UNIT III GAME PLAYING AND CSP

Game theory –Optimal decisions in games –Alpha-beta search –Monte-carlo tree search Stochastic games –Partially observable games-Constraint satisfaction problems –Constraint propagation –Backtracking search for CSP –Local search for CSP –Structure of CSP

Introduction

When two or more agents have conflicting goals in a competitive environment it gives rise to **adversarial search** problems.

Game Theory

A game can be defined by the **initial state** (how the board is set up), the legal **actions** in each state, the **result** of each action, a **terminal test** (which says when the game is over), and a **utility function** that applies to terminal states to say who won and what the final score is.
The three opinions that can be considered for a multiagent environment are

- The first stance, appropriate when there are a very large number of agents, is to consider them in the aggregate as an **economy**, without having to predict the action of any individual agent.

- Second, an adversarial agents can be considered as just a part of the environment—a part that makes the environment nondeterministic.
- The third stance is to explicitly model the adversarial agents with the techniques of adversarial game-tree search.

Pruning is a technique that makes the search more efficient by ignoring portions of the search tree that make no difference to the optimal move.
A heuristic **evaluation function** estimates who is winning based on features of the state or the outcomes of many fast simulations of the game are averaged from that state all the way to the end

Two-player zero-sum games

The games most commonly studied within AI (such as chess and Go) are what game theorists call deterministic, two-player, turn-taking, **perfect information**, **zero-sum games**. "Perfect information" is a synonym for "fully observable," and "zero-sum" means that what is good for one player is just as bad for the other: there is no "win-win" outcome. For games the term **move is used** as a synonym for "action" and **position** as a synonym for "state."

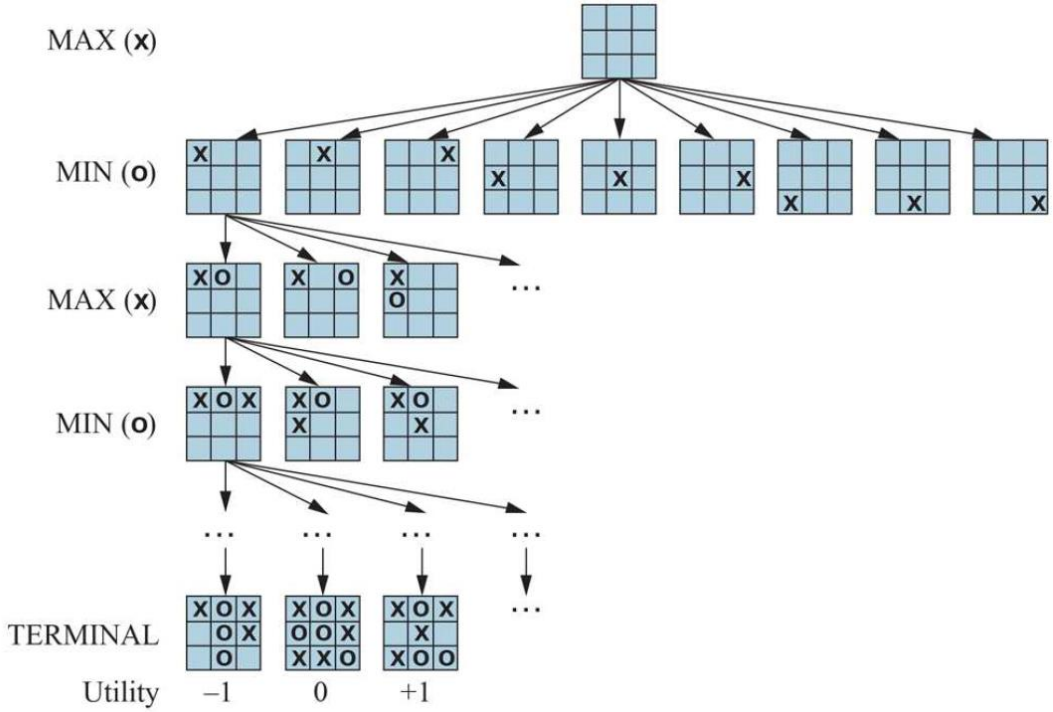
Types of Game

Perfect Information Game: In which player knows all the possible moves of himself and opponent and their results. E.g. Chess.
Imperfect Information Game: In which player does not know all the possible moves of the opponent. E.g. Bridge since all the cards are not visible to player.

A game can be formally defined with the following elements:
S_0: The **initial state**, which specifies how the game is set up at the start.
TO-MOVE(s) : The player whose turn it is to move in state s.
ACTIONS (s): The set of legal moves in state s .

RESULT(s,a) : The **transition model**, which defines the state resulting from taking action a in state s .
IS-TERMINAL(s) : A **terminal test**, which is true when the game is over and false otherwise. states where the game has ended are called **terminal states**.
UTILITY(s,p): A **utility function** (also called an objective function or payoff function), which defines the final numeric value to player p when the game ends in terminal state s

In chess, the outcome is a win, loss, or draw, with values 1 , 0, or 1/2
1.Some games have a wider range of possible outcomes—for example, the payoffs in backgammon range from 0 to 192 .
2 Chess is considered a "zero-sum" game, even though the sum of the outcomes for the two players is +1 for each game , not zero. "Constant sum" would have been a more accurate term, but zero-sum is traditional and makes sense when each player is charged an entry fee of ½
The **state space graph in a game is** a graph where the vertices are states, the edges are moves and a state might be reached by multiple paths. the complete **game tree can be defined** as a search tree that follows every sequence of moves all the way to a terminal state. The game tree may be infinite if the state space itself is unbounded or if the rules of the game allow for infinitely repeating positions.
Figure shows part of the game tree for tic-tac-toe (noughts and crosses).



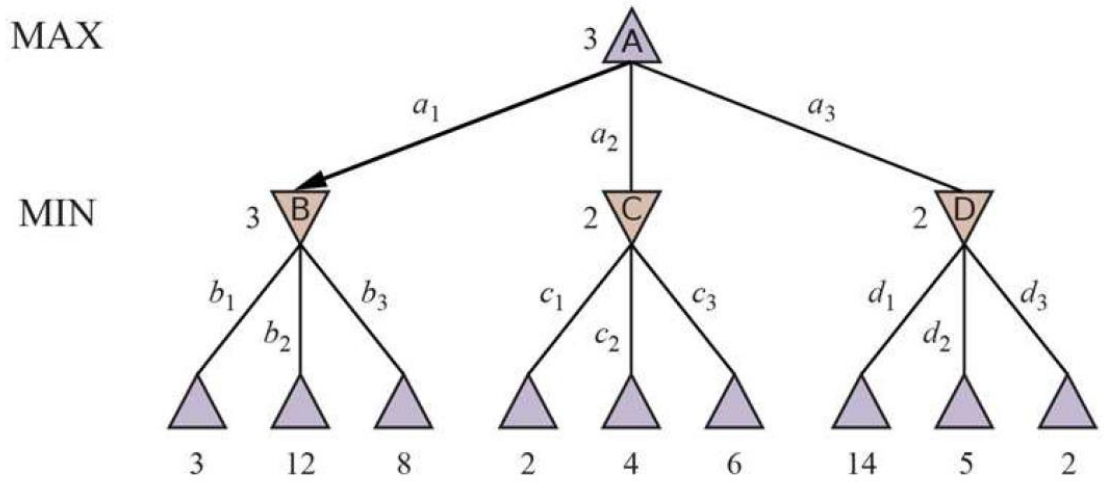
From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing an O until the leaf nodes corresponding to terminal states are reached such that one player has three squares in a row or all the squares are filled. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are good for MAX and bad for MIN .

Optimal Decisions in Games

MAX wants to find a sequence of actions leading to a win, but MIN has something to say about it. This means that MAX's strategy must be a conditional plan—a contingent strategy specifying a response to each of MIN's possible moves. In games that have a binary outcome (win or lose), AND–OR search can be used to generate the conditional plan. In fact, for such games, the definition of a winning strategy for the game is identical to the definition of a solution for a nondeterministic planning problem: in both cases the desirable outcome must be guaranteed no matter what the "other side" does. For games with multiple outcome scores, a more general algorithm called **minimax search is used**.

Consider the trivial game in Figure. A two-ply game tree. The Δ 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 a1, because it leads to the state with the highest minimax value, and MIN's best reply is b1, because it leads to the state with the lowest minimax value.

The possible moves for MAX at the root node are labeled a1, a2, and a3. The possible replies to for MIN are b1, b2, b3 and so on. This particular game ends after one move each by MAX and MIN. (**NOTE:** In some games, the word "move" means that both players have taken an action; therefore the word **ply** is used to unambiguously mean one move by one player, bringing us one level deeper in the game tree.) The utilities of the terminal states in this game range from 2 to 14.



Given a game tree, the optimal strategy can be determined by working out the **minimax value** of each state in the tree, which can be written as MINIMAX(*s*). The minimax value is the utility (for MAX) of being in that state, *assuming that both players play optimally* from there to the end of the game. The minimax value of a terminal state is just its utility. In a non-terminal state, MAX prefers to move to a state of maximum value when it is MAX's turn to move, and MIN prefers a state of minimum value (that is, minimum value for MAX and thus maximum value for MIN).Hence:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s,\text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,\ a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,\ a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

The terminal nodes on the bottom level get their utility values from the game's UTILITY function. The first MIN node, labeled B , has three successor states 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 successor states have minimax values 3, 2, and 2; so it has a minimax value of 3. The **minimax decision** at the root can be also identified: action a1 is the optimal choice for MAX because it leads to the state with the highest minimax value.This definition of optimal play for MAX assumes that MIN also plays optimally.

The minimax search algorithm

MINIMAX(*s*) can be computed which in turn can be converted into a search algorithm that finds the best move for MAX by trying all actions and choosing the one whose resulting state has the highest MINIMAX value.

Figure shows the algorithm.

