Unit II

INFORMED SEARCH AND EXPLORATION

Informed (heuristic) Search strategies: one that uses problem-specific knowledge beyond the definition of the problem itself-can find solutions more efficiently than an uninformed strategy

Best-first search - an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function,** f (n) .

Traditionally, the node with the *lowest* evaluation is selected for expansion, because the evaluation measures distance to the goal. It can be implemented using priority queue, a data structure that will maintain the fringe in ascending order of f -values

All we can do is choose the node that ***appears*** to be best according to the evaluation function. If the evaluation function is exactly accurate, then this will indeed be the best node;

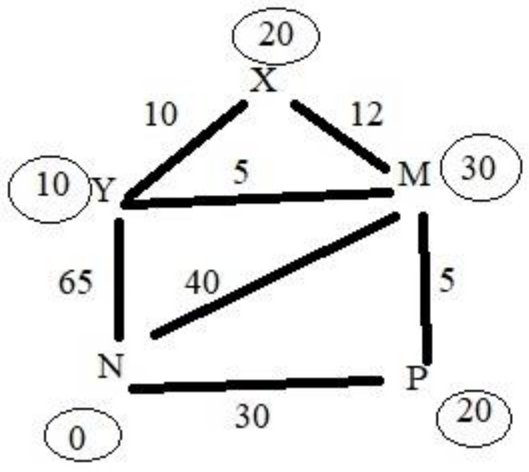
Family of Best first search algorithm with different evaluation functions. **A** key component of these algorithms is a heuristic function, denoted *h(n)*= estimated cost of the cheapest path from node *n* to a goal node.

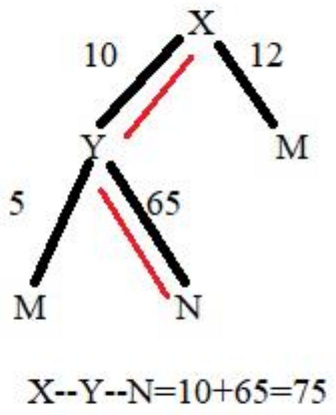
Greedy best-first search

Greedy best-first search3 tries to expand the node that is closest to the goal, on the: grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function: f *(n)* = *h(n).*

The following fig shows the state space graph and tree construction for the graph. S is the initial node, G is the goal node and the arcs are showing the step cost between nodes. The circled values are Straight Line Distance of the node to the goal node.

h_{SLD} = X-20; Y-10; M-30; P-20; N-0





(Give explanation for the tree, how the greedy search select the node from successor)

In this example though greedy search provides the solution, it is not the best solution since X→M→P→N=47 is the best solution for this. It checks only the current SLD to proceed. This is the reason for the name ‘Greedy’.
Greedy best-first search resembles depth-first search in the way it prefers to follow a single path all the way to the goal, but will back up when it hits a dead end.
It suffers from the same defects as depth-first search-it is not optimal, and it is incomplete (because it can start down an infinite path and never return to try other possibilities).
The worst-case time and space complexity is O(bm), where m is the maximum depth of the search space. With a good heuristic function, however, the complexity can be reduced substantially. The amount of the reduction depends on the particular problem and on the quality of the heuristic.

A* search: Minimizing the total estimated solution cost

The most widely-known form of best-first search is called **A*** search.
Evaluation function:

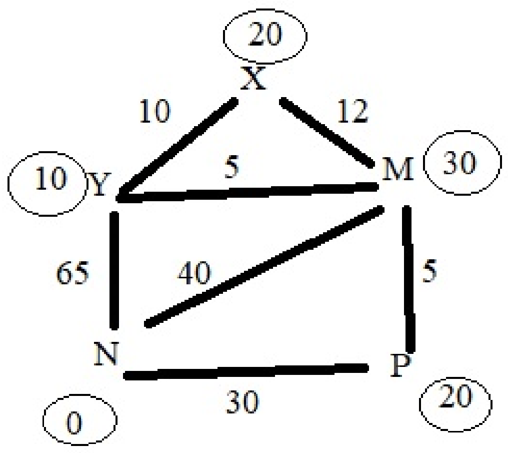
$$f(n)=g(n)+h(n)$$

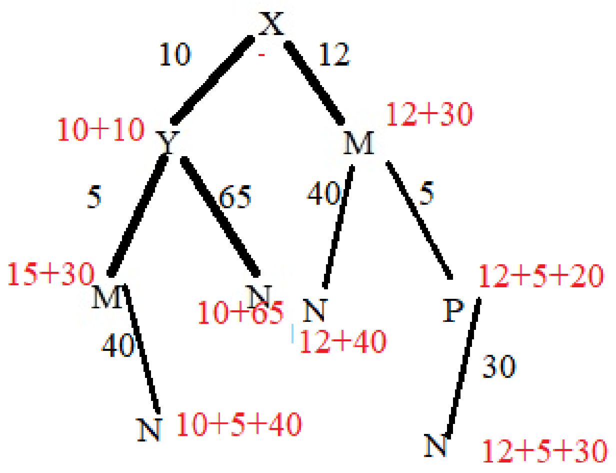
- It evaluates nodes by combining *g(n)*,t he cost to reach the node, and *h(n.)*,the cost to get from the node to the goal
- Since *g(n)* gives the path cost from the start node to node *n*, and *h(n)* is the estimated cost of the cheapest path from n to the goal, we have f (n) = estimated cost of the cheapest solution through n.
- Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of *g(n)* + *h(n).*
- It turns out that this strategy is more than just reasonable: provided that the heuristic function *h(n)* satisfies certain conditions, **A*** search is both complete and optimal.

Optimality and admissible:

- The optimality of A* is straightforward to analyze if it is used with TREE-SEARCH.
- In this case, A* is optimal if *h(n)* is an admissible heuristic-that is, provided that *h(n)**__never overestimates__* the cost to reach the goal.
- Admissible heuristics are by nature optimistic, because they think the cost of solving the problem is less than it actually is.
- Since *g(n)* is the exact cost to reach *n*, we have as immediate consequence that *f (n)* never overestimates the true cost of a solution through n.
- An obvious example of an admissible heuristic is the straight-line distance *hsLD* that we used in getting to goal node.
- Straight-line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot be an overestimate.

(we show the progress of an A* tree search)
h_{SLD} = S-10; A-20; B-30; C-40; G-0





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

1: f(Y)=(X—Y): 10+10=20

f(M)=(X—M): 12+30=42 h1

20<42, select Y

2: f(M)= (X—Y—M): 10+5+30=45 h2

f(N)=(X—Y—N) 10+65+0=75 h3

h1<h2<h3, goto h1

3: f(N)=(X—M—N) 12+40+0=52 h4

f(P)= (X—M—P) 12+5+20=37, select P

4: f(N)= (X—M—P—N) 12+5+30+0=47 h5

h5>h2

Goto h2

5: f(M)=(X—Y—M—N)=10+5+40=55>h5

Got to h5: f(N)= (X—M—P—N): 12+5+30+0=47, Goal found with best solution cost of 47.

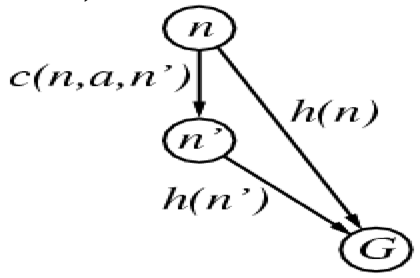
we can extract a general proof that A* ***using*** TREE-SEARCH ***is optimal*** *if h(n)* ***is admissible.***
Suppose a Then, because *G2*i s suboptimal and because *h(G2=)* 0 (true for any goal node), vie know
$f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$.

Now consider a fringe node *n* that is on an optimal solution path-for example, Pitesti in the example of the preceding paragraph. (There must always be such a node if a solution exists.)
If *h(n)* does not overestimate the cost of completing the solution path, then we know that
$f(n) = g(n) + h(n) <= C^*$.
Now we have shown that f *(n)* <= C* < f (G_2)s,o G_2 will not be expanded anti A* must return an optimal solution.

If we use the GRAPH-SEARCH algorithm of Figure 3.19 instead of TREE-SEARCH, then this proof breaks down

Consistency:

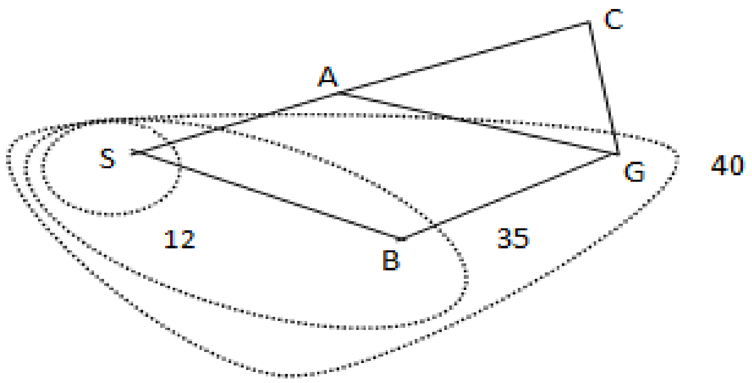
- The second solution is to ensure that the optimal path to any repeated state is always the first one followed-as is the case with uniform-cost search. This property holds if we impose an extra requirement on *h(n)*,n amely the requirement of **consistency** (also called **monotonicity).**
- **A** heuristic *h(n)* is consistent if, for every node *n* and every successor *n'* of *n* generated by any action a, the estimated cost of reaching the goal from *n* is no greater than the step cost of getting to *n'* plus the estimated cost of reaching the goal from *n':*
  $$h(n) <= c(n,a , n') + h(n').$$
- This is a form of the general **triangle inequality,** which stipulates that each side of a triangle cannot be longer than the sum of the other two sides.
- Here, the triangle is formed by *n, n'*, and the goal closest to *n*. It is fairly easy to show (Exercise 4.7) that every consistent heuristic is also admissible.
- The most important consequence of consistency is the following: A* *using* GRAPH-SEARCH *is optimal if h(n) is consistent.*
- Although consistency is a stricter requirement than admissibility, one has to work quite hard to concoct heuristics that are admissible but not consistent.
- All the admissible heuristics we discuss in this chapter are also consistent.
- Consider, for example, h_{sLD}. We know that the general triangle inequality is satisfied when each side is measured by the straight-line distance, and that the straight-line distance between n and *n'* is no greater than *c(n,* a, *n')*.
- Hence, h_{sLD} is a consistent heuristic.



