Game Playing

Chapter 8

Playing games like *Chess* well has long been considered a hallmark of intelligence amongst humans. The game of *Chess* was most likely invented in India[1] (Murray, 1985), and has long been considered as a training ground for strategic thinking. *Chess* requires strategic and tactical skills, and in countries like the erstwhile USSR, it was actively promoted to help develop analytic skills (Kotov and Youdovich, 2002).

The computer science community quickly took up games from the earliest times. The first paper on computer chess was published by Claude Shannon (Shannon, 1950). In this paper, he discussed the merits and demerits of complete versus selective search. The first dramatic success in implementing games came, however, in checkers (also known as draughts). In the Dartmouth Conference (McCarthy et al., 1955) where the term AI was coined in 1956, one of the big exhibits was Arthur Samuel's checkers playing program. The striking feature of Samuel's *Checkers* program (Samuel, 1959) was that it learnt from experience, and grew better and better at the game, eventually defeating Samuel himself. In 1968, the British grandmaster David Levy had wagered a bet that no machine could beat him in *Chess*. The duration of the bet, fortunately for Levy, was 10 years. In 1989, Levy was soundly beaten by the computer program *Deep Thought* in an exhibition match. Computer programs improved steadily in performance, and by the mid-nineties were of world champiomship calibre. In 1996, IBM's *Deep Blue* (Campbell et al., 2002; Hsu, 2004) did achieve the feat of beating world champion Garry Kasparov, and in the following year beat him 3.5–2.5 in a six match series. But no one has yet bestowed the quality of intelligence[2] upon computers because of that!

Apart from the fascination that humans naturally have for games, they are very good as platforms for experimentation. Games provide a well defined environment in which states are intrinsically discrete. This means that one does not have to worry about processing input or effecting output in a complex environment, and can focus entirely on the decision making strategy. One can circumvent perception and action in the real world; problems that one would have to address if one were building a robot to play golf or tennis. Moreover, absolutely nothing is lost in abstraction. Furthermore, in games, the rules are well defined and success or failure can be measured easily. Good programs will beat inferior programs or humans in the game. Moreover, as we will see in this chapter, in spite of

the simplicity of the domain, they provide us with problems that are hard to solve.

Game playing is also interesting because it allows us to reason about multi-agent activity. The problem-solving activity studied in the preceding chapters is characterized by the fact that only a single agent is involved. Single agent situations do occur in the real world. For example, organizations can be seen as agents pursuing their goals in isolation. They could be in industrial organizations planning their design and manufacturing of products, or they could be government organizations planning infrastructure, or sending a man to the moon, or a rover to Mars. Or an individual may be planning a meal, or building a house.

Many real world situations, however, have multiple agents involved. In such situations, the problem-solving agents have to consider the actions of the other agents too, because they affect the world the agent is operating in. The agents may be collaborating with each other, they may be competing with each other, or they may be antagonistic to each other.

Interaction between agents has most commonly been studied by abstracting them as games. Games are formalisms that are used in various fields of study, ranging from economics to war. The common feature is that they attempt to devise models of rationality in the face of other agents being active. John von Neumann (Neumann and Morgenstern, 1944), the prolific computer scientist, is also credited with pioneering work in formalizing games and is often referred to as the father of Game Theory. The following example, known as the Prisoner's Dilemma, illustrates the kind of problems posed by von Neumann (Poundstone, 1993).

Prisoner's Dilemma

Imagine that the police have in their custody two suspects of a bank robbery, but no real evidence. Their only chance lies in a confession from one or both of them. They interrogate them in separate chambers, and offer each a lighter sentence if they confess, and betray the partner. Imagine, for a moment, that you are one of the two suspects. In the *normal* form of game (McCain, 2004) representation, a payoff matrix can be constructed, as follows:

Payoff: Yours / his	You confess	You deny
He confesses	–100 / –100	–200 / –10
He denies	–10 / –200	–50 / –50

FIGURE 8.1 A payoff matrix for Prisoner's Dilemma.

If both confess, they get –100 each. If only one confesses, he escapes with –10 but the other gets –200. If both deny, each gets –50 for possession of an illegal firearm. As one can see, the combined optimal

payoff is achieved by both denying and getting a total penalty of 100. However, there is the temptation to betray the other and escape with –10. Seen from an individual perspective, the following *extensive* form[3] of the game is represented as shown in Figure 8.2. Even though the choices are made concurrently, one views this as a sequential phenomenon in which the two players act one after the other. Since the outcome depends upon the other player's action, the rational choice is to choose a move in which the worst penalty is as small as possible. Thus, the safer choice for you is to confess because it could at worst lead to the penalty of –100. If you deny the crime and the other player confesses, you will be in for a –200 penalty. Obviously, for habitual[4] bank robbers, the more profitable strategy is to stand by each other and deny any involvement.

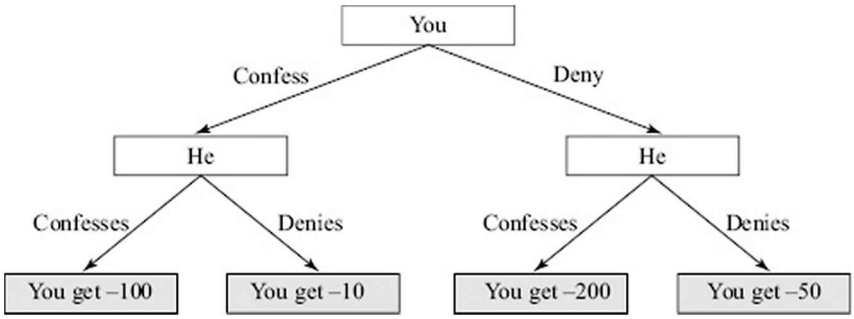

FIGURE 8.2 In the extensive form of Prisoner's Dilemma, the payoff is seen to be a function of the other person's choice after you have made your decision.

This can be perhaps seen more clearly by looking at an equivalent extensive form in which the other player plays first, as shown in Figure 8.3. If the opponent[5] confesses, you are better off confessing yourself to get a penalty of –100, instead of –200. And if the opponent denies, you can get away with –10 by confessing.

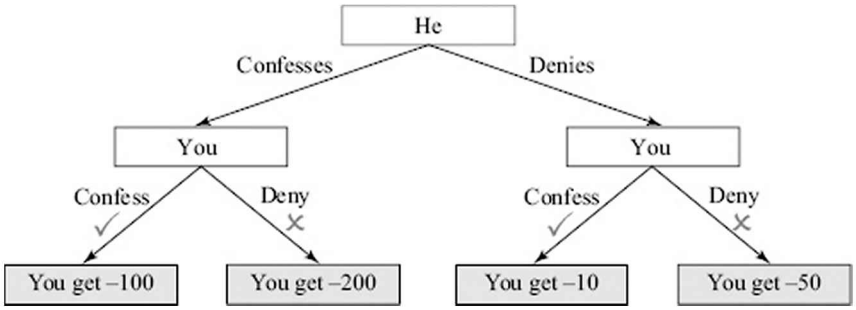

FIGURE 8.3 In an equivalent extensive representation, one can see that whatever the other player does; in each case, confessing is a better option for you.

Observe that in the *Prisoner's Dilemma*, the two players choose without really knowing what the other player has chosen. This is in

contrast to many situations where a player is aware of the opponent's choice while making a move. To distinguish between these two cases, one often draws the extensive form of the game (McCain, 2004) as shown in Figure 8.4. The idea is that the second player does not know which part of the tree he is operating in. This is depicted by merging the nodes at the second level into an *information set*.