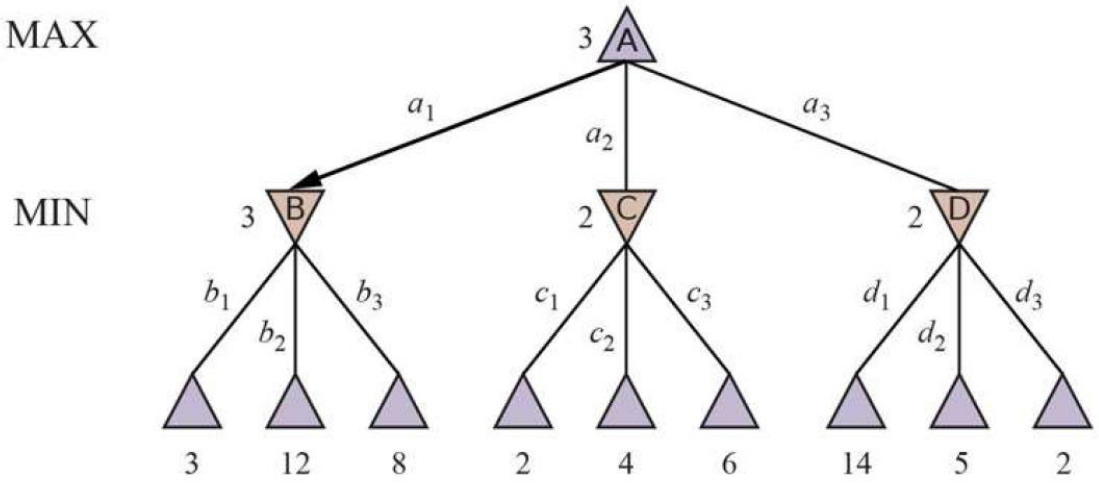
```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

It is a recursive algorithm that proceeds all the way down to the leaves of the tree and then **backs up** the minimax values through the tree as the recursion unwinds.

For example, consider the figure



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, the maximum of 3, 2, and 2 is taken to get the backed-up value of 3 for the root node.

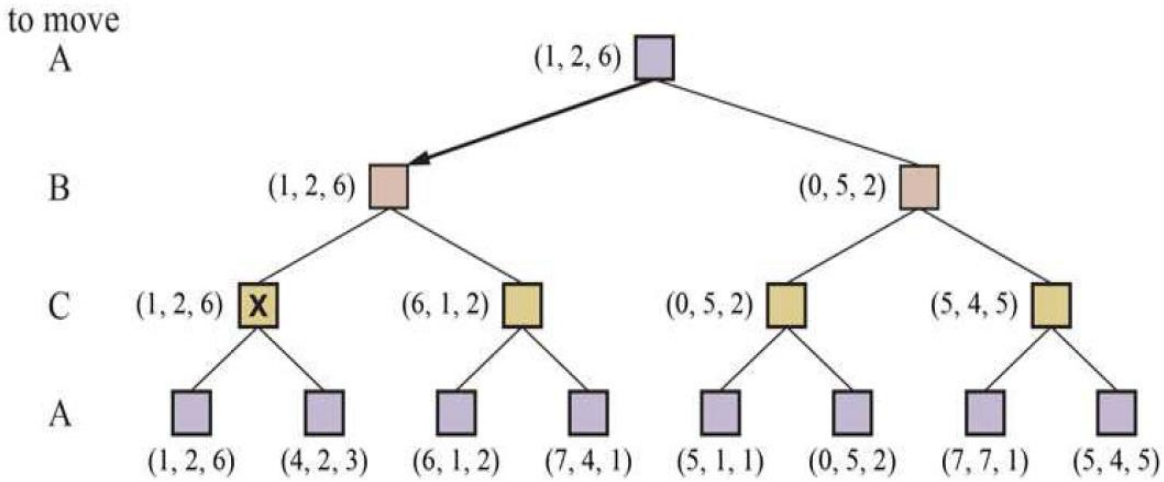
The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time . The exponential complexity makes MINIMAX impractical for complex games.

Optimal decisions in multiplayer games

Many popular games allow more than two players. The minimax idea can be extended to multiplayer games. First, the single value for each node has to be replaced with a *vector* of values. For example, in a three-player game with players A,B , and C , a vector $< V_A, V_B, Vc >$ is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint.

(In two-player, zero-sum games, the two-element vector can be reduced to a single value because the values are always opposite.) The simplest way to implement this is to have the UTILITY function return a vector of utilities.

Consider the nonterminal states. Consider the node marked X in the game tree shown in **Figure**.



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.

In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors $< V_A=1, V_B=2, Vc=6 >$ and $< V_A=4, V_B=2, Vc=3 >$. Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to a terminal state with utilities $< V_A=1, V_B=2, Vc=6 >$. 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 the successor state with the highest value for the player choosing at n .

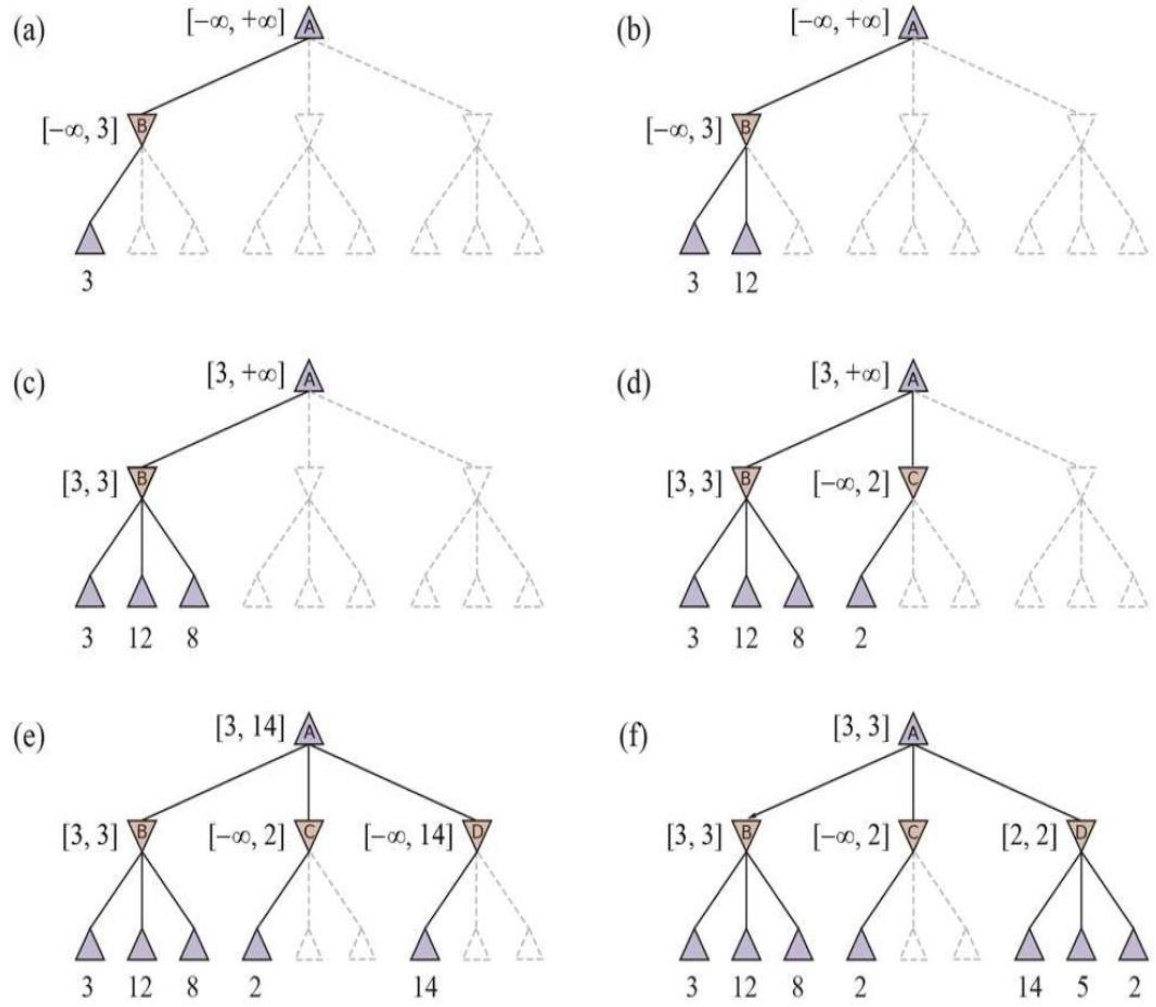
Multiplayer games usually involve **alliances(unions)**, whether formal or informal, among the players. Alliances are made and broken as the game proceeds. For example, suppose A and B are in weak positions and C is in a stronger position. Then it is often optimal for both A and B to attack rather than each other, before C destroys each of them individually. In this way, collaboration emerges from purely selfish behavior. Of course, as soon as C weakens under the joint attack, the alliance loses its value, and either A or B could violate the agreement.

Alpha– beta pruning

MINIMAX searches the entire tree, even if in some cases the rest can be ignored. But, this alpha beta cutoff returns appropriate minimax decision without exploring entire tree. The minimax search procedure is slightly modified by including branch and bound strategy one for each players. This modified strategy is known as alpha beta pruning. It requires maintenance of two threshold values, one representing a lower bound on the value that a maximizing node may ultimately be assigned (alpha) and another representing upper bound on the value that a minimizing node may be assigned (beta).

Alpha– beta pruning, when applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision

Consider again the two-ply game tree from Figure .



At each point, range of possible values for each node is shown. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all 's B successor states, so the value of B is exactly 3. Now we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence,C , which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C .Therefore, there is no point in looking at the other successor states of C . This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we nowhave bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of is worth 5, so again we need to keep exploring. The third successor is worth 2, so now is worth exactly 2. MAX's decision at the root is to move to , giving a value of 3

Let the two unevaluated successors of node C in Figure have the values x and y respectively. Then the value of the root node is given by
MINIMAX(root) = max(min(3, 12, 8), min(2, x, y), min(14, 5, 2))
= max(3, min(2, x, y), 2)
= max(3, z, 2) where z = min(2, x, y) ≤ 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. Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves.

The general principle is this: consider a node n somewhere in the tree ,such that Player has a choice of moving to that node. If Player has a better choice m either at the parent node of n or at any choice point further up, then n *will never be reached in actual play.*

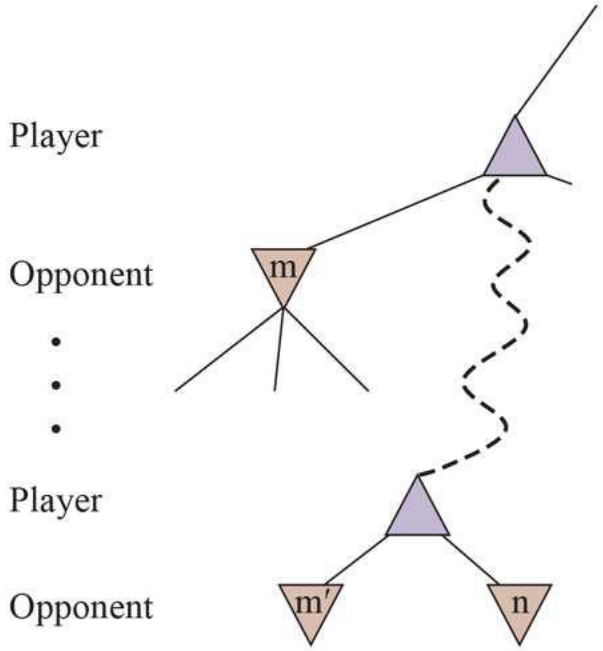


Figure: The general case for alpha–beta pruning.

Alpha–beta pruning gets its name from the 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.

The complete algorithm is given in Figure

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively

Move ordering

The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined. For example, in Figure (e) and (f) , any successors of D could not be pruned at all because the worst successors (from the point of view of MIN) were generated first. If the third successor D of had been generated first, with value 2, then it would be able to prune the other two successors. This suggests that it might be worthwhile to try to first examine the successors that are likely to be best.

If this could be done perfectly, alpha–beta would need to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$for minimax.

Alpha–beta with perfect move ordering can solve a tree roughly twice as deep as minimax in the same amount of time. With random move ordering, the total number of nodes examined will be roughly $O(b^{3m/4})$ for moderate . If *perfect* move ordering is not achieved—in that case the ordering function could be used to play a perfect game!

The process of **iterative deepening** can be used for dynamic move-ordering schemes**.** First, search one ply deep and record the ranking of moves based on their evaluations. Then search one ply deeper, using the previous ranking to inform move ordering; and so on. The increased search time from iterative deepening can be more than made up from better move ordering. The best moves are known as **killer moves**, and to try them first is called the killer move heuristic.

Redundant paths to repeated states can cause an exponential increase in search cost, and that keeping a table of previously reached states can address this problem. In game tree search, repeated states can occur because of **transpositions**—different permutations of the move sequence that end up in the same position, and the problem can be addressed with a **transposition table** that caches the heuristic value of states.

Even with alpha–beta pruning and clever move ordering, minimax won't work for games like chess and Go, because there are still too many states to explore in the time available. Two strategies are proposed to address this problem :
Type A strategy considers all possible moves to a certain depth in the search tree, and then uses a heuristic evaluation function to estimate the utility of states at that depth. It explores a *wide but shallow* portion of the tree. A **Type B strategy** ignores moves that look bad, and follows promising lines "as far as possible." It explores a *deep but narrow* portion of the tree. Historically, most chess programs have been Type A whereas Go programs are more often Type B because the branching factor is much higher in Go,

Heuristic Alpha–Beta Tree Search
To limit computation time, the search can be cut off early and a heuristic **evaluation function is applied** to states, effectively treating nonterminal nodes as if they were terminal. In other words, the UTILITY function is replaced with EVAL, which estimates a state's utility and also replace the terminal test by a **cutoff test**, which must return true for terminal states, but is otherwise free to decide when to cut off the search, based on the search depth and any property of the state that it chooses to consider. This gives the formula HMINIMAX(s, d) for the heuristic minimax value of state s at search depth d :

$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s,\text{MAX}) & \text{if IS-CUTOFF}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,\ a), d+1) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,\ a), d+1) & \text{if TO-MOVE}(s) = \text{MIN}. \end{cases}$$

Evaluation functions

A heuristic evaluation function EVAL(s,p)returns an *estimate* of the expected utility of state s to player p. For terminal states, it must be EVAL(s,p)=UTILITY(s,p)and for nonterminal states, the evaluation must be somewhere between a loss and a win:
UTILITY(loss,p)≤ EVAL(s,p) ≤ UTILITY(win,p)

Requirements for a good evaluation function:

First, the computation must not take too long!
Second, the evaluation function should be strongly correlated with the actual chances of winning. But if the search must be cut off at nonterminal states, then the algorithm will necessarily be *uncertain* about the final outcomes.

Mathematically, a **weighted linear function of an** evaluation function can be expressed as

$$\mathrm{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)\,,$$

where each f_i is a feature of the position and each w_i is a weight .The weights should be normalized so that the sum is always within the range of a loss (0) to a win (+1).

The evaluation function should be strongly correlated with the actual chances of winning, but it need not be linearly correlated: for instance, if state s is twice as likely to win as state s' it doesn't require that EVAL(S) be twice EVAL(S'); all it requires is EVAL(s)> EVAL(s').

Cutting off search

The next step is to modify ALPHA-BETA-SEARCH so that it will call the heuristic EVAL function when it is appropriate to cut off the search. This can be done by replacing IS TERMINAL with the following line in the algorithm of Alpha-Beta Search

if *game*.IS-CUTOFF(*state*, *depth*) **then return** *game*.EVAL(*state*, *player*), *null*

Some bookkeeping (record) must be maintained so that the current *depth* is incremented on each recursive call. The most straightforward approach to controlling the amount of search is to set a fixed depth limit so that IS-CUTOFF(*state*, *depth*) returns *true* for all *depth* greater than some fixed depth (as well as for all terminal states). The depth d is chosen so that a move is selected within the allocated time. A more robust approach is to apply iterative deepening. When time runs out, the program returns the move selected by the deepest completed search. As a bonus, if in each round of iterative deepening entries are kept in the transposition table, subsequent rounds will be faster, and evaluations can be used to improve move ordering. These simple approaches can lead to errors due to the approximate nature of the evaluation function.

