $\rightarrow z = 8 \text{ or } 9 \text{ and } \text{prev_carry} = 1$
4. $x + y = z + 10$ $\rightarrow (x, y) = (8, 9) \mid (7, 9) \mid (6, 9) \mid (5, 9) \mid (4, 9) \mid (3, 9) \mid (2, 9) \mid (1, 9) \mid (7, 8) \mid (6, 8) \mid (5, 8) \mid (4, 8) \mid (3, 8) \mid (2, 8) \mid (6, 7) \mid (5, 7) \mid (4, 7) \mid (3, 7) \mid (5, 6) \mid (4, 6).$

To solve the problem $SEND + MORE = MONEY$ using the above rules we proceed

as follows:

Applying rule 1

$$x = S, Y = M, z = 0$$

$$S + M = MO \rightarrow M = 1, O = 0 \text{ and } S = 8 \text{ or } 9$$

Applying rule 2

$$a = E, x = N, y = R, c = N, z = E$$

$$EN + OR = NE \rightarrow N = E + 1, N + R = E + 10$$

Also, as $E + O + 1 = N$ and $S + M$ generates carry

$$S = 9 \text{ and not } 8$$

Applying rule 3

$$y = N, x = E, z = R$$

$$N = E + 1, N + R = E + 10 \rightarrow R = 8 \text{ (as } S = 9)$$

$$\text{prev_carry} = 1$$

$$\text{prev_carry} = 1 \rightarrow D + E = Y + 10$$

Applying rule 4

$$x = D, y = E, z = Y$$

$$D + E = Y + 10 \rightarrow (D, E) = (7, 5)$$

We conclude that $(D, E) = (7, 5)$ because any other choice always leads to violation of the constraint that - no two distinct alphabates can have same numeric code.

Thus $D = 7$ and $E = 5$

$$\text{Also, as } N = E + 1 = 5 + 1 = 6$$

$$\text{Also, as } D + E = Y + 10 \Rightarrow Y = 7 + 5 - 10 = 2$$

Thus,

$$S = 9, E = 5, N = 6, D = 7,$$

$$+ M = 1, O = 0, R = 8, E = 5,$$

$$M = 1, O = 0, N = 6, E = 5, Y = 2.$$

3.5 Means – end Analysis

Means-ends analysis allows both backward and forward searching. This means we could solve major parts of a problem first and then return to smaller problems when assembling the final solution.

The means-ends analysis algorithm can be said as follows:

1. Until the goal is reached or no more procedures are available:

- ☐ Describe the current state the goal state and the differences between the two.
- ☐ Use the difference the describes a procedure that will hopefully get nearer to goal.
- ☐ Use the procedure and update current state

If goal is reached then success otherwise fail.

For using means-ends analysis to solve a given problem, a mixture of the two directions, forward and backward, is appropriate. Such a mixed strategy solves the major parts of a problem first and then goes back and solves the small problems that arise by putting the big pieces together. The means-end analysis process detects the differences between the current state and goal state. Once such difference is isolated an operator that can reduce the difference has to be found. The operator may or may not be applied to the current state. So a sub problem is set up of getting to a state in which this operator can be applied.

In operator sub goaling backward chaining and forward chaining is used in which first the operators are selected and then sub goals are set up to establish the preconditions of the operators. If the operator does not produce the goal state we want, then we have second sub problem of getting from the state it does produce to the goal. The two sub problems could be easier to solve than the original problem, if the difference was chosen correctly and if the operator applied is really effective at reducing the difference. The means-end analysis process can then be applied recursively.

This method depends on a set of rules that can transform one problem state into another. These rules are usually not represented with complete state descriptions on each side. Instead they are represented as a left side that describes the conditions that must be met for the rules to be applicable and a right side that describes those aspects of the problem state that will be changed by the application of the rule. A separate data structure called a difference table which uses the rules by the differences that they can be used to reduce.

Example Solve the following problem using means-ends analysis

A farmer wants to cross the river along with fox, chicken and grain. He can take only one along with him at a time. If fox and chicken are left alone the fox may eat the chicken. If chicken and grain are left alone the chicken may eat the grain. Give the necessary solution.

Solution:

	Operator	Preconditions	Results
1.	Arrive (location)	—	At(farmer,chicken,fox, grain,location)
2.	select(object,source)	At(object,source)	At(farmer,object)
3.	Go(object,source)	At(farmer,object) ^ At(object,source) ^ Boat(source)	At(obj,farmer,dest)
4.	Keep(object,destination)	At(object,destination)	At(object,destination)
5.	Come(object, destination)	At(farmer,object) ^ At(object,destination) ^ Boat(destination)	At(chicken ,destination) ^ At(farmer,source)
6.	At(farmer,chicken,fox,grain,location)	At(all,destination)	At(all,home)

Difference table.