One observation that one makes about the game is that simply acting rationally from one player's perspective, does not yield the best possible result. In this example, if both players were to deny then they would have both got a penalty of –50, known as the Pareto optimal (after Vilfredo Pareto), but by *rationally* choosing to confess they both end up incurring a penalty of –100 each, reaching the Nash equilibrium (after John Nash). That is because rationality here is taking a pessimistic or conservative view trying to cater for the worst that can happen. In practice, human beings are often optimistic, they dream, they gamble and take chances, and they cooperate with each other. And on the average, they are better off as a result.

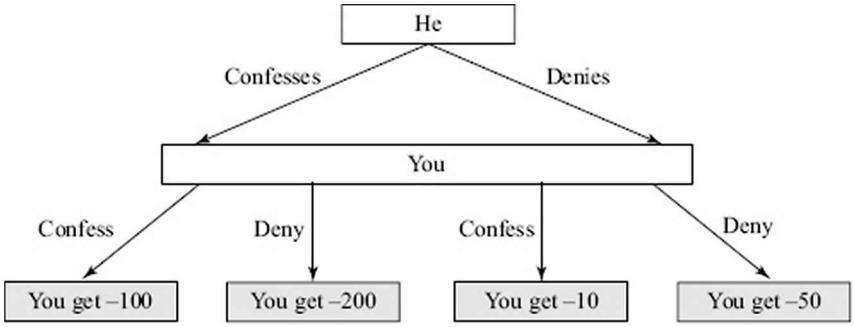

FIGURE 8.4 An information set is a node in an extensive form that hides the choice of the first player. The second player has to choose without knowledge of the move made by the other player.

Games[6] thus are abstractions of interactions between agents. In the *Prisoner's Dilemma* example we saw above, the two players had to play simultaneously. Also, the strategy is concerned with choosing that one move. We will focus instead on board games like *Chess*, in which the two players play a *sequence of alternating moves* and the outcome is determined only when the game ends. Such games may be called *board games*. These games could in principle be represented and analysed by constructing the payoff table, but in practice, such a process would be cumbersome and computationally demanding. We will focus instead on more efficient strategies to play board games.

8.1 Board Games

The games we focus on primarily in this chapter are classified as

- two person,
- zero sum,
- complete information,
- alternate move, and
- deterministic games.

Two person games have exactly two players. In zero sum games, the total payoff is zero. One player's gain is the other player's loss. One wins, the other loses. In complete information games, both the players have access to all the information. That is, both can see the board, and thus know the options the other player has. In alternate move games, the players take turns to make their moves. In deterministic games, there is no element of chance in the moves that one can make.

All the above properties can be relaxed to produce more complex multi-agent environments. Adding a dice to a board game introduces an element of chance. The player cannot deterministically make a move. The essential feature of most card games is that they are multiple-player incomplete information games. Cards of other players are hidden, but the pack is known and finite. So a player does not know the options available to other players, but the possible options are bounded by knowledge of the cards that have not yet been played. In games like contract bridge, players draw inferences from the known information to glean as much additional information as possible. The game has four players, with partnerships of two each, and the strategy also involves communication of information between partners. Since this information is also available to opponents, the tactics often involve misinformation and deception too. Army generals fighting wars operate similarly with genuinely incomplete information. They also attempt to glean information about the resources and options of the enemy, and likewise the flow of information yields opportunities of misinformation and deception. War and spies provide a multitude of engaging stories concerning information exchange and deception[7]. Since both sides usually suffer casualties, wars can be seen as negative sum games, though sometimes one side may have some positive payoff. A price war, likewise, inflicts losses on the competing sellers, and can be modelled as a negative sum game, though if the buyers are included in the game too then it becomes a multi-player zero sum game. Cooperation is an example of a positive sum game, whether it is between students studying together for an examination, or when large corporations collude to jack up prices. Observe that some of the examples above do not have alternating moves, and neither is the outcome of their actions deterministic.

In the domain of recreation, *Checkers* (also known as *Draughts), Chess, Othello* and *Go* are examples of board games that have received the attention of programmers. One of the earliest to make a mark was Arthur Samuel's Checkers playing program (Samuel, 1959). It was a program that improved its performance with experience, and became news when it was able to beat its creator! The learning it did was essentially parametric *reinforcement learning*, tuning weights of its

evaluation function based on the outcome of each game. But it contributed to the wild notion of computers taking over the world, *à la* Victor Frankenstein's robotic creature (Shelley, 2001), in the novel written by Mary Shelley in 1818.

Amongst the incomplete information games, the most successful has been the implementation of *Scrabble.* In fact, the program *Maven*, now commercially available, can easily outplay human players (Sheppard, 2002). This is not surprising given that the machine can have access to a large vocabulary. In addition, it can speedily run through different combinations, and pick the one yielding maximum score. One could use a heuristic function that evaluates a board position by the points it yields, along with some measure of the cost of openings it creates for the opponent. A game that is met with less success is *Contract Bridge.* Though there have been several attempts at implementing the game (see: Throop, Frank, Khemani, Smith, Ginsberg, Sterling and Nygate), a truly world class player is yet to emerge.

8.1.1 Game Trees

We now look at some well known, search based algorithms to play complete information games. Our focus will be on alternate move games. In some games, players are allowed to move again in certain situations. For example, multiple captures in checkers, or potting a ball in snooker. We will assume that the move sequence can be modelled as a single (macro) move. We also assume the games to be two-player games. But the ideas presented can easily be extended to multi-player games. Finally, we assume that the game is a zero sum game. Our two players are traditionally named *MAX* and *MIN*, indicating that their goals are opposite of each other. The algorithms we study could be adapted to players whose goals are not diametrically opposite each other.

A game is represented by a *game tree.* A game tree is a layered tree in which at each alternating level, one or the other player makes the choices. The layers are called *MAX* layers and *MIN* layers as shown in Figure 8.5. Traditionally, we draw *MAX* nodes as square boxes *and MIN* nodes as circles in the tree[8], and *MAX* is at the root. A game starts at the root with *MAX* playing first and ends at a leaf node.

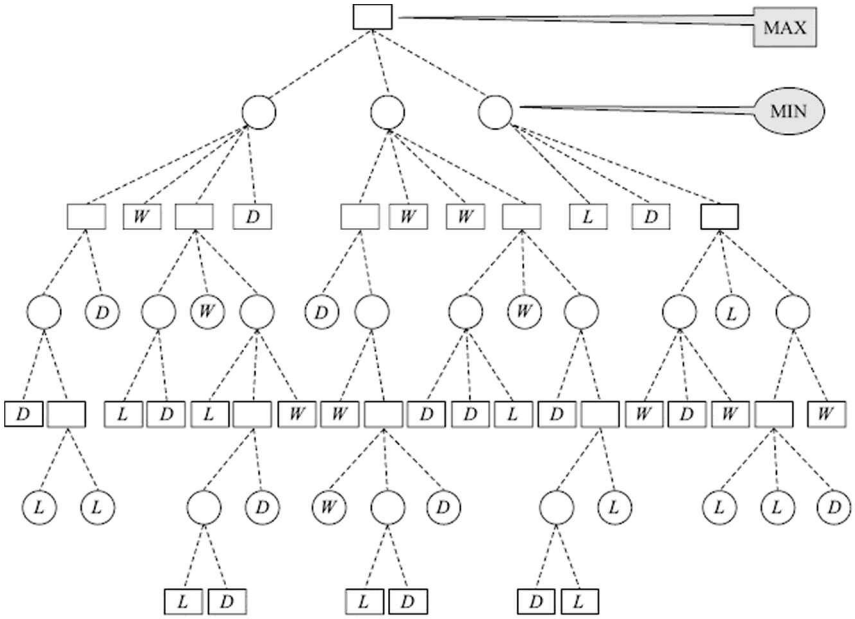

FIGURE 8.5 A small game tree. Leaf nodes are labelled with "W" for win, "D" for draw, and "L" for loss for Max. 1, 0, and –1 would be equivalent labels.