The evaluation function should be applied only to positions that are **quiescent**—that is, positions in which there is no pending move (such as a capturing the queen) that would wildly swing the

evaluation. For nonquiescent positions the IS-CUTOFF returns false, and the search continues until quiescent positions are reached. This extra **quiescence search** is sometimes restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

Quiescence search is an algorithm typically used to extend search at unstable nodes in minimax game trees in game-playing computer programs. It is an extension of the evaluation function to defer evaluation until the position is stable enough to be evaluated statically, that is, without considering the history of the position or future moves from the position. It mitigates(reduces) the effect of the horizon problem faced by AI engines for various games like chess and Go.

Horizon effect: It arises when the program is facing an opponent's move that causes serious damage and is ultimately unavoidable, but can be temporarily avoided by the use of delaying tactics.

One strategy to mitigate the horizon effect is to allow **singular extensions** moves that are clearly better than all other moves in a given position even when the search would be normally cut off at that point

Forward pruning

Alpha–beta pruning prunes branches of the tree that can have no effect on the final evaluation, but **forward pruning** prunes moves that appear to be poor moves, but might possibly be good ones. Thus, the strategy saves computation time at the risk of making an error. This is a Type B strategy.

One approach to forward pruning is **beam search** : on each ply, consider only a "beam" of the n best moves (according to the evaluation function) rather than considering all possible moves. Unfortunately, this approach is rather dangerous because there is no guarantee that the best move will not be pruned away.

The PROBCUT, or probabilistic cut, algorithm is a forward-pruning version of alpha–beta search that uses statistics gained from prior experience to lessen the chance that the best move will be pruned. Alpha–beta search prunes any node that is *provably* outside the current $(\propto, \beta)$ window. PROBCUT also prunes nodes that are *probably* outside the window. It computes this probability by doing a shallow search to compute the backed-up value v of a node and then using past experience to estimate how likely it is that a score of v at depth d in the tree would be outside$(\propto, \beta)$.

Another technique, **late move reduction**, works under the assumption that move ordering has been done well, and therefore moves that appear later in the list of possible moves are less likely to be good moves. But rather than pruning them away completely, just reduce the depth to which the moves are to be searched, thereby saving time. If the reduced search comes back with a value above the current $\propto$ value, then re-run the search with the full depth.

Search versus lookup

Many game-playing programs use *table lookup* rather than search for the opening and ending of games. In addition, computers can gather statistics from a database of previously played games to see which opening sequences most often lead to a win. For the first few moves there are few possibilities, and most positions will be in the table. Usually after about 10 or 15 moves a rarely seen position is ended up and the program must switch from table lookup to search. A computer, on the other hand, can completely *solve* the endgame by producing a **policy**, which is a mapping from every possible state to the best move in that state.

Limitations:

The game of Go illustrates two major weaknesses of heuristic alpha–beta tree search:

- First, Go has a branching factor that starts at 361, which means alpha–beta search would be limited to only 4 or 5 ply.
- Second, it is difficult to define a good evaluation function for Go because material value is not a strong indicator and most positions are in flux until the endgame.

In response to these two challenges, modern Go programs have abandoned alpha–beta search and instead use a strategy called **Monte Carlo tree search (MCTS)**

Monte Carlo tree search (MCTS)

The basic MCTS strategy does not use a heuristic evaluation function. Instead, the value of a state is estimated as the average utility over a number of **simulations** of complete games starting from the state. A simulation (also called a **playout** or **rollout**) chooses moves first for one player, then for the other, repeating until a terminal position is reached. At that point the rules of the game determine who has won or lost, and by what score. For games the only outcomes are a win or a loss, "average utility" is the same as "win percentage."

Choosing the moves during Playout

To get useful information from the playout a **playout policy is needed** that biases the moves towards good ones. For Go and other games, playout policies have been successfully learned from self-play by using neural networks. Sometimes game-specific heuristics are used, such as "consider capture moves" in chess or "take the corner square" in Othello.

Given a playout policy, the next step is to decide two things:

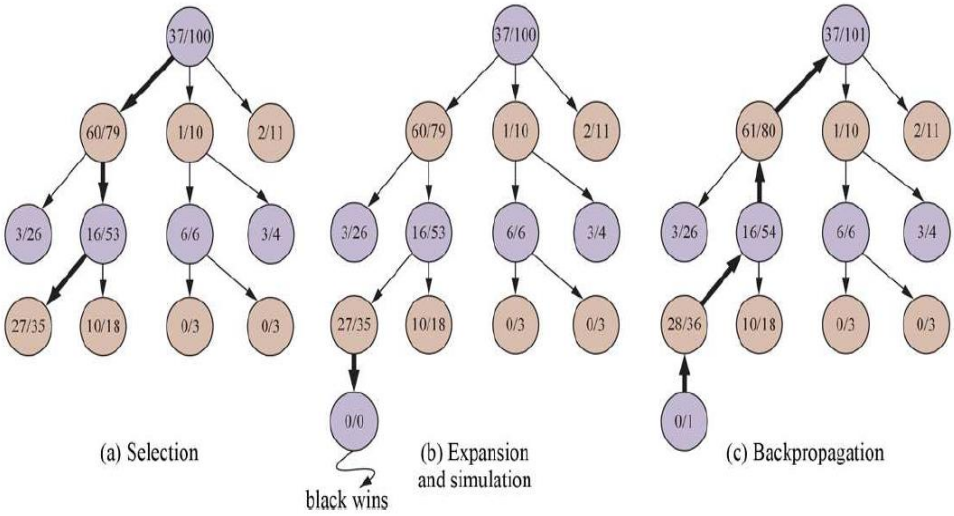
- From what positions to start the playouts, and
- how many playouts to be allocated to each position?

The simplest answer, called **pure Monte Carlo search**, is to do N simulations starting from the current state of the game, and track which of the possible moves from the current position has the highest win percentage.

For some stochastic games this converges to optimal play as N increases, but for most games it is not sufficient—in this case a **selection policy is needed** that selectively focuses the computational resources on the important parts of the game tree. It balances two factors: **exploration** of states that have had few playouts, and **exploitation** of states that have done well in past playouts, to get a more accurate estimate of their value. Monte Carlo tree search does that by maintaining a search tree and growing it on each iteration of the following four steps, as shown in figure

SELECTION: Starting at the root of the search tree, a move is chosen (guided by the selection policy), leading to a successor node, and repeat that process, moving down the tree to a leaf. Figure (a) shows a search tree with the root representing a state where white has just moved, and white has won 37 out of the 100 playouts done so far.

The thick arrow shows the selection of a move by black that leads to a node where black has won 60/79 playouts. This is the best win percentage among the three moves, so selecting it is an example of exploitation. But it would also have been reasonable to select the 2/11 node for the sake of exploration—with only 11 playouts, the node still has high uncertainty in its valuation, and might end up being best . Selection continues on to the leaf node marked 27/35.



EXPANSION: The search tree is grown by generating a new child of the selected node; **Figure** (b) shows the new node marked with 0/0. (Some versions generate more than one child in this step.)

SIMULATION: A playout is performed from the newly generated child node, choosing moves for both players according to the playout policy. These moves are *not* recorded in the search tree. In the figure, the simulation results in a win for black.

BACK-PROPAGATION: The result of the simulation is used to update all the search tree nodes going up to the root. Since black won the playout, black nodes are incremented in both the number of wins and the number of playouts, so 27/35 becomes 28/26 and 60/79 becomes 61/80. Since white lost, the white nodes are incremented in the number of playouts only, so 16/53 becomes 16/54 and the root 37/100 becomes 37/101.

These four steps are repeated either for a set number of iterations, or until the allotted time has expired, and then return the move with the highest number of playouts.

One very effective selection policy is called "upper confidence bounds applied to trees" or **UCT**. The policy ranks each possible move based on an upper confidence bound formula called **UCB1**. For a node , the formula is:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

Where U(n) is the total utility of all playouts that went through node n ,N(n) is the number of playouts through node n , and PARENT(n)is the parent node of n in the tree. Thus U(n)/N(n) is the exploitation term: the average utility of n . The term with the square root is the exploration term: it has the count N(n) in the denominator, which means the term will be high for nodes that have only been explored a few times. In the numerator it has the log of the number of times the parent of n is explored . This means that if n is selected some non-zero percentage of the time , the exploration term goes to zero as the counts increase, and eventually the playouts are given to the node with highest average utility.

C is a constant that balances exploitation and exploration. There is a theoretical argument that C should be $\sqrt{2}$, but in practice, game programmers try multiple values for C and choose the one that performs best
For example With C=1.4 , the 60/79 node in Figure has the highest UCB1 score, but with C=1.5, it would be the 2/11 node.

Figure shows the complete UCT MCTS algorithm. When the iterations terminate, the move with the highest number of playouts is returned. It would be better to return the node with the highest average utility, but the idea is that a node with 65/100 wins is better than one with 2/3 wins, because the latter has a lot of uncertainty. In any event, the UCB1 formula ensures that the node with the most playouts is almost always the node with the highest win percentage, because the selection process favors win percentage more and more as the number of playouts goes up.

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
  tree ← NODE(state)
  while IS-TIME-REMAINING() do
    leaf ← SELECT(tree)
    child ← EXPAND(leaf)
    result ← SIMULATE(child)
    BACK-PROPAGATE(result, child)
  return the move in ACTIONS(state) whose node has highest number of playouts
```