	Actions	Arrive	Select	GO	Keep	Come
1.	Arrive at the location	*				
2.	Take Chicken		*			
3.	Travel by boat to destination			*		
4.	Leave chicken at destination				*	
5.	Come back at source					*
6.	Take fox	*				
7.	Travel by boat to destination		*			
8.	Keep fox at destination and take chicken back to source.			*	*	
9.	Keep chicken at source and take grain to destination.		*	*		
10.	Keep grain to destination			*		

Unit -3

Using predicates logic :

Computable functions and predicates

- ☐ Some of the computational predicates like Less than, Greater than used in knowledge representation.
- ☐ It generally return true or false for the inputs.
Examples: Computable predicates
 $gt(1,0)$ or $lt(0,1)$

$gt(5,4)$ or $gt(4,5)$

Computable functions: $gt(2+4, 5)$

Consider the following set of facts, again involving Marcus:

1. marcus was a man

$man(Marcus)$

2. Marcus was a pompeian

$Pompeian(Marcus)$

3. Marcus was born in 40 A.D

$born(marcus, 40)$

4. All men are mortal

$\forall x: men(x) \rightarrow mortal(x)$

5. All Pompeians died when the volcano erupted in 79 A.D

$erupted(volcano, 79) \ \& \ x : pompeian(x) \rightarrow died(x, 79)$

6. No mortal lives longer than 150 years

$\forall x: \forall t1: \forall t2: mortal(x) \ \& \ born(x, t1) \ \& \ gt(t2-t1, 150) \rightarrow dead(x, t1)$

7. It is Now 1991

$Now=1991$

8. Alive means not dead

$\forall x: \forall t: [alive(x, t) \rightarrow \sim dead(x, t)] \ \& \ [\sim dead(x, t) \rightarrow alive(x, t)]$

9. If someone dies then he is dead at all later times

$\forall x: \forall t1: \forall t2: died(x, t1) \ \& \ gt(t2, t1) \rightarrow dead(x, t2)$

This representation says that one is dead in all years after the one in which one died. It ignores the question of whether one is dead in the year in which one died.

1. $man(Marcus)$

2. $Pompeian(Marcus)$

3. $born(marcus, 40)$

4. $\forall x: men(x) \rightarrow mortal(x)$

5. $\forall x: \text{pompeian}(x) \rightarrow \text{died}(x, 79)$

6. $\text{erupted}(\text{volcano}, 79)$

7. $\forall x: \forall t_1: \forall t_2: \text{mortal}(x) \ \& \ \text{born}(x, t_1) \ \& \ \text{gt}(t_2 - t_1, 150) \rightarrow$
 $\text{dead}(x, t_1)$

8. $\text{Now} = 1991$

9. $\forall x: \forall t: [\text{alive}(x, t) \rightarrow \sim \text{dead}(x, t)] \ \& \ [\sim \text{dead}(x, t) \rightarrow \text{alive}(x, t)]$

10. $\forall x: \forall t_1: \forall t_2: \text{died}(x, t_1) \ \& \ \text{gt}(t_2, t_1) \rightarrow \text{dead}(x, t_2)$

$\neg \text{alive}(\text{Marcus}, \text{now})$
 $\uparrow$ (9, substitution)
 $\text{dead}(\text{Marcus}, \text{now})$
 $\uparrow$ (10, substitution)
 $\text{died}(\text{Marcus}, t_1) \wedge \text{gt}(\text{now}, t_1)$
 $\uparrow$ (5, substitution)
 $\text{Pompeian}(\text{Marcus}) \wedge \text{gt}(\text{now}, 79)$
 $\uparrow$ (2)
 $\text{gt}(\text{now}, 79)$
 $\uparrow$ (8, substitute equals)
 $\text{gt}(1991, 79)$
 $\uparrow$ (compute gt)
 nil

One Way of Proving That Marcus Is Dead

$$\begin{array}{rcl}
\neg \text{alive}(\text{Marcus}, \text{now}) & & \\
\uparrow & (9, \text{substitution}) & \\
\text{dead}(\text{Marcus}, \text{now}) & & \\
\uparrow & (7, \text{substitution}) & \\
\text{mortal}(\text{Marcus}) \wedge & & \\
\text{born}(\text{Marcus}, t_1) \wedge & & \\
\text{gt}(\text{now} - t_1, 150) & & \\
\uparrow & (4, \text{substitution}) & \\
\text{man}(\text{Marcus}) \wedge & & \\
\text{born}(\text{Marcus}, t_1) \wedge & & \\
\text{gt}(\text{now} - t_1, 150) & & \\
\uparrow & (1) & \\
\text{born}(\text{Marcus}, t_1) \wedge & & \\
\text{gt}(\text{now} - t_1, 150) & & \\
\uparrow & (3) & \\
\text{gt}(\text{now} - 40, 150) & & \\
\uparrow & (8) & \\
\text{gt}(1991 - 40, 150) & & \\
\uparrow & (\text{compute minus}) & \\
\text{gt}(1951, 150) & & \\
\uparrow & (\text{compute gt}) & \\
\text{nil} & &
\end{array}$$

Another Way of Proving That Marcus is Dead

Two things should be clear from the proofs we have just shown:

- ☐ Even very simple conclusions can require many steps to prove.
- ☐ A variety of processes, such as matching, substitution, and application of modus ponens are involved in the production of a proof, This is true even for the simple statements we are using, It would be worse if we had implications with more than a single term on the right or with complicated expressions involving ands and ors on the left.