The leaves of the game tree are labelled with the outcome of the game and the game ends there. The task of each player is to choose the move when its turn comes. In the game tree, *MAX* chooses at *MAX* levels and *MIN* chooses at *MIN* levels. Thus, a game is a path from the root to some leaf node, chosen at alternating levels by the two players. For our zero sum game, the outcomes are defined by a set {*win, draw, loss*} and the values are as seen from the perspective of *MAX*, the player at the root. Thus, the value "*win*" means that *MAX* wins the game, and "*loss*" means that *MAX* loses or equivalently *MIN* wins. The leaves can also be labelled equivalently with numbers {1, 0, –1}. That is, it is a function that returns the outcome at a leaf node.

value(*leaf*)	= 1	if *MAX* wins
	= 0	if the game is a draw
	= −1	if *MIN* wins

One can now see the rationale of naming them *MAX* (the one who prefers the maximum valued outcome) and *MIN* (the one who prefers the minimum valued outcome).

Given a game tree, it is possible to analyse the game and determine the outcome when both players play perfectly. We can do this by backing up values from the leaf nodes up to the root. The backup rule is as follows.

Minimax rule

- If the node is a *MAX* node, back up the maximum of the values of its children.

$$\text{value(node)} = \max \{\text{value}(c) \mid c \text{ is a child of node}\}$$

- If the node is a *MIN* node, back up the minimum of the values of its children.

$$\text{value(node)} = \min \{\text{value}(c) \mid c \text{ is a child of node}\}$$

FIGURE 8.6 The *minimax* rule backs up values from the children of a node. For a *MAX* node, it backs up the maximum of the values of the children, and for a *MIN* node, the minimum.

The rationale of the rules is that given a set of choices with known outcomes, *MAX* will choose a move that yields the value = 1 (or *win*) if available, else 0 if available, and will have to choose a –1 (or *loss*), only if all its children are labelled with –1. The backup rule for *MIN* is exactly the opposite, given that *MIN* is also trying to win the game. For *MIN* to win the game, the node must be labelled with –1 or *loss*. The *minimax value* of the game is the backed-up value of the root from all the leaves, and represents the outcome when both the players play perfectly. Figure 8.7 shows the above game tree with backed-up values. Observe that *MAX* wins *this* game when both players play their best.

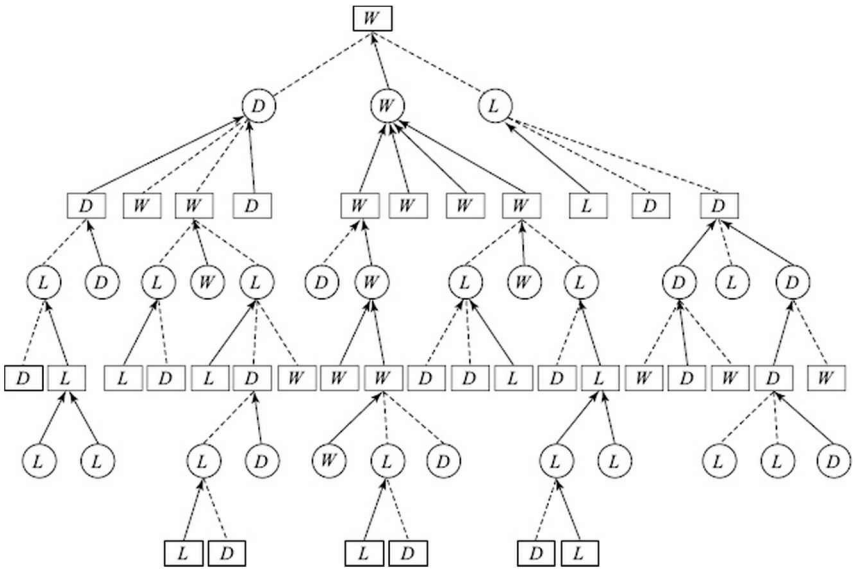

FIGURE 8.7 The game tree with backed-up values. Arrows show the values backed up. They identify the best move for each player. Where more than one move is best, all are marked. *MAX* wins the game because the backed-up value is *W*.

A game playing program is required to produce the moves for a player, traditionally *MAX*. Since the *minimax* value determines the best that *MAX* can do against a perfect opponent, this involves computing the *minimax* value, and the choice of moves that leads to it. The choices of a

player can be represented as a *strategy*. A strategy is a subtree of the game tree that freezes the choices for the player. A strategy can be constructed by the following procedure.

```
ConstructStrategy(MAX)
  1 Traverse the tree starting at the root
  2 if level is MAX
  3   then Choose one branch below it
  4 if level is MIN
  5   then Choose all branches below it
  6 return the subtree constructed
```

FIGURE 8.8 A Strategy for *MAX*.

The idea is that the strategy freezes the choices for the player, in our case *MAX*. That is, *MAX* has already decided the strategy, or the set of choices. The following figure shows two strategies for *MAX* in the given game tree.

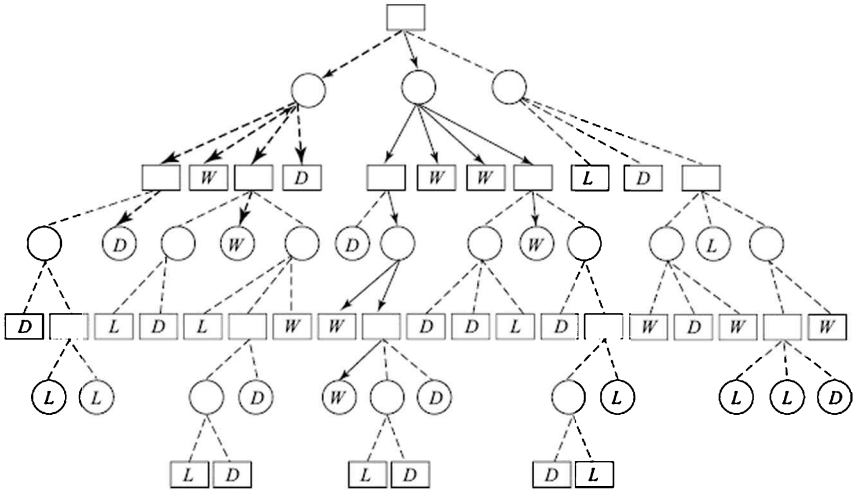

FIGURE 8.9 Strategies in a game tree are subtrees that represent choices of one player. The figure shows two strategies for *MAX*.