Non decreasing values

- Another important consequence of consistency is the following: if *h(n) is consistent, then the values* off *(n) along any path are nondecueasing.*
- The proof follows directly from the definition of consistency. Suppose *n'* is a successor of *n;* then *g(n')= g(n)+ c(n,a , n')* for some *a*, and we have
  $$f(n')= g(n')+ h(n')= g(n)+ c(n,a , n') + h(n')>= g(n)+ h(n)= f (n).$$
- It follows that the sequence of nodes expanded by A* using GRAPH-SEARCH is in nondecreasing
  order of f (n). Hence, the first goal node selected for expansion must be an optimal solution, since all later nodes will be at least as expensive.
- The fact that *f* -costs are nondecreasing along any path also means that we can draw **contours** in the state space, just like the contours in a topographic map.

Figure 4.4 shows an example. Inside the contour labeled 12 (35,40), all nodes have f *(n)* less than or equal to (12 (35,40), and so on.



- Then, because A* expands the fringe node of lowest *f* -cost, we can see that an A* search fans out from the start node, adding nodes in concentric bands of increasing f -cost.
- With uniform-cost search (A* search using *h(n)* = O), the bands will be "circular" around the start state. With more accurate heuristics, the bands will stretch toward the goal state and become more narrowly focused around the optimal path. If C* is the cost of the optimal solution path, then we can say the following:

- A* expands all nodes with *f* (n) < C*.
- A* might then expand some of the nodes right on the "goal contour" (where f *(n)* = C*) before selecting a goal node.

Performance measures

- Intuitively, it is obvious that the first solution found must be an optimal one, because goal nodes in all subsequent contours will have higher *f* -cost, and thus higher g-cost (because all goal nodes have *h(n)* = 0).
- Intuitively, it is also obvious that A* search is complete. As we add bands of increasing *f*, we must eventually reach a band where *f* is equal to the cost of the path to a goal state.4
- One final observation is that among optimal algorithms of this type-algorithms that extend search paths from the root-A* is **optimally efficient** for any given heuristic function.
- That is, no other optimal algorithm is guaranteed to expand fewer nodes than A* (except possibly through tie-breaking among nodes with f *(n)* = C*).
- This is because any algorithm that *does not* expand all nodes with f *(n)* < C* runs the risk of missing the optimal solution.
- That A* search is complete, optimal, and optimally efficient among all such algorithms is rather satisfying.

One can use variants of A* that find suboptimal solutions quickly, or one can sometimes design heuristics that are more accurate, but not strictly admissible. In any case, the use of a good heuristic still provides enormous savings compared to the use of an uninformed search.

Book Example for understanding:

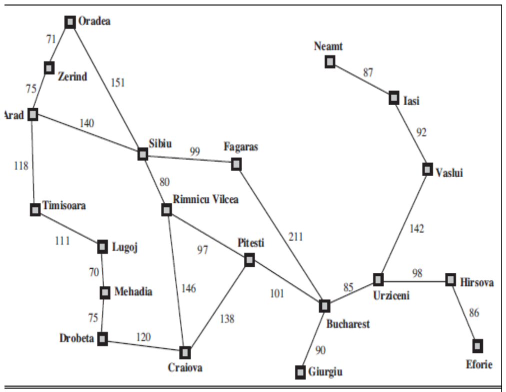
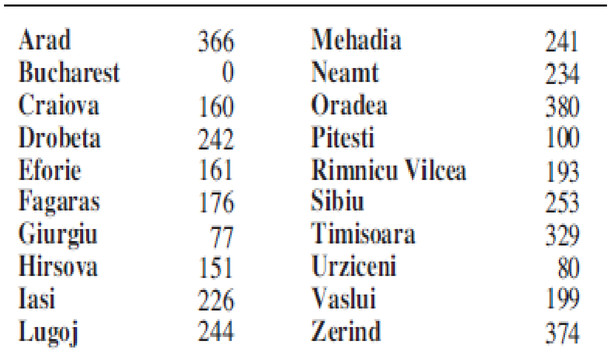
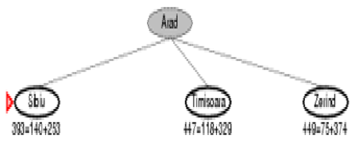
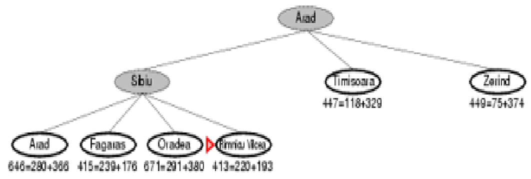


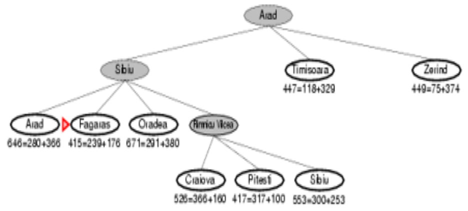
Figure 3.2 A simplified road map of part of Romania.

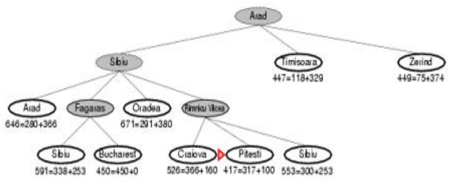


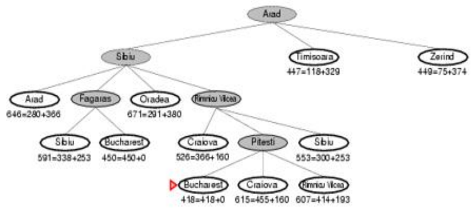
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374





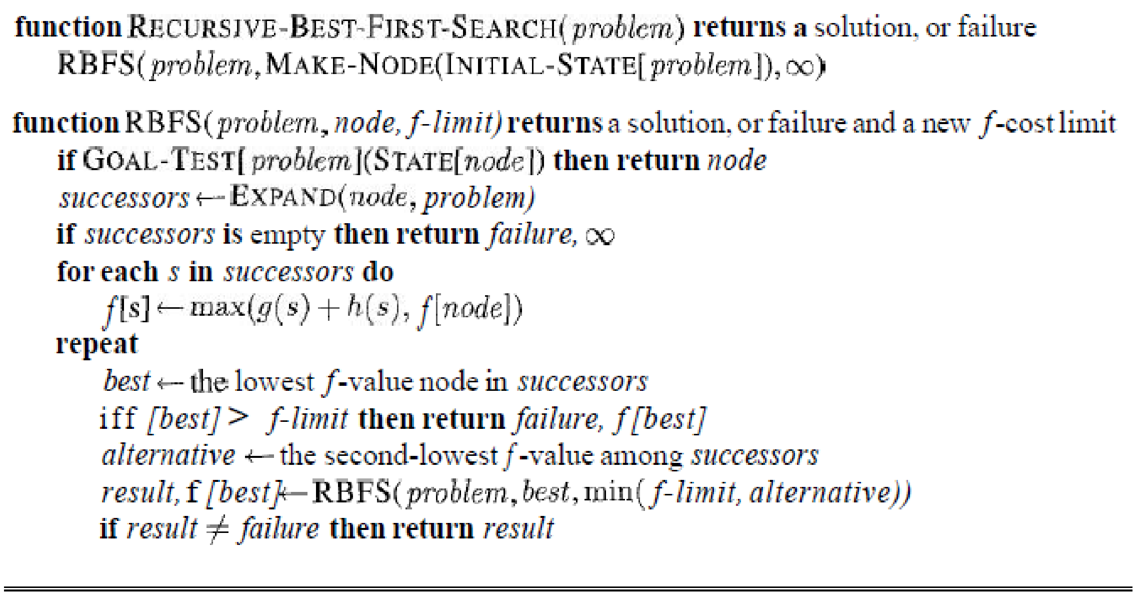






Example 3:

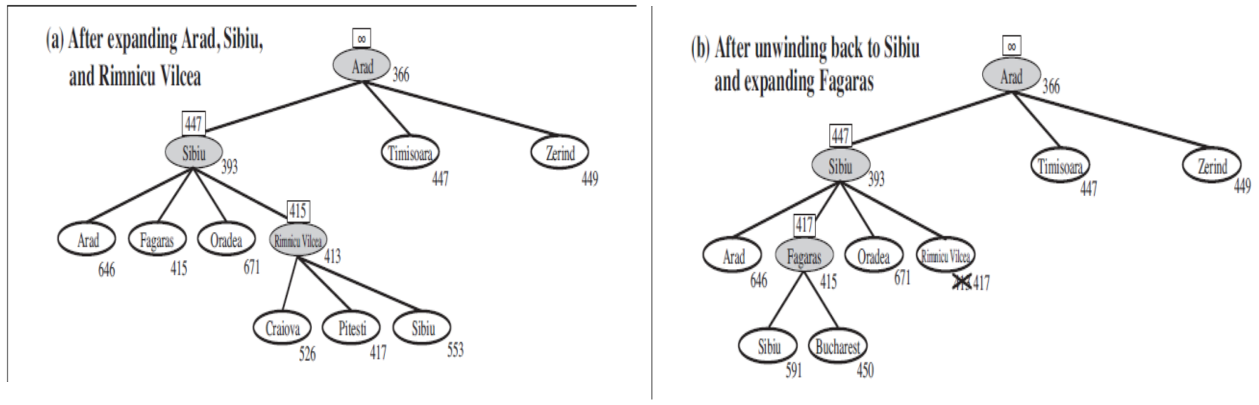
Recursive best-first search (RBFS)- simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space
Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it keeps track of the f-value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path.
As the recursion unwinds, RBFS replaces the *f* -value of each node along the path with the best *f* -value of its children.
In this way, RBFS remembers the f -value of the best leaf in the forgotten subtree and can therefore decide whether it's worth reexpanding the subtree at some later time. Figure 4.6 shows how RBFS reaches Bucharest.



```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    RBFS(problem, MAKE-NODE(INITIAL-STATE[problem]), ∞)

function RBFS(problem, node, f-limit) returns a solution, or failure and a new f-cost limit
    if GOAL-TEST[problem](STATE[node]) then return node
    successors ← EXPAND(node, problem)
    if successors is empty then return failure, ∞
    for each s in successors do
        f[s] ← max(g(s) + h(s), f[node])
    repeat
        best ← the lowest f-value node in successors
        if f [best] > f-limit then return failure, f [best]
        alternative ← the second-lowest f-value among successors
        result, f [best] ← RBFS(problem, best, min(f-limit, alternative))
        if result ≠ failure then return result
```

Figure 4.5 The algorithm for recursive best-first search.

Book Example:



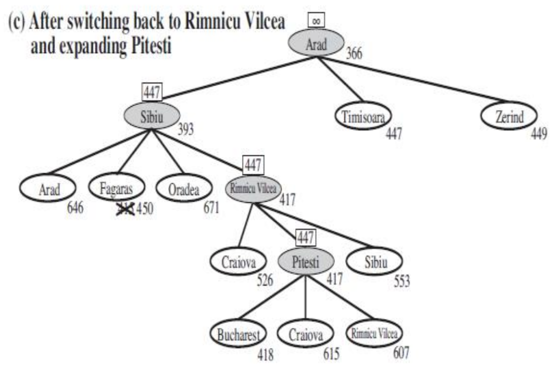


Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

AO* Algorithm

Problem Reduction with AO* Algorithm. When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND - OR trees are used for representing the solution. The decomposition of the problem or problem reduction generates AND arcs. One AND are may point to any number of successor nodes. All these must be solved so that the arc will rise to many arcs, indicating several possible solutions. Hence the graph is known as AND - OR instead of AND. Figure shows an AND - OR graph.