Disadvantage:

- ☐ Many steps required to prove simple conclusions
- ☐ Variety of processes such as matching and substitution used to prove simple conclusions

2.5 Resolution

- ☐ Resolution is a proof procedure by refutation.
- ☐ To prove a statement using resolution it attempt to show that the negation of that statement.

Algorithm: Convert to Clause Form

1. Eliminate $\rightarrow$, using the fact that $a \rightarrow b$ is equivalent to $\neg a \vee b$. Performing this transformation on the wff given above yields

$$\forall x: \neg [\text{Roman}(x) \wedge \text{know}(x, \text{Marcus})] \vee$$
$$[\text{hate}(x, \text{Caesar}) \vee (\forall y : \neg(\exists z : \text{hate}(y, z)) \vee \text{thinkcrazy}(x, y))]$$

2. Reduce the scope of each $\neg$ to a single term, using the fact that $\neg(\neg p) = p$, deMorgan's laws [which say that $\neg(a \wedge b) = \neg a \vee \neg b$ and $\neg(a \vee b) = \neg a \wedge \neg b$], and the standard correspondences between quantifiers [$\neg \forall x: P(x) = \exists x: \neg P(x)$ and $\neg \exists x: P(x) = \forall x: \neg P(x)$]. Performing this transformation on the wff from step 1 yields

$$\forall x: [\neg \text{Roman}(x) \vee \neg \text{know}(x, \text{Marcus})] \vee$$
$$[\text{hate}(x, \text{Caesar}) \vee (\forall y: \forall z: \neg \text{hate}(y, z) \vee \text{thinkcrazy}(x, y))]$$

3. Standardize variables so that each quantifier binds a unique variable. Since variables are just dummy names, this process cannot affect the truth value of the wff. For example, the formula

$$\forall x: P(x) \vee \forall x: Q(x)$$

would be converted to

$$\forall x: P(x) \vee \forall y: Q(y)$$

4. Move all quantifiers to the left of the formula without changing their relative order. This is possible since there is no conflict among variable names. Performing this operation on the formula of step 2, we get

$$\forall x: \forall y: \forall z: [\neg \text{Roman}(x) \vee \neg \text{know}(x, \text{Marcus})] \vee$$
$$[\text{hate}(x, \text{Caesar}) \vee (\neg \text{hate}(y, z) \vee \text{thinkcrazy}(x, y))]$$

At this point, the formula is in what is known as prenex normal form. It consists of a prefix of quantifiers followed by a matrix, which is quantifier-free.

5. Eliminate existential quantifiers. A formula that contains an existentially quantified variable asserts that there is a value that can be substituted for the variable that makes the formula true. We can eliminate the quantifier by substituting for the variable a reference to a function that produces the desired value. Since we do not necessarily know how to produce the value, we must create a new function name for every such replacement. We make no assertions about these functions except that they must exist. So, for example, the formula

$$\exists y : \text{President}(y)$$

can be transformed into the formula

President(S1)

where S1 is a function with no arguments that somehow produces a value that satisfies President. If existential quantifiers occur within the scope of universal quantifiers, then the value that satisfies the predicate may depend on the values of the universally quantified variables. For example, in the formula

$\forall x: \exists y: \text{father-of}(y, x)$

the value of y that satisfies father-of depends on the particular value of x. Thus we must generate functions with the same number of arguments as the number of universal quantifiers in whose scope the expression occurs. So this example would be transformed into

$\forall x: \text{father-of}(S2(x), x)$

These generated functions are called Skolem functions. Sometimes ones with no arguments are called Skolem constants.

6. Drop the prefix. At this point, all remaining variables are universally quantified, so the prefix can just be dropped and any proof procedure we use can simply assume that any variable it sees is universally quantified. Now the formula produced in step 4 appears as

$[\neg \text{Roman}(x) \vee \neg \text{know}(x, \text{Marcus})] \vee$
 $[\text{hate}(x, \text{Caesar}) \vee (\neg \text{hate}(y, z) \vee \text{thinkcrazy}(x, y))]$

7. Convert the matrix into a conjunction of disjuncts. In the case of our example, since there are no and's, it is only necessary to exploit the associative property of or [i.e., $(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$] and simply remove the parentheses, giving

$\neg \text{Roman}(x) \vee \neg \text{know}(x, \text{Marcus}) \vee$
 $\text{hate}(x, \text{Caesar}) \vee \neg \text{hate}(y, z) \vee \text{thinkcrazy}(x, y)$

However, it is also frequently necessary to exploit the distributive property [i.e., $(a \wedge b) \vee c = (a \vee c) \wedge (b \vee c)$]. For example, the formula