Once a player decides her strategy, the outcome of the game depends upon the opponent. Assuming that the opponent plays rationally, the value of the strategy for *MAX* will be the minimum value of a leaf node in the strategy, because that is where *MIN* will drive the game. Given a game tree, the *optimal strategy* for *MAX* is the strategy with the highest value. If this happens to be 1 (or win) then *MAX* has a *winning strategy*. In any case, the objective of both the players is to find their optimal strategies. Once both have chosen their strategies, the game played will be the path that is the intersection of the two strategies (the two

subtrees), as shown in Figure 8.10. The thick grey lines are a strategy for *MAX*, the thick dotted lines the strategy of *MIN*, and the thick black lines the resulting game path. Observe that the strategy for *MAX* shown in the figure is not an optimal one, leading to a draw.

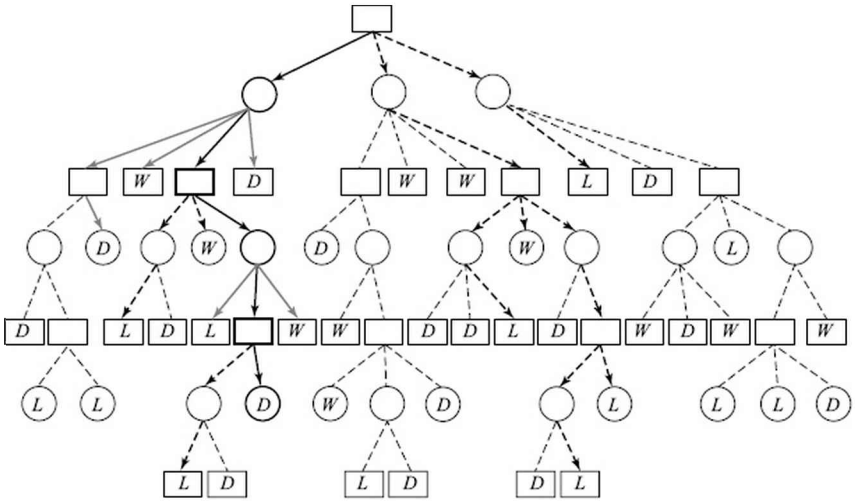

FIGURE 8.10 The subtree in grey arrows represents a strategy for *MAX*, and the one in dotted arrows, a strategy for *MIN*. The game played as a result of these two strategies is the path shown with black arrows, which is the intersection of the two subtrees.

Selection of the optimal strategy requires solving the game tree. Observe that the solution is a subtree, being a strategy, like the solution of an *And-Or* problem. In fact, finding the strategy is like solving the game tree as an *And-Or* problem. Each player has to choose her own moves, the *Or* choices, and cater to all possible responses by the opponent, the *And* nodes. With the cost of solving, the *And* node being the maximum of the costs of solving its children, instead of the sum, the question one might ask is can one use the *AO** algorithm to play games? One rarely hears of algorithms like *AO** being used to play games. The answer is that these algorithms could be adapted to play the games if it were possible to reach all the leaf nodes in the game tree. For most interesting games, the trees turn out to be too large to be traversed completely. Games with small trees can be completely solved. For example, the well-known game of *Tic-Tac-Toe* (also known as *Noughts and Crosses*, see Figure 8.11) is known to end in a draw when both players play correctly. But such statements cannot be made for games like *Checkers*, *Chess* and *Go*, because their game trees are too big. That is why these games are still fascinating for us to play. In *Chess*, for example, many people believe that *White*, the first to play, has an advantage. But this is only speculation. Let us see why.

The starting position in a *Chess* game[9] has twenty possible moves for each player. As the game proceeds, the board opens up and the number

of choices increases further. Still further in the game as the number of pieces on the board reduce, the number of choices gradually comes down in the end game. It has been estimated that the average branching factor in *Chess* is thirty five, and that a typical game lasts about fifty moves. This means that the *Chess* tree has about 35^{50} leaves. This is roughly equal to 10^{120}leaves, a number that is difficult even to comprehend. One followed by one hundred twenty zeroes. Let us make a rough estimate of how long it will take to inspect just the leaves, forgetting about the internal tree. Let us assume that we have a fast machine on which ten billion leaves can be examined every second. Thus, it will take 10^{110} seconds to examine 10^{120} nodes. Like in Chapter 2, we can conservatively assume a hundred thousand seconds in a day, and a thousand days in a year. We will then need 10^{102} years, or 10^{100} (also known as a Googol) centuries. Compare this number with the total number of fundamental particles in the entire universe, which is estimated to be about 10^{75}, and one can see that the task of inspecting 10^{120} nodes is clearly in the realm of the impossible, with due respect to all who think otherwise[10]. Even if every one of these fundamental particles was a machine working in tandem with the others, it would still take 10^{27} years, which is much longer than the estimated age of the universe.

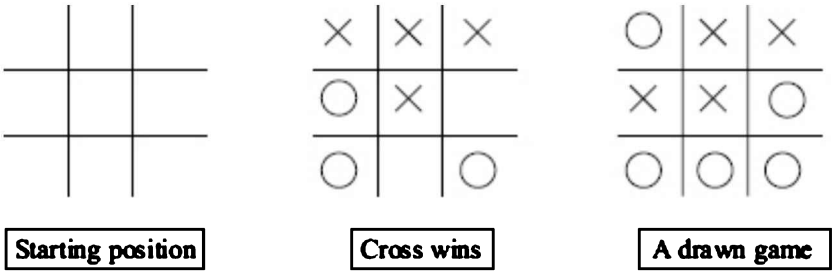

FIGURE 8.11 The game *Noughts and Crosses* is played on a 3 × 3 board. One player places a cross and the other a nought, alternately. The objective is to place three in a line first; row, column or diagonal. The first figure shows the empty board, the second a game one by *Cross*, and the third figure shows a drawn game.

The back of the envelope analysis done above, gives us a couple of insights. First, that even toy problems can be computationally hard. This means that real world problems will be harder to solve, unless they are posed very tightly. That is, only the minimal key features of the problem are identified and abstracted. That is why problem formulation is important to successful problem solving. The second insight is that we humans still do tackle many complex problems. And we do it with the aid of knowledge. Knowledge thus, has to be a key component for problem solving, if we are to build intelligent agents.

8.1.2 The Evaluation Function

Since we cannot inspect the complete game tree and compute the *minimax* value, we have to resort to other means of selecting the move to make. If we could have computed the *minimax* value, we would have selected the move that would have been *known* to be the best move. Now instead, we have to look for methods with which we will select moves that *appear* to be the best. This is done by using a function to evaluate the goodness of a state, like we did with heuristic functions in Chapter 3. In game playing terminology, we call it the *evaluation function*, and it tells us how good a given position (state) is from the perspective of *MAX*.

The outcome of a game is a value from the set {–1, 0, 1}. The evaluation function, however, is usually applied to an intermediate node, and we are not in a position to choose a value from the set, since the game has not ended, and we cannot evaluate the full tree. Instead, we define the range of the evaluation function as the real interval [–1, 1]. The extreme values –1 and 1 still represent wins for the two players. But other values *estimate* how close to winning each side is. For example, the value 0.5 says that *MAX* is better off than *MIN*, and 0.75 says that *MAX* is even more better off. A value of –0.9 says that *MIN* is doing very well. Note that the value zero says that both players are equally placed. It does not say that the game has ended in a draw. In practice, we extend the range to something like [–10000, 10000] which is more conducive to devise evaluation functions. We will refer to it as [–*Large*, +*Large*]. Where do we get the values from? This is where the knowledge of an expert comes in. Generally, the evaluation function is computed as a sum of values of different features, and we add or subtract values for each good feature or bad feature. This kind of knowledge may be acquired from an expert, or one could devise experiments to learn from experience.