The time to compute a playout is linear, not exponential, in the depth of the game tree, because only one move is taken at each choice point. That gives plenty of time for multiple playouts.

The conventional wisdom has been that Monte Carlo search has an advantage over alpha– beta for games like Go where the branching factor is very high or when it is difficult to define a good evaluation function.
Alpha–beta chooses the path to a node that has the highest achievable evaluation function score, given that the opponent will be trying to minimize the score. Thus, if the evaluation function is inaccurate, alpha–beta will be inaccurate. A miscalculation on a single node can lead alpha–beta to erroneously choose (or avoid) a path to that node. But Monte Carlo search relies on the aggregate of many playouts, and thus is not as vulnerable to a single error. It is possible to combine MCTS and evaluation functions by doing a playout for a certain number of moves, but then truncating the playout and applying an evaluation function.

Monte Carlo search uses **early playout termination**, where a stop is taken at a playout that is taking too many moves, and either evaluate it with a heuristic evaluation function or just declare it a draw.

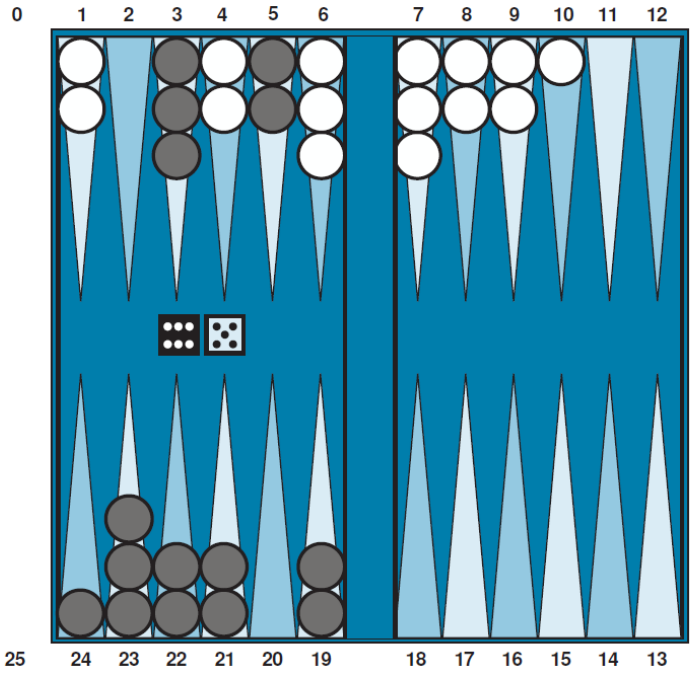
Monte Carlo search can be applied to brand-new games, in which there is no need of experience to draw upon to define an evaluation function. Once the rules of the game is known, Monte Carlo search does not need any additional information. The selection and playout policies can make good use of hand-crafted expert knowledge when it is available, but good policies can be learned using neural networks trained by self-play alone.

Monte Carlo search has a disadvantage when it is likely that a single move can change the course of the game, because the stochastic nature of Monte Carlo search means it might fail to consider that move. In other words, Type B pruning in Monte Carlo search means that a vital line of play might not be explored at all. Monte Carlo search also has a disadvantage when there are game states that are "obviously" a win for one side or the other (according to human knowledge and to an evaluation function), but where it will still take many moves in a playout to verify the winner. It was long held that alpha–beta search was better suited for games like chess with low branching factor and good evaluation functions, but recently Monte Carlo approaches have demonstrated success in chess and other games.

The general idea of simulating moves into the future, observing the outcome, and using the outcome to determine which moves are good ones is one kind of **reinforcement learning**

Stochastic Games

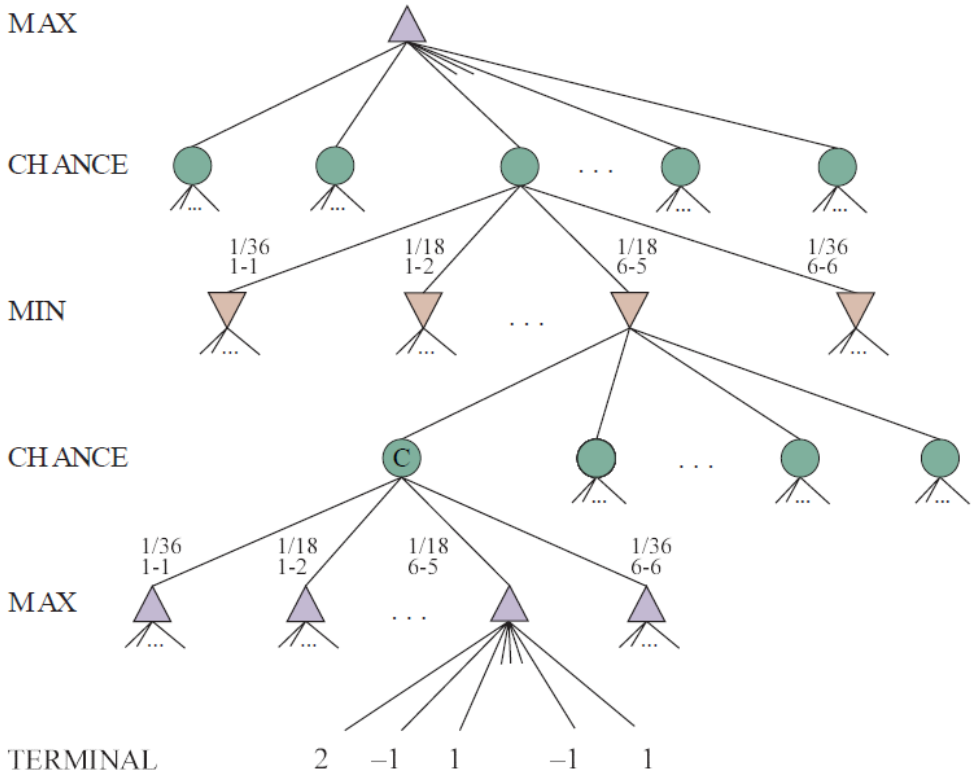
Stochastic games brings a little closer to the unpredictability of real life by including a random element, such as the throwing of dice. Backgammon is a typical stochastic game that combines luck and skill. In the backgammon position of Figure



The goal of the game is to move all one's pieces off the board. Black moves clockwise toward 25, and White moves counter clockwise toward 0. A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over. In the position shown, Black has rolled 6–5 and must choose among four legal moves: (5–11,5–10), (5–11,19–24), (5–10,10–16), and (5–11,11–16), where the notation (5–11,11–16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.

At this point Black knows what moves can be made, but does not know what White is going to roll and thus does not know what White's legal moves will be. That means Black cannot construct a standard game tree as in chess and tic-tac-toe. A game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes. Chance nodes are shown as circles in Figure.

Figure: Schematic game tree for a backgammon position.



The branches leading from each chance node denote the possible dice rolls; each branch is labeled with the roll and its probability. There are 36 ways to roll two dice, each equally likely; but because a 6–5 is the same as a 5–6, there are only 21 distinct rolls. The six doubles (1–1 through 6–6) each have a probability of 1/36, hence P(1-1)=1/36 . The other 15 distinct rolls each have a 1/18 probability.

