Chapter 3

Multiagent Search

"Friendships born on the field of athletic strife are the real gold of competition. Awards become corroded, friends gather no dust."– Jesse Owens

3.1 Introduction فاطمة+ رنيم

The previous chapter focuses on search involving a single agent. However, many real settings are associated with environments involving multiple agents. In multiagent environments, there are multiple agents interacting with the environment and affecting the state of the environment, which in turn affects the actions of all agents. Therefore, an agent cannot take actions without accounting for the effects of the actions of other agents. Since the actions of other agents cannot always be exactly predicted, a multi-agent environment works with partial knowledge; however, some information is available about the goals and utility functions of the other agents. Therefore, this type of setting is considered to be related to but distinct from uncertain or probabilistic environments.

The different agents might be mutually cooperative, independent, or they may be competitive (depending on the application at hand). Some examples of multi-agent environments are as follows:

- *Competitive agents:* Games like chess represent competitive environments in which the agents are in competition with one another. The actions of the agents cause transitions in the state of the environment (which correspond to the position of the pieces on the chessboard). Such environments are also referred to as *adversarial*. Adversarial agents do not communicate with one another beyond affecting the environment in an adversarial way.

- *Independent agents:* In a car-driving environment, one might have multiple car-driving agents, each of which is optimizing its own objective function of safely driving on the road and reaching its specific destination. However, the agents are not adversarial; in fact, the agents are partially cooperative because neither agent would like to have an

C. C. Aggarwal, *Artificial Intelligence*, https://doi.org/10.1007/978-3-030-72357-6_3

accident with the other. However, the cooperation is only implicit because each agent is optimizing its own objective function, and the cooperation is a side effect of their mutual benefit.

- *Cooperative agents:* In this case, the agents are communicating with one another, and they are optimizing a global objective function that defines a shared goal. For example, it is possible to envisage a game-playing environment in which a large number of agents perform a collaborative task and keep each other informed about their progress. In the event that the game is defined by two teams playing against one another, the setting has aspects of both competition and cooperation.

This chapter will propose methods that can work in these different types of environments. Some methods are able to work in all three types of environments, whereas other methods are designed only for adversarial environments. We will discuss both generic methods (which can work in all types of environments) as well as specific methods that are geared towards adversarial environments. The special focus on adversarial environments is caused by its importance in key artificial intelligence applications like game playing.

In many settings, the agents might alternate in their actions, although this is not always necessary. For example, in the case of games like chess, the agents usually alternate to make moves. However, in some environments, the agents do not alternate in their move choices. Since much of the focus of this chapter is on adversarial environments, it will be assumed that the moves of the agents alternate. Furthermore, we will specially focus on environments containing two agents, although many of the ideas can be generalized to environments containing multiple agents.

Multiagent environments necessitate different types of search settings as compared to single-agent environments. One issue is that a given agent has no control over the choices made by another agent, and therefore it might have to *plan for all possible contingencies* arising from the other agent's action. In other words, it is no longer sufficient to find a single path from the starting state to the goal in order to determine whether it is possible to achieve success with a particular action — rather, one must consider all possible choices that other agents might make while evaluating the long-term effect of a particular action. For example, while evaluating whether a particular move in chess will yield a win, one must consider all possible moves that the opponent might make (and not just a single one). Ideally, the agent must assume that the adversary will make the best possible move available to it. It is noteworthy that this setting is very similar to what happens in probabilistic environments where an action made by an agent has consequences that are stochastic in nature. Therefore, one might need to consider all possible stochastic outcomes while evaluating the effect of a particular action.

Multiagent environments are inherently online because one cannot make a decision about future actions of an agent without considering *all possible* choices made by the other agents (i.e., all possible contingencies). Since it is usually too expensive to enumerate all possibilities (i.e., all contingencies), one simply has to look ahead for a few transitions and then perform a heuristic evaluation in order to determine which action is the most appropriate one in the current state. Therefore, multiagent algorithms are mostly useful for only making choices about the next action in a particular state; subsequent actions require one to observe the actions of other agents before reevaluating how to make further choices. For example, a chess-playing agent can only make a decision about the next move, and there is considerable uncertainty about its next move without knowing what the adversary might play. It is only after the adversary makes the next move that the chess-playing agent may reevaluate the specific choice of move that it might make. This type of search is referred to as *contingency-centric search*.

One useful way of performing contingency-centric search is to use AND-OR trees. Such trees have OR nodes corresponding to the choices made by the agent making a particular decision (where the agent has free choice and can select any one), and they have AND nodes corresponding to the actions of the other agents (i.e., contingencies where the agent has no choice). For the case of competitive agents, these trees can be converted into utility-centric variants, which are referred to as minimax trees. Before discussing such types of adversarial search, we will address the more general issue of contingency planning, which occurs in situations beyond adversarial environments. Most of this chapter will, however, focus on competitive environments and game playing, which are adversarial environments. The focus on game-playing environments is because of their special importance as microcosms of artificial intelligence settings, where many real-world situations can be considered in a manner similar to games. While games are obviously far simpler (and more crisply defined) than real-life situations, they provide a testing ground for the basic principles that can be used for more complex and general scenarios.

This chapter is organized as follows. The next section introduces AND-OR trees for addressing contingencies, which occur quite frequently in the context of multiagent environments. The use of AND-OR search represents a scenario of uninformed search, which can take a long time in most realistic environments. Therefore, the use of informed search with state utilities is discussed in Section 3.3. This approach is then optimized with alpha-beta pruning in Section 3.4. Monte Carlo tree search is introduced in Section 3.5. A summary is given in Section 3.6.

3.2 Uninformed Search: AND-OR Search Trees

Single-agent environments correspond to search settings in which finding *any* path from the starting state to the goal state can be thought of as a success. In other words, one can choose *any* action at a given state that eventually leads to a goal node. Therefore, *all* nodes can be thought of as OR nodes, and finding a single path from the starting node to the goal is sufficient. This is not the case in multi-agent environments, where one cannot control the actions of other agents. One consequence of the multiagent setting is that finding a single path through the state-space graph is no longer sufficient to ensure that choosing a particular course of action will yield success. For example, if a particular move in chess leads to a win only because of the opponent making suboptimal moves, whereas another move leads to a guaranteed win, the first of the two moves may not be preferable. As a result, one has to account for the different choices (i.e., paths) that result from this uncertainty, while choosing a course of action in a particular state. In other words, one must work with a tree of choices for the paths from any particular state to the goal; the adversarial nodes are AND nodes, where all possible paths must be followed to account for the worst-case scenario where the opponent makes the choices that are most suitable to her. Therefore, unlike single-agent environments, a multi-agent environment contains both AND nodes and OR nodes.

In this section, we will primarily consider the case of the two-agent environment. In the two-agent environments, each node of the tree corresponds to the choices made by a particular agent, and alternate levels of the tree correspond to the two different agents. The decisions made by the agents are represented by a tree of choices (such as a tree of moves in chess). The placement of AND and OR nodes depends on the choice of the agent from the perspective of which a particular action is being performed. We refer to the two agents as the *primary agent* and the *secondary agent*, respectively. The root of the tree corresponds

to the choices made by the primary agent, and it is always an OR node. The entire tree is being constructed from the perspective of the primary agent. At the levels of the tree, where the secondary agent is due to make a transition, the primary agent cannot predict what action the secondary agent might take, and therefore has to account for all possibilities of actions taken by the secondary agent; such a node is, therefore, an AND node since the tree is constructed from the perspective of the primary agent. The use of the AND node ensures that one explores *all* possibilities for the actions chosen by the secondary agent. This leads to AND-OR search, in which the OR nodes correspond to the primary agent (from whose perspective the problem is being solved), and the AND nodes correspond to the choices made by the secondary agent(s). For simplicity, we first consider a two-agent problem in which the goal is defined from the perspective of (primary) agent A, whereas (secondary) agent B might make transitions that agent A cannot control. Furthermore, agent A and agent B make alternate transitions. Although this model might seem simplistic, it is the model for large classes of two-person-game settings. can capture more powerful settings, such as multi-agent settings, by relatively minor modifications (cf. Section 3.2.1). In the AND-OR tree, alternate actions are made by agent A and agent B using nodes at alternate levels. Therefore, directed edges exist only between the nodes corresponding to the transitions of the two agents. We assume that the state-space graph is structured as a tree, which is a simplification of the assumptions made in the previous chapter. One consequence of this simplification is that the same state may be repeated many times in the tree, whereas each state is generally represented only once in the state-space graphs of the previous chapter. For example, the same position in chess may be reached by using different sequences of moves, and this is especially evident during the opening portion of the game, where alternate move sequences to reach the same opening position are referred to as *transpositions*. However, each of these states is a different node in the tree, and therefore a given state may be represented multiple times in the tree. It is also possible to collapse this tree into a state-space graph in which each state is represented exactly once (see Exercise 9), and doing so can lead to a more efficient representation, which is typically not a tree. The simplification of using a tree structure assures that one does not have to keep track of previously visited nodes with a separate data-structure. In practice, keeping such a data structure can lead to considerable computational savings, although it leads to increased algorithmic complexity and loss of simplicity in exposition.