$(\text{winter} \wedge \text{wearingboots}) \vee (\text{summer} \wedge \text{wearingsandals})$

Becomes, after one application of the rule

$[\text{winter} \vee (\text{summer} \wedge \text{wearingsandals})]$
 $\wedge [\text{wearingboots} \vee (\text{summer} \wedge \text{wearingsandals})]$

And then, after a second application, required since there are still conjuncts joined by OR's,

(winter $\vee$ summer) $\wedge$

(winter $\vee$ wearingsandals) $\wedge$

(wearingboots $\vee$ summer) $\wedge$

(wearingboots $\vee$ wearingsandals)

8. Create a separate clause corresponding to each conjunct. In order for a wff to be true, all the clauses that are generated from it must be true. If we are going to be working with several wff's, all the clauses generated by each of them can now be combined to represent the same set of facts as were represented by the original wff's.

9. Standardize apart the variables in the set of clauses generated in step 8. By this we mean rename the variables so that no two clauses make reference to the same variable. In making this transformation, we rely on the fact that

$$(\forall x: P(x) \wedge Q(x)) = \forall x: P(x) \wedge \forall x: Q(x)$$

Thus since each clause is a separate conjunct and since all the variables are universally quantified, there need be no relationship between the variables of two clauses, even if they were generated from the same wff.

Performing this final step of standardization is important because during the resolution procedure it is sometimes necessary to instantiate a universally quantified variable (i.e., substitute for it a particular value). But, in general, we want to keep clauses in their most general form as long as possible. So when a variable is instantiated, we want to know the minimum number of substitutions that must be made to preserve the truth value of the system.

After applying this entire procedure to a set of wff's, we will have a set of clauses, each of which is a disjunction of *literals*. These clauses can now be exploited by the resolution procedure to generate proofs.

2.5.1 Resolution in Propositional Logic

In propositional logic, the procedure for producing a proof by resolution of proposition P with respect to a set of axioms F is the following.

ALGORITHM: PROPOSITIONAL RESOLUTION

1. Convert all the propositions of F to clause form
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found or no progress can be made:
 - a) Select two clauses. Call these the parent clauses.

b) Resolve them together. The resulting clause, called the resolvent, will be the disjunction of all of the literals of both of the parent clauses with the following exception: If there are any pairs of literals L and $\neg L$ such that one of the parent clauses contains L and the other contains $\neg L$, then select one such pair and eliminate both L and $\neg L$ from the resolvent.

c) If the resolvent is the empty clause, then a contradiction has been found. If it is not then add it to the set of clauses available to the procedure.

Suppose we are given the axioms shown in the first column of Table 1 and we want to prove R . First we convert the axioms to clause which is already in clause form. Then we begin selecting pairs of clauses to resolve together. Although any pair of clauses can be resolved, only those pairs that contain complementary literals will produce a resolvent that is likely to lead to the goal of sequence of resolvents shown in figure 1. We begin by resolving with the clause ϕR since that is one of the clauses that must be involved in the contradiction we are trying to find.

One way of viewing the resolution process is that it takes a set of clauses that are all assumed to be true and based on information provided by the others, it generates new clauses that represent restrictions on the way each of those original clauses can be made true. A contradiction occurs when a clause becomes so restricted that there is no way it can be true. This is indicated by the generation of the empty clause.

Sl. No	Given Axioms	Converted to Clause Form
1	P	P
2	$(P \wedge Q) \rightarrow R$	$\neg P \vee \neg Q \vee R$
3	$(S \vee T) \rightarrow Q$	$\neg S \vee \neg T \vee Q$
4		$\neg T \vee Q$
5	T	T

Table 1 Some Facts in Propositional Logic

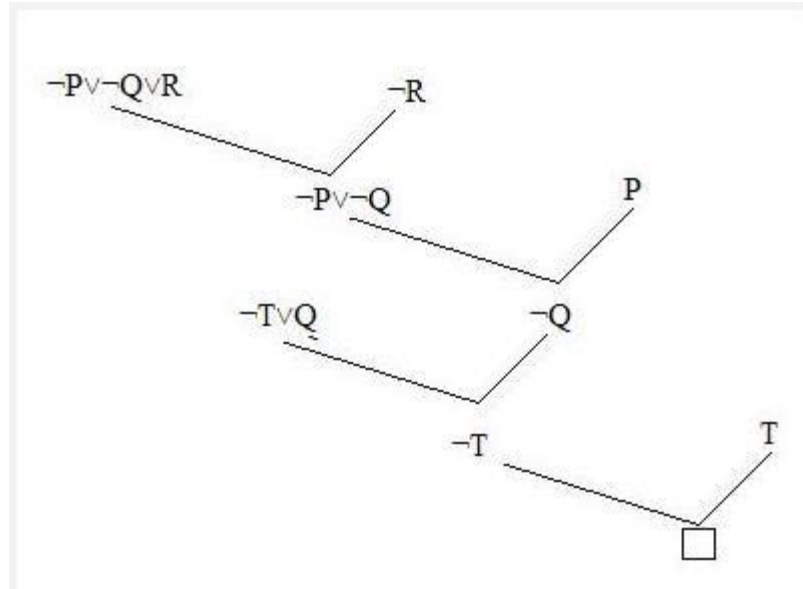


Figure : Resolution In Propositional Logic

2.5.3 UNIFICATION ALGORITHM

In propositional logic it is easy to determine that two literals can not both be true at the same time. Simply look for L and $\sim L$. In predicate logic, this matching process is more complicated, since bindings of variables must be considered.