Typically, the evaluation function is split into different components. In *Chess*, for example, one may count the *material* value and add to that the *positional* value.

Chess players are used to assigning values to the pieces in the range of 1 to 10. Typically, pawns are valued at one point, knights and bishops about three points each, rooks about five points, and the queen about nine points. The *fighting value* of the king in the end game is equivalent to about four points. The material value of the king would be *large*, if it were counted, because its capture ends the game in a loss. *Chess* programmers however choose numbers in a larger range, thus enabling them to add positional values also more precisely. Thus, in the evaluation function of a program, a queen may be worth 900 points, a rook 500, a bishop 325, a knight 300 and a pawn 100. The material balance on a given board position is arrived at by adding the value of all the pieces on our side, and subtracting the value of pieces on the opponent's side. One may observe that the value of the evaluation function in the starting position is zero.

The positional part of the evaluation function looks at many aspects. These are concerned with mobility, board control, development, pawn

formations, piece combinations, king protection, etc. For example, two rooks in the same column have an added value; a pair of bishops is better than a bishop and a knight; knights are valuable in certain kinds of end positions. Traditionally, development and centre control have been given great importance; one's pieces must become as mobile as possible and either occupy the centre or control it. Pawn formations are also subject to evaluation; chained pawns that support each other are better than isolated pawns; pawns in the same column or opposing pawns head to head are not good; and as most players know, pawns heading for promotion have added value. The king needs protection in the opening and middle games, and structures like those obtained by castling are valued high; while in the end game, the king may be an offensive piece adding to the fighting strength. Pins, forks and discovered attacks also need to be considered while computing positional value. The evaluation function of the program *Deep Blue* has about 8000 components (Campbell et al., 2002).

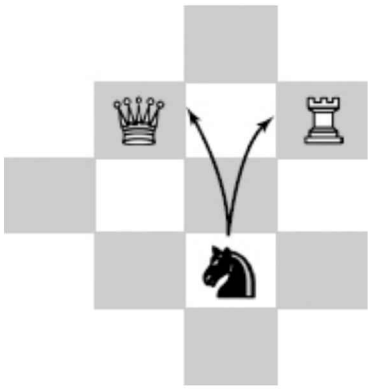

FIGURE 8.12 The fork is a pattern in which a Knight attacks two pieces simultaneously. The opponent can only move one piece away. In the example, the Red Knight attacks the White Rook as well the White Queen.

The value determined by a component of the evaluation function could have also been determined by searching further. For example, if one ascribes a value to the existence of a fork *pattern* on a chessboard (see Figure 8.12), it says that a fork tilts the value in favour of the player by a certain amount. If this was not part of the evaluation function, the advantage would have become evident after a few plies search. In the example in the figure, the Red Knight attacks the White Queen and the White Rook simultaneously. White can move only one of the pieces away, and in the next move the Red Knight could capture the other piece, gaining material advantage, even if the knight is captured in turn. This estimate of material gain could be encoded as the value of the pattern. The key thing is that this value is available from the given board position directly, without having to look ahead. This also illustrates how knowledge can be used to trade off search.

One obvious way to play the game now would be to evaluate all the positions that result from the moves one can make, and choose the best one. This would be called *one-ply* look-ahead and is depicted in Figure 8.13.

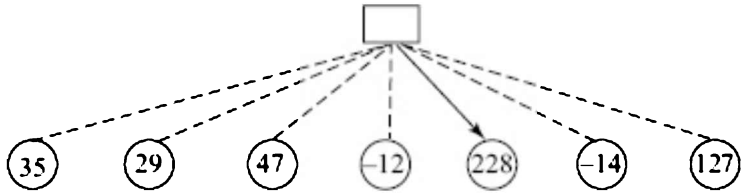

FIGURE 8.13 A one-ply look-ahead picks the best looking successor.

This would be fine if the evaluation function was very good. While it has emerged that grandmasters do store tens of thousands of *Chess schemata* and evaluate them directly (Sowa, 1983), it is difficult to devise an evaluation function that is good enough to play with one-ply look-ahead. In practice, *Chess* programmers rely on a combination of evaluation and look-ahead. While we cannot write programs to look ahead till the end of the game, we can still do so to look ahead a smaller distance. Figure 8.14 below gives you a feel of the exploding search space with a game tree involving four choices per board position.

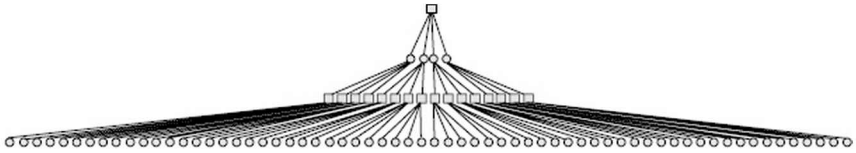

FIGURE 8.14 A game tree with a branching factor 4. A three-ply search needs to inspect 64 nodes. The next level will have 256, the one after that 1024, and the next one 4096; trees that are too large to draw on these pages. Most games have an even higher branching factor.

Look-ahead takes care of the combinatorial aspect of the game, like piece exchanges in *Chess*, which cannot be captured easily in a heuristic (evaluation) function, while the evaluation function provides a mechanism for evaluating the material and positional properties of nodes at the end of an incomplete look-ahead. The amount of look-ahead would basically depend upon the resources available to the program. The faster the machine, the more is the look-ahead possible in the same time. The more the program looks ahead, the better it is likely to play. Experts hypothesize that even with a simple evaluation function, a program that looks ahead fourteen to sixteen plies will play at a grandmaster level (Newborn, 2003). The extent of look-ahead is determined by the computing resources available. Most commercial programs do an eight- or nine-ply search. More sophisticated machines try and harness parallel computing with specialized hardware (Berliner, 1987; Campbell et al., 2002). One does not have to do a fixed look-ahead rigidly. We can write

programs to explore critical regions deeper. While playing competitive *Chess*, one is constrained by the clock. One can then do a flexible amount of look-ahead, depending on the amount of time available. This is typically done using an iterative deepening approach, like the one we studied in Chapter 2. The search component keeps searching deeper and deeper, and when the controlling program needs to play a move, it simply picks the best known move available.

8.2 Game Playing Algorithms

The basic procedure for game playing is then the following. Explore the tree up to a finite ply depth. Compute the evaluation function of the nodes on the frontier. Use the *minimax* backup rule described earlier to determine the value of the partial game tree, and the best move. Make the chosen move, wait for the opponent's move, and then again search for your best move.

```
GamePlay(MAX)
   1 while game not over
   2   do Call k-ply search
   3      Make move
   4      Get MIN's move
```

FIGURE 8.15 The top level game playing algorithm makes a call to search algorithm that backs up the evaluation function values from the horizon at *k*-ply depth. It makes the move that yields the *minimax* value, and after getting the opponents move, does another *k-ply* search.

If we could have searched the entire tree, the search would have to be done only once. But constrained to search only a part of a tree, we do a series of searches, one every time the program has to make a move. Every subsequent search starts two plies deeper than the previous one, and explores two more plies in the game tree. But, as shown in Figure 8.16, since it searches only below the chosen moves, it only looks at a fixed number of nodes at each level in the game tree. The complexity of algorithm can be depicted by the area of the search tree, which is proportional to the number of nodes in the tree. The figure below gives one an intuitive idea that the series of fixed ply searches explore only a small part of the entire game tree. Assuming that each search looks at P nodes, the game playing program will look at a total of $PN/2$ nodes during the entire time, where N is the number of moves made by both sides.