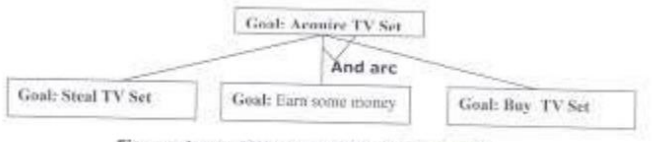


Figure shows AND - Or graph - an example.

An algorithm to find a solution in an AND - OR graph must handle AND area appropriately. A* algorithm can not search AND - OR graphs efficiently. This can be understand from the give figure.

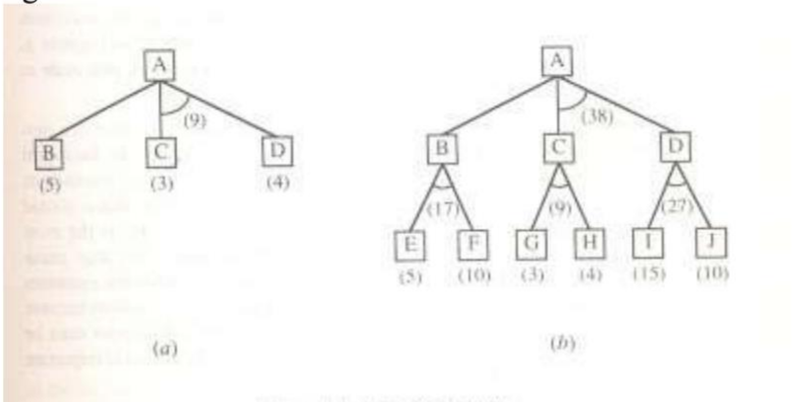
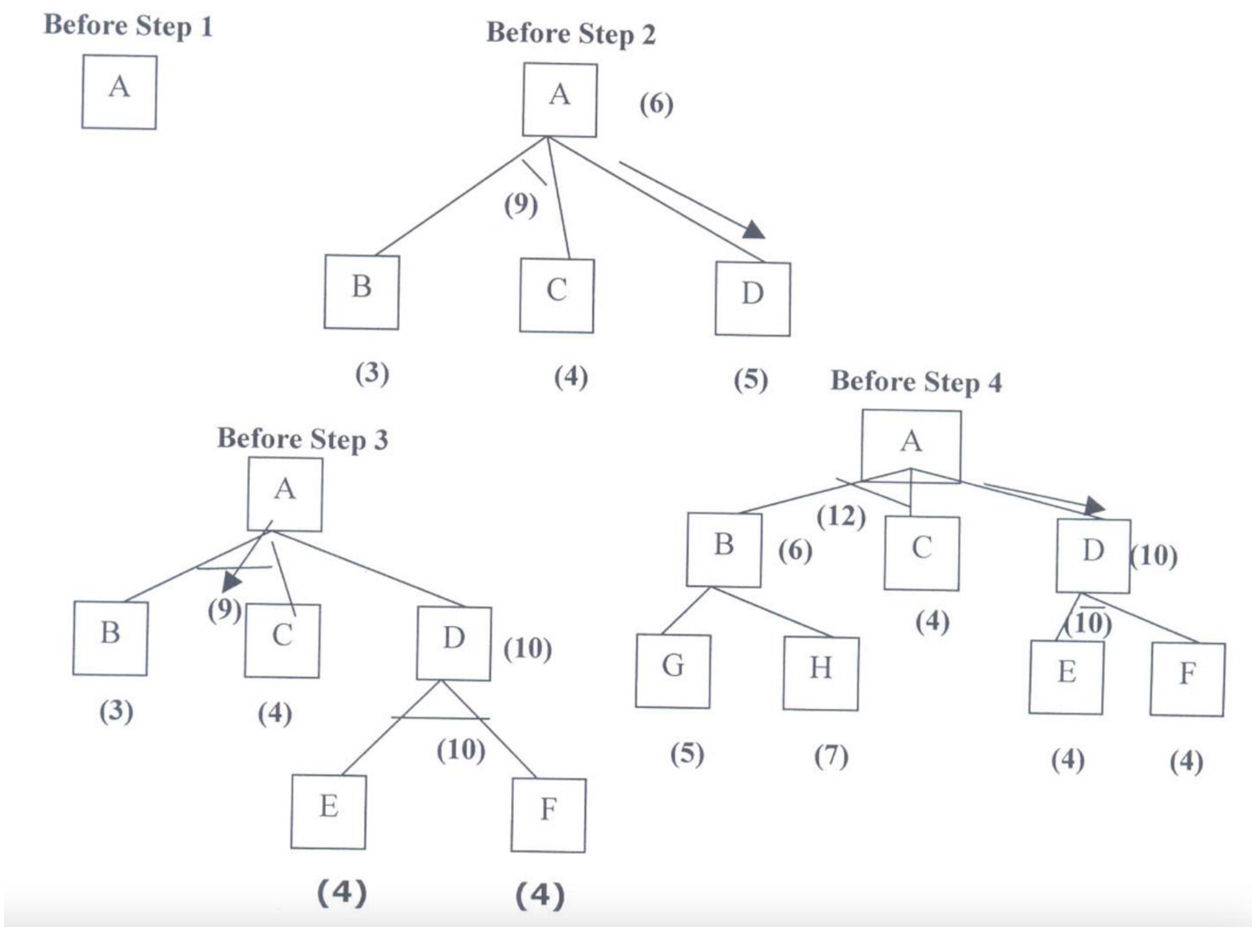


Figure 3.7: AND-OR Graphs

FIGURE : AND - OR graph

In figure (a) the top node A has been expanded producing two area one leading to B and leading to C-D . the numbers at each node represent the value of f ' at that node (cost of getting to the goal state from current state). For simplicity, it is assumed that every operation(i.e. applying a rule) has unit cost, i.e., each are with single successor will have a cost of 1 and each of its components. With the available information till now , it appears that C is the most promising node to expand since its f ' = 3 , the lowest but going through B would be better since to use C we must also use D' and the cost would be 9(3+4+1+1). Through B it would be 6(5+1). Thus the choice of the next node to expand depends not only n a value but also on whether that node is part of the current best path form the initial mode. Figure (b) makes this clearer. In figure the node G appears to be the most promising node, with the least f ' value. But G is not on the current beat path, since to use G we must use GH with a cost of 9 and again this demands that arcs be used (with a cost of 27). The path from A through B, E-F is better with a total cost of (17+1=18). Thus we can see that to search an AND-OR graph, the following three things must be done. 1. traverse the graph starting at the initial node and following the current best path, and accumulate the set of nodes that are on the path and have not yet been expanded. 2. Pick one of these unexpanded nodes and expand it. Add its successors to the graph and computer f ' (cost of the remaining distance) for each of them.

3. Change the f ' estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path. The propagation of revised cost estimation backward is in the tree is not necessary in A* algorithm. This is because in AO* algorithm expanded nodes are re-examined so that the current best path can be selected. The working of AO* algorithm is illustrated in figure as follows:



Referring the figure. The initial node is expanded and D is Marked initially as promising node. D is expanded producing an AND arc E-F. f ' value of D is updated to 10. Going backwards we can see that the AND arc B-C is better . it is now marked as current best path. B and C have to be expanded next. This process continues until a solution is found or all paths have led to dead ends, indicating that there is no solution. An A* algorithm the path from one node to the other is always that of the lowest cost and it is independent of the paths through other nodes. The algorithm for performing a heuristic search of an AND - OR graph is given below. Unlike A* algorithm which used two lists OPEN and CLOSED, the AO* algorithm uses a single structure G. G represents the part of the search graph generated so far. Each node in G points down to its immediate successors and up to its immediate predecessors, and also has with it the value of h' cost of a path from itself to a set of solution nodes. The cost of getting from the start nodes to the current node "g" is not stored as in the A* algorithm. This is because it is not possible to compute a single such value since there may be many paths to the same state. In AO* algorithm serves as the estimate of goodness of a node. Also a there should value called FUTILITY is used. The estimated cost of a solution is greater than FUTILITY then the search is abandoned as too expansive to be practical. For representing above graphs AO* algorithm is as follows

AO* ALGORITHM:

1. Let G consists only to the node representing the initial state call this node INTT. Compute h' (INIT). 2

2. . Until INIT is labeled SOLVED or hi (INIT) becomes greater than FUTILITY, repeat the following procedure.

(I) Trace the marked arcs from INIT and select an unbounded node NODE.

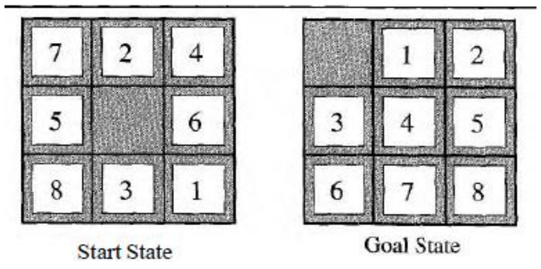
(II) Generate the successors of NODE . if there are no successors then assign FUTILITY as h' (NODE). This means that NODE is not solvable. If there are successors then for each one called SUCCESSOR, that is not also an ancester of NODE do the following (a) add SUCCESSOR to graph G (b) if successor is not a terminal node, mark it solved and assign zero to its h ' value. (c) If successor is not a terminal node, compute it h' value.

(III) propagate the newly discovered information up the graph by doing the following . let S be a set of nodes that have been marked SOLVED. Initialize S to NODE. Until S is empty repeat the following procedure;
(a) select a node from S call if CURRENT and remove it from S.
(b) compute h' of each of the arcs emerging from CURRENT , Assign minimum h' to CURRENT.
(c) Mark the minimum cost path a s the best out of CURRENT.
(d) Mark CURRENT SOLVED if all of the nodes connected to it through the new marked are have been labeled SOLVED. (e) If CURRENT has been marked SOLVED or its h ' has just changed, its new status must be propagate backwards up the graph . hence all the ancestors of CURRENT are added to S.

Search Procedure.
1. Place the start node on open.
2. Using the search tree, compute the most promising solution tree TP .
3. Select node n that is both on open and a part of tp, remove n from open and place it no closed.
4. If n is a goal node, label n as solved. If the start node is solved, exit with success where tp is the solution tree, remove all nodes from open with a solved ancestor.
5.. If n is not solvable node, label n as unsolvable. If the start node is labeled as unsolvable, exit with failure. Remove all nodes from open ,with unsolvable ancestors.
6. Otherwise, expand node n generating all of its successor compute the cost of for each newly generated node and place all such nodes on open.
7. Go back to step(2) Note: AO* will always find minimum cost solution.