For example $\text{man}(\text{john})$ and $\text{man}(\text{john})$ is a contradiction while $\text{man}(\text{john})$ and $\text{man}(\text{Himalayas})$ is not. Thus in order to determine contradictions we need a matching procedure that compares two literals and discovers whether there exist a set of substitutions that makes them identical. There is a recursive procedure that does this matching. It is called Unification algorithm.

In Unification algorithm each literal is represented as a list, where first element is the name of a predicate and the remaining elements are arguments. The argument may be a single element (atom) or may be another list. For example we can have literals as

(tryassassinate Marcus Caesar)

(tryassassinate Marcus (ruler of Rome))

To unify two literals, first check if their first elements are same. If so proceed. Otherwise they can not be unified. For example the literals

(tryassassinate Marcus Caesar)

(hate Marcus Caesar)

Can not be Unfied. The unification algorithm recursively matches pairs of elements, one pair at a time. The matching rules are :

- i) Different constants , functions or predicates can not match, whereas identical ones can.
- ii) A variable can match another variable , any constant or a function or predicate expression, subject to the condition that the function or [predicate expression must not contain any instance of the variable being matched (otherwise it will lead to infinite recursion).
- iii) The substitution must be consistent. Substituting y for x now and then z for x later is inconsistent. (a substitution y for x written as y/x)

The Unification algorithm is listed below as a procedure UNIFY (L1, L2). It returns a list representing the composition of the substitutions that were performed during the match. An empty list NIL indicates that a match was found without any substitutions. If the list contains a single value F, it indicates that the unification procedure failed.

The empty list, NIL, indicates that a match was found without any substitutions. The list consisting of the single value FAIL indicates that the unification procedure failed.

Algorithm: Unify(L1, L2)

1. If L1 or L2 are both variables or constants, then:

- (a) If L1 and L2 are identical, then return NIL.
- (b) Else if L1 is a variable, then if L1 occurs in L2 then return {FAIL}, else return (L2/L1).
- (c) Else if L2 is a variable, then if L2 occurs in L1 then return {FAIL} , else return (L1/L2).
- (d) Else return {FAIL}.

2. If the initial predicate symbols in L1 and L2 are not identical, then return {FAIL}.

3. If L1 and L2 have a different number of arguments, then return {FAIL}.

4. Set SUBST to NIL. (At the end of this procedure, SUBST will contain all the substitutions used to unify L1 and L2.)

5. For $i \leftarrow 1$ to number of arguments in L1 :

- (a) Call Unify with the i th argument of L1 and the i th argument of L2, putting result in S.
- (b) If S contains FAIL then return {FAIL}.
- (c) If S is not equal to NIL then:
 - (i) Apply S to the remainder of both L1 and L2.

(ii) SUBST: = APPEND(S, SUBST).

6. Return SUBST.

2.5.4 Resolution in Predicate Logic

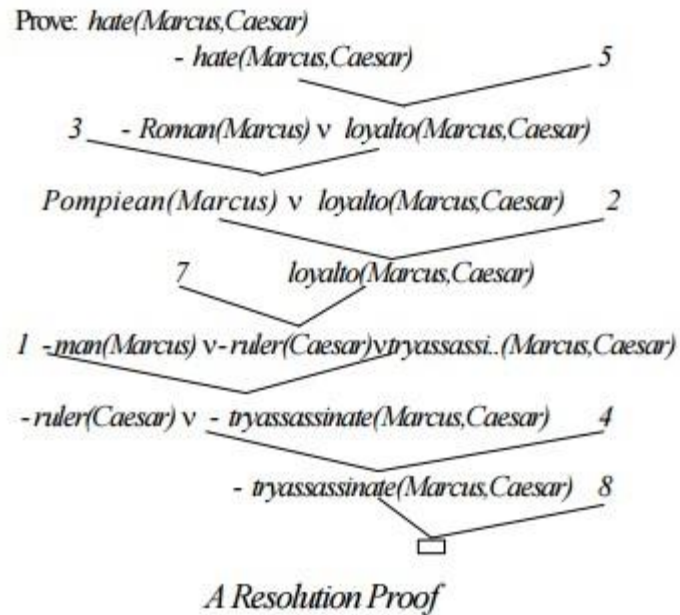
ALGORITHM: RESOLUTION IN PREDICATE LOGIC