The next step is to understand how to make correct decisions. Obviously, the move that leads to the best position has to be picked up. However, positions do not have definite minimax values.

Instead, only the **expected value** of a position is calculated: the average over all possible outcomes of the chance nodes.

This leads to the **expectiminimax value** for games with chance nodes, a generalization of the minimax value for deterministic games. Terminal nodes and MAX and MIN nodes work exactly the same way as before . For chance nodes the expected value is computed, which is the sum of the value over all outcomes, weighted by the probability of each chance action:

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s,\text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_a \text{if EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_a \text{if EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if TO-MOVE}(s) = \text{MIN} \\ \sum_r P(r)\text{EXPECTIMINIMAX}\Big(\text{RESULT}\Big(s,r\Big)\Big) & \text{if TO-MOVE}(s) = \text{CHANCE} \end{cases}$$

where r represents a possible dice roll (or other chance event) and RESULT(s,r) is the same state as r , with the additional fact that the result of the dice roll is r .

Evaluation functions for games of chance

As with minimax, the obvious approximation to make with expectiminimax is to cut the search off at some point and apply an evaluation function to each leaf.

Figure shows how an evaluation function that assigns the values [1, 2, 3, 4] to the leaves, move a1 is best; with values [1, 20, 30, 400], move a2 is best. Hence, the program behaves totally differently if some of the evaluation values are changed, even if the preference order remains the same.

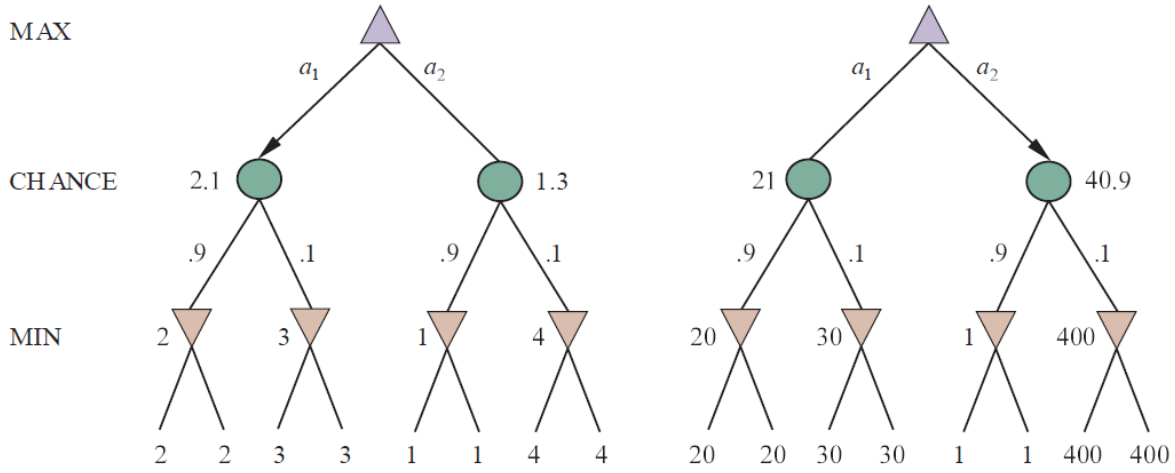


Figure :An order-preserving transformation on leaf values changes the best move.

To avoid this problem, the evaluation function must return values that are a positive linear transformation of the **probability** of winning (or of the expected utility, for games that have outcomes other than win/lose). This relation to probability is an important and general property of situations in which uncertainty is involved

Expectiminimax considering all the possible dice-roll sequences, will take $O(b^m n^m)$ to solve the game, where n is the number of distinct rolls, b is the branching factor

PARTIALLY OBSERVABLE GAMES

In *deterministic* partially observable games, uncertainty about the state of the board arises entirely from lack of access to the choices made by the opponent. This class includes children's games such as Battleships (where each player's ships are placed in locations hidden from the opponent but do not move) and Stratego (where piece locations are known but piece types arehidden). Lets consider the game of **Kriegspiel**, a partially observable variant of chess in which pieces can move but are completely invisible to the opponent.

The rules of Kriegspiel are as follows:

- White and Black each see a board containing only their own pieces.
- A referee, who can see all the pieces, adjudicates the game and periodically makes announcements that are heard by both players.
- On his turn, White proposes to the referee any move that would be legal if there were no black pieces.
- If the move is in fact not legal (because of the black pieces), the referee announces "illegal."
- In this case, White may keep proposing moves until a legal one is found—and learns more about the location of Black's pieces in the process.

For a partially observable game, the notion of a **strategy** is altered; instead of specifying a move to make for each possible *move* the opponent might make, hence a move is needed for every possible *percept sequence* that might be received. For Kriegspiel, a winning strategy, or **guaranteed checkmate**, is one that, for each possible percept sequence, leads to an actual checkmate for every possible board state in the current belief state, regardless of how the opponent moves.

In addition to guaranteed checkmates, Kriegspiel admits an entirely new concept that makes no sense in fully observable games: **probabilistic checkmate**. Such checkmates are still required to work in every board state in the belief state; they are probabilistic with respect to randomization of the winning player's moves

It is quite rare that a guaranteed or probabilistic checkmate can be found within any reasonable depth, except in the endgame. Sometimes a checkmate strategy works for *some* of the board states in the current belief state but not others. Trying such a strategy may succeed, leading to an **accidental checkmate**—accidental in the sense that White could not *know* that it would be checkmate—if Black's pieces happen to be in the right places. (Most checkmates in games

between humans are of this accidental nature.) This idea leads naturally to the question of *how likely* it is that a given strategy will win, which leads in turn to the question of *how likely* it is that each board state in the current belief state is the true board state.

The probabilities associated with the board states in the current belief state can only be calculated given an optimal randomized strategy; in turn, computing that strategy seems to require knowing the probabilities of the various states the board might be in. This problem can be resolved by adopting the game theoretic notion of an **equilibrium** solution. An equilibrium specifies an optimal randomized strategy for each player.

Card games

Card games such as bridge, whist, hearts, and poker feature *stochastic* partial observability, where the missing information is generated by the random dealing of cards. At first sight, it might seem that these card games are just like dice games: the cards are dealt randomly and determine the moves available to each player, but all the "dice" are rolled at the beginning! Even though this analogy turns out to be incorrect, it suggests an algorithm: treat the start of the game as a chance node with every possible deal as an outcome, and then use the EXPECTIMINIMAX formula to pick the best move. Note that in this approach the only chance node is the root node; after that the game becomes fully observable. This approach is sometimes called *averaging over clairvoyance* because it assumes that once the actual deal has occurred, the game becomes fully observable to both players.

Constraint Satisfaction Problems

A problem is solved when each variable has a value that satisfies all the constraints on the variable. Such a problem is called a **constraint satisfaction problem**, or **CSP**

Defining Constraint Satisfaction Problems

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, , one for each variable {D1,…,Dn}.

C is a set of constraints that specify allowable combinations of values.

A domain,D_i , consists of a set of allowable values, {v1,…,vk} , for variable Xi . For example, a Boolean variable would have the domain {true,false}. Different variables can have different domains of different sizes. Each constraint consists of a pair Cj ⟨scope,rel⟩ , where scope is a tuple of variables that participate in the constraint and rel is a **relation** that defines the values that those variables can take on.

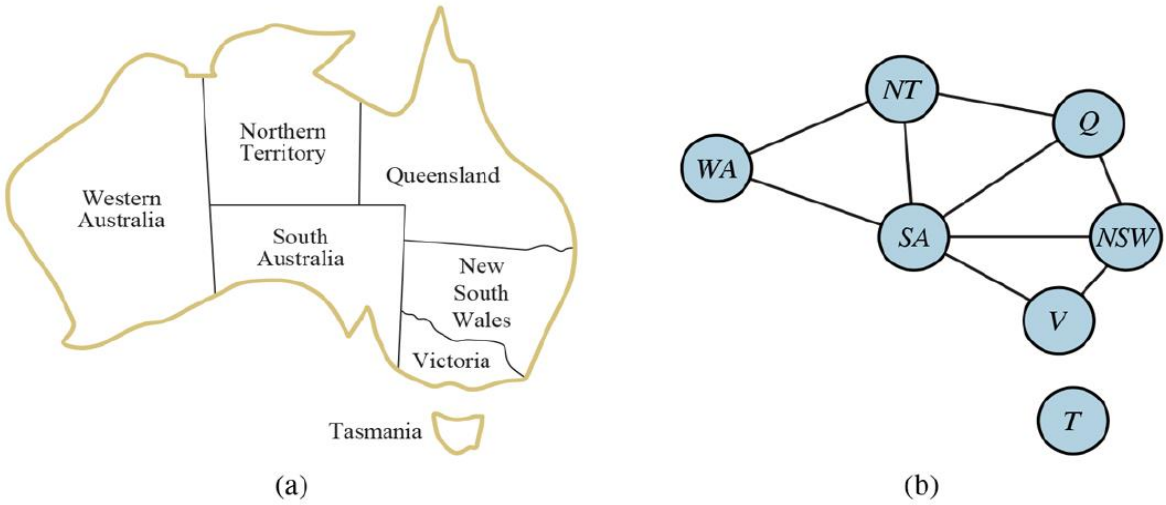
For example, if X1 and X2 both have the domain, {1,2,3}then the constraint saying that X1 must be greater than X2 can be written as ⟨(X1,X2),{(3,1),(3,2),(2,1)}⟩ ⟨(X1,X2),X1 > X2⟩

CSPs deal with **assignments** of values to 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 a value, and a **solution** to a CSP is a consistent, complete assignment. A **partial assignment** is one that leaves some variables unassigned, and a **partial solution** is a partial assignment that is consistent.

Example problem: Map coloring

Consider a map of Australia showing each of its states and territories .The task is to color each region with either red, green, or blue in such a way that no two neighboring regions have the same color. To formulate this as a CSP, the variables are the regions defined as follows:

X = {WA,NT,Q,NSW,V ,SA,T}.



The domain of every variable is the set Di = {red,green,blue} The constraints require 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 }.

There are many possible solutions to this problem, such as

{WA = red,NT = green,Q = red,NSW = green,V = red,SA = blue,T = red }.

It can be helpful to visualize a CSP as a **constraint graph** . The nodes of the graph correspond to variables of the problem, and an edge connects any two variables that participate in a constraint.

Example problem: Job-shop scheduling

Factories have the problem of scheduling a day's worth of jobs, subject to various constraints. In practice, many of these problems are solved with CSP techniques. Consider the problem of scheduling the assembly of a car. The whole job is composed of tasks. Constraints can assert that one task must occur before another—for example, a wheel must be installed before the hubcap is put on. Constraints can also specify that a task takes a certain amount of time to complete.