The starting state s is the root of the tree corresponding to the state-space graph being examined. The initial state for the problem is denoted by s, in which agent A takes the first action. The goal state is assumed to be reached when the condition $\mathcal{G}$ is assumed to be *True* from the perspective of agent A. The overall procedure is shown in Figure 3.1, and it is a modification of the procedure used in Figure 2.2 of Chapter 2. This algorithm is structured as a recursive algorithm and its output is the Boolean value of *True* or *False*, depending on whether or not any goal node (defined by constraint set $\mathcal{G}$) is reachable from the starting node s. Note that returning a Boolean value is sufficient in order to determine which course of action to take at the current node, because one can always print out the branch that was followed at the top level of the recursion in order to determine what action agent A should make. While it is possible to identify the relevant branch at the top level of the tree, it is no longer possible to report the full path of actions from the starting state to the goal node because of the inability to predict the actions of the secondary agent B. After all, all AND nodes need to be explored, even though OR nodes allow the flexibility of following only a single path. In general, one can only predict the next action from the current node, and the approach is typically used in online environments (as the secondary agent takes actions and subsequent states become known).

```
Algorithm AO-Search(Initial State: s, Goal Condition: G)
begin
  i = s;
  if i satisfies G then return True;
       else initialize resultA= False;
  for all nodes j ∈ A(i) reachable via agent A action from i do
  begin
    resultB= True;
    if (j satisfies G) then return True;
    for all nodes k ∈ A(j) reachable via agent B action from j do
    begin
      resultB= (resultB AND (AO-Search(k, G)));
      if (resultB = False) then break from inner loop;
    end;
  resultA = (resultA OR resultB);
  if (resultA= True) then return True;
  end;
  return False;
end
```

Figure 3.1: The AND-OR search algorithm

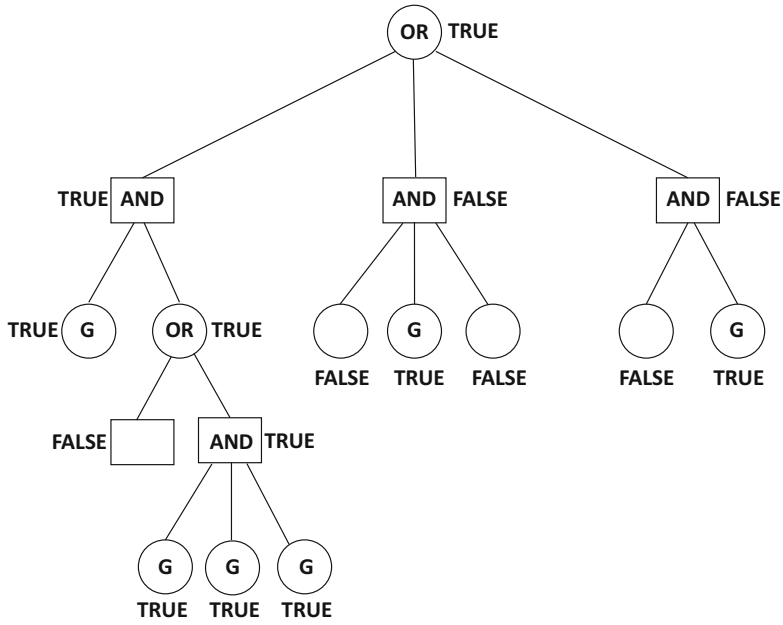


Figure 3.2: The AND-OR tree for multiagent search

A single recursive call of the AND-OR search algorithm processes two levels of the tree, the first of which corresponds to the actions of agent A and the second corresponds to the actions of agent B. Therefore, two levels are processed in each recursive call. The algorithm starts at node s and checks whether the starting node satisfies the goal conditions in $\mathcal{G}$. If this is indeed the case, the algorithm returns the Boolean value of *True*. Otherwise, the algorithm initiates a recursive call for each node two levels below the current node. This is achieved by first retrieving all states $j \in A(i)$ that are accessible via a single action from current state i, and then calling the procedure from each child of j. Note that each such child of i is also one in which the primary agent is due to make an action. The set S of grandchildren of node i is defined as follows:

$$S = \cup_{j \in A(i)} A(j)$$

The recursive call is initiated for each such node $k \in S$, unless the goal condition $\mathcal{G}$ is satisfied by a recursive call from one of these nodes. The recursive call returns either *True* or *False*, depending on whether or not the goal is reachable from that node using the AND-OR structure of recursive search. The results of all the recursive calls at the nodes two levels below are then consolidated into a single Boolean value of *True* if for even one of the actions of agent A from the current node, all actions for the agent B at the child node return a value of *True*. In other words, the OR condition is applied at the current node, and the AND condition is applied at the nodes one level below. Therefore, the two Boolean operators are applied at alternate levels of the tree. The recursive description of Figure 3.1 only returns a single Boolean value, and it can also be used to decide the action at the top level of the tree. In particular, if the Boolean value of *True* is returned by the algorithm, the top level of any OR branch that yields the value of *True* can be reported as the action choice. Note that subsequent choices of actions of either agent at deeper levels of the tree are not reported; this is because the choices made by the secondary agent are not predictable and can sometimes be suboptimal.

An example of an AND-OR tree is shown in Figure 3.2. The goal nodes are marked by 'G.' The goal node is always a node that returns the value of *True*. It is possible for a goal node to occur at the action of either agent A or agent B, although goal nodes might correspond to only one of the two agents in some game-centric settings. For example, a win (i.e., occurrence of goal node) in the game of tic-tac-toe occurs only after the move by the primary agent. It is evident that many more branches and paths of the tree need to be traversed because of the AND condition (as compared to single-agent settings in which only a single path needs to be traversed). Note that a single-agent setting can be viewed as a search tree with only OR nodes, which is the main reason that only a single path needs to be traversed.

For large state spaces, the need to traverse multiple paths in AND-OR trees can be a problem, because one would need to evaluate a large number of paths in order to determine whether the goal condition can be reached. The number of possible paths is directly proportional to the number of nodes in the tree. Since the state space can be massive in real-world settings, this can be impractical in many cases.

3.2.1 Handling More than Two Agents

The description of the AND-OR search algorithm in the previous section involves only two agents. What happens in settings where there are more than two agents? A multi-agent environment might seem to be quite onerous at first glance, especially if the secondary agents can make transitions in arbitrary order, which are not necessarily alternately interleaved

with those of the primary agent. For example, when one has three agents, A, B, and C, the transition by agent B could occur before agent C and vice versa. In such cases, the transitions made by multiple secondary agents can be modeled by using the union of all possible actions by other agents and treating it as the action made by a single dummy (secondary) agent; similarly, a missing step by the primary agent between consecutive steps by the secondary agents can be modeled with a dummy action that transitions from a state to itself. The resulting problem can then be reduced to the two-agent problem in which alternate steps are taken by the two agents. There are, of course, limitations to this approach; the limitations are caused by the fact that combining multiple actions into a single action can blow up the number of possible actions at a particular state. This will result in a tree with very high degree, whose size increases rapidly with increasing depth. As a result, one can practically address this type of scenario only when the underlying trees are shallow.

3.2.2 Handling Non-deterministic Environments

Interestingly, AND-OR search is also useful in non-deterministic environments, where the transitions resulting from a single agent are probabilistic in nature. In probabilistic settings, an agent cannot fully control the state resulting from choosing a particular outcome. This is similar in principle to the multi-agent setting in which the primary agent cannot fully control the state they will find themselves in as a result of choices made by other agents. In other words, each action made by an agent has a different distribution of outcomes in terms of the final state in which the agent lands after making a transition. This type of randomness can be captured by allowing each action a of the agent in state i to move from state i to a dummy action-specific state (i, a). This dummy state is an AND node. From this node, one moves to the different states corresponding to the various possibilities allowed by action a in state i. The probability of moving to a particular state from a dummy state is governed by a probability distribution specific to state (i, a). This setting is exactly similar to the two-agent setting discussed in the previous section. Therefore, multiagent environments are quite similar to non-deterministic settings in many ways (from the perspective of a single agent who cannot predict the other agent); however, we do not formally consider an environment to be probabilistic simply by virtue of being a multi-agent environment. Similarly, a probabilistic environment is not formally considered a multiagent environment. Nevertheless, the algorithms in the two cases turn out to be quite similar from the perspective of a single agent (who cannot completely control the transitions at all levels of the tree).

3.3 Informed Search Trees with State-Specific Loss Functions

The AND-OR trees discussed in the previous section represent a case of uninformed search in which large parts of the search space may need to be explored. Unfortunately, this type of setting is not very suitable for large state spaces, in which the size of the tree is very large as well. For example, consider the two-agent game of chess. There are more than 100 billion possible states after each player has made four moves. The number of states for a 40-move game of chess is more than a number of atoms in the observable universe. Clearly, one cannot hope to search the entire tree of moves with the use of AND-OR search. The main problem is that the depth of the tree can be quite large in the worst case, as a result

of which it is impossible to explore the tree down to *leaf nodes* which have termination outcomes and crisply defined Boolean values. Therefore, it is useful to have a method by which one can control the depth (and number of states) of the tree that need to be explored. In other words, there needs to be aa way to explore the upper portions of the tree effectively and make decisions about transitions without having access to the eventual outcomes at leaf nodes.

In such cases, it is only possible to perform search over a subset of the tree with the help of state-specific loss functions. State-specific loss functions represent the heuristic "goodness" of each node from the perspective of each agent, and therefore free the agent from the intractable task of having to know the termination outcomes at leaf nodes. By using state-specific loss functions, one is able to evaluate intermediate states in which a lower numerical value is indicative of a more desirable state. This evaluation can be used for various types of pruning, such as for reducing the depth of the explored tree or for pruning specific branches of the tree. Note that the utility of a particular state from the perspective of each agent is typically different; in the case of adversarial game-playing settings, the utilities of the two players might be completely antithetical to each other. A state that is good for one agent is poor for the other, and vice versa. In fact, in the case of adversarial settings like chess, the evaluation of a particular position from the perspective of one agent is exactly the negative of the evaluation from the perspective of the adversary.

In the following, we will work with a two-agent environment, denoted by agent A and agent B respectively. Each state is associated with a loss function. The loss of state i for agent A is denoted by $L(i, A)$, whereas the loss of state i for agent B is denoted by $L(i, B)$. Smaller loss values are considered desirable from the perspective of each agent. *An important assumption in this case is that both players are aware of each other's evaluation functions.* This knowledge can be used by an agent to explore the tree from the perspective of the other agent. The cases in which agents are unaware of each other's evaluation functions is generally not feasible to address with the use of reasoning methods; in those cases, it becomes important to use learning methods in which agents learn optimal choices from past experiences.

Loss functions associated with states are often constructed using heuristic evaluations of a state with the aid of domain knowledge. In order to illustrate this point, we will use the game of chess as an example. In such a case, a state corresponds to a board position, and an agent corresponds to a chess player. In such a case, an evaluation of the board position corresponds to an evaluation of the goodness of a position from the perspective of the player who is about to make a move. These types of goodness values are often constructed on the basis of algorithms by human experts, who use their knowledge of chess in order to give a numerical score to the position from the corresponding player's perspective (since evaluations are specific to each player in an adversarial game). This evaluation might aggregate numerical scores for material, positional factors, king safety, and so on. The evaluation function is almost always constructed and coded into the system with the help of a human domain expert; the use of domain knowledge is, after all, the hallmark of all deductive reasoning methods. One problem is that such evaluations are hard to encode in an interpretable way with explicit material and positional factors (without losing some level of accuracy). Human chess players often make moves on the basis of intuition and insight that is hard to encode in an interpretable way. This general principle is true of most real-world settings, where domain-specific proxies can encode relevant factors to the problem at hand in an incomplete way. In other words, such evaluations are inherently imperfect by design and are therefore bound to be a source of inaccuracy in the decisions made by the underlying system. Recent years have also seen the proliferation of inductive

learning methods for board evaluation in the form of reinforcement learning. This topic will be discussed later in this chapter and also in a dedicated chapter on reinforcement learning (see Chapter 10).