Heuristic Functions

To find a solution in proper time rather than a complete solution in unlimited time we use heuristics. ‘A heuristic function is a function that maps from problem state descriptions to measures of desirability, usually represented as numbers’.



- The average solution cost for a randomly generated 8-puzzle instance is about 22 steps.
- The branching factor is about 3. (When the empty tile is in the middle, there are four possible moves; when it is in a corner there are two; and when it is along an edge there are three.)
- This means that an exhaustive search to depth 22 would look at about $\mathbf{3^{22}}$ = 3.1 x 10^{10} states.
- By keeping track of repeated states, we could cut this down by a factor of about 170,000, because there are only 9!/2 = 181,440 distinct states that are reachable.
    - *h1* = the number of misplaced tiles. In the above figure all of the eight tiles are out of position, so the start state would have *hl* = 8. *h1* is an admissible heuristic, because it is clear that any tile that is out of place must be moved at least once.
    - *h2* = the sum of the distances of the tiles from their goal positions. Because tiles cannot move along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the **city block distance** or **Manhattan distance.**
      h2 is also admissible, because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of *h2=3+l+2+2+2+3+3+2=18.*
      As we would hope, neither of these overestimates the true solution cost, which is 26.

LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

The search algorithms that we have seen so far are designed to explore search spaces systematically.
This systematicity is achieved by keeping one or more paths in memory and by recording which alternatives have been explored at each point along the path and which have not. When a goal is found, the *path* to that goal also constitutes a solution to the problem.
In many problems, however, the path to the goal is irrelevant. For example, in the 8-queens problem what matters is the final configuration of queens, not the order in which they are added.
This class of problems includes many important applications such as integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing, and portfolio management.If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all.

Local search algorithms operate using a single **current state** (rather than multiple paths) and generally move only to neighbors of that state. Typically, the paths followed by the search are not retained. Although local search algorithms are not systematic, they have two key advantages:
(1) they use very little memory-usually a constant amount; and
(2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

In addition to finding goals, local search algorithms are useful for solving pure **optimization problems,** in which the aim is to find the best state according to an **objective function.**

To understand local search, we will find it very useful to consider the **state space land scape** (as in Figure 4.10). A landscape has both "location" (defined by the state) and "elevation" (defined by the value of the heuristic cost function or objective function).

If elevation corresponds to cost, then the aim is to find the lowest valley-a **global minimum;**

if elevation corresponds to an objective function, then the aim is to find the highest peak-a **global maximum,.** (You can convert from one to the other just by inserting a minus sign.)

Local search algorithms explore this landscape. A **complete,** local search algorithm always finds a goal if one exists; an **optimal** algorithm always finds a, global minimum/maximum.

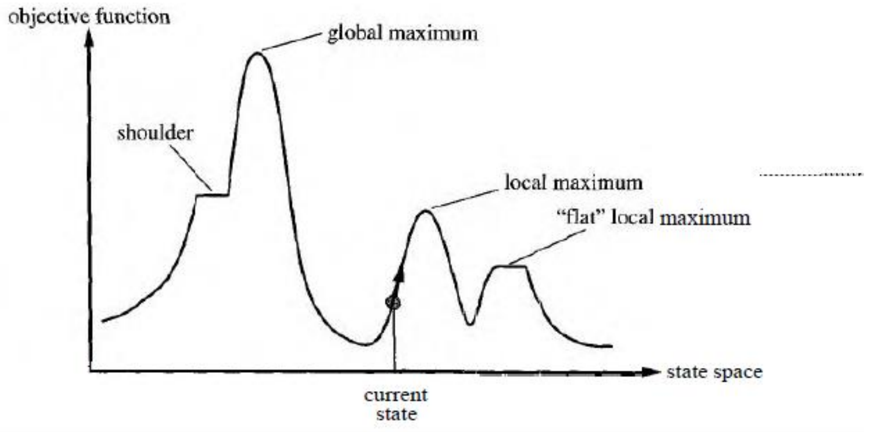


Figure 4.10 A one-dimensional state space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

Hill-climbing search

The **hill-climbing** search algorithm is shown in Figure 4.11. It is simply a loop that continually moves in the direction of increasing value-that is, uphill. It terminates when it reaches a "top" where no neighbor has a higher value. The algorithm does not maintain a search tree, so the current node data structure need only record the state and its objective function value.

Hill-climbing does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia. To illustrate hill-climbing, we will use the **8-queens problem**. Local-search algorithms typically use a **complete-state formulation,** where each state has 8 queens on the board, one per column.

The successor function returns all possible states generated by moving a single queen to another square in the same column (so each state has 8 x 7 = 56 successors).

The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly.

The global minimum of this function is zero, which occurs only at perfect solutions.

Figure 4.12(a) shows a state with h = 17.

The figure also shows the values of all its successors, with the best successors having h = 12. Hill-climbing algorithms typically choose randomly among the set of best successors, if there is more than one.

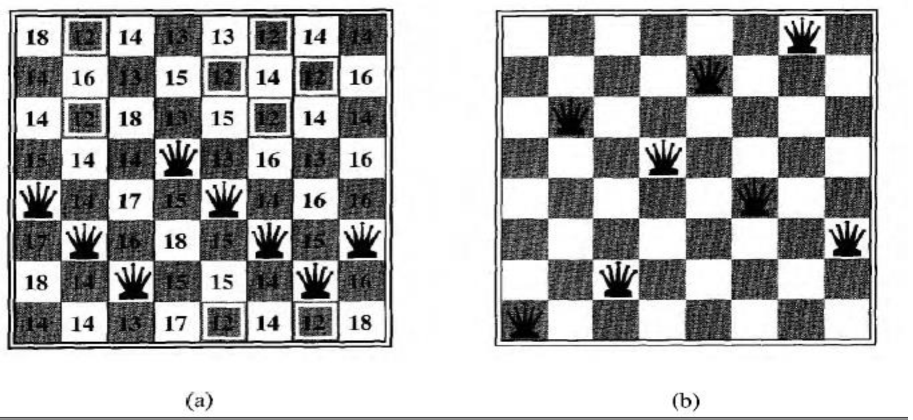


Figure 4.12 (a) An 8-queens state with heuristic cost estimate h = 17, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has h = 1 but every successor has a higher cost.

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

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

Figure 4.11 The hill-climbing search algorithm (**steepest ascent** version), which is the most basic local search technique. At each step the current node is replaced by the best neighbor; in this version, that means the neighbor with the highest VALUE, but if a heuristic cost estimate h is used, we would find the neighbor with the lowest h.

Hill climbing is sometimes called **greedy local s'earch** because it grabs a good neighbor state without thinking ahead about where to go next. Hill climbing often makes very rapid progress towards a solution, be'cause it is usually quite easy to improve a bad state.

For example, from the state in Figure 4.12(a), it takes just five steps to reach the state in Figure 4.12(b), which has h = 1 and is very nearly a solution. Unfortunately, hill climbing often gets stuck for the following reasons:

Local maxima: a local maximum is a top point that is higher than each of its neighboring states, but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards towards the peak, but will then be stuck with nowhere else to go. Figure 4.10 illustrates the problem schematically.
More concretely, the state in Figure 4.12(b) is in fact a local maximum (i.e., a local minimum for the cost h); every move of a single queen makes the situation worse.

Ridges: a ridge is shown in Figure 4.13. Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.

Plateaux: a plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum, from which no uphill exit exists, or a **shoulder,** from which it is possible to make progress. (See Figure 4.10.) A hill-climbing search might be unable to find its way off the plateau.

In each case, the algorithm reaches a point at which no progress is being made. Starting from a randomly generated 8-queens state, steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances. It works quickly, taking just 4 steps on average when it succeeds and 3 when it gets stuck-not bad for a state space with 8^8 = 17 million states.

The algorithm in Figure 4.11 halts if it reaches a plateau where the best successor has the same value as the current state. Might it not be a good idea to keep going-to allow a **sideways move** in the hope that the plateau is really a shoulder, as shown in Figure 4. L3? The answer is usually yes, but we must take care. If we always allow sideways moves when there are no uphill moves, an infinite loop will occur when ever the algorithm reaches a flat local maximum that is not a shoulder. One common solution is to put a limit on the number of consecutive sideways moves allowed. For example, we could allow up to, say, 100 consecutive sideways moves in the 8-queens problem. This raises the percentage of problem instances solved by hill-climbing from 14% to 94%. Success comes at a cost: the algorithm averages roughly 21 steps for each successful instance and 64 for each failure.

Many variants of hill-climbing:

a) **Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes it finds better solutions.
b) **First-choice hill climbing** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state. This is a good strategy when a state has many (e.g., thousands) of successors.

The hill-climbing algorithms described so far are incomplete-they often fail to find a goal when one exists because they can get stuck on local maxima.

c) **Random-restart hill climbing** adopts the well known adage, "If at first you don't succeed, try, try again." It conducts a series of hill-climbing searches from randomly generated initial state, stopping when a goal is found. It is complete with probability approaching 1, for the trivial reason that it will eventually generate a goal state as the initial state if there are few local maxima and plateaux, random-restart hill climbing will find a good solution very quickly. a reasonably good local maximum can often be found after a small number of restarts.

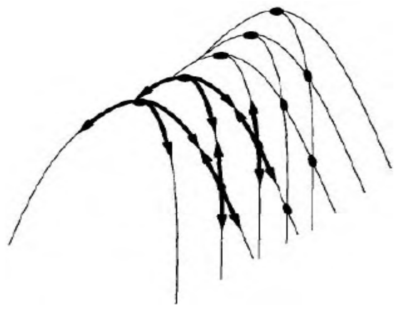


Figure 4.13 Illustration of why ridges cause difficulties for hill-climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

Local beam search

The **local beam search** algorithm keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.

At first sight, a local beam search with k states might seem to be nothing more than running k random restarts in parallel instead of in sequence. In fact, the two algorithms are quite different.

In a random-restart search, each search process runs independently of the others. In *a local beam search, useful information is passed among the k parallel search threads.*

For example, if one state generates several good successors and the other *k* - 1 states all generate bad successors, then the effect is that the first state says to the others, "Come over here, the grass is greener!" The algorithm quickly abandons unfruitful searches and moves its resources to where the most progress is being made.

In its simplest form, local beam search can suffer from a lack of diversity among the *k* states-they can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing. A variant called **stochastic beam search,** analogous to stochastic hill climbing, helps to alleviate this problem. Instead of choosing the best k from the the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value.