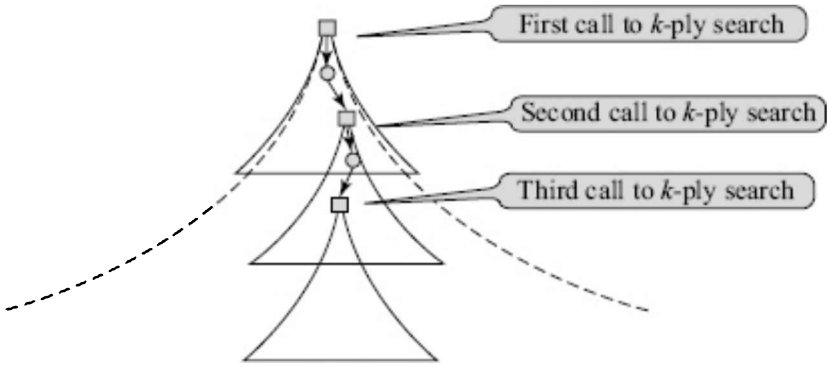

FIGURE 8.16 A game playing program does a *k*-ply look-ahead search for each move. It makes the best move, waits for the opponent to move, and does another *k*-ply search to decide upon the next move.

We now look at the basic algorithm for doing the fixed ply search. The algorithm uses an evaluation function *e*(*J*) when considering the nodes at the frontier.

8.2.1 Algorithm *Minimax*

The algorithm *Minimax* searches the game tree till depth *k* in a depth-first manner from left to right. It applies the *minimax* rule (Figure 8.6) to determine the value of the root node. The following algorithm (Figure 8.17) is a recursive version adapted from (Pearl, 1984). The algorithm uses a test *Terminal*(*node*) to determine whether it is looking at a frontier node, and therefore should apply the evaluation function *e*(*J*) instead of making a recursive call. A node is a terminal of a leaf node of the game, and will evaluate to one of {–*Large*, 0, +*Large* } or it is a node on the horizon, and in that case the evaluation function *e*(*J*) will be applied. Not shown in the algorithm is the implementation of the test for terminal node. It will need incorporation of a depth parameter *k*, perhaps passed along with the node, decremented at each recursive call. It will become zero when the node is on the horizon. This is left as an exercise (number 5) for the reader.

```
Minimax(j)
  1     /* To return the minimax value V(j) of a node j */
  2 if Terminal(j)
  3     then return V(j) ← e(j)
  4     else for i ← 1 to b            /* b is the branching factor */
  5            do
  6              Generate j_i the i^th successor of j
  7              V(j_i) ← Minimax(j_i)    /* recursive call */
  8              if i = 1
  9                   then CV(j) ← V(j_i)
  10                  else if j is MAX
  11                           then   CV(j) ← Max(CV(j), V(j_i))
  12                           else   CV(j) ← Min(CV(j), V(j_i))
  13 return V(j) ← CV(j)
```

FIGURE 8.17 The *MINIMAX* algorithm recursively calls itself till it reaches a terminal node. A terminal node is either a leaf of the game tree or a node at depth *k*. The algorithm does a *k*-ply search from left to right. Note that the recursive calls are of decreasing ply depth. One will need to keep track of depth of a node.

In the above algorithm, the *minimax* value is returned but not the best move that leads to that value. Since the objective is to play the game, the following version returns the best move. It calls the above *Minimax* algorithm for each successor of the root, and keeps track of the best move as well as the best board value.

```
BestMove(j)
  1    /* To return the best successor b of a node j */
  2 b ← NIL
  3 value ← -LARGE
  4 for i ← 1 to b
  5     do V(j_i) ← Minimax(j_i)
  6     if V(j_i) > value
  7          then   value ← V(j_i)
  8                 b ← j_i
  9 return b
```

FIGURE 8.18 The algorithm *BestMove* accepts a board position and returns the best move for *MAX*. It calls algorithm *Minimax* with each of its successors and keeps track of which successor yields the best value.

Figure 8.19 depicts the tree searched by the algorithm *Minimax* and *BestMove* for a synthetic game tree. The tree is a binary tree, with two choices to each player at each level. The values for the evaluation at the 4-ply level have been arbitrarily chosen.

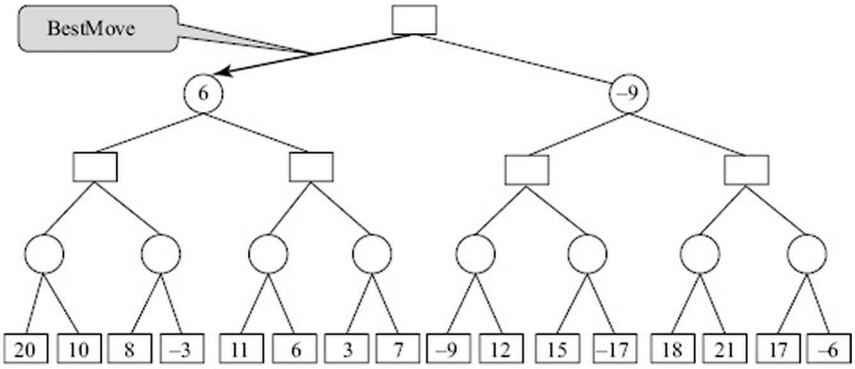

FIGURE 8.19 The algorithm *BestMove* calls algorithm *Minimax* for each of the successors of root, which computes the *minimax* value of each of them. It then chooses the best successor and returns that as the best move.

The *Minimax* algorithm above is the one that is doing the search. The *BestMove* algorithm is simply a modification to keep track of the best move found by *Minimax*. In Exercise 6, the reader is asked to modify the algorithm, so that the moves available to *MAX* at the next turn (after two plies) are stored along with their backed-up values. We will see in the following section how to exploit this information.

The algorithm *Minimax* finds the best move after searching the entire tree *k*-ply deep. There are, however, situations when it is not necessary to continue searching. This happens when it is known that searching further does have any scope of improvement. The simplest case is when a winning move has already been found, as shown in Figure 8.20.

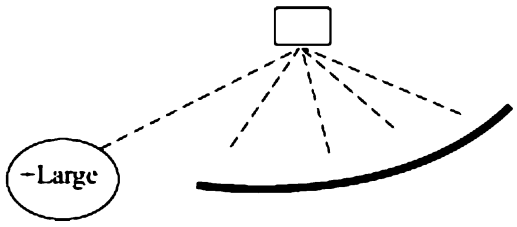

FIGURE 8.20 When a winning move has been found, the other options need not be explored.

8.2.2 Algorithm *AlphaBeta*

In the above figure, *MAX* has found a successor that evaluates to *+Large*. Since one cannot hope to improve upon, it does not make sense to search any further. We say that the search tree has been pruned. For pruning to happen, it is not necessary that a winning move has been found. It can also happen that during exploration it becomes clear that a particular child of a node cannot offer to improve upon the value delivered by a sibling. In that case, that node need not be explored further.

To understand this, we can view the game tree as a supply-chain process. At the top level, *MAX* has a set of *MIN* suppliers, from which it will select the one with the maximum value. Likewise, each *MIN* has *MAX* suppliers, from which the one with the lowest valued one will be selected. This process continues down the tree.