1. Convert all the statements of F to clause form
2. Negate P and convert the result to clause form. Add it to the set of clauses obtained in step 1.
3. Repeat until either a contradiction is found or no progress can be made or a predetermined amount of effort has been expended:
 - a) Select two clauses. Call these the parent clauses.
 - b) Resolve them together. The resulting clause, called the resolvent, will be the disjunction of all of the literals of both the parent clauses with appropriate substitutions performed and with the following exception: If there is one pair of literals T1 and T2 such that one of the parent clauses contains T1 and the other contains T2 and if T1 and T2 are unifiable, then neither T1 nor T2 should appear in the resolvent. We call T1 and T2 complementary literals. Use the substitution produced by the unification to create the resolvent. If there is one pair of complementary literals, only one such pair should be omitted from the resolvent.
 - c) If the resolvent is the empty clause, then a contradiction has been found. If it is not then add it to the set of clauses available to the procedure.

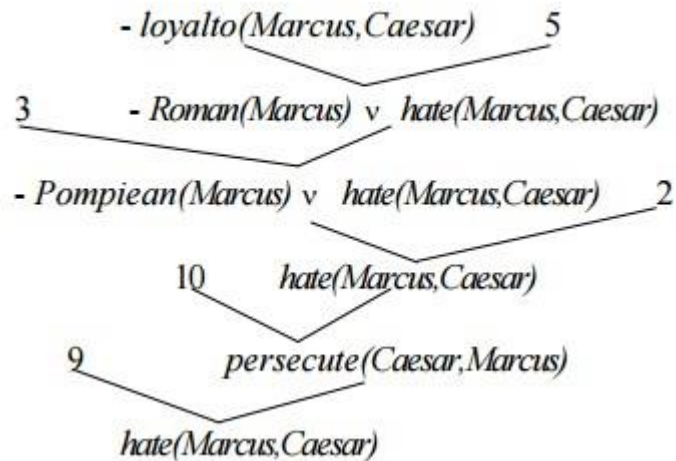
If the choice of clauses to resolve together at each step is made in certain systematic ways, then the resolution procedure will find a contradiction if one exists. However, it may take a very long time. There exist strategies for making the choice that can speed up the process considerably as given below.

Axioms in clause form

1. *man(Marcus)*
 2. *Pompiean(Marcus)*
 3. *- Pompiean(x1) v Roman(x1)*
 4. *ruler(Caesar)*
 5. *- Roman(x2) v loyalto(x2,Caesar) v hate(x2,Caesar)*
 6. *loyal(x3,f(x3))*
 7. *- man(x4) v - ruler(y1) v - tryassassinate(x4,y1) v loyalto(x4,y1)*
 8. *tryassassinate(Marcus,Caesar)*
-



Prove: $\text{loyalto}(\text{Marcus}, \text{Caesar})$



An Unsuccessful Attempt at Resolution

9. $\text{persecute}(x, y) \sqcup \text{hate}(y, x)$

10. $\text{hate}(x, y) \sqcup \text{persecute}(y, x)$

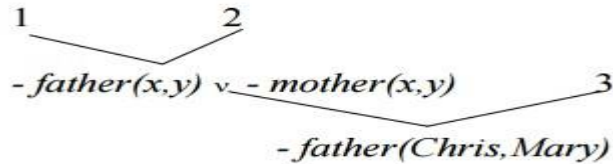
Converting to clause form, we get

9. $\sqcup \text{persecute}(x_5, y_2) \sqcup \text{hate}(y_2, x_5)$

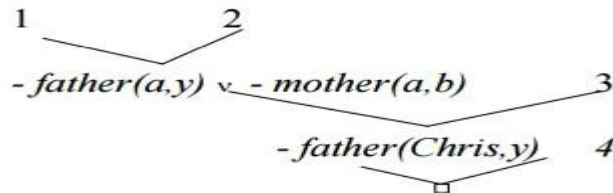
10. $\sqcup \text{hate}(x_6, y_3) \sqcup \text{persecute}(y_3, x_6)$

Given

1. $father(x,y) \vee \neg women(x)$
2. $mother(x,y) \vee women(x)$
3. $mother(Chris, Mary)$
4. $father(Chris, Bill)$



The need to Standardize Variables



2.5.5 Procedural v/s Declarative Knowledge

- ☐ A Declarative representation is one in which knowledge is specified but the use to which that knowledge is to be put in, is not given.
- ☐ A Procedural representation is one in which the control information that is necessary to use the knowledge is considered to be embedded in the knowledge itself.
- ☐ To use a procedural representation, we need to augment it with an interpreter that follows the instructions given in the knowledge.
- ☐ The difference between the declarative and the procedural views of knowledge lies in where control information resides.

man(Marcus)

man(Caesar)

person(Cleopatra)

$\forall x: man(x) \rightarrow person(x)$

person(x)?