Genetic algorithms

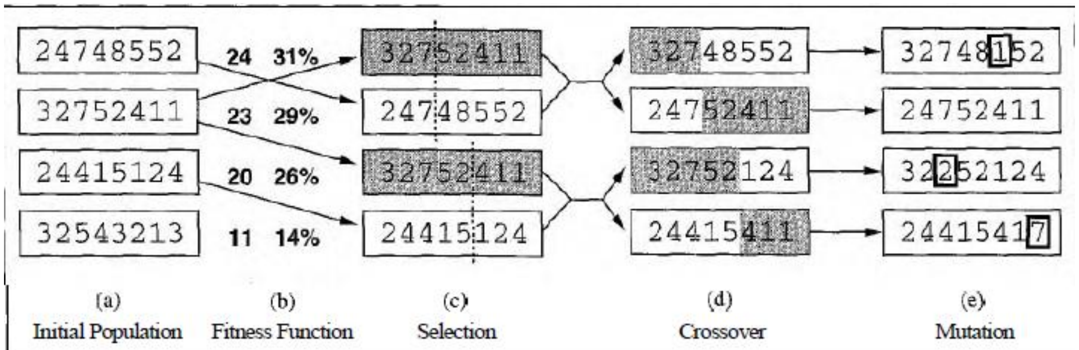


Figure 4.15 The genetic algorithm. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

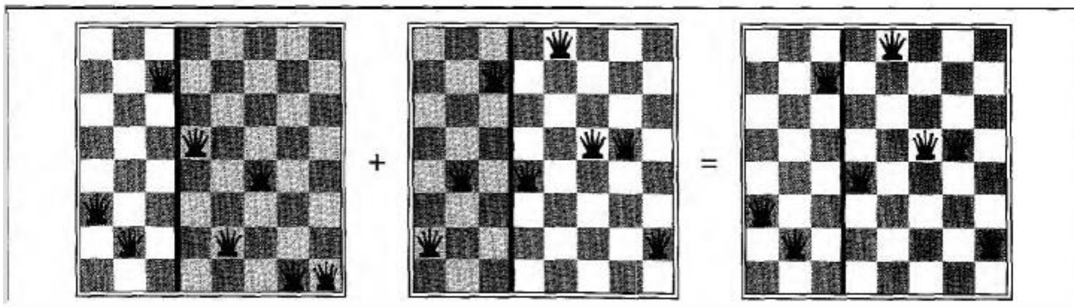


Figure 4.16 The 8-queens states corresponding to the first two parents in Figure 4.15(c) and the first offspring in Figure 4.15(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

A genetic algorithm (or **GA)** is a variant of stochastic beam search in which successor states are generated by combining *two* parent states, rather than by modifying a single state. Like beam search, GAS begin with a set of k randomly generated states, called the **population.**

Each state, or **individual,** is represented as a string over a finite alphabet-most commonly, a string of 0s and 1s. For example, an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires 8 x $\log_2 8 = 24$ bits. Alternatively, the state cloud be represented as 8 digits, each in the range from 1 to 8. (We will see laterthat the two encoding behave differently.)

Figure 4.15(a) shows a population of four 8-digit strings representing 8-queens states.The production of the next generation of states is shown in Figure 4.15(b)-(e). In (b), each state is rated by the evaluation function or (in **C;A** terminology) the **fitness function.** A fitness function should return higher values for better states, so, for the 8-queens problem we use the number of *nonattacking* pairs of queens, which has a value of 28 for a solution.The values of the four states are 24, 23, 20, and 11. In this particular variant of the genetic algorithm, the probability of being chosen for reproducing is directly proportional to the fitness score, and the percentages are shown next to the raw scores.

In (c), a random choice of two pairs is selected for reproduction, in accordance with the probabilities in (b). Notice that one individual is selected twice and one not at all. For each pair to be mated, a **crossover** point is randomly chosen from the positions in the string. In Figure 4.15 the crossover points are after the third digit in the first pair and after the fifth digit in the second pair. In (d), the offspring themselves are created by crossing over the parent strings at the crossover point.

For example, the first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent, whereas the second child gets the first three digits from the second parent and the rest from the first parent. The 8 queens states involved in this reproduction step are shown in Figure 4.16. The example illustrates the fact that, when two parent states are quite different, the crossover operation can produce a state that is a long way from either parent state. It is often the case that the population is quite diverse early on in the process, so crossover (like simulated annealing) frequently takes large steps in the state space early in the search process and smaller steps later on when most individuals are quite similar.

Finally, in (e), each location is subject to random **mutation** with a small independent probability. One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column. Figure 4.17 describes an algorithm that implements all these steps.

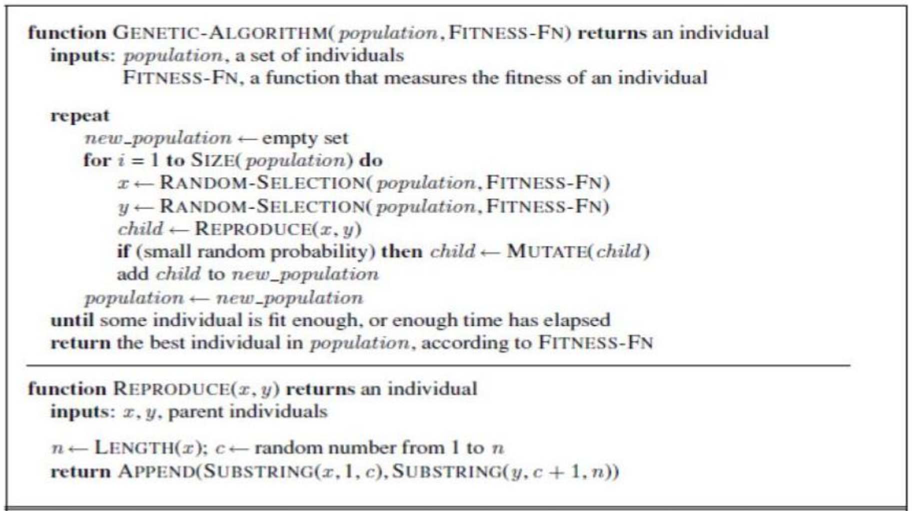
Like stochastic beam search, genetic algorithms combine an uphill tendency with random exploration and exchange of information among parallel search threads. The primary advantage, if any, of genetic algorithms comes from the crossover operation. Yet it can be shown mathematically that, if the positions of the genetic code is permuted initially in a random order, crossover conveys no advantage.

Intuitively, the advantage comes from the ability of crossover to combine large blocks of letters that have evolved independently to perform useful functions, thus raising the level of granularity at which the search operates. For example, it could be that putting the first three queens in positions 2, 4, and 6 (where they do not attack each other) constitutes a useful block that can be combined with other blocks toconstruct a solution.

The theory of genetic algorithms explains how this works using the idea of a **schema,** which is a substring in which some of the positions can be left unspecified. For example, the schema 246***** describes all 8-queens states in which the first three queens are in positions 2, 4, and 6 respectively. Strings that match the schema (such as 24613578) are called **instances** of the schema.

Genetic algorithms work best when schemas correspond to meaningful components of a solution. For example, if the string is a representation of an antenna, then the schemas may represent components of the antenna, such as reflectors and deflectors. A good component is likely to be good in a variety of different designs. This suggests that successful use of genetic algorithms requires careful engineering of the representation.