As the search in algorithm *Minimax* continues from the left to right, each node on the search frontier has been partially evaluated as shown in Figure 8.21. We will call the partially (or fully) known values of *MAX* nodes as *a* values. These values are lower bounds on the value of the *MAX* node. This is because that the *MAX* node will only accept higher values from the unevaluated successors. *MAX* nodes are also known as *Alpha* nodes. Likewise, *MIN* nodes are also called *Beta* nodes and store *b* values, which are upper bounds on values of the concerned *MIN* nodes. Remember that the *a* and *b* values are values that are already available to the respective nodes. They are not going to be interested in any successors that offer something inferior. Not only that, they are not going to be interested in any descendant that does not offer a better value.

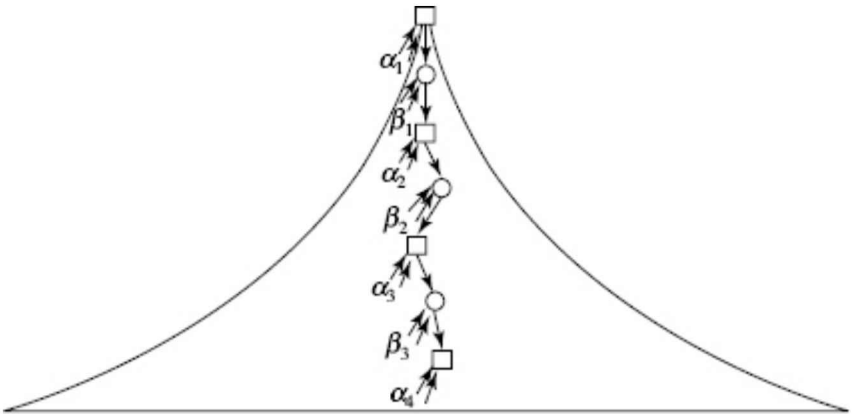

FIGURE 8.21 The search frontier contains a partial path in the game tree in which nodes have been partially evaluated. As this frontier sweeps to the right, these node will get fully evaluated.

The following example is from the *Noughts and Crosses* game. We assume a 2-ply search, in which the following evaluation function is used,

$e(J)$ = (numbers of rows, columns and diagonals available to *MAX*) – (number of rows, columns and diagonals available to *MIN*)

We look at the progress of the search to understand the need to prune search, shown in Figure 8.22. The algorithm starts with *MIN* child *A* by placing a cross in a corner. The *MIN* node *A* looks at all children and evaluates to a value –1. Note that while there are seven successors of *A*, only the five distinct ones are shown in the figure. Now the root has $a = -1$, which means that it will not go lower than –1. In the figure, this is

expressed by the inequality $a \geq -1$. It then turns to *MIN* child *B*. The first successor of *B* sends back a value –1, which becomes an upper bound on the value of *B*. Since this is an upper bound and the root is already getting a value –1, the root is not going to be influenced by *B*, and the *rest* of the *tree* below *B* can be pruned away. Note that even with *k*-ply search, where *k* is larger, the same pruning will happen as long as the backed-up values are as shown. It is also important to note that a node can be pruned only after it has been partially evaluated.

We call the pruning shown in the above figure an *α cutoff.* An *α* cutoff occurs below a *β* node, when it is constrained to evaluate to a lower (or equal) value than the *α* value above it. Correspondingly, when an *α* node has a lower bound (*α* value) that is higher than the *β* value of an ancestor than the rest of the tree below, it is pruned. This is known as a *β* cutoff.

In fact, this conflict between the bounds of two nodes need not be between a parent and child only. As described in (Pearl 1984), the *Alpha* node *J* in Figure 8.21 (with value α_4) will *influence* the root, *only* if it is greater that all the *α* values in the (*Alpha*) ancestors, and smaller than all the *β* values in the (*Beta*) ancestors. That is, the value *V*(*J*) of a node *J* must satisfy,

$$\alpha < V(J) < \beta$$

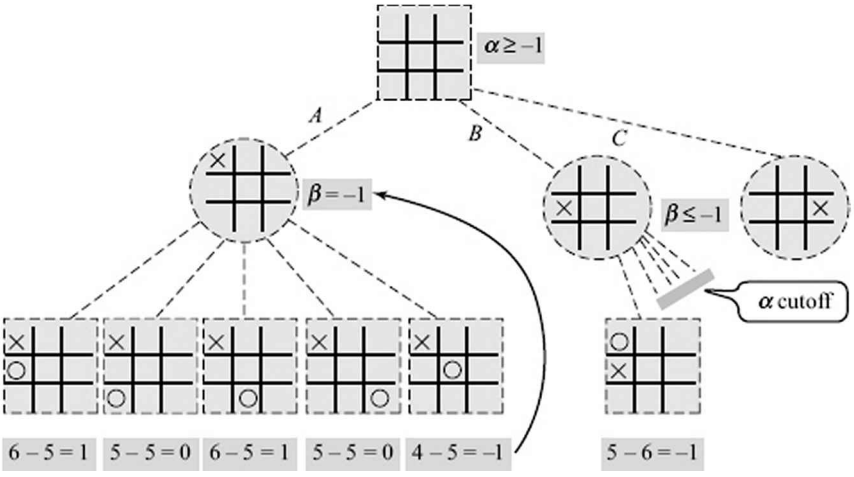

FIGURE 8.22 After evaluating the child *A*, the root node gets an *α* value of –1. When it starts on move *B*, it sees a *β* value of –1. Since this is an upper bound, the node *B* does not need to be explored further, and an *α* cutoff takes place.

where,

$$\alpha = \max \{\alpha_1, \alpha_2, \alpha_3, \ldots\}, \text{ and}$$
$$\beta = \min \{\beta_1, \beta_2, \beta_3, \ldots\}$$

When we are about to solve for a node *J*, we can propagate these bounds to the algorithm from its ancestors, and terminate the search if

these bounds are crossed. Note that an *Alpha* node can only increase in value, and can only cross a β bound. If it does then the search below the *Alpha* node is discontinued, and a β cutoff takes place. Likewise, a *Beta* node can only cross a lower bound α, and can be pruned using an α cutoff. The two kinds of cutoffs are illustrated in Figure 8.23.

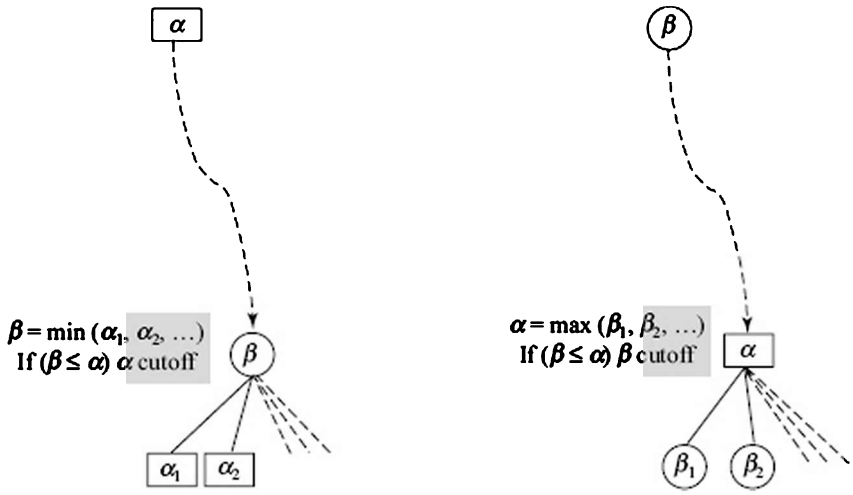

FIGURE 8.23 The α cutoff is induced by an α-bound from a *MAX* ancestor below a β-node. The β cutoff is induced by a β-bound from a *MIN* ancestor below an α-node.

The resulting algorithm known as the *AlphaBeta* algorithm is given in Figure 8.24 below.

```
AlphaBeta(j, α, β)
  1     /* To return the minimax value of a node j */
  2     /* Intially α = -LARGE, and β = +LARGE */
  3 if Terminal(j)
  4     then  return e(j)
  5     else  if j is a MAX node
  6                then for i ← 1 to b      /* j_i is the j^th child of j */
  7                       do α ← Max(α, AlphaBeta(j_i, α, β))
  8                          if α ≥ β   then return β
  9                          if i = b     then return α
  10               else   /*j is MIN */
  11                  for i ← 1 to b
  12                       do β ← Min(β, AlphaBeta(j_i, α, β))
  13                          if α ≥ β   then return α
  14                          if i = b      then return β
```

FIGURE 8.24 The *AlphaBeta* algorithm searches the tree like *Minimax* from left to right. It passes two bounds α and β to each call, and continues searching, only if the node has a value within those bounds.

The *AlphaBeta* algorithm is called initially with bounds α = –*Large*, and β = +*Large*. As search progresses, these bounds come closer, and eventually converge on the *minimax* value of the tree. At any point, when it is recursively called with bounds α and β, it returns *V*(*J*) if the value lies

between the two bounds. Otherwise, it returns a value β if J is an *Alpha* node, and returns α, if J is a *Beta* node.

An Example

The following figure shows the subtree in a 4-ply binary game. As in the earlier example, the values of the evaluation function have been filled in randomly.

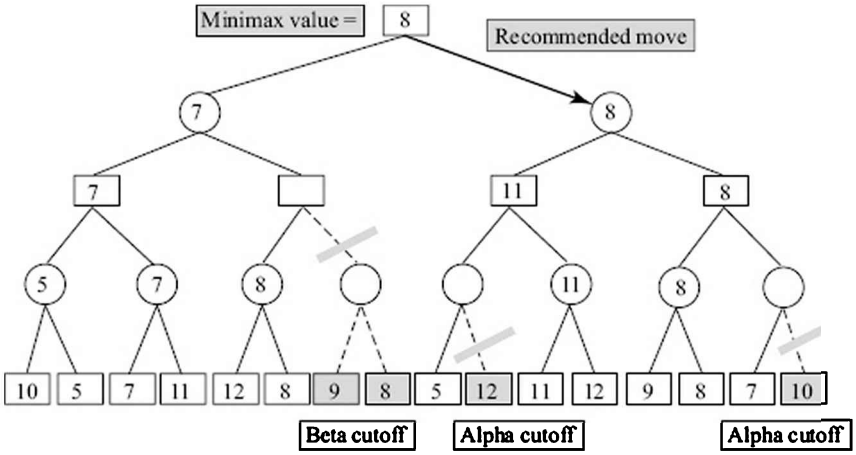

FIGURE 8.25 The *AlfBeta* algorithm evaluates twelve out of the sixteen leaves. It does one *b* cutoff and two *a* cutoffs as shown. The *minimax* value is 8.

The reader is encouraged to verify that the *minimax* value of the game tree is indeed 8, as reported by the *AlphaBeta* algorithm. This is irrespective of the values that are in the pruned leaves, shown as shaded nodes. Based on the leaves the algorithm has seen earlier in the search, which is in the left part of a tree, some leaves have no influence on the *minimax* value. The reader is encouraged to try different values in the shaded nodes and verify that the *minimax* value does not change.

Since the cutoffs are dictated by values of nodes seen earlier, the amount of pruning by *AlphaBeta* algorithm will depend upon the leaves seen earlier. If the *better* moves for both sides are explored earlier then the window of α-bound and β-bound will shrink faster and more cutoffs will take place. In particular, if the node whose value is backed up to the root, and which represents the best moves from both sides, is found earlier, the number of cutoffs will increase significantly. In the above figure, one can observe that the *minimax* value comes from the right half of the tree. Let us flip the tree about the root by reversing the order of the leaves and run the *AlphaBeta* procedure on the reversed tree. The resulting cutoffs are shown in Figure 8.26.

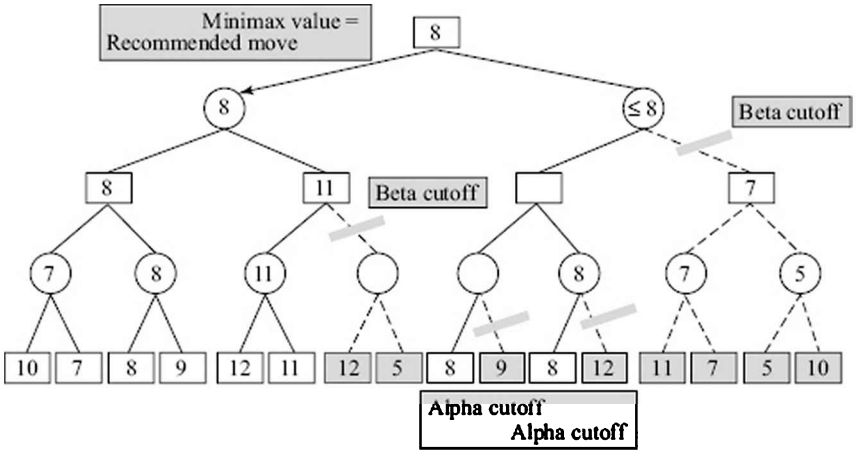

FIGURE 8.26 When the tree of Figure 8.25 is flipped about the root, the algorithm inspects only eight nodes out of sixteen.

As can be seen, the number of cutoffs in the same game tree, but with the order reversed, has gone up to eight, and only eight of the sixteen nodes are inspected. In Exercise 8, the reader is asked to construct a tree in which *AlphaBeta* is forced to evaluate all leaves, and then try the algorithm on the flipped tree.

It is evident that the performance of the *AlphaBeta* algorithm depends upon the order in which the moves are generated, and when the better moves are generated earlier, the cutoffs will be greater in number. But how do we generate the better moves first? One way could be to somehow put in domain-specific heuristics to order the moves. For example, in the *Noughts and Crosses* game, corner moves might be preferred over side moves.

Another, and a domain independent, way would be to use one instance of search to order the moves in the next instance of search. Let us say *MAX* is to play in some board position X. *MAX* calls the *AlphaBeta* algorithm and along with finding the best moves, also keeps track of the moves it explored at the third ply. These are moves it will start searching with the next time it has to make a move. *MAX* utilizes the *current* search to order the moves in preparation of the next one (see exercises 6 and 9).

Another way would be to give the algorithm a sense of direction, like in the transition from depth first search to heuristic search in Chapter 3. The algorithm *AlphaBeta* searches blindly from left to right. In the algorithm *SSS**, described in the next section, the search is guided towards the better looking nodes. The interesting thing is that the guidance does not come (directly) from a domain specific heuristic function. Instead, it is generated by a preliminary exploration of the game tree up to *k*-ply depth. This preliminary exploration yields some information which is used to guide the search towards better nodes. This notion of domain independent heuristics is an exciting one, and we shall