Now we want to extract from this knowledge base the ans to the question:

$\exists y : \text{person}(y)$

Marcus, Ceaser and Cleopatra can be the answers

- As there is more than one value that satisfies the predicate, but only one value is needed, the answer depends on the order in which the assertions are examined during the search of a response.
- If we view the assertions as declarative, then we cannot depict how they will be examined. If we view them as procedural, then they do.
- Let us view these assertions as a non deterministic program whose output is simply not defined, now this means that there is no difference between Procedural & Declarative Statements. But most of the machines don't do so, they hold on to what ever method they have, either sequential or in parallel.
- The focus is on working on the control model.

$\text{man}(\text{Marcus})$

$\text{man}(\text{Ceaser})$

$\forall x : \text{man}(x)$

$\text{person}(x)$

$\text{Person}(\text{Cleopatra})$

- If we view this as declarative then there is no difference with the previous statement. But viewed procedurally, and using the control model, we used to get Cleopatra as the answer, now the answer is marcus.
- The answer can vary by changing the way the interpreter works.
- The distinction between the two forms is often very fuzzy. Rather than trying to prove which technique is better, what we should do is to figure out what the ways in which rule formalisms and interpreters can be combined to solve problems.

2.5.6 Logic Programming

- Logic programming is a programming language paradigm in which logical assertions are viewed as programs, e.g : PROLOG
 - A PROLOG program is described as a series of logical assertions, each of which is a Horn Clause.
 - A Horn Clause is a clause that has at most one positive literal.
-

- Eg $p, \neg p \vee q$ etc are also Horn Clauses.
- The fact that PROLOG programs are composed only of Horn Clauses and not of arbitrary logical expressions has two important consequences.
- Because of uniform representation a simple & effective interpreter can be written.
- The logic of Horn Clause systems is decidable.
- Even PROLOG works on backward reasoning.
- The program is read top to bottom, left to right and search is performed depth-first with backtracking.
- There are some syntactic difference between the logic and the PROLOG representations as mentioned
- The key difference between the logic & PROLOG representation is that PROLOG interpreter has a fixed control strategy, so assertions in the PROLOG program define a particular search path to answer any question. Whereas Logical assertions define set of answers that they justify, there can be more than one answers, it can be forward or backward tracking.
- Control Strategy for PROLOG states that we begin with a problem statement, which is viewed as a goal to be proved.
- Look for the assertions that can prove the goal.
- To decide whether a fact or a rule can be applied to the current problem, invoke a standard unification procedure.
- Reason backward from that goal until a path is found that terminates with assertions in the program.
- Consider paths using a depth-first search strategy and use backtracking.
- Propagate to the answer by satisfying the conditions.

2.5.7 Forward v/s Backward Reasoning

- The objective of any search is to find a path through a problem space from the initial to the final one.
- There are 2 directions to go and find the answer

- ☐ Forward
- ☐ Backward
- ☐ 8-square problem
- ☐ Reason forward from the initial states: Begin building a tree of move sequences that might be solution by starting with the initial configuration(s) at the root of the tree. Generate the next level of tree by finding all the rules whose left sides match the root node and use the right sides to create the new configurations. Generate each node by taking each node generated at the previous level and applying to it all of the rules whose left sides match it. Continue.
- ☐ Reason backward from the goal states: Begin building a tree of move sequences that might be solution by starting with the goal configuration(s) at the root of the tree. Generate the next level of tree by finding all the rules whose right sides match the root node and use the left sides to create the new configurations. Generate each node by taking each node generated at the previous level and applying to it all of the rules whose right sides match it. Continue. This is also called Goal-Directed Reasoning.
- ☐ To summarize, to reason forward, the left sides (pre-conditions) are matched against the current state and the right sides (the results) are used to generate new nodes until the goal is reached.
- ☐ To reason backwards, the right sides are matched against the current node and the left sides are used to generate new nodes.
- ☐ Factors that influence whether to choose forward or backward reasoning:
 - ☐ Are there more possible start states or goal states? We would like to go from smaller set of states to larger set of states.
 - ☐ In which direction is the branching factor (the average number of nodes that can be reached directly from a single node) greater? We would like to proceed in the direction with the lower branching factor.
 - ☐ Will the program be asked to justify its reasoning process to the user? If so, it is important to proceed in the direction that corresponds more closely with the way user will think.
 - ☐ What kind of event is going to trigger a problem-solving episode? If it is the arrival of a new fact, forward reasoning should be used. If it is a query to which response is desired, use backward reasoning.
- ☐ Home to unknown place example.
- ☐ MYCIN

- Bidirectional Search (The two searches must pass each other)
- Forward Rules: which encode knowledge about how to respond to certain input configurations.
- Backward Rules: which encode knowledge about how to achieve particular goals.
- Backward- Chaining Rule Systems
 - PROLOG is an example of this.
 - These are good for goal-directed problem solving.
 - Hence Prolog & MYCIN are examples of the same.
- Forward - Chaining Rule Systems
 - We work on the incoming data here.
 - The left sides of rules are matched with against the state description.
 - The rules that match the state dump their right side assertions into the state.
 - Matching is more complex for forward chaining systems.
 - OPS5, Brownston etc. are the examples of the same.
- Combining Forward v/s Backward Reasoning
 - Patients example of diagnosis.
 - In some systems, this is only possible in reversible rules.