In practice, genetic algorithms have had a widespread impact on optimization problems, such as circuit layout and job-shop scheduling. Much work remains to be done to identify the condition under which genetic algorithms perform



```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     for i = 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Figure 4.8 A genetic algorithm. The algorithm is the same as the one diagrammed in Figure 4.6, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

CONSTRAINT SATISFACTION PROBLEM

A constraint satisfaction problem consists of three components, X,D, and C:
X is a set of variables, {X1, . . . ,Xn}.
D is a set of domains, {D1, . . . ,Dn}, one for each variable.
C is a set of constraints that specify allowable combinations of values.
Each domain Di consists of a set of allowable values, {v1, . . . , vk} for variable Xi.
Each constraint Ci consists of a pair scope, rel ,
scope :tuple of variables that participate in the constraint.
rel :relation that defines the values that those variables can take on.
A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation.
State is defined by variables X_i with values from domain D_i

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

Solution is a complete, consistent assignment

For example, if X1 and X2 both have the domain {A,B}, then the constraint saying the two variables must have different values can be written as (X1,X2), [(A,B), (B,A)] or as (X1,X2),X1 = X2.Each state in a CSP is defined by an assignment of values to some or all of the variables, {Xi =vi,Xj = vj , . . .}.

- An assignment that does not violate any constraints is called a consistent or legal assignment.
- A complete assignment is one in which every variable is assigned, anda solution to a CSP is a consistent, complete assignment.
- A partial assignment is one that assigns values to only some of the variables.

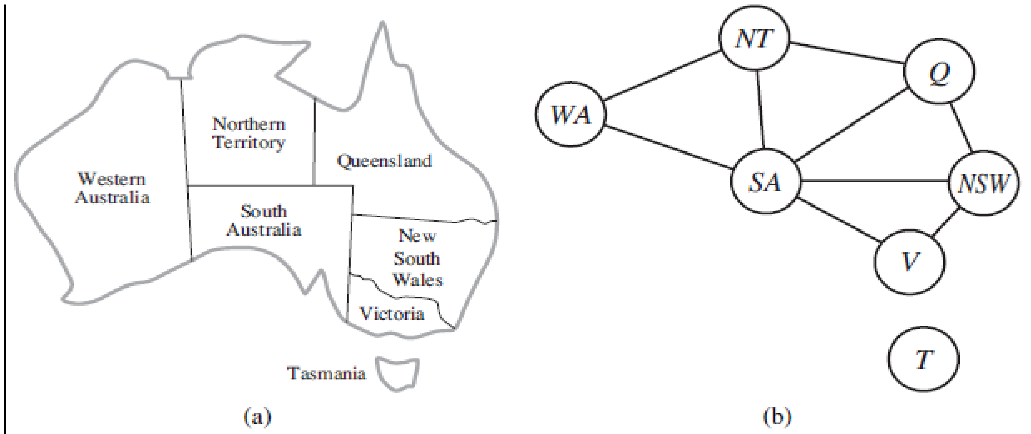
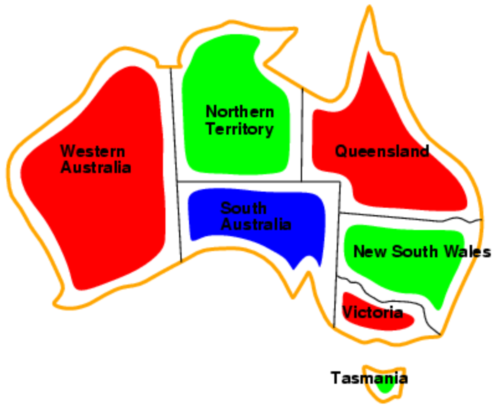


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

- **Variables:** WA, NT, Q, NSW, V, SA, T
- **Domains:** {red, green, blue}

Constraints: adjacent regions must have different colors
e.g., WA ≠ NT, or (WA, NT) in {(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)}

- coloring each region either red, green, or blue in such a way that no neighboring regions have the same color.
- To formulate this as a CSP, we define the variables to be the regions :X = {WA,NT,Q,NSW, V,SA, T} .
- The domain of each variable is the set Di = {red , green, blue}.
- The constraints: neighboring regions- to have distinct colors.
- C = {SA = WA,SA = NT,SA = Q,SA = NSW,SA = V,WA = NT,NT = Q,Q = NSW,NSW = V } .
- SA = WA can be fully enumerated in turn as
  {(red , green), (red , blue), (green, red), (green, blue), (blue, red), (blue, green)} .
- There are many possible solutions to this problem, such as
  {WA=red ,NT =green,Q=red ,NSW =green, V =red ,SA=blue, T =red }.



-
- **Solutions** are *complete* and *consistent* assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

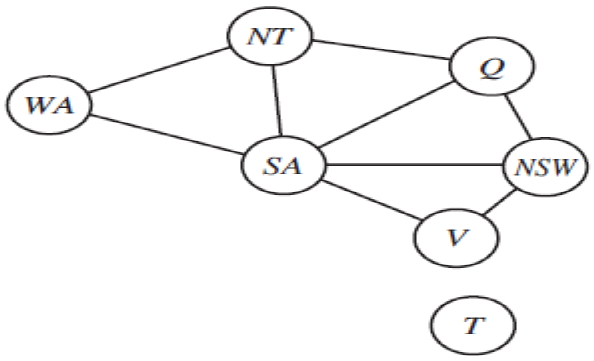
- 3^5 =243 assignments
- Easy elimination process in CSP : ex: {SA=blue}
- None of them will get blue
- only 25 =32 assignments to look at, a reduction of 87%.

depth-limited search: partial assignment, and an action would be adding var = value to the assignment.

CSP with n variables of domain size d, : the branching factor at the top level is nd because any of d values can be assigned to any of n variables. Atthe next level, the branching factor is (n − 1)d, and so on for n levels. We generate a tree with n! · dn leaves, even though there are only dn possible complete assignments!

COMMUTATIVITY : A problem is commutative if the order of application of any given set of actions has no effect on the outcome. need only consider a single variable at each node in the search tree.
For example, at the root node of a search tree for coloring the map of Australia, we might make a choice between SA=red, SA=green, and SA=blue, but we would never choose between SA=red and WA=blue.
With this restriction, the number of leaves is dn, as we would hope.

An algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of inference called constraint propagation: using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.
The key idea is local consistency. If we treat each variable as a node in a graph and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph



NODE CONSISTENT

A single variable (corresponding to a node in the CSP network) is node-consistent if all the values in the variable's domain satisfy the variable's unary constraints.
For example: SA dislike green, the variable SA starts with domain {red , green, blue}, make it node consistent by eliminating green, leaving SA with the reduced domain {red , blue}.

ARC CONSISTENT of var

- every value in its domain satisfies the variable's binary constraints.
- Xi is arc-consistent with respect to another variable Xj if for every value in the current domain Di there is some value in the domain Dj that satisfies the binary constraint on the arc (Xi,Xj).
- A network is arc-consistent if every variable is arc consistent with every other variable.
- For example, consider the constraint Y = X2 where the domain of both X and Y is the set of digits.
- We can write this constraint explicitly as (X, Y), {(0, 0), (1, 1), (2, 4), (3, 9))} .
- To make X arc-consistent with respect to Y , we reduce X's domain to {0, 1, 2, 3}.
- make Y arc-consistent with respect to X, then Y 's domain becomes {0, 1, 4, 9} and the whole CSP is arc-consistent.

PATH CONSISTENT

A two-variable set {Xi,Xj} is path-consistent with respect to a third variable Xm if, for every assignment {Xi = a,Xj = b} consistent with the constraints on {Xi,Xj}, there is an assignment to Xm that satisfies the constraints on {Xi,Xm} and {Xm,Xj}.
This is called path consistency because one can think of it as looking at a path from Xi to Xj with Xm in the middle.
Ex: coloring the Australia map with two colors.
make the set {WA,SA} path consistent with respect to NT.
only two enumerations : {WA = red ,SA = blue} and {WA = blue,SA = red}.
with both of these assignments NT can be neither red nor blue (because it would conflict with either WA or SA). Because there is no valid choice for NT, eliminate both assignments, and end up with no valid assignments for {WA,SA}.

Global constraints

-resource constraint- atmost constraint
-For example, in a scheduling problem, let P1, . . . , P4 denote the numbers of personnel assigned to -each of four tasks.
-Constraint:no more than 10 personnel are assigned in total is written as Atmost(10, P1, P2, P3, P4).
for example, if each variable has the domain {3, 4, 5, 6}?

Real – World CSP’s:-

1. Assignment Problems
   Eg:- who teaches what class
2. Timetable problems
   Eg: which class is offered when and where?
3. Transportation Scheduling
4. Factory Scheduling

CRYPT ARTHEMATIC PROBLEM

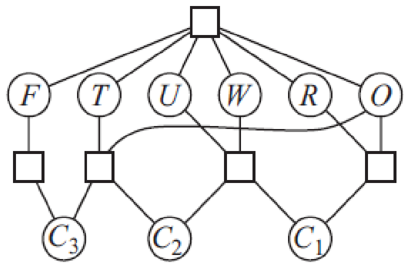