Consider a small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels (right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect the final assembly. The tasks can be represented with 15 variables:

X = {AxleF, AxleB, WheelRF ,WheelLF ,WheelRB, WheelLB,NutsRF , NutsLF ,NutsRB,NutsLB,CapRF ,CapLF ,CapRB,CapLB,Inspect}.

Next **precedence constraints are represented** between individual tasks. Whenever a task T1 must occur before task T2 , and task takes duration d to complete, an arithmetic constraint of the form can be added

$$\mathbf{T1 + d1 \leq T2.}$$

In this example, the axles have to be in place before the wheels are put on, and it takes 10 minutes to install an axle, hence the constraints can be written as

AxleF + 10 ≤ WheelRF ; AxleF + 10 ≤ WheelLF ;
AxleB + 10 ≤ WheelRB; AxleB + 10 ≤ WheelLB.

Variations on the CSP formalism

The simplest kind of CSP involves variables that have **discrete**, **finite domains**. Mapcoloring problems and scheduling with time limits are both of this kind. The 8-queens problem can also be viewed as a finite-domain CSP, where the variables Q1….Q8 correspond to the queens in columns 1 to 8, and the domain of each variable specifies the possible row numbers for the queen in that column, Di = {1,2,3,4,5,6,7,8}.The constraints say that no two queens can be in the same row or diagonal.

A discrete domain can be **infinite**, such as the set of integers or strings. With infinite domains implicit constraints must be used rather than explicit tuples of values. Special solution algorithms exist for **linear constraints** on integer variables—that is, constraints, such as the one just given, in which each variable appears only in linear form. It can be shown that no algorithm exists for solving general **nonlinear constraints** on integer variables—the problem is undecidable.

Constraint satisfaction problems with **continuous domains** are common in the real world and are widely studied in the field of operations research. The best-known category of continuous domain CSPs is that of **linear programming** problems, where constraints must be linear equalities or inequalities. Linear programming problems can be solved in time polynomial in the number of variables. Problems with different types of constraints and objective functions have also been studied—quadratic programming, second-order conic programming, and so on. These problems constitute an important area of applied mathematics.

Types of Constraints

The simplest type is the **unary constraint**, which restricts the value of a single variable. For example, in the map-coloring problem it could be the case that South Australians won't tolerate the color green; thus the unary constraint .<(SA),SA≠green>

A **binary constraint** relates two variables. For example, SA≠ NSW is a binary constraint. A **binary CSP** is one with only unary and binary constraints; The ternary constraint between(X,Y,Z) for can be defined as ⟨(X,Y ,Z),X < Y < Z or X > Y > Z⟩.

A constraint involving an arbitrary number of variables is called a **global constraint**. One of the most common global constraints is Alldiff , which says that all of the variables involved in the constraint must have different values.

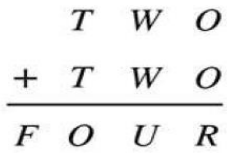
Another example is provided by **cryptarithmetic** puzzles . Each letter in a cryptarithmetic puzzle represents a different digit. For the case in Figure (a) , would be represented as the global constraint Alldiff(F,T,U,W,R,O) . The addition constraints on the four columns of the puzzle can be written as the following n-ary constraints:

$O + O = R + 10 \cdot C1$

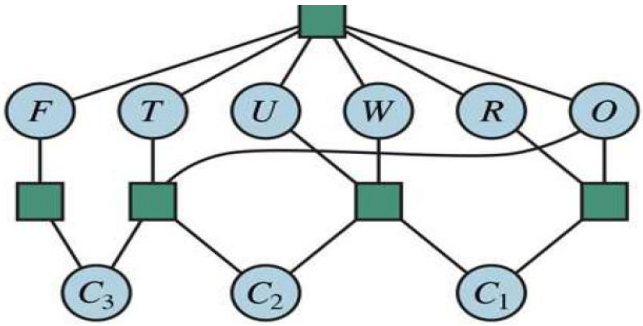
$C1+W +W =U + 10 \cdot C2$

$C2+T +T = O+10 \cdot C3$

$C3 = F$,



(a)



(b)

(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 constraint (square box at the top) as well as the column addition constraints (four square boxes in the middle). The variables C1 ,C2 ,C3 and represent the carry digits for the three columns from right to left

These constraints can be represented in a **constraint hypergraph**, such as the one shown in Figure (b) .A hypergraph consists of ordinary nodes (the circles in the figure) and hypernodes (the squares), which represent n-ary constraints—constraints involving n variables.

Another way to convert an n-ary CSP to a binary one is the **dual graph** transformation: create a new graph in which there will be one variable for each constraint in the original graph, and one binary constraint for each pair of constraints in the original graph that share variables.

A global constraint is preferred for two reasons rather than a set of binary constraints.

- First, it is easier and less error-prone to write the problem description using Alldiff .
- Second, it is possible to design special-purpose inference algorithms for global constraints that are more efficient than operating with primitive constraints.

Many real-world CSPs include **preference constraints** indicating which solutions are preferred.

Preference constraints can often be encoded as costs on individual variable assignments—for example, assigning an afternoon slot for Prof. R costs 2 points against the overall objective function, whereas a morning slot costs 1. With this formulation, CSPs with preferences can be

solved with optimization search methods, either path-based or local. Such problem are called as **constrained optimization problem**, or COP. Linear programs are one class of COPs.

Cryptarithmetic Problems : Examples

1) SEND + MORE = MONEY

```
  5  4   3   2   1
     S   E   N   D
+    M   O   R   E
     c3  c2  c1
----------------------
 M  O   N   E   Y
```

Cryptarithmetic Problems: It is an arithmetic problem represented using letters. It involves the decoding of digits represented by a character.

Constraints:

- Assign a decimal digit to each of the letters in such a way that the answer is correct
- Assign decimal digit to letters
- Cannot assign different digits to same letters
- No two letters have the same digit
- Unique digit assigned to each letter

Rules:

- From Column 5, M=1, since it is only carry-over possible from sum of 2 single digit number in column 4.

```
  5  4   3   2   1
     S   E   N   D
+    1   O   R   E
     c3  c2  c1
----------------------
 1  O   N   E   Y
```

- To produce a carry from column 4 to column 5 ‘S + M’ is at least 9 so ‘S=8or9’ so ‘S+M=9 or10’ & so ‘O=0 or 1’ . But ‘M=1’, so ‘O=0’.

```
  5  4   3   2   1
     9   E   N   D
+    1   0   R   E
     c3  c2  c1
----------------------
 1  0   N   E   Y
```

- If there is carry from column 3 to 4 then ‘E=9’ & so ‘N=0’. But ‘O=0’ so there is no carry & ‘S=9’ & ‘c3=0’.
- If there is no carry from column 2 to 3 then ‘E=N’ which is impossible, therefore there is carry & ‘N=E+1’ & ‘c2=1’.
- If there is carry from column 1 to 2 then ‘N+R=E mod 10’ & ‘N=E+1’ so ‘E+1+R=E mod 10’, so ‘R=9’ but ‘S=9’, so there must be carry from column 1 to 2. **Therefore ‘c1=1’ & ‘R=8’.**
- To produce carry ‘c1=1’ from column 1 to 2, there must be ‘D+E=10+Y’ as Y cannot be 0/1 so D+E is at least 12. As D is at most 7 & E is at least 5(D cannot be 8 or 9 as it is already assigned). N is atmost 7 & ‘N=E+1’ so ‘E=5or6’.
- If E were 6 & D+E atleast 12 then D would be 7, but ‘N=E+1’ & N would also be 7 which is impossible. Therefore **‘E=5’ & ‘N=6’.**
- D+E is atleast 12 hence ‘D=7’ & ‘Y=2’

SOLUTION:

```
   9  5  6  7
+  1  0  8  5
-----------------
1  0  6  5  2
```

Letter	Digit Value
S	9
E	5
N	6
D	7
M	1
O	0
R	8
Y	2

Example 2: CROSS +ROADS =DANGER

```
    CROSS
+   ROADS
  DANGER
```

Solution
D=1,
When two same numbers are added ,the result is an even number therefore R={0,2,4,6,8}

Letter	Digit Value
C	9
R	6
O	2
S	3
A	5
D	1
N	8
G	7
E	4

Example 3: BASE +BALL =GAMES
Solution

Letter	Digit Value
B	7
A	4
S	8
E	3
L	5
G	1
M	9

Constraint Propagation : Inference in CSP

In CSPs there is a choice: 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 each variable is treated 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. There are different types of local consistency, which are as follows.

Node consistency

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, in the variant of the Australia map-coloring problem where South Australians dislike green, the variable SA starts with domain {red , green, blue}, and this can be made node consistent by eliminating green, leaving SA with the reduced domain {red , blue}. Thus a network is node-consistent if every variable in the network is node-consistent. It is always possible to eliminate all the unary constraints in a CSP by running node consistency.

Arc consistency

A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints. More formally, 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 $Y = X^2$ where the domain of both X and Y is the set of digits. This constraint can be explicitly written as {(X, Y), {(0, 0), (1, 1), (2, 4), (3, 9)) })

To make X arc-consistent with respect to Y , reduce X's domain to {0, 1, 2, 3}. and also to make Y arc-consistent with respect to X, then Y 's domain becomes {0, 1, 4, 9} and the whole CSP is arc-consistent.