Matching

- Till now we have used search to solve the problems as the application of appropriate rules.
- We applied them to individual problem states to generate new states to which the rules can then be applied, until a solution is found.
- We suggest that a clever search involves choosing from among the rules that can be applied at a particular point, but we do not talk about how to extract from the entire collection of rules those that can be applied at a given point.
- To do this we need matching.

Indexing

- Do a simple search through all the rules, comparing each one's precondition to the current state and extracting all the ones that match.
- But this has two problems
- In order to solve very interesting problems, it will be necessary to use a large number of rules, scanning through all of them at every step of the search would be hopelessly inefficient.
- It is not always immediately obvious whether a rule's preconditions are satisfied by a particular state.
- To solve the first problem, use simple indexing. E.g. in Chess, combine all moves at a particular board state together.

Matching with Variables

- The problem of selecting applicable rules is made more difficult when preconditions are not stated as exact descriptions of particular situations but rather describe properties that the situations must have.
- Then we need to match a particular situation and the preconditions of a given situation.
- In many rules based systems, we need to compute the whole set of rules that match the current state description. Backward Chaining Systems usually use depth-first backtracking to select individual rules, but forward chaining systems use Conflict Resolution Strategies.
- One efficient many to many match algorithm is RETE

Complex & Approximate Matching

- A more complex matching process is required when the preconditions of a rule specify required properties that are not stated explicitly in the description of the current state. In this case, a separate set of rules must be used to describe how some properties can be inferred from others.
 - An even more complex matching process is required if rules should be applied if their preconditions approximately match the current situation. Example of listening to a recording of a telephonic conversation.
 - For some problems, almost all the action is in the matching of the rules to the problem state. Once that is done, so few rules apply that the remaining search is trivial. Example ELIZA
-

Conflict Resolution

- The result of the matching process is a list of rules whose antecedents have matched the current state description along with whatever variable binding were generated by the matching process.
- It is the job of the search method to decide on the order in which the rules will be applied. But sometimes it is useful to incorporate some of the decision making into the matching process. This phase is called conflict resolution.
- There are three basic approaches to the problem of conflict resolution in the production system
 - Assign a preference based on the rule that matched.
 - Assign a preference based on the objects that matched.
 - Assign a preference based on the action that the matched rule would perform.

STRUCTURED REPRESENTATION OF KNOWLEDGE

- Representing knowledge using logical formalism, like predicate logic, has several advantages. They can be combined with powerful inference mechanisms like resolution, which makes reasoning with facts easy. But using logical formalism complex structures of the world, objects and their relationships, events, sequences of events etc. cannot be described easily.
 - A good system for the representation of structured knowledge in a particular domain should possess the following four properties:
 - (i) Representational Adequacy:- The ability to represent all kinds of knowledge that are needed in that domain.
 - (ii) Inferential Adequacy :- The ability to manipulate the represented structure and infer new structures.
 - (iii) Inferential Efficiency:- The ability to incorporate additional information into the knowledge structure that will aid the inference mechanisms.
 - (iv) Acquisitional Efficiency :- The ability to acquire new information easily, either by direct insertion or by program control.
 - The techniques that have been developed in AI systems to accomplish these objectives fall under two categories:
-

1. Declarative Methods:- In these knowledge is represented as static collection of facts which are manipulated by general procedures. Here the facts need to be stored only one and they can be used in any number of ways. Facts can be easily added to declarative systems without changing the general procedures.

2. Procedural Method:- In these knowledge is represented as procedures. Default reasoning and probabilistic reasoning are examples of procedural methods. In these, heuristic knowledge of “How to do things efficiently “can be easily represented.

- In practice most of the knowledge representation employ a combination of both. Most of the knowledge representation structures have been developed to handle programs that handle natural language input. One of the reasons that knowledge structures are so important is that they provide a way to represent information about commonly occurring patterns of things . such descriptions are some times called schema. One definition of schema is

- “Schema refers to an active organization of the past reactions, or of past experience,
which must always be supposed to be operating in any well adapted organic response”.

- By using schemas, people as well as programs can exploit the fact that the real world is not random. There are several types of schemas that have proved useful in AI programs. They include

(i) Frames:- Used to describe a collection of attributes that a given object possesses (eg: description of a chair).

(ii) Scripts:- Used to describe common sequence of event (eg:- a restaurant scene). (iii) Stereotypes :- Used to described characteristics of people.

(iv) Rule models:- Used to describe common features shared among a set of rules in a production system.

- Frames and scripts are used very extensively in a variety of AI programs. Before selecting any specific knowledge representation structure, the following issues have to be considered.

(i) The basis properties of objects , if any, which are common to every problem domain must be identified and handled appropriately.

(ii) The entire knowledge should be represented as a good set of primitives.

(iii) Mechanisms must be devised to access relevant parts in a large knowledge base.