(The name is traditional but confusing because it need not involve *all* the variables in a problem). One of the most common global constraints is *Alldiff*, which says that all of the variables involved in the constraint must have different values. In Sudoku problems (see Section 6.2.6), all variables in a row or column must satisfy an *Alldiff* constraint. Another example is provided by **cryptarithmetic** puzzles. (See Figure 6.2(a).) Each letter in a cryptarithmetic puzzle represents a different digit. For the case in Figure 6.2(a), this would be represented as the global constraint $Alldiff(F,T,U,W,R,O)$. The addition constraints on the four columns of the puzzle can be written as the following n-ary constraints:

$$
\begin{aligned}
&O + O = R + 10 \cdot C_{10}\\
&C_{10} + W + W = U + 10 \cdot C_{100}\\
&C_{100} + T + T = O + 10 \cdot C_{1000}\\
&C_{1000} = F\ ,
\end{aligned}
$$

where C_{10}, C_{100}, and C_{1000} are auxiliary variables representing the digit carried over into the tens, hundreds, or thousands column. These constraints can be represented in a **constraint hypergraph**, such as the one shown in Figure 6.2(b). A hypergraph consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent n-ary constraints.

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

(a)



(b)

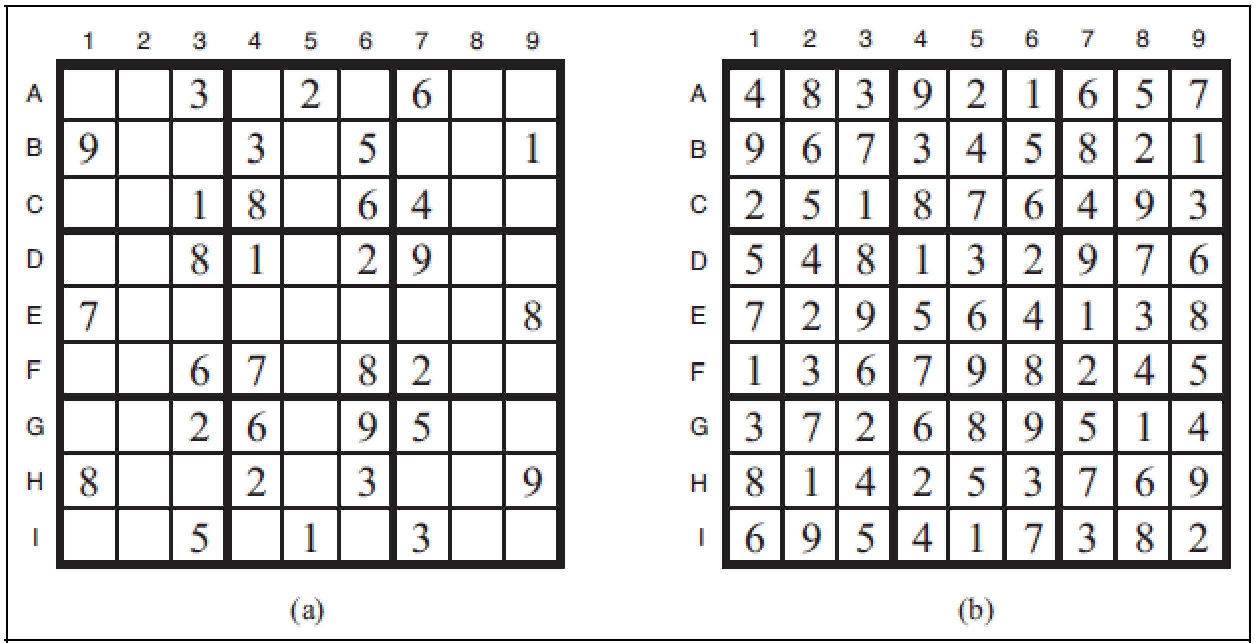
Figure 6.2 (a) A cryptarithmetic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed. (b) The constraint hypergraph for the cryptarithmetic problem, showing the *Alldiff* constraint (square box at the top) as well as the column addition constraints (four square boxes in the middle). The variables C_1, C_2, and C_3 represent the carry digits for the three columns.

bounds propagation

For example, in an airline-scheduling problem

let's suppose there are two flights, F1 and F2, for which the planes have capacities 165 and 385, respectively.
The initial domains for the numbers of passengers on each flight are then
    D1 = [0, 165] and D2 = [0, 385] .
Now suppose we have the additional constraint that the two flights together must carry 420 people: F1 + F2 = 420.
Propagating bounds constraints, we reduce the domains to ?
D1 = [35, 165] and D2 = [255, 385] .

SUDOKU EXAMPLE:-



(a)

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

(b)

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

Figure 6.4 (a) A Sudoku puzzle and (b) its solution.

Alldiff constraints: one for each row, column, and box of 9 squares.

$$Alldiff(A1, A2, A3, A4, A5, A6, A7, A8, A9)$$
$$Alldiff(B1, B2, B3, B4, B5, B6, B7, B8, B9)$$
$$\cdots$$
$$Alldiff(A1, B1, C1, D1, E1, F1, G1, H1, I1)$$
$$Alldiff(A2, B2, C2, D2, E2, F2, G2, H2, I2)$$
$$\cdots$$
$$Alldiff(A1, A2, A3, B1, B2, B3, C1, C2, C3)$$
$$Alldiff(A4, A5, A6, B4, B5, B6, C4, C5, C6)$$
$$\cdots$$

Adversarial Search

Optimal decisions in games

We will consider games with two players, whom we will call **MAX** and MIN for reasons that will soon become obvious. **MAX** moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A game can be formally defined as a kind of search problem with the following components:

- The initial state, which includes the board position and identifies the player to move.
- A successor function, which returns a list of *(move, state)* pairs, each indicating a legal move and the resulting state.
- **A terminal test,** which determines when the galme is over. States where the game has ended are called **terminal states.**
- **A utility function** (also called an objective function or payoff function), which gives a numeric value for the terminal states. In chess, the outcome is a win, loss, or draw, with values +1, -1, or 0. Some games have a wider ,variety of possible outcomes; the payoffs in backgammon range from +I92 to -192. This chapter deals mainly with zero-sum games, although we will briefly mention non-zero-sum games

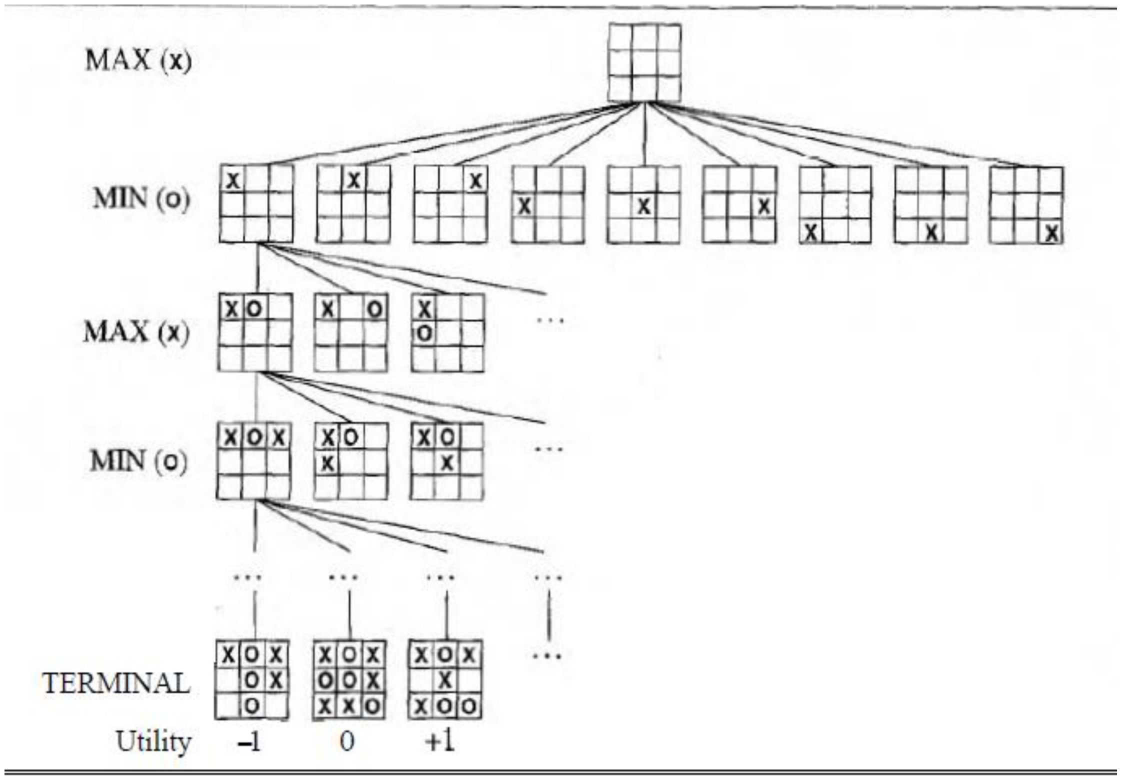


Figure 6.1 A (partial) search tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the search tree, giving alternating moves by MIN (O) and MAX, until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

Figure6.1 shows part of the game tree for tic-tac-toe (noughts and crosses). From the initial state, MAX has nine possible moves. Play alternates between MAX'S placing an X and MIN'S placing an O until we reach leaf nodes corresponding to ter.mina1 states such that one player has three in a row or all the squares are filled. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN (which is how the players get their names). It is MAX'S job to use the search tree (particularly the utility of terminal states) to determine the best move.)

Optimal strategies

Minimax Process:

Given a game tree, the optimal strategy can be determined by examining the **minimax value** of each node, which we write as MINIMAX- VALUE(n). The minimax value of a node is the utility (for MAX) of being in the corresponding state, *assuming that both players play optimally* from there to the end of the game. Obviously, the minimax value of a terminal state is just its utility. Furthermore, given a choice, MAX will prefer to move to a state of maximum value, whereas MIN prefers a state of minimum value. So we have the following:

MINIMAX- VALUE(n) =

UTILITY(n) if n is a terminal state

$max_{s \epsilon E Successor(n)}$MINIMAX-VALUE(S)f *n* is a MAX node

$min_{s \epsilon E Successor(n)}$*MINI*MAX-VALUE(S)if *n* is a MIN node.

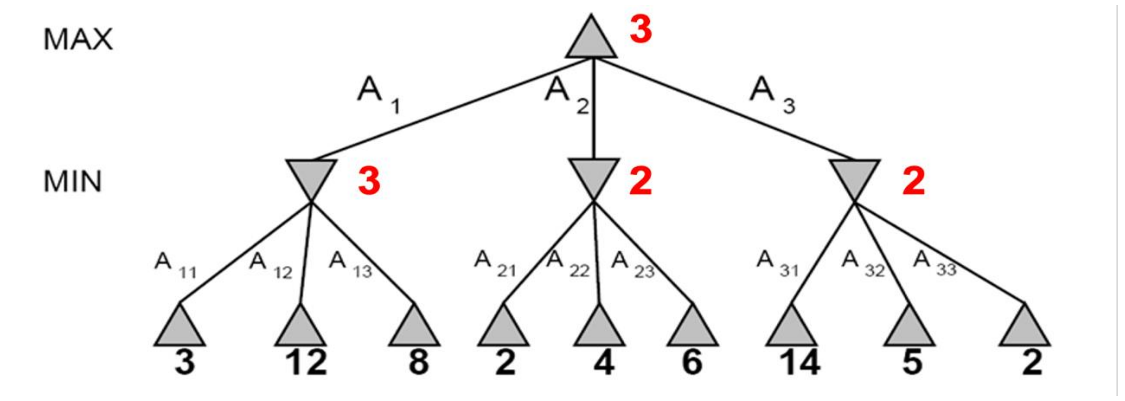


Figure 6.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the successor with the highest minimax value, and MIN's best reply is b_1, because it leads to the successor with the lowest minimax value.

Let us apply these definitions to the game tree in Figure 6.2. The terminal nodes on the bottom level are already labeled with their utility values. The first MIN node, labeled B, has three successors with values 3, 12, and 8, so its minimax value is **3.** Similarly, the other two MIN nodes have minimax value 2. The root node is a MAX node; its successors have minimax values 3, 2, and 2; so it has a minimax value of 3. We can also identify the **minimax decision** at the root: action *a1* is the optimal choice for MAX because it leads to the successor with the highest minimax value.

The minimax algorithm

(The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed** up through the tree as the recursion unwinds. For example, in Figure 6.2, the algorithm first recurses down to the three bottom left nodes, and uses the UTILITY function on them to discover that their values are 3, 12, and 8 respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B. A similar process gives the backed up values of 2 for *C* and 2 for *D*.

Finally,we take the maximum of 3,2, and 2 to get the backed-up value of 3 for the root node. The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is *m*, and there are *b* legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is $O(bm)$ for an algorithm that generates all successors at once, or $O(m)$ for an algorithm that generates successors one at a time (see page 76).)