The most popular algorithm for arc consistency is called AC-3 .

```
function AC-3( csp) returns false if an inconsistency is found and true otherwise
  queue ← a queue of arcs, initially all the arcs in csp

  while queue is not empty do
    (X_i, X_j) ← POP(queue)
    if REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true

function REVISE( csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

To make every variable arc-consistent, the AC-3 algorithm maintains a queue of arcs to consider. Initially, the queue contains all the arcs in the CSP. AC-3 then pops off an arbitrary arc (Xi,Xj)

from the queue and makes Xi arc-consistent with respect to Xj . If this leaves Di unchanged, the algorithm just moves on to the next arc. But if this revises Di (makes the domain smaller), then add all arcs (Xk,Xi) to the queue where Xk is a neighbor of Xi. This has to be done because the change in Di might enable further reductions in the domains of Dk, even if Xk is considered previously. If Di is revised down to nothing, then the whole CSP has no consistent solution, and AC-3 can immediately return failure. Otherwise, it keeps checking, trying to remove values from the domains of variables until no more arcs are in the queue. At that point, a CSP that is equivalent to the original CSP is left—they both have the same solutions—but the arc-consistent CSP will in most cases be faster to search because its variables have smaller domains

Path Consistency

Path consistency tightens the binary constraints by using implicit constraints that are inferred by looking at triples of variables.
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.

Let's consider the path consistency fares in coloring the Australia map with two colors. Consider the set {WA,SA}which is path consistent with respect to NT. By enumerating the consistent assignments to the set. there are only two assignments: {WA = red ,SA = blue}and {WA = blue, SA = red}.In 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, both assignments can be eliminated there is no valid assignments for {WA,SA}. Therefore, there can be no solution to this problem.

K -consistency
Stronger forms of propagation can be defined with the notion of k **-consistency**. A CSP is k-consistent if, for any set of k-1 variables and for any consistent assignment to those variables, a consistent value can always be assigned to any kth variable

A CSP is **strongly k -consistent** if it is k-consistent and is also(k-1) -consistent,(k-2) - consistent, all the way down to 1-consistent.

Global constraints
A global constraint is one involving an arbitrary number of variables (but not necessarily all variables). Global constraints occur frequently in real problems and can be handled by special-purpose algorithms. For example, the Alldiff constraint says that all the variables involved must have distinct values (as in the cryptarithmetic problem and Sudoku puzzles).

One simple form of inconsistency detection for Alldiff constraints works as follows: if m variables are involved in the constraint, and if they n have possible distinct values altogether, and m>n , then the constraint cannot be satisfied.

This leads to the following simple algorithm:

- First, remove any variable in the constraint that has a singleton domain, and delete that variable's value from the domains of the remaining variables.
- Repeat as long as there are singleton variables.
- If at any point an empty domain is produced or there are more variables than domain values left, then an inconsistency has been detected.

Sudoku

- The popular **Sudoku** puzzle has introduced millions of people to constraint satisfaction problems, although they may not realize it.
- A Sudoku board consists of 81 squares, some of which are initially filled with digits from 1 to 9. The puzzle is to fill in all the remaining squares such that no digit appears twice in any row, column, or 3x3 box
- A row, column, or box is called a **unit**.

	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		

(a)

	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

(b)

Fig:(a) A Sudoku puzzle and (b) its solution.

A Sudoku puzzle can be considered a CSP with 81 variables, one for each square. The variable names A1 through A9 is used for the top row (left to right), down to I1 through I9 for the bottom row. The empty squares have the domain{1,2,3,4,5,6,7,8,9} and the pre-filled squares have a domain consisting of a single value. In addition, there are 27 different Alldiff constraints, one for each unit (row, column, and box of 9 squares):

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

Backtracking Search for CSP

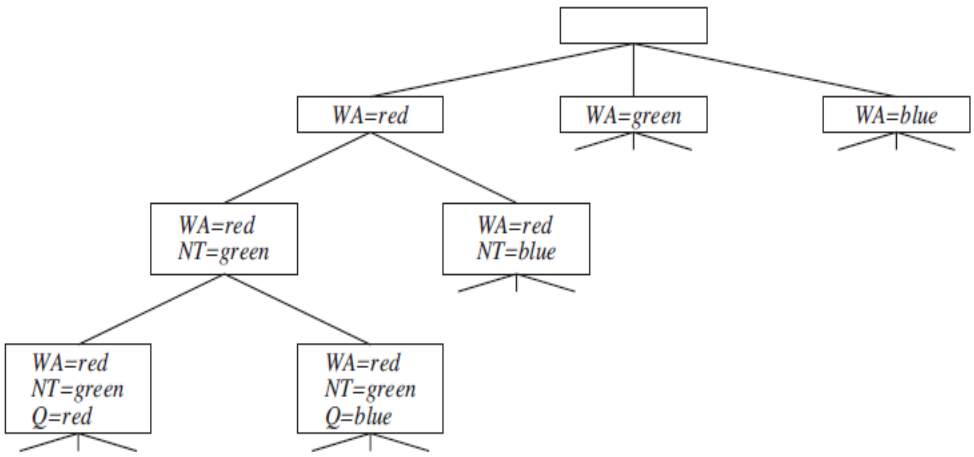
The term **backtracking search** is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in Figure.

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

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

It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value.

Part of the search tree for the Australia problem is shown in Figure ,



where variables are assigned in the order WA,NT,Q, ... Because the representation of CSPs is standardized, there is no need to supply BACKTRACKING-SEARCH with a domain-specific initial state, action function, transition model, or goal test.

Variable and value ordering

The backtracking algorithm contains the line
var ←SELECT-UNASSIGNED-VARIABLE(csp) .
The simplest strategy for SELECT-UNASSIGNED-VARIABLE is to choose the next unassigned variable in order, {X1,X2, . . .}. This static variable ordering seldom results in the most efficient search. For example, after the assignments for WA=red and NT =green in Figure , there is only one possible value for SA, so it makes sense to assign SA=blue next rather than assigning Q. In fact, after SA is assigned, the choices for Q, NSW, and V are all forced. This intuitive idea—choosing the variable with the fewest "legal" values—is called the **minimum remaining-values** (MRV) heuristic. It also has been called the "most constrained variable" or "fail-first" heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree.

Once a variable has been selected, the algorithm must decide on the order in which to examine its values. For this, the **least-constraining-value** heuristic can be effective in some cases. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph. For example, suppose that in Figure generated the partial assignment with WA=red and NT =green and the next choice is for Q. Blue would be a bad choice because it eliminates the last legal value left for Q's neighbor, SA. The least-constraining-value heuristic therefore prefers red to blue.

Interleaving search and inference

One of the simplest forms of inference is called **forward checking**. Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y 's domain any value that is inconsistent with the value chosen for X.
Figure shows the progress of backtracking search on the Australia CSP with forward checking.

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA=red*	(R)	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	(R)	B	(G)	R B	R G B	B	R G B
After *V=blue*	(R)	B	(G)	R	(B)		R G B

There are two important points to notice about this example.

- First, notice that after WA=red and Q=green are assigned, the domains of NT and SA are reduced to a single value;
- A second point to notice is that after V =blue, the domain of SA is empty.
- Hence, forward checking has detected that the partial assignment {WA=red,Q=green, V =blue} is inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately.

Intelligent backtracking: Looking backward

The BACKTRACKING-SEARCH algorithm has a very simple policy when a branch of the search fails: back up to the preceding variable and try a different value for it. This is called **chronological backtracking** because the *most recent* decision point is revisited. In this subsection, we consider better possibilities.

A more intelligent approach to backtracking is to backtrack to a variable that might fix the problem— To do this, keep track of a set of assignments that are in conflict with some value for SA. The set (in this case {Q=red ,NSW =green, V =blue, }), is called the **conflict set** for SA. The **backjumping** method backtracks to the *most recent* assignment in the conflict set; in this case, backjumping would jump over Tasmania and try a new value for V. This method is easily implemented by a modification to BACKTRACK such that it accumulates the conflict set while checking for a legal value to assign. If no legal value is found, the algorithm should return the most recent element of the conflict set along with the failure indicator. A backjumping algorithm that uses conflict sets defined is called **conflict-directed backjumping**

To summarize: let Xj be the current variable, and let conf (Xj) be its conflict set. If every possible value for Xj fails, backjump to the most recent variable Xi in conf (Xj), and set conf (Xi) ← conf (Xi) ∪ conf (Xj) −{Xi} .
When a contradiction is reached, backjumping tells how far to back up, hence time is not wasted in changing variables . When the search arrives at a contradiction, some subset of the conflict set is responsible for the CONSTRAINT problem. **Constraint learning** is the idea of finding a minimum set of variables from the conflict set that causes the problem. This set of variables, along with their corresponding values, is called a **no-good**.

No-goods can be effectively used by forward checking or by backjumping. Constraint learning is one of the most important techniques used by modern CSP solvers to achieve efficiency on complex problems.

LOCAL SEARCH FOR CSPS

Local search algorithms turn out to be effective in solving many CSPs. They use a complete-state formulation: the initial state assigns a value to every variable, and the search changes the value of one variable at a time. For example, in the 8-queens problem the initial state might be a random configuration of 8 queens in 8 columns, and each step moves a single queen to a new position in its column. Typically, the initial guess violates several constraints. The point of local search is to eliminate the violated constraints.

In choosing a new value for a variable, the most obvious heuristic is to select the value MIN-CONFLICTS that results in the minimum number of conflicts with other variables—the **min-conflicts** heuristic. The algorithm is and its application to an 8-queens problem is shown in figure

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
      if current is a solution for csp then return current
      var ← a randomly chosen conflicted variable from csp.VARIABLES
      value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
      set var = value in current
  return failure
```

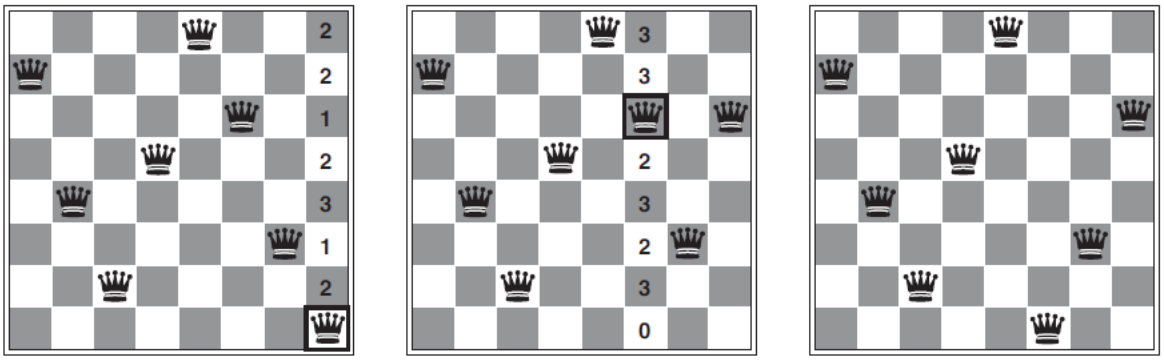


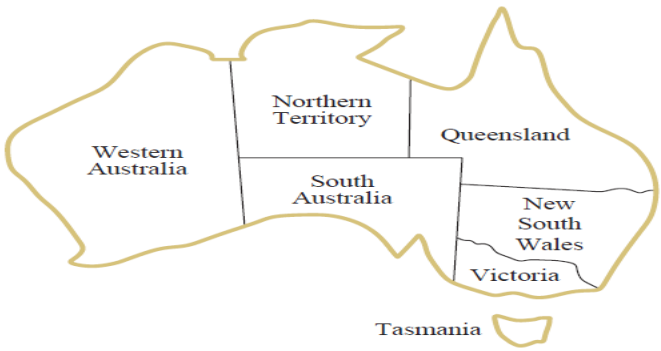
Figure: A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

Another technique, called **constraint weighting**, can help concentrate the search on the important constraints. Each constraint is given a numeric weight, Wi, initially all 1. At each step of the search, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints. The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment.