For some adversarial settings like chess, we might have $L(i, A) = -L(i, B)$. In such cases, these types of search reduce to *min-max search* in which the objective of search at alternative levels is either minimized or maximized by defining a single loss. However, we do not make this assumption in this section, and work with the general case in which the objectives of the two agents might be independent of one another. Therefore, no relationship is assumed between $L(i, A)$ and $L(i, B)$. The agent taking the action at the root of the tree is referred to as the primary agent A, and the agent taking action just below the root is the secondary agent B. In game-playing situations, it is possible for the system to use one of the two agents (say, agent B) to take decisions about actions, and the other agent (say, agent A) to play an anticipatory role in predicting the optimum actions the human might make. However, in other situations, it is possible for each agent to be full automated (and part of the same system). Furthermore, the objectives of the two agents can be partially or wholly independent. For appropriate choices of the losses, it is even possible for the agents to cooperate with one another partially or completely.

The use of loss functions associated with states helps in creating an informed exploration process in which the number of states visited is greatly reduced with pruning. A key point is that such informed search algorithms also use the domain-specific loss functions in order to explore the tree of possibilities up to a particular depth d (as an input parameter) in order to reduce the size of the tree being explored. The depth d corresponds to the number of actions on the path of maximum length from the root to the leaf. When d is odd, the last transition at the lowest level of the tree corresponds to the primary agent A. For example, using $d = 1$ corresponds to making the best possible move in chess after using a heuristic evaluation with the primary agent making a single move. When d is even, the last transition is made by the secondary agent B. The computational advantages of doing so are significant, because most of the nodes in search trees are at the lower levels of the tree.

By restricting the depth of exploration, one is exposed to the inaccuracies caused by imperfections in the evaluation function. The goal of search is only to sharpen the evaluation of a possibly imperfect loss function by using lookaheads down the tree of moves. For example, in a game of chess, one could simply apply the evaluation function after performing each possible legal move and simply selecting the best evaluation from the agent's perspective. However, this type of move would be suboptimal because it fails to account for the effects of subsequent moves, which are hard to account for with an evaluation that is inherently suboptimal. After all, it is hard to evaluate the goodness of a position in chess, if a long sequence of piece exchanges follow from the current board position. By applying a *depth-sharpened evaluation*, one can explore the entire tree of moves up to a particular depth, and then report the best move at the top level of the tree (where each move is optimal from the perspective of the player being considered at a particular depth). This evaluation is of much better quality because of the use of lookaheads. Therefore, the choice made at the root of the tree is also of much better quality (because of deep lookaheads) than a move made only with a difficult-to-design loss function (but no lookahead).

The overall algorithm for informed search with two agents, denoted by agent A and agent B, is shown in Figure 3.3. The input to the algorithm is the initial state s and the maximum depth d of exploration of the tree. Note that each action of an agent contributes 1 to the depth. Therefore, a single action by each agent corresponds to a depth of 2. The losses from the perspective of agent A and agent B in state i are $L(i, A)$ and $L(i, B)$, respectively. In adversarial environments, these losses are the negations of one another, but

Algorithm *SearchAgentA* (Initial State: s, Depth: d)
begin
  $i = s$;
  if (($d = 0$) or (s is termination leaf)) **then return** s;
  $min_a = \infty$;
  for $j \in A(i)$ **do**
  begin
    $OptState_b(j) = SearchAgentB(j, d-1)$;
    if $(L(OptState_b(j), A) < min_a)$ **then** $min_a = L(OptState_b(j), A)$; $beststate_a = OptState_b(j)$;
  end;
  return $beststate_a$;
end

Algorithm *SearchAgentB* (Initial State: s, Depth: d)
begin
  $i = s$;
  if (($d = 0$) or (s is termination leaf)) **then return** s;
  $min_b = \infty$;
  for $j \in A(i)$ **do**
  begin
    $OptState_a(j) = SearchAgentA(j, d-1)$;
    if $(L(OptState_a(j), B) < min_b)$ **then** $min_b = L(OptState_a(j), B)$; $beststate_b = OptState_a(j)$;
  end;
  return $beststate_b$;
end

Figure 3.3: The multi-search algorithm in a two-agent environment

this might not be the case in general settings. The notation $A(i)$ denotes the set of the states directly reachable from state i via a single action of the agent at that state. There, $A(i)$ represents the adjacency list of node i in the tree of moves. The algorithm is structured as a *mutually* recursive algorithm in which each node for the search algorithm for agent A calls the search algorithm for agent B and vice-versa. The search call for agent A is denoted by *SearchAgentA*, and the search call for agent B is denoted by *SearchAgentB*. The main difference between the two procedures is that different losses $L(i, A)$ and $L(i, B)$ are used to make key choices in the two algorithms, and these choices govern the behaviors of the algorithms at hand in terms of the preferred actions. The two algorithms are denoted by the mutually recursive subroutine calls *SearchAgentA* and *SearchAgentB*, respectively. The former algorithm returns the best possible state obtained after exploring d moves down the tree from the perspective of agent A, whereas the second algorithm returns the best possible state from the perspective of agent B after exploring d moves down the tree. In other words, the two pseudocodes are almost identical in structure, except that they minimize over different loss functions. Each call returns the best possible state d transitions down the tree from the perspective the agent concerned. It is also possible to rewrite each pseudocode to return the value of the optimal node evaluation along with the optimal state d levels down the tree (which is how it is normally done in practice). This is achieved by keeping track of the optimal state value of each node during the current sequence of calls, and it is helpful in avoiding repeated evaluation of the same node during backtracking. *Unlike AND-OR trees, a single, optimal path can indeed be found d levels down the tree by using the loss-function.* Therefore, aside from reducing the depth of the tree being considered,

the approach reduces the amount of bookkeeping required by the algorithm as compared to an AND-OR algorithm (which requires exploration of multiple paths and corresponding bookkeeping).

The recursive algorithm works as follows. If the input depth d is 0 or the current node i is a termination leaf, then the current state is returned. On the other hand, when each algorithm is called for a non-terminal nodes i, the corresponding algorithm for a particular agent recursively calls the algorithm for the other agent from each node $j \in A(i)$ with depth parameter fixed to $(d-1)$. The minimum of the evaluations of the state is returned. Therefore, a state from $d-1$ levels down the tree will be returned by each of these $|A(i)|$ recursive calls. The final state is selected out of these $|A(i)|$ possible states by selecting the state with the least loss among these states. When agent A calls *SearchAgentB* for each state $j \in A(i)$, the optimal state from the perspective of agent B is stored in $OptState_b(j)$, and the best of these states from over the different values of j from the perspective of agent A is returned. Returning the best multi-agent state d levels down the tree does not necessarily yield the best action at the current node directly. However, one can separately keep track of which branch was followed to reach the best state d levels down the tree.

Although this algorithm can be used for adversarial environments by selecting the loss function for one agent to be the negative of the loss function for the other agent, it can also be used in other type of multiagent environments as long as the loss functions of the two agents are properly defined. In such cases, it is possible to have a certain level of cooperation between the two agents, if the corresponding loss functions are sufficiently aligned towards a particular goal.

3.3.1 Heuristic Variations

The algorithm of Figure 3.3 is the most basic version of the approach, and it is not optimized for performance. One issue is that many of the explored branches of the tree are often redundant. For example, in a game of chess some obviously suboptimal moves can be ruled out very quickly by humans only by exploring the tree of possibilities to a very shallow depth. One possibility is to explore a branch only if its evaluation is better than a particular threshold from the perspective of the agent for whom the evaluation is being performed. By using this approach, large parts of the search space are heuristically pruned early on. This can be achieved by using an additional loss threshold parameter l. When the loss function at the current node is greater than this quality threshold, it is unlikely that this particular branch will yield a good evaluation (although there can always be surprises deeper down the tree). Therefore, in the interest of practical efficiency, the algorithm simply returns the state corresponding to this node directly (by applying the loss function at this state) without exploring down further. This type of change is extremely minor, and it tends to prunes out branches that are unlikely to be germane to the evaluation process. There are many ways of further sharpening the pruning by using a shallow depth of exploration for pruning evaluations (such as 2 or 3) and a deep depth for the primary evaluation (as in the previous section).

Another natural optimization is that many states will be reached via alternative paths in the tree. For example, in a game of chess, different orders of moves[1] might result in the same position. It is helpful to maintain a hash table of previously visited positions, and use the evaluations of such positions when required. This can sometimes be tricky, as the earlier evaluation of a position at depth d_1 might not be as accurate as the evaluation of

[1]For those chess players who are familiar with formal move notation, the following two sequences $(1.e4, Nc6, 2.NF3)$, and $(1.Nf3, Nc6, 2.e4)$ lead to the same position.

```
Algorithm MaxPlayer(Initial State: s, Depth: d)
begin
  i = s;
  if ((d = 0) or (s is termination leaf)) then return s;
  max_a = −∞;
  for j ∈ A(i) do
  begin
    OptState_b(j) = MinPlayer(j, d − 1);
    if (U(OptState_b(j)) > max_a) then max_a = U(OptState_b(j)); beststate_a = OptState_b(j);
  end;
  return beststate_a;
end

Algorithm MinPlayer(Initial State: s, Depth: d)
begin
  i = s;
  if ((d = 0) or (s is termination leaf)) then return s;
  min_b = ∞;
  for j ∈ A(i) do
  begin
    OptState_a(j) = MaxPlayer(j, d − 1);
    if (U(OptState_a(j)) < min_b) then min_b = U(OptState_a(j)); beststate_b = OptState_a(j);
  end;
  return beststate_b;
end
```

Figure 3.4: Minimax search algorithm in a two-agent adversarial environment

a position at depth $d_2 > d_1$. Therefore, the depth of evaluation of the position also needs to be maintained in the hash table. In many cases, the same position may be obtained in different branches of the tree (by simple transpositions of moves), in which case the stored evaluation in the hash table can save a lot of time. This type of position is referred to as a *transposition table*. While it is not practical to maintain all previously seen positions, it is possible to cache a reasonable subset of them, if they are deemed to be relevant in the future. For example, it does not make sense to cache a position in chess, when a subset of the pieces have already been exchanged, or if the current position has already moved sufficiently far away from a cached position. In games like chess, some positions are unreachable from others. For example, if a pawn is moved from its starting state, the corresponding state cannot be reached again.

3.3.2 Adaptation to Adversarial Environments

The informed search approach is naturally suited to adversarial two-player environments like chess. The main difference of adversarial search with the informed search algorithm is that the loss functions of the two agents are related to one another in the former case; the loss function $L(i, A)$ and $L(i, B)$ are related to one another by negation:

$$L(i, A) = -L(i, B)$$

The resulting trees are referred to as *minimax trees*, since they minimize and maximize at alternate levels. Such trees form the basis of most traditional chess programs over the past

two decades, such as *Stockfish* and *DeepBlue*, although newer algorithms such as *AlphaZero* tend to use ideas from reinforcement learning. The former is a deductive method that uses domain knowledge (e.g., evaluation function), whereas the latter is an inductive learning method that learns from data and experience. Many methods combine ideas from both schools of thought.

One convention that is commonly used in chess-playing programs is to work with utility functions rather than loss functions, where the objective needs to be maximized rather than minimized. Therefore, we restate the algorithm of Figure 3.3 in minimax form, where a maximization of the utility function $U(\cdot)$ is performed in even layers (starting at the level of the base recursion) and a minimization of the same function $U(\cdot)$ is performed in odd layers (starting one layer below the base recursion). Therefore, the utility function is maximized with respect to the primary agent at the root of the tree. We conform with this different convention (as opposed to a minimization-centric loss function) in this section in order to make it consistent with the most common use of this approach in chess-playing programs.

The resulting algorithm is shown in Figure 3.4. As in the case of Figure 3.3, the input to the algorithm is the initial state s and the depth d of the search. It is assumed that agent A occurs as the maximizing agent at the base of the recursion, and agent B occurs as the minimizing agent one level below the base of the recursion. This algorithm is almost exactly identical to the pseudocode shown in Figure 3.3 as a mutually recursive algorithm, except that the two agents do not have different loss functions; the *same* utility function $U(\cdot)$ is maximized by agent A by the *MaxPlayer* call, and it is minimized by the agent B by the *MinPlayer* call. All other variables are similar in Figures 3.3 and 3.4. The control flow is also similar, except that maximization happens in one pseudocode and minimization happens in the other pseudocode. The subroutine *Maxplayer* describes the approach from the maximization player's perspective, whereas the subroutine *Minplayer* describes the approach from the minimization player's perspective. It is noteworthy that minimax trees are often implemented so that the best move is not always returned, especially if the next best moves have very similar evaluation values. After all, the domain-specific utility function is only a heuristic, and allowing slightly "worse" moves results in greater diversity and unpredictability of the game-playing program. This ensures that the opponent cannot play by simply learning from previous programs in order to identify the weaknesses in the program.

An example of a minimax tree is shown in Figure 3.5. The nodes at alternate levels of the tree are denoted by either circles or rectangles, depending on whether the node performs minimization or whether it performs minimization. The circular nodes perform maximization (player A), whereas the rectangular nodes perform minimization (player B). A leaf node might either be one at which the game terminates (e.g., a checkmate or a draw-by-repetition in chess), or it might be one at which the node is evaluated by the evaluation function (because the maximum depth of search has been reached). Each node in Figure 3.5 is annotated with a numerical value corresponding to the utility function. The numerical values at circular nodes are computed using the maximum of the values of their children, whereas the values at rectangular nodes are computed using the minimum of the values of their children. The utilities are computed using the domain-specific function at the leaf nodes, which might either be terminal nodes (e.g., checkmate in chess), or they might be nodes present at the maximum depth at which the minimax tree is computed. The computation of the domain-specific evaluation function, therefore, needs to be efficiently computable, as it regulates the computational complexity of the approach. The tree of Figure 3.5 is not perfectly balanced, because the terminal state might be reached early in

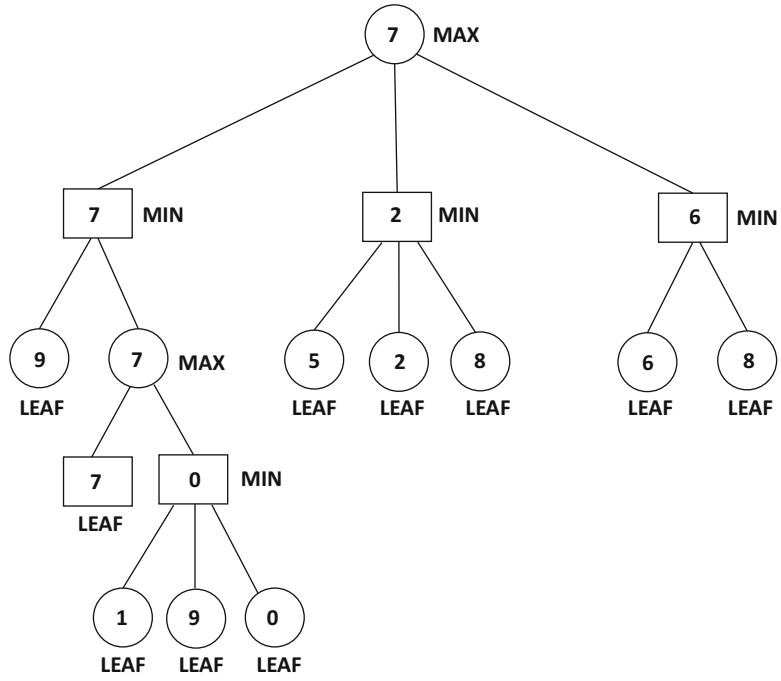


Figure 3.5: The minimax tree for adversarial search

many cases. For example, it is possible to arrive at a checkmate in chess after less than 10 moves or more than 100 moves. Therefore, such trees may be highly unbalanced in real settings. If the recursive algorithm of Figure 3.4 is used, the nodes will be visited in depth-first order in order to compute the values at various nodes. It is also possible to organize the algorithm of Figure 3.4 in order to use other strategies for exploring the nodes (such as breadth-first search). However, depth-first search is desirable because it is space efficient, and it enables a particular type of pruning strategy, which will be discussed later. For problems like chess, breadth-first search is too inefficient to be used at all.

3.3.3 Prestoring Subtrees

In many settings, certain portions of the tree are repeated over and over again. For example, in chess, the upper portion of the tree is fairly standard in terms of the portions of the tree that are known to be nearly optimal. There are a fairly large number of paths in the opening portion of the tree that result in nearly equal positions for both opponents. Each of these sequences is often named in chess by a distinctive name such as *Sicilian Defense*, *Queen's Indian*, and so on. This set of sequences is collectively referred to as *opening theory*, and they can be strung together in a tree structure that is a very sparsely selected subset of the (opening portion of the) minimax tree. Human grandmasters spend a significant amount of time studying opening theory, and also discovering novelties (i.e., small variations of known theory) that can provide them an advantage in an actual game because of the complexity of analyzing the long-term ramifications of a particular opening. It is generally hard for search trees (or other deductive methods) to discover such novelties easily because the effect of choosing a particular opening persist long into the game (beyond the depth to which a search typically works). At the same time, it is also hard for minimax trees to discover known theory (and the best opening moves) with the use of the search methods, as

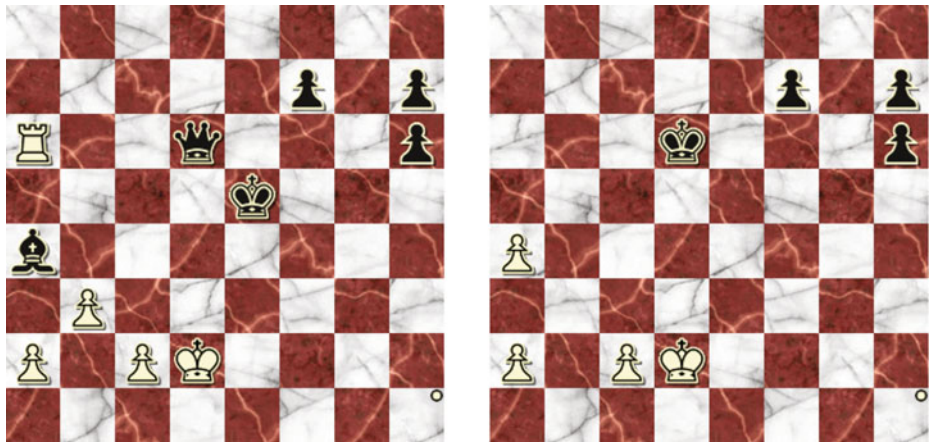
a minimax tree can make no distinction between known theory and a novelty. A key point is that human grandmasters have collectively spent hundreds of years in understanding the long-term effects of different openings, which is hard for search methods to fully match. As a result, this portion of the tree (referred to as the *opening book*) is pre-stored up front and used to make quick moves early on by selecting one of the branches in the opening book. Doing so significantly improves the strength of the resulting game-playing system. This is a classical example of an informal knowledge base, where the collective wisdom of hundreds of years of human play is pre-stored up front as a hypothesis for the system to use for better play. Indeed, minimax trees with opening books perform significantly better than those that do not use opening books.

At the lower levels of the tree, the board contains a smaller number of pieces for which outcomes are known with perfect play. For example, a queen versus rook end game (i.e., four pieces on the board including the kings) always results in a win for the player with the queen. However, it is not always a simple matter to play such endings perfectly, and even human grandmasters have been known to make mistakes[2] in the process, and reach suboptimal outcomes. When working with computers, the lower portions of such trees are tractable enough that the trees can be expanded to the leaves. In many cases, expanding the trees might require a few days of computational effort, which cannot be achieved during real-time game play. Therefore, the optimal moves for all positions with at most five pieces have been explicitly worked out computationally and have been pre-stored in *tablebases*. These tablebases are typically implemented as massive hash tables. Simply speaking, a hash table can be used to map from the position to the optimal move. A significant number of six-piece endgames have been pre-stored as well. These optimal moves have been worked out computationally (which can be viewed as a form of inductive learning). However, since they are pre-stored as tablebases (which are informal forms of knowledge bases), they are considered deductive methods as well, where the hypothesis (tablebase) is provided as an input to the system (no matter how it was actually derived). In some of these cases, an optimal endgame sequence might be longer than 100 moves, which is difficult even for the best grandmasters to figure out over the board during play. Therefore, these additional knowledge bases greatly improve the power of minimax trees.

3.3.4 Challenges in Designing Evaluation Functions

It is not always a simple matter to design domain-specific evaluation functions for game-centric or other settings, and some states are inherently resistant to evaluation without further expansion to lower levels. In order to understand the challenges associated with evaluation function design, we provide an example based on the inherent instability of trying to predict the value of a position in chess; some positions require a sequence of piece exchanges, and an evaluation of such positions in the middle of a dynamic exchange can lead to an evaluated value that reflects the true balance of power between the players rather poorly. A key point is that it is often hard to encode the complex dynamics between pieces with the use of a particular function. This is because each exchange leads to a wild swing in any (material-centric) domain-specific evaluation, as the pieces are rapidly removed from the board. Such volatile positions are not "quiet" and are therefore referred to as *non-quiescent*

[2] A great example of this situation corresponds to the round 5 playoffs during the 2001 FIDE World Chess Championship in Moscow. The grandmaster Peter Svidler reached a queen versus rook end game against Boris Gelfand, who refused to resign and effectively challenged his opponent to prove that he knew how to play the end game perfectly. Peter Svidler made mistakes in the process, and the game resulted in a draw.



(a) A non-quiescent position (white to move)

(b) A quiescent position after exchanges (black to move)

Figure 3.6: Black seems to have significantly more material in position (a). After a series of exchanges, the position is shown in (b), which is a draw with optimal play from both sides

in chess parlance. An example of a non-quiescent position is shown in Figure 3.6(a). In the position of Figure 3.6(a), it seems that black has significant material advantage, and also has the white king in check. However, after a series of forced exchanges, the position becomes quiet (as shown in Figure 3.6(b)), and it is theoretically equal between white and black (as a draw with optimal play from both sides). Therefore, a common approach to evaluating board positions is by first finding the "most likely" quiescent position after a short sequence of moves. Finding quiescent positions in chess is an important area of research in its own right, and it remains an imperfect science in spite of significant improvements over the years. This problem is not specific to chess and it arises in all types of game-playing settings. The algorithms used to arrive at quiescence are not perfect, and can often be fooled by a particular board position. This often has a devastating effect on the overall evaluation. A poor evaluation also results in mistakes during play, especially when a shallow minimax tree is used.

Beyond the more obvious issue of non-quiescent positions, challenges arise because of subtle aspects of evaluations that cannot be easily hand-crafted. For example, a human chess grandmaster evaluates a board position in terms of complex combinations of patterns and intuition gained from past experience. It is hard for evaluation functions to encode this type of knowledge. This is a problem with any deductive reasoning method that relies on interpretable hypotheses (e.g., numerical value of a pawn versus the numerical value of a bishop) to hand-craft knowledge into the system. Therefore, beyond overcoming these challenges, it is important to know which positions are harder to evaluate and than others so that those specific portions of the tree can be explored more deeply. Indeed, this is already done in most game-playing programs with various heuristics, which tends to make the trees somewhat unbalanced. Recent experiences seem to suggest that one can *learn* robust evaluations in a data-driven manner, rather than hand-crafting them. This is the task of *reinforcement learning* methods like *Monte Carlo search trees*, which will be discussed in a later section. However, these methods fall into the school of thought belonging to inductive learning, rather than deductive reasoning. This will be topic of discussion in Section 3.5.

3.3.5 Weaknesses of Minimax Trees

Minimax trees are typically too deep to be exhaustively evaluated down to the final outcome, which is why heuristic evaluations are used at leaf nodes. By using minimax search, one is effectively sharpening the evaluation function by looking ahead a few levels down the tree. For deep enough evaluations, a weak evaluation function can become extremely strong. For example, in the case of a chess-playing program, consider the use of the trivial evaluation function of simply setting the evaluation to either $+1$ or -1 when the game is in a checkmate position (depending on which player is winning), and 0, otherwise.

Chess is a game of finite length, and an upper-bound on the maximum length of a chess game is known to be 17,697 half-moves. In such a case, a minimax tree of depth 17,698 (counting moves by each player as an additional depth) will cause a chess-playing program with the aforementioned trivial evaluation to strengthen to perfect evaluations at the root node (via minimax computations). This is because a sequence of at most 17,698 moves will either lead to a checkmate or will lead to a draw based on the termination rules of chess. However, such an approach cannot be implemented in any computationally practical setting, as the number of nodes in such a tree will be larger than the number of atoms in the universe. The main problem is that it is impossible to use a very large depth of evaluation because of the explosion in the number of nodes that need to be evaluated. After all, the number of nodes increases by a factor of more than 10 for each additional level of the tree. Therefore, one needs to evaluate intermediate positions with heuristic evaluation functions. The depth of this evaluation is referred to as the *horizon* or *half-ply*[3] in the context of chess-playing programs. As a practical matter, the horizon can often be quite constrained because of the combinatorial explosion of the number of possible positions. In certain positions as the starting point of the search, a game of chess can have more than 100 billion possible outcome positions to evaluate after four moves from each player. In practice, most chess programs need to evaluate the positions much deeper in order to obtain moves of higher quality. With the trivial evaluation function discussed above, any minimax tree of computationally feasible depth will make random moves, as all leaf nodes will be usually non-terminal; therefore, the algorithm will make random moves in most positions. It is important to develop better heuristics for evaluating chess positions in order to reduce the depth of the tree at which high-quality moves are made. For example, if one had an oracle evaluation function to exactly predict the value of a position as a win, loss, or draw with optimal play, a tree of depth 1 would suffice for perfect play. The reality is that while evaluation functions have become increasingly sophisticated, there are severe limitations on how well they can be evaluated with domain-specific heuristics. Therefore, the minimax evaluation with increased depth is essential for high-quality chess play. Furthermore, executing very sophisticated evaluation functions can itself be computationally expensive, which can ultimately result in lower depth of evaluation in a time-constrained setting. In such cases, it is possible to lose the computational gains obtained from using sophisticated evaluation functions (because of reduced depth) to the additional cost of evaluating each function. Therefore, it often becomes a sensitive balancing act in designing a sophisticated evaluation function that can also be evaluated efficiently. The trade-off between spending time for better evaluation functions versus exploring the minimax tree more deeply continues to be a subject of considerable interest among practitioners and researchers in computer chess.

When the depth of the evaluation is limited, the weaknesses of imperfect evaluation functions become more apparent. Over the years, the depth of evaluation has improved in

[3] A ply corresponds to one move by each player. A half-ply corresponds to a single move by one of the two players.

game-playing systems (because of sophisticated hardware), and the evaluation function at the leaf nodes has also improved as programmers have learned novel ways of encoding domain knowledge into chess-playing software. The combination of the two adds significantly to the playing strength of the program. As a specific example, *Deep Blue* used specialized hardware together with carefully designed evaluation functions to defeat the world champion, Garry Kasparov, in a match of six games in 1997. The hardware was highly specialized, and the chips were specifically designed for chess-specific evaluation. *Deep Blue* had the capability of evaluating 200 million positions per second using specialized computer chips, which was very impressive for the computer hardware available at the time. This was the first time that such a powerful machine was combined with a sophisticated evaluation function in order to create a machine playing chess at the highest level. However, Kasparov did win one game and draw three games over the course of the six-game match. In the modern era, it would be unlikely[4] for a world champion to hope to draw even one game out of six games with an off-the-shelf chess software like *Stockfish* running on a commodity laptop or mobile phone. Simply speaking, artificial intelligence has far leapfrogged humans, as far as chess is concerned. A large part of this success stems from the ability to implement increasingly sophisticated evaluation functions and search mechanisms in an efficient way. Improvements in the state-of-the-art of computer hardware have also helped significantly in being able to construct and explore minimax trees in an efficient way. There have also been many advancements in designing strategies for pruning fruitless branches of minimax trees in order to reduce the computational burden of exploration.

Chess has been a success story for minimax trees, because of the large amount of interpretable domain knowledge available in order to perform utility function evaluations of high quality from given positions on the board. In spite of this fact, many weaknesses in modern chess programs do exist; these weaknesses become particularly apparent during computer-to-computer play resulting in chess positions where the evaluations require a greater level of intuitive pattern recognition. An imperfect heuristic evaluation can turn out to be an even greater problem in board positions where the tree has a large degree, and therefore one cannot create a very deep tree. Therefore, it is important to develop a general strategy to prune branches that are not very promising from each player's perspective in order to be able to evaluate the positions more deeply. One naïve way of doing this is to prune branches for which the immediate evaluation is extremely poor from the perspective of the agent that is making the next move. However, doing so can sometimes inadvertently prune relevant subtrees because more promising moves may be hidden lower down the tree. On the other hand, certain types of pruning are guaranteed to not lose relevant subtrees. This approach uses some special properties of minimax trees, and it is referred to as alpha-beta pruning.

3.4 Alpha-Beta Pruning

The main goal of alpha-beta pruning is to rule out irrelevant branches of the tree, so that the search process becomes more efficient. We present alpha-beta pruning with the use of a utility function $U(\cdot)$ which maximizes and minimizes at alternate levels of the tree. Therefore, all notations in this section are the same as those used in Section 3.3.2. The modified algorithm is illustrated in Figure 3.7. The alpha-beta method is very similar to

[4]The odds can be computed using the Elo rating system for both chess players and computers. As of the writing of this book, the Elo rating of the best chess program exceeded that of the current world champion by about 530 points. This difference translates to ten wins by the computer for every *draw* by the world champion.

Algorithm *AlphaBetaMaxPlayer*(Initial State: s, Depth: d, α, β)
begin
  $i = s$;
  if (($d = 0$) or (s is termination leaf)) **then return** s;
  for $j \in A(i)$ **do**
  begin
    $OptState_b(j) = AlphaBetaMinPlayer(j, d-1, \alpha, \beta)$;
    if $(U(OptState_b(j)) > \alpha)$ **then** $\alpha = U(OptState_b(j))$; $beststate_a = OptState_b(j)$;
    if $(\alpha > \beta)$ **then return** $beststate_a$; **{ Alpha-Beta Pruning }**
  end;
  return $beststate_a$;
end

Algorithm *AlphaBetaMinPlayer*(Initial State: s, Depth: d, α, β)
begin
  $i = s$;
  if (($d = 0$) or (s is termination leaf)) **then return** s;
  for $j \in A(i)$ **do**
  begin
    $OptState_a(j) = AlphaBetaMaxPlayer(j, d-1, \alpha, \beta)$;
    if $(U(OptState_a(j)) < \beta)$ **then** $\beta = U(OptState_a(j))$; $beststate_b = OptState_a(j)$;
    if $(\alpha > \beta)$ **then return** $beststate_b$; **{ Alpha-Beta Pruning }**
  end;
  return $beststate_b$;
end

Figure 3.7: The alpha-beta search algorithm

the minimax approach discussed in the previous section, except that it adds a *single line of code* that prunes out irrelevant branches, when it is understood that the current branch cannot find a better action at a node present at a higher level of the tree from the perspective of the corresponding player.

In order to understand alpha-beta pruning, we will provide an example of how some branches can be pruned in particular situations. Consider the example shown in Figure 3.5 in which maximization is performed at the top level of the tree, and minimization is performed at the next level. After processing the entirety of the first subtree hanging from the root of the tree, a value of 7 is returned to the root (which is the current maximization value from the perspective of agent A at the root of the tree). When the second subtree at the root is being processed, the minimization agent B at the next level encounters the first leaf, which has a value of 5. While processing further leaves can lead to an even lower value from the perspective of the minimization agent B, it knows that the maximization agent A already has a value of 7 available from the first subtree (and would never choose an action with a lower utility). Therefore, processing the entire subtree further is fruitless from the perspective of agent B, and further branches (corresponding to the leaf nodes containing values 2 and 8) can be pruned. The relevant portion of the tree from Figure 3.5 is shown in Figure 3.8.

Next, we describe the alpha-beta search algorithm more formally. The reason that the approach is historically referred to as alpha-beta search is because the best evaluations from the perspective of the two adversarial agents are often denoted by α and β, respectively. The overall algorithm is shown in Figure 3.7, and it is very similar to the algorithm in

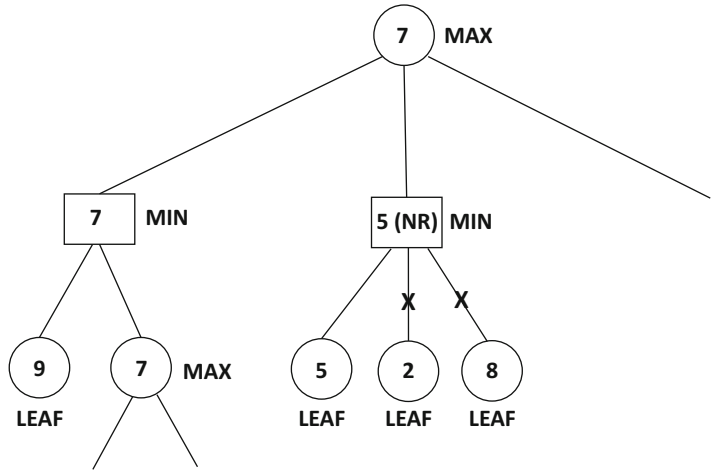


Figure 3.8: An example of how fruitless branches can be pruned

Figure 3.4 The parameters α and β serve the same role as the variables max_a and min_b in Figure 3.4, except that they are no longer initialized from scratch in each call of the mutually recursive pseudocodes. Rather α is a *running estimate* of best evaluation from the perspective of agent A over the course of the entire algorithm, and β is a running estimate of the best evaluation from the perspective of agent B over the course of the algorithm. Therefore, these parameters are passed down the recursion, as the parameters are updated. As a result, one works with tighter estimates of the optimal states from the perspective of each agent, given that more information is passed down the mutually recursive calls. The value of α is set to $-\infty$ at the base call of the algorithm, whereas the value of β is set to ∞. Throughout the course of the algorithm, we will always have $\alpha \leq \beta$ at a node or else the children of that node no longer need to be explored. The best way to understand α and β is that they are both pessimistic bounds on the evaluations from the perspectives of the maximizing and minimizing players, respectively. In an optimal minimax setting the optimal move that agent A makes is the same as the optimal move that agent B expects A to make when the same utility function is used by both. Therefore, at the end of the process, one must have $\alpha = \beta$ at the root of the tree. However, for pessimistic bounds one must always have $\alpha \leq \beta$. Therefore, whenever we have $\beta < \alpha$ at a node of the tree, the action sequence leading to such a node will never be selected by at least one of the two players and can be pruned. The overall algorithm in Figure 3.7 is very similar to the approach used in Figure 3.4, except for the additional parameters α and β, which provide the pessimistic estimates for the two players. Furthermore, a single line of code in each of the pseudocodes for the two agents performs the alpha-beta pruning, which is not present in the pseudocodes for the basic minimax algorithm. One consequence of carrying forward α and β as a recursive parameter is that the algorithm is able to use bounds established in distant regions of the tree for pruning.

An example of alpha-beta pruning is shown in Figure 3.9, and the evaluation of each node is also shown. The nodes that are not relevant for computation of the optimal state are marked as '(NR)' and the evaluations within these nodes are not exact. The acronym '(NR)' stands for "(Not Relevant)". The reason for the non-relevance of this node is that one or more of the descendent subtrees rooted at this node have already been explored, and

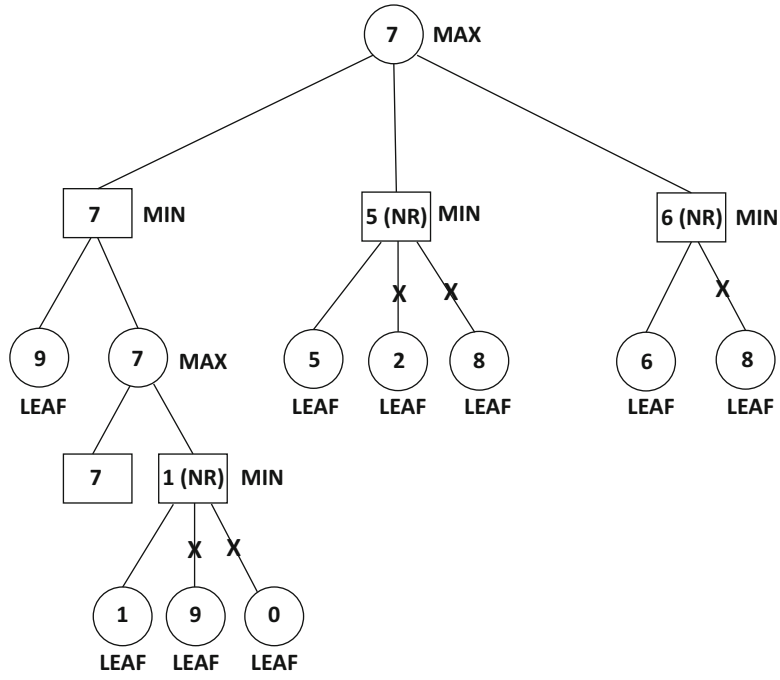


Figure 3.9: The minimax tree for adversarial search

the optimal state rooted at that node is deemed to be unacceptable to the opponent (based on other options available to the opponent higher up the tree). Therefore, other subtrees that descend from this tree are not explored further (as the entire subtree rooted at that node is deemed to be irrelevant). Therefore, the value within the node marked by '(NR)' is based on explored branches, and is a pessimistic bound. Correspondingly, many of the branches rooted at these nodes are marked by 'X', and they are not explored further. Even though only a small number of nodes are pruned in Figure 3.9, it is possible to prune entire subtrees using this approach in many cases.

In all of the examples we have shown so far, the pruning is done based on bounds developed in alternate levels of the tree. However, since α and β are carried down to lower levels of the tree, it is also possible for pruning to occur at a lower level of the tree because of bounds that were developed higher up in the tree.

3.4.1 Importance of Branch Evaluation Order

An important point is that the effectiveness of the pruning depends heavily on the order in which the branches of the minimax tree are explored. It makes a lot more sense to explore promising branches of the tree first from each agent's perspective, because it leads to sharper evaluations earlier on in the process. The notion of "promising" branches is evaluated in a myopic fashion by computing the utility of a state immediately after taking a single action from the perspective of the agent taking that action. The state with the best utility from the perspective of the agent making the transition is explored first. This is achieved by applying the domain-specific utility function (e.g., evaluation of chess position) after making one transition (e.g., move in chess) and then picking the best one among all these

possibilities. More sophisticated methods for branch ordering might use multiple transitions for a shallow lookahead in evaluation. By exploring branches in this order, the agent is more likely to obtain a more favorable evaluation at deeper levels of the tree early on the process of depth-first search. This leads to a tighter pessimistic bound for each agent earlier in the process, and therefore a larger fraction of redundant branches is pruned.

3.5 Monte Carlo Tree Search: The Inductive View

Although variations of Monte Carlo tree search can be used for any decision problem, it has historically been used only for adversarial settings. Therefore, the discussion in this section is based on the adversarial setting. The main problem with the minimax approach of the previous section is that it is based on domain-specific evaluation functions, which might sometimes not be available in more complex games. The weaknesses of domain-specific evaluation functions are particularly notable in games where the evaluation of a position is not easily interpretable and depends on a large level of human intuition. This is particularly evident in games like Go, which depend on a high level of spatial pattern recognition. Even in games like chess, where domain-specific evaluation functions are available, they tend to have significant weaknesses. As a result, the tree of possibilities often needs to be evaluated to significant depth (with some level of heuristic pruning) in order to ensure that good solutions are not missed. This can be very expensive in chess, and turns out to be intractable in games like Go. Even in chess, computers tend to be weak in certain types of positions that require a high level of intuition to understand. Indeed, humans were able to consistently defeat minimax trees in chess during the early years by using these weaknesses of minimax trees. The difference in style of play by minimax trees from humans is very apparent in terms of lacking creativity in game play. Creativity is a hallmark of learning systems that can generalize past experiences to new situations in a novel way.

Monte Carlo tree search uses the inductive perspective of *learning* from past explorations down the tree structure in order to learn promising paths over time. Furthermore, Monte Carlo tree search does not require an explicit evaluation function, which is consistent with the fact that it is a learning approach rather than a knowledge-based approach. This type of statistical approach requires fewer evaluations from an empirical perspective although it is not guaranteed to provide an optimal solution from the minimax perspective. Note that the minimax tree is designed to provide an optimal solution from each opponent's perspective, if *one were to construct a tree down to termination nodes*. However, the meaningfulness of such an "optimal" solution is questionable when one has to stop at a particular depth of the tree in order to evaluate a questionable utility function. It might sometimes make more sense to explore fewer branches all the way down to goal states, and then empirically select the most promising branches.

Monte Carlo tree search is based on the principles of reinforcement learning, which are discussed in detail in Chapter 10. We will first discuss a basic version of Monte Carlo tree search, as it was proposed during the early years. This version is referred to as the *expected outcome model*, and it captures the important principles of the approach in spite of some obvious limitations; modern versions of Monte Carlo tree search are based on these principles, and they use various modifications to address these limitations. In a later section, we will discuss how modern versions of Monte Carlo tree search have improved over this model.

Monte Carlo tree search derives its inspiration from the fact that making the best move in a given position will often result in improved winning chances over multiple paths of the

tree. Therefore, the basic approach expands sampled paths *to the very end* until the game terminates. By sampling the rollout to the very end, one is able to avoid the weaknesses associated with the domain-specific evaluation function. As we will see later, the methodology of sampling uses some knowledge of past experiences in order to narrow down the number of branches that need to be explored.

At the root of the tree, one considers all possible states reachable by a single transition and then selects a particular node with the use of Monte Carlo sampling. At this newly selected node, all possible actions are sampled from the perspective of the opponent and one of these nodes is randomly selected for further expansion. Therefore, this process is continually repeated until one reaches a state that satisfies the goal condition. At this point, the sampling process terminates. Note that the repeated process of node sampling will result in a single path, along with all the immediate children of the nodes on the path. The sampled path is, therefore, much deeper than any of the paths considered by a relatively balanced minimax model. This process of repeated sampling up to termination of the game tree is referred to as a *rollout*. The process of sampling paths is repeated over the course of the algorithm in order to create multiple rollouts. The multiple rollouts provide the empirical data needed to make choices at the top level of the tree. After a sufficient number of paths have been sampled, the algorithm terminates.

After the algorithm terminates, the win-loss-draw statistics are *backpropagated* along each branch right up to root. For each branch, the number of times that it is played is stored, along with the number of wins, draws, and losses. For a given branch b, which is traversed N_b times, let the number of wins and losses be W_b and L_b, respectively. Then, one possible heuristic evaluation E_b for the branch b is as follows:

$$E_b = \frac{W_b - L_b}{N_b}$$

This evaluation favors branches that have a favorable win-to-loss ratio. The value will always range between -1 and $+1$, with larger values being reached by branches that have higher win to loss ratios. At the end of the Monte Carlo simulation, the evaluation of all branches at the root of the tree is used in order to recommend a move. In particular, the branch with the largest value of E_b. This approach is essentially the most primitive form of Monte Carlo tree search, although significant modifications are required in order to make it work well. In particular, we will see that the branches sampled later on in the process are not independent from the ones that were sampled earlier on. This process of continuously learning from past experience is referred to as reinforcement learning.

It is noteworthy that Monte Carlo tree search successively builds upon the tree that it has already expanded. Therefore, it might contain multiple branches corresponding to multiple rollouts. The resulting tree is therefore bushy and unbalanced. The number of leaf nodes (with a final outcome) in the resulting Monte Carlo tree is therefore bounded above by the number of rollouts (as some rollouts might be occasionally[5] replayed). Examples of Monte Carlo trees after successive rollouts are shown in Figure 3.10.

The success of Monte Carlo tree search heavily depends on the fact that the different evaluations from a particular branch of the tree are correlated with one another. Therefore, the statistical win-loss ratio from a particular branch of the tree often leads to accurate evaluations at the top level, even though a particular rollout might not be remotely optimal from the perspective of either opponent. This means that one is often able to estimate the

[5]For many complex games with long sequences of moves, it is highly unlikely to replay the same rollout. Repeats are more likely in simple games like tic-tac-toe.

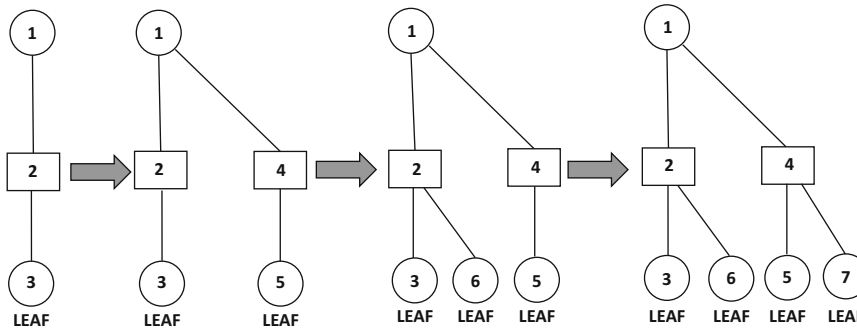


Figure 3.10: Results of Monte Carlo rollouts in the expected outcome model

value of a particular move by statistical sampling. This is different from a minimax tree, which tries to find optimal paths from the perspective of each opponent. It has been shown (under certain assumptions) that the prediction of the basic version of Monte Carlo tree search converges to that of a minimax tree *constructed all the way down to the termination nodes*. However, the number of samples required is too large for the Monte Carlo approach to even remotely approach convergence, and minimax trees cannot be (practically) constructed down to termination nodes anyway. In practice, the effects of an empirical approach are quite different from the domain-specific strategy of minimax trees.

One advantage of inductive learning methods like Monte Carlo tree search is that they do not suffer from the hardcoded weaknesses of a domain-specific evaluation function, as in the case of a minimax tree. Monte Carlo tree search is particularly suitable for games with very large branching factors (like Go) and also in the case of games that do not have well-developed theory in order to perform domain-specific evaluations. Compared to chess, Go tends to require a greater level of intuition from human players, and it is often hard to perform explicit calculation of the quality of a specific position. In such cases, simulating the game up to the very end seems like a reasonable solution, given that it is not realistically possible either to construct the tree up to a large depth (as required by minimax) or to reasonably evaluate an intermediate node in an accurate way.

While Monte Carlo tree search theoretically evaluates positions to the very end, it can lead to infinite (or very long) loops in certain types of games. Therefore, a practical limit is often imposed on the length of the tree path in order to avoid impractical situations. Such nodes can be ignored in the evaluation process.

As discussed above, convergence to a minimax tree requires a very large number of rollouts. To address this problem, many variants of Monte Carlo tree search do not perform the rollouts completely randomly. Rather, a certain level of bias may be used in choosing branches wither by using a domain-specific evaluation function, or based on the success of previous rollouts. The goal of incorporating this type of bias is to speed up convergence of the evaluation of branches; the trade-off is that the final solution may be suboptimal, and a better solution may be obtained with unbiased sampling in cases where it is possible to perform a larger number of simulations. Therefore, a trade-off between *exploration* and *exploitation* is natural. The next section discusses some of these enhancements, which defines how such trees are used in practice.

3.5.1 Enhancements to the Expected Outcome Model

The previous section describes the expected outcome model, which was the progenitor of Monte Carlo tree search, but was not actually referred to by that name. There are several problems with the expected outcome model, the most important of which is the fact that it takes too long to converge to the results obtained from a minimax tree. As a result, this basic version of the model performs rather poorly. Modern variants of Monte Carlo tree search rely on the experience of past explorations in order to grow the tree in promising directions. These promising directions are learned by identifying better win to loss ratios in the past. The precise degree of biasing is a key trade-off in the process of construction of the Monte Carlo tree, and it is based on the idea on reinforcement learning, where one *reinforces* past experiences during the learning process. Modern versions of Monte Carlo tree search improve the power of the approach by making several modifications to the original version of the expected outcome model.

The vanilla version of Monte Carlo tree search (i.e., the expected outcome model) explores branches completely randomly. The main challenge in doing so is that only a few paths might be promising, whereas most branches might be disastrously suboptimal. The drawback of this situation is that purely random search might have some of the properties of searching for a needle in a haystack. In reality, some branches are likely to be more favorable than others based on the win-loss ratio of earlier traversals down that branch. Therefore, a level of determinism in exploration is incorporated at the upper levels of the tree. However, by exploiting only the win-loss ratio of previous traversals (after performing a small number of them), one would repeatedly favor specific branches (which might be suboptimal) and fail to explore new branches that might eventually lead to better choices. Therefore, a trade-off between exploration and exploitation is used, which is based on ideas drawn from the principle of *multi-armed bandits*. In multi-armed bandits, a gambler repeatedly tries to discover the best of two slot machines to play on by trying them repeatedly (assuming that the expected payoffs from the two machines are not the same). Playing the slot machines alternately to learn their win-loss ratio is known to be wasteful, especially if one of the machines yields repeated rewards early on and the other yields no rewards. Therefore, the multi-armed bandit approach works by regulating the trade-off between exploration (by stochastically trying different slot machines) and exploitation (by stochastically favoring the one that has performed better so far).

The multiarmed bandits method uses a variety of strategies to learn the best slot machine over time, one of which is the *upper-bounding method*. This method is described in detail in Chapter 10, and it explores branches based on their "optimistic" potential. In other words, the gambler compares the most optimistic statistical outcomes from each slot machine, and selects the slot machine with the highest reward in the optimistic sense. Note that the gambler would be likely to favor machines that are played less often, since there is greater variability in their predicted statistical performance — greater variability is always favored by optimists. At the same time, machines with superior historical outcomes will also be favored, because past performance does add to the optimistic estimate of performance. In the context of tree search, this method is also referred to as the *UCT algorithm*, and it stands for *upper-bounding algorithm applied to trees*. The basic idea is that branches are evaluated as a sum of a quantity based on the earlier win-loss experience from that branch (and a bonus based on how frequently that branch has been explored earlier). Infrequently explored

branches receive a higher bonus in order to encourage greater exploration. For example, a possible evaluation of node i (corresponding to a particular branch) is as follows:

$$u_i = \underbrace{\frac{w_i}{n_i}}_{\text{Exploit}} + \underbrace{c\frac{\sqrt{N_i}}{n_i}}_{\text{Explore}} \tag{3.1}$$

Here, N_i is the number of times the parent of node i is played, w_i is the number of wins starting from node i out of the n_i times that it is played. The quantity u_i is referred to as the "upper bound" on the evaluation quality. The reason for referring to it as an upper bound, is that we are adding an exploration bonus to the win record of a particular branch, and this exploration bonus favors infrequently visited branches. The key point here is that the portion w_i/n_i is a direct reward for good performance, whereas the second part of the expression is proportional to $1/n_i$, which favors infrequently explored branches. In other words, the first part of the expression favors exploitation, whereas the second part of the expression favors exploration. The parameter c is a balancing parameter. It is also important to note that the value of u_i is evaluated from the perspective of the player making the move, and the value of w_i/n_i is also computed from the perspective of that player (which is different at alternate levels of the tree).

The presence of $\sqrt{N_i}$ in the numerator (in contrast to the value of n_i in the denominator) ensures that the exploration component reduces in relative value, if both n_i and N_i grow at the same rate. In general, the denominator always needs to grow faster than the numerator for this to occur. In the extreme case, when the value of n_i is 0 and N_i is positive, the value of $\sqrt{N_i}/n_i$ is set to the value of ∞ (which is the right-hand side limit of the expression as the (positive value) n_i goes to 0^+. Therefore, a branch that has never been selected is always prioritized for exploration over those that have been selected at least once.

Another possible example of the evaluation of u_i is as follows:

$$u_i = \frac{w_i}{n_i} + c\frac{\sqrt{\ln(N_i)}}{\sqrt{n_i}}$$

This expression also satisfies the property that the numerator of the exploration component grows slower with increasing N_i than does the denominator with increasing n_i.

A tree starts off with only the root node, and increases in size over multiple iterations. Branches with the largest upper bound are *deterministically* constructed until a new node in the tree is found that was never explored earlier. This new node is created and is referred to as a *leaf node* from the perspective of the current tree. It is here that the probabilistic evaluation of the leaf node starts (which is backpropagated to each branch in the tree statistics). Note that the overall tree constructed will still have a random component to it, since the evaluations at leaf nodes are performed using Monte Carlo simulations. The value of the upper bound in the next iteration does depend on the results of these simulations, which will affect the structure of the tree. Although Monte Carlo rollouts are used for evaluation of leaf nodes, the path taken by the rollouts is not added to the Monte Carlo tree.

The above description of the tree construction process suggests that a distinction is made between the deterministic approach to each iteration of the tree construction (using the above exploration-exploitation trade-off) and the evaluation at leaf nodes using Monte Carlo rollouts. In other words, like minimax trees, the leaf nodes might be intermediate positions in the specific adversarial setting (e.g., chess game) rather than terminal positions

(which correspond to wins, losses or draws). However, there are several important difference from minimax trees in terms of how the structure of these trees is arrived at and how Monte Carlo rollouts are used from these leaf nodes in order to evaluate the position down to terminal nodes. The use of Monte Carlo rollouts for evaluations ensures that inductive learning is prioritized over domain knowledge. One typically performs multiple Monte Carlo rollouts from the same leaf node for greater robustness in evaluation. This type of approach for evaluating leaf nodes uses the general principle that such rollouts can often lead to robust evaluations in the expected sense, even though the rollouts will obviously make suboptimal moves.

There are also significant differences in terms of the nature of the evaluation with respect to the expected outcome model. Unlike the case of the expected outcome model, one does not use uniform probabilities over all valid moves in a given position in order to perform the Monte Carlo rollouts. Rather, the moves are typically predicted using a machine learning algorithm that is trained to predict the probability of the best move, given the current state. The machine learning algorithm can be trained on a database of chess positions together with the evaluations of various moves as eventually resulting in a win, loss, or draw. The goodness evaluation of the machine learning algorithm is used to sample a move at each step of the Monte Carlo rollout.

Machine learning algorithms can be used not only to bias the Monte Carlo rollouts (which are performed at lower levels and do not add to the Monte Carlo tree), but they can also be used to influence the best branch to explore during the upper levels of the deterministic tree construction process. For example, the method in [168] uses a heuristic upper bound quantification of each branch, which is enhanced with a machine learning method. Specifically, a machine learning method predicts the probability of each action in a particular state (chess position). This probability is learned using a training database of the performance in similar positions in the past. Higher probabilities indicate more desirable branches. The exploration bonus for each state is multiplied with this probability p_i in order to yield a modified value of u_i as follows:

$$u_i = \frac{w_i}{n_i} + c \cdot p_i \frac{\sqrt{N_i}}{n_i}$$

This relationship is the same as Equation 3.1, except that the value of p_i is used in order to weight the exploration component. The value of p_i is computed using a machine learning algorithm. As a result, desirable branches tend to be selected more often while creating the Monte Carlo tree. This general principle is referred to as that of adding *progressive bias* [36].

Many of these enhancements were used in *AlphaZero* [168], which is a general-purpose reinforcement learning method for Go, chess, and shogi. The resulting program (and other programs based on similar principles) have been shown to outperform chess programs that are based purely on minimax search with domain-specific heuristics. An important point about inductive methods is that they are often able to implicitly learn subtle ways of evaluating states *from experience*, which goes beyond the natural horizon-centric limitations of domain-specific methods. By using Monte Carlo rollouts, they explore fewer paths, but the paths are deeper and are explored in an expected sense. In order to understand why such an approach works better, it is important to note that a poor move in chess will often lead to losses over many possible subsequent continuations of the go, and one does not really need the optimal minimax path to evaluate the position. In other words, the poor nature of the move will show up in the win-loss ratios of the Monte Carlo rollouts.

Past experience with the use of such methods in chess has been that the style of play is extremely creative. In particular, *AlphaZero* was also able to recognize subtle positional

(a) Minimax recommends moving threatened knight (white to move)

(b) *AlphaZero* moves rook (black to move)

Figure 3.11: A deep knight sacrifice by *AlphaZero* (white) in its game against *Stockfish* (black)

factors during play. As a specific example, a chess position from the *AlphaZero-Stockfish* game [168] is shown in Figure 3.11, where the Monte Carlo program *AlphaZero* is playing white and the minimax program *Stockfish* is playing black. This position is an opening variation of the Queen's Indian defense, and white's knight is under threat. All minimax programs at the time recommended moving the knight back to safety. Instead, *AlphaZero* moved its rook (see Figure 3.11(b)), which allowed the capture of the knight. However, a strategic effect of this choice was that it caused black's position to stay underdeveloped over the longer term with many of its pieces in their starting positions. This fact was eventually exploited by *AlphaZero* to win the game over the longer term. These types of strategic choices are often harder for minimax programs to make, because of the natural horizon effects associated with minimax computation. On the other hand, inductive methods learn form experience as they explore the Monte Carlo tree deeper than a typical minimax program, and also learn important patterns on the chess board from past games. These patterns are learned by combining the rollouts with deep learning methods. Although *AlphaZero* might not always have seen the same position in the past, it might have encountered similar situations (in terms of board patterns), and favorable outcomes gave it the knowledge that underdeveloped positions in the opponent were sufficiently beneficial for it in the longer term. This information was indirectly encoded in the parameters of its neural network. This is exactly how humans learn from past experience, where the outcomes from past games change the patterns stored in the neural network of our brain. Some of these methods will be discussed in Chapter 10.

AlphaZero was able to discover all of the opening theory of chess on its own, unlike minimax programs that require the known opening tree of moves (based on past human play) to be hard-coded as an "opening book." This is a natural difference between a deductive reasoning method (which often leverages domain knowledge from human experience) and an inductive learning method (which is based on data-driven learning from machine experience). In some cases, *AlphaZero* discovered opening novelties or resurrected much-neglected lines of play in opening theory. This led to changes in the approach used by human players.

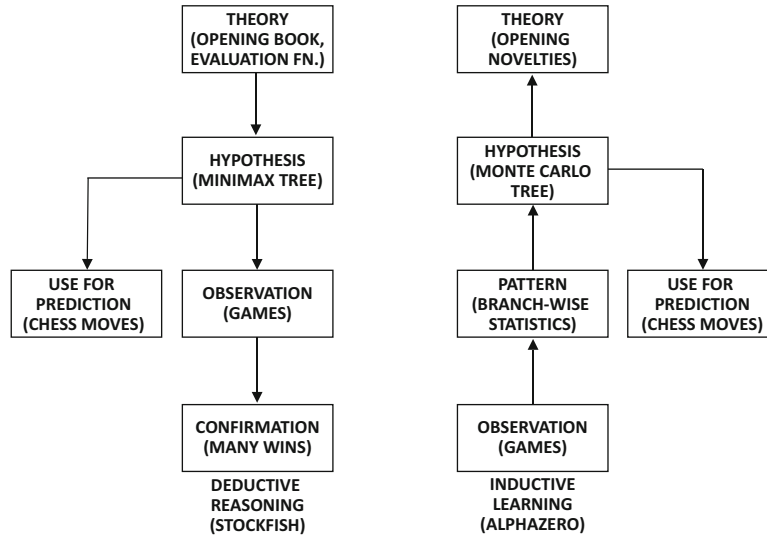


Figure 3.12: Revisiting Figure 1.1: The two schools of thought in artificial intelligence as applied to chess

The reason is that repeated Monte Carlo rollouts from specific opening positions encounter win-loss ratios that are very informative to the learning process. This is also how humans learn from playing many chess games. A similar observation was made for backgammon, where a reinforcement learning program by Tesauro [186] led to changes in the style of play by human players. The ability to go beyond human domain knowledge is usually achieved by inductive learning methods that are unfettered from the imperfections in the domain knowledge of deductive methods.

It is also noteworthy that even though Monte Carlo methods were tried well before the *AlphaZero* method by chess programs such as *KnightCap* [17], they were never successful in outperforming chess programs based on the minimax method. It was only in 2016 that *AlphaZero* was able to achieve a victory over one of the best minimax programs at the time (*Stockfish*). An important point is that inductive learning methods require a *lot of data and computational power*. While one can generate an unlimited amount of data in the chess domain by computer self-play, limitations in computational power can stand in the way of training such programs effectively. Because of this limitation, inductive methods were not trained or optimized (before 2016) at the scale required to outperform domain-specific minimax trees (which were already quire powerful in their own right). After all, the human world champion, Garry Kasparov, had already been defeated by a domain-specific minimax program (*Deep Blue*) as far back as 1997. *AlphaZero* also increased the power of Monte Carlo tree search methods by using deep learning methods to predict the best moves in each position. Currently, the best human players are typically unable to defeat either minimax chess programs or inductive Monte Carlo methods, even after handicapping the chess program with two removed pawns at the starting position. Among the two classes of methods, it is only recently that Monte Carlo methods have overtaken minimax trees.

3.5.2 Deductive Versus Inductive: Minimax and Monte Carlo Trees

Minimax trees create a hypothesis with the use of a domain-specific evaluation function (and an opening book for the initial moves) and then predict using this hypothesis. This is a deductive reasoning method because it incorporates the hypothesis of the expert chess player into the evaluation function and the opening book. While this hypothesis can be fallible, its quality can be confirmed by playing the chess program multiple times and registering many wins. While it is sometimes thought that deductive reasoning methods only deal with absolute truths, these truths hold in the context of the knowledge initially fed into the system. If the knowledge base is imperfect (such as a bad opening book or evaluation function), it will also show up in the quality of the chess playing. The model created with a minimax tree is the hypothesis, which is used to predict moves at each position. The mapping of different elements of a minimax chess program to the different steps in deductive reasoning are shown on the left-hand side of Figure 3.12.

On the other hand, Monte Carlo tree search is an inductive learning method. Monte Carlo tree search makes predictions by generating *examples* (rollouts) and then makes a prediction based on the experience of this algorithm with these examples. Modern variations of Monte Carlo tree search further combine tree search with reinforcement learning and deep learning in order to improve predictions. Deep learning is used in order to learn the evaluation function instead of using domain knowledge. For example, *AlphaZero* uses Monte Carlo tree search for rollouts, while also using deep learning to evaluate the quality of the moves and guide the search. The Monte Carlo tree together with the deep learning models form the hypothesis used by the program in order to move moves. Note that these hypotheses are created using the *statistical patterns* of empirical behavior in games. *AlphaZero* was given zero opening knowledge of chess theory, and *it discovered chess theory on its own* in a few hours of training. In many cases, it discovered exciting opening novelties that had not been known to human grandmasters. Clearly, such capabilities are outside the ambit of deductive reasoning methods, which have a ceiling to their performance based on what is already known (or considered "correct").

In general, Monte Carlo trees tend to explore a few promising branches deeper based on evaluations from previous experience, whereas minimax trees explore all unpruned branches in a roughly similar way. The human approach to chess is similar to the former, wherein humans evaluate a small number of promising directions of play rather than exhaustively considering all possibilities. The result is that the style of chess from Monte Carlo tree search is more similar to humans than that from minimax trees. The programs resulting from Monte Carlo trees can often take more risks in game playing, if past experience has shown that such risks are warranted over the longer term. On the other hand, minimax trees tend to discourage any risks beyond the horizon of tree exploration, especially since the evaluations at leaf levels