```
function MINIMAX-DECISION(state) returns an action
   inputs: state, current state in game

   $v \leftarrow$ MAX-VALUE(state)
   return the action in SUCCESSORS(state) with value $v$
```

```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$
   for a, s in SUCCESSORS(state) do
      $v \leftarrow$ MAX($v$, MIN-VALUE($s$))
   return $v$
```

```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow \infty$
   for a, s in SUCCESSORS(state) do
      $v \leftarrow$ MIN($v$, MAX-VALUE($s$))
   return $v$
```

Figure 6.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions *MAX-VALUE* and *MIN-VALUE* go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state.

Optimal decisions in multiplayer games

Let us examine how to extend the minimax idea to multiplayer games. For example, in a three-player game with players A, B, and *C*, a vector (v_A, v_B, v_C) is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint.

Now we have to consider nonterminal states. Consider the node marked X in the game tree shown in Figure 6.4. In that state, player *C* chooses what to do. The two choices lead to terminal states with utility vectors $(v_A=1, v_B=2, v_C=6)$ and $(v_A=4, v_B=2, v_C=3)$. Since 6 is bigger than 3, *C* should choose the first move. This rneans that if state X is reached, subsequent play will lead to a terminal state with utilities $(v_A=1, v_B=2, v_C=6)$. Hence the backed-up value of X is this vector. In general, the backed-up value of a node n is the utility vector of whichever successor has the highest value for the player choosing at n.

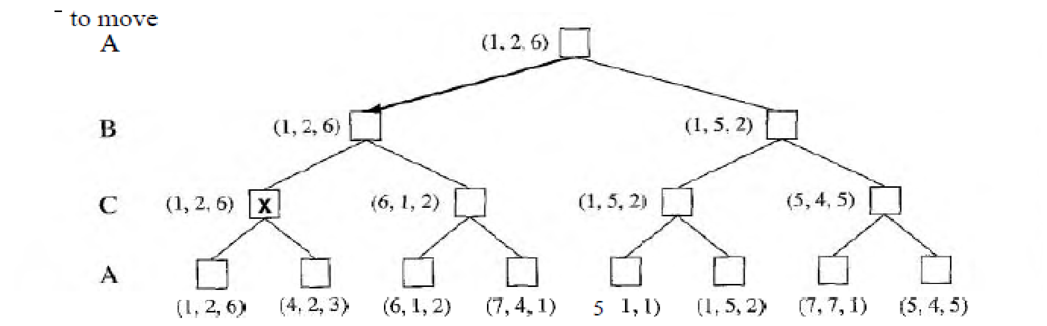
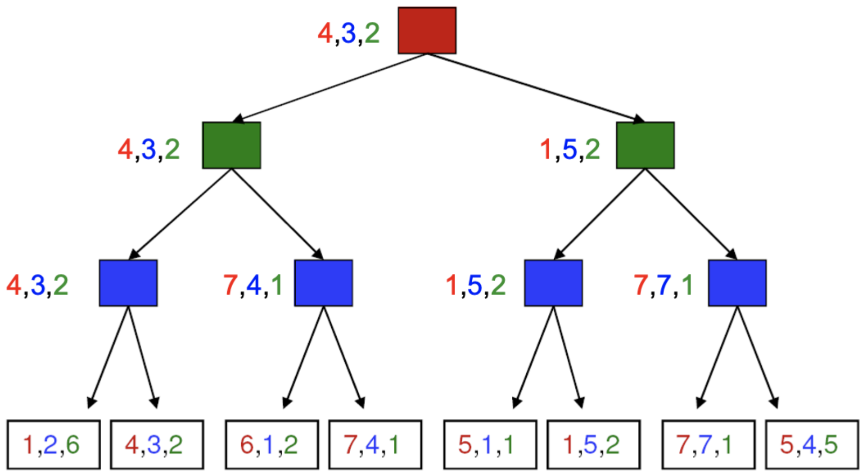


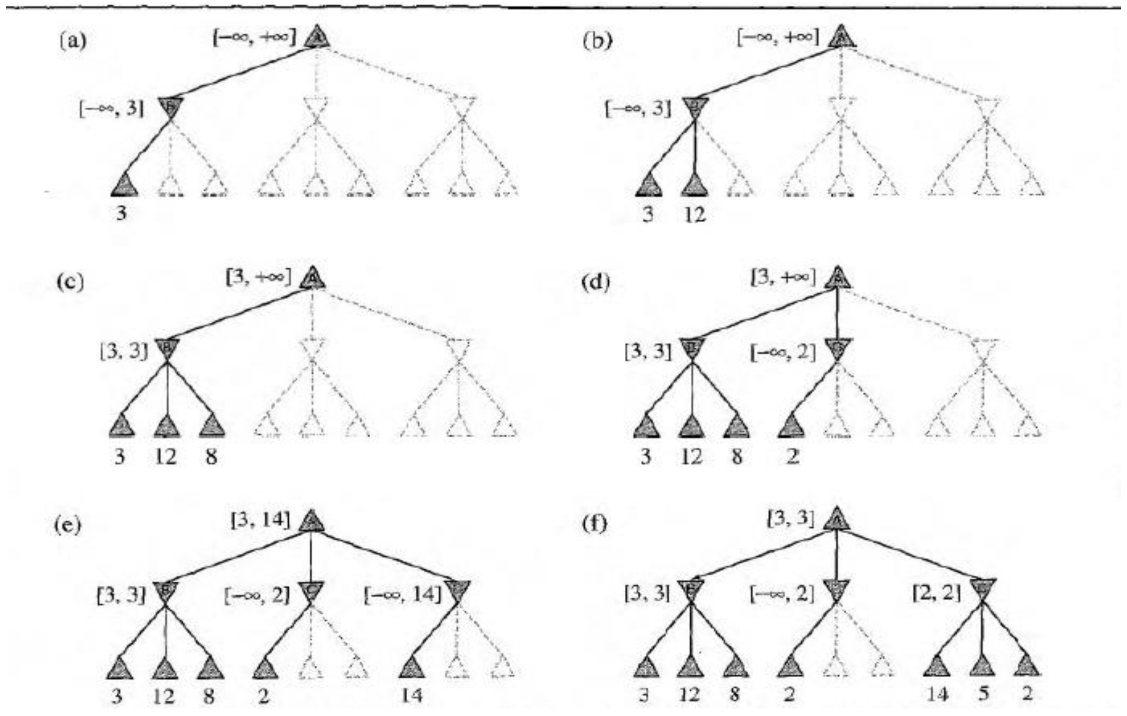
Figure 6.4 The first three ply of a game tree with three players (A, B, C). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

problem with minimax-number of game states it has to examine is exponential in the number of moves.-> can't eliminate the exponent, but we can effectively cut it in half.



Alpha-beta pruning Process

standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision
have values *x* and *y* and let *z* be
the minimum of x and *y*. The value of the root node is given by
*MINIMAX-VALUE(root)=max (min(3,12,8),m in(2,*x *, y) ,m in(14,5,2))*
= *max(3, min(2, x,* y), *2)*
= *max(3, z, 2)* where **;z<=2**
= **3.**
In other words, the value of the root and hence the minimax decision are *independent* of the values of the pruned leaves *x* and *y*.



following two parameters that describe bounds on the backed.-up values that appear anywhere along the path:

***α* = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.**

***β* = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.**

alpha-beta needs to examine only *0($b^{m/2}$)* nodes to pick the best move, instead of *O(b^m)* for minimax. This means that the effective branching factor becomes instead of b--for chess, 6 instead of 35.

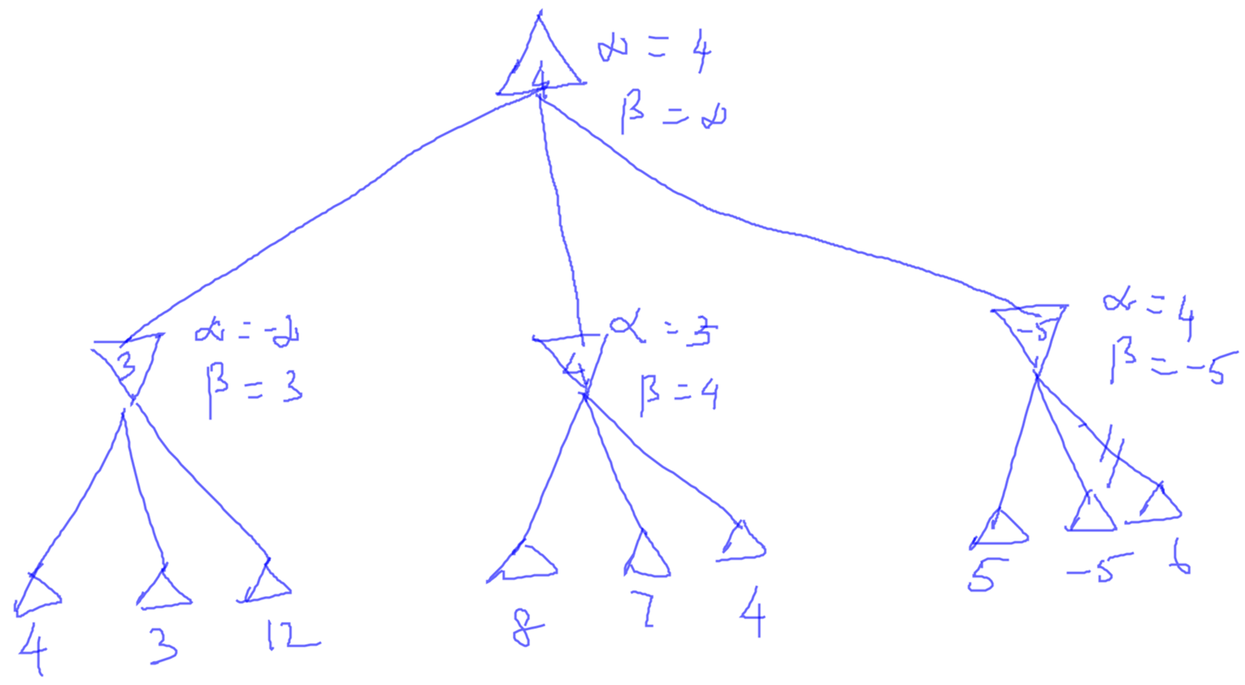
```
function ALPHA-BETA-SEARCH(state) returns an action
   inputs: state, current state in game

   $v \leftarrow$ MAX-VALUE(state, $-\infty$, $+\infty$)
   return the action in SUCCESSORS(state) with value $v$
```

```
function MAX-VALUE(state, $\alpha$, $\beta$) returns a utility value
   inputs: state, current state in game
           $\alpha$, the value of the best alternative for MAX along the path to state
           $\beta$, the value of the best alternative for MIN along the path to state

   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$
   for $a$, $s$ in SUCCESSORS(state) do
      $v \leftarrow$ MAX($v$, MIN-VALUE($s$, $\alpha$, $\beta$))
      if $v \geq \beta$ then return $v$
      $\alpha \leftarrow$ MAX($\alpha$, $v$)
   return $v$
```

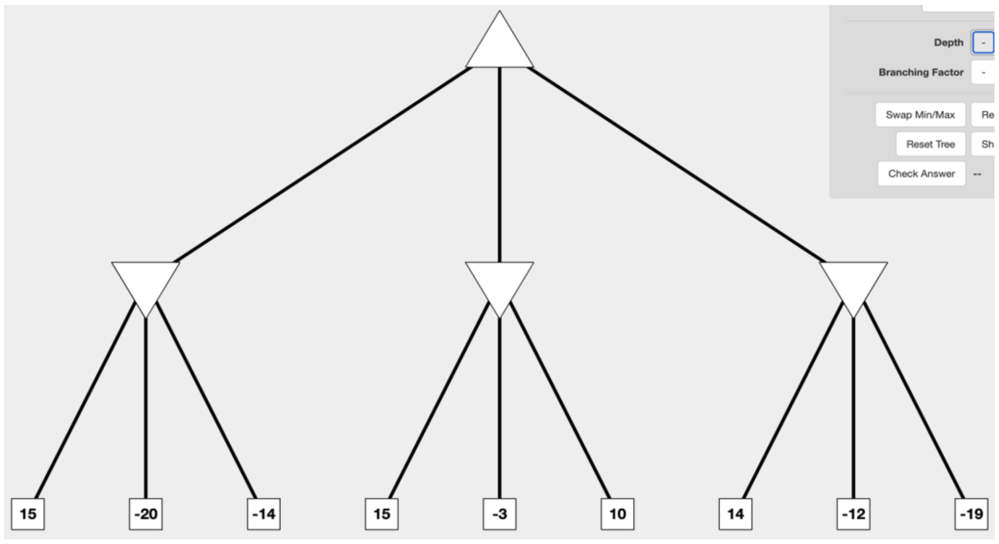
```
function MIN-VALUE(state, $\alpha$, $\beta$) returns a utility value
   inputs: state, current state in game
           $\alpha$, the value of the best alternative for MAX along the path to state
           $\beta$, the value of the best alternative for MIN along the path to state

   if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow +\infty$
   for $a$, $s$ in SUCCESSORS(state) do
      $v \leftarrow$ MIN($v$, MAX-VALUE(%$\alpha$, $\beta$))
      if $v \leq \alpha$ then return $v$
      $\beta \leftarrow$ MIN($\beta$, $v$)
   return $v$
```

- Pruning does not affect final result
- Amount of pruning depends on move ordering
    - Should start with the “best” moves (highest-value for MAX or lowest-value for MIN)
    - For chess, can try captures first, then threats, then forward moves, then backward moves
    - Can also try to remember “killer moves” from other branches of the tree
- With perfect ordering, the time to find the best move is reduced to $O(b^{m/2})$ from $O(b^m)$
    - Depth of search is effectively doubled

Example:2



Exercise 1:



For more practice use the web site: https://pascscha.ch/info2/abTreePractice/