This has two benefits: it adds topography to plateaux, making sure that it is possible to improve from the current state, and it also, over time, adds weight to the constraints that are proving difficult to solve. Another advantage of local search is that it can be used in an online setting when the problem changes. This is particularly important in scheduling problems. A backtracking search with the new set of constraints usually requires much more time and might find a solution with many changes from the current schedule.

The Structure of Problems

The *structure* of the problem represented by the constraint graph, can be used to find solutions quickly. Consider the Australia Constraint graph, Tasmania is not connected to the mainland. Intuitively, it is obvious that coloring Tasmania and coloring the mainland are **independent subproblems**—any solution for the mainland combined with any solution for Tasmania yields a solution for the whole map.



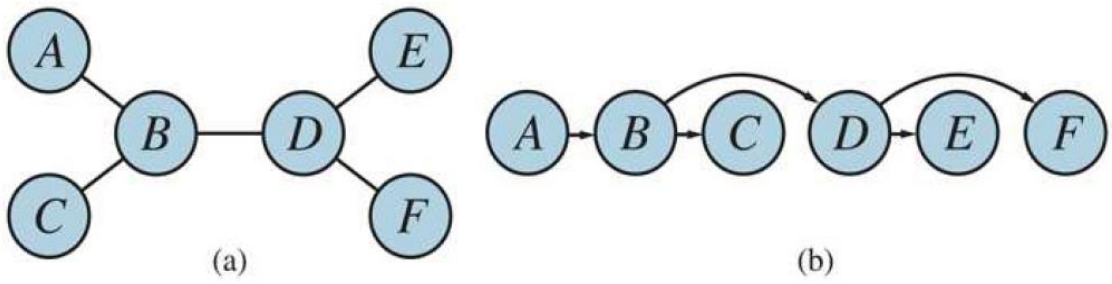
Independence can be ascertained simply by finding **connected components** of the constraint graph. Each component corresponds to a subproblem CSP_i. If assignment S_i is a solution of CSP_i, then U_iS_i is a solution of U_iCSP_i.

Suppose each CSP_i has c variables from the total of n variables, where c is a constant. Then there are n/c subproblems, each of which takes at most d^c work to solve, where d is the size of the domain. Hence, the total work is $O(d^cn/c)$, which is *linear* in n ; without the decomposition, the total work is $O(d^{n)}$, which is exponential in n .

A constraint graph is a **tree** when any two variables are connected by only one path. A CSP is defined to be directional arc-consistent or DAC under an ordering of variables $X_1,X_2…….X_n$ if and only if every X_i is arc-consistent with each X_j for j>i.

To solve a tree-structured CSP, first pick any variable to be the root of the tree, and choose an ordering of the variables such that each variable appears after its parent in the tree. Such an ordering is called a **topological sort.**

Figure shows a sample tree and (b) shows one possible ordering.



(a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with *A* as the root. This is known as a **topological sort** of the variables.

Any tree with n nodes has n-1 edges, hence this graph can be converted to directed arc consistent in O(n) steps, each of which must be compared up to d possible domain values for two variables, for a total time of $O(nd^2)$.

Once a directed arc-consistent graph is constructed, the list of variables can be noted and the remaining values can be chosen. Since each edge from a parent to its child is arc-consistent, for any value chosen for the parent, there will be a valid value left to choose for the child. Hence there is no need of backtracking as the variables are moved linearly.

The complete algorithm is shown in **Figure**

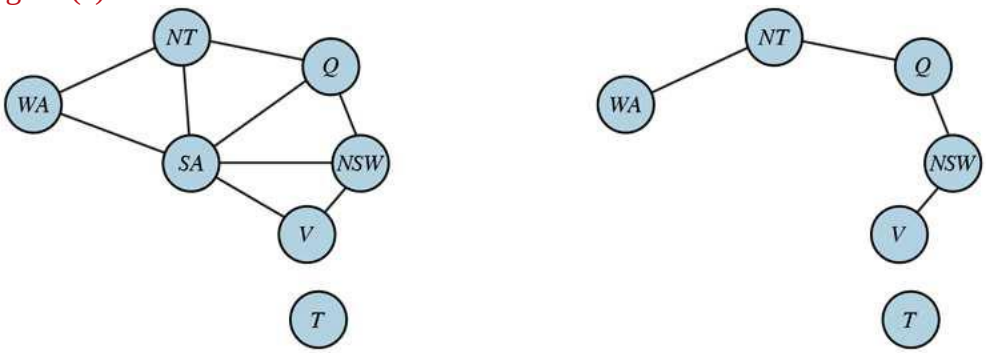
```
function TREE-CSP-SOLVER(csp) returns a solution, or failure
  inputs: csp, a CSP with components X, D, C

  n ← number of variables in X
  assignment ← an empty assignment
  root ← any variable in X
  X ← TOPOLOGICALSORT(X, root)
  for j = n down to 2 do
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent then return failure
  for i = 1 to n do
    assignment[X_i] ← any consistent value from D_i
    if there is no consistent value then return failure
  return assignment
```

There are two ways to reduce a constraint graph into tree This can be done by removing nodes or by collapsing nodes together

Cutset conditioning

The first way to reduce a constraint graph to a tree involves assigning values to some variables so that the remaining variables form a tree. Consider the constraint graph for Australia, shown again in Figure (a) .



Without South Australia, the graph would become a tree, as in (b). Fortunately, South Australia can be deleted (in the graph) by fixing a value for SA and deleting from the domains of the other variables any values that are inconsistent with the value chosen for SA . Now, any solution for the CSP after SA and its constraints are removed will be consistent with the value chosen for SA.

The general algorithm is as follows:

1. Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S .S is called a **cycle cutset**.

2. For each possible assignment to the variables in S that satisfies all constraints on S,

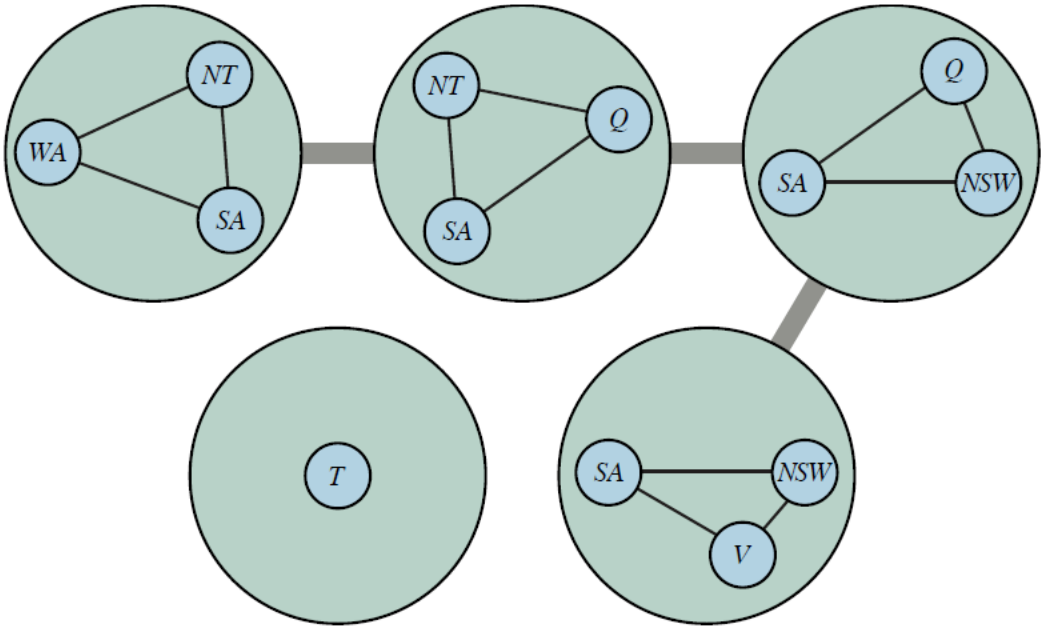
a. remove from the domains of the remaining variables any values that are inconsistent with the assignment for S , and

b. if the remaining CSP has a solution, return it together with the assignment for S

If the cycle cutset has size c, then the total run time is $O(d^c.(n-c)d^2)$: then try each of the d^c combinations of values for the variables in S and for each combination the tree has to be solved for a problem of size n-c . The overall algorithmic approach is called **cutset conditioning** where the aim is to find the smallest cycle cutset which is NP-hard problem and can be used for reasoning about probabilities.

Tree decomposition

The second way to reduce a constraint graph to a tree is based on constructing a **tree decomposition** of the constraint graph: a transformation of the original graph into a tree where each node in the tree consists of a set of variables, as in **Figure** .



A tree decomposition must satisfy these three requirements:

- Every variable in the original problem appears in at least one of the tree nodes.
- If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the tree nodes.
- If a variable appears in two nodes in the tree, it must appear in every node along the path connecting those nodes.

The first two conditions ensure that all the variables and constraints are represented in the tree decomposition. The third condition seems rather technical, but implies that any variable from the original problem must have the same value wherever it appears: the constraints in the tree say that a variable in one node of the tree must have the same value as the corresponding variable in the adjacent node in the tree.

For example, SA appears in all four of the connected nodes in Figure, so each edge in the tree decomposition therefore includes the constraint that the value of in one node must be the same as the value of in the next.

Once a tree-structured graph is got then TREE-CSP-SOLVER is applied to get a solution in $O(nd^2)$ time, where n is the number of tree nodes and d is the size of the largest domain.

A given graph admits many tree decompositions; in choosing a decomposition, the aim is to make the subproblems as small as possible. The **tree width** of a tree decomposition of a graph is one less than the size of the largest node; the tree width of the graph itself is defined to be the minimum width among all its tree decompositions. If a graph has tree width w then the problem can be solved in $O(nd^{w+1})$time given the corresponding tree decomposition. Hence, *CSPs with constraint graphs of bounded tree width are solvable in polynomial time.*

Value symmetry

Consider the map-coloring problem with d colors. For every consistent solution, there is actually a set of d! solutions formed by permuting the color names. For example, Consider the three regions of Australia map WA,NT and SA all must have different colors, but there are 3!=6 ways to assign three colors to three regions. This is called **value symmetry**. Thus the search space can be reduced by a factor of d! by breaking the symmetry in assignments.This is done by **symmetry-breaking constraint** where an arbitrary ordering constraint, that requires the three values to be in alphabetical order. This constraint ensures that only one of the d! solutions is possible:
{NT = blue,SA = green,WA = red}

For map coloring, it was easy to find a constraint that eliminates the symmetry. In general it is NP-hard to eliminate all symmetry, but breaking value symmetry has proved to be important and effective on a wide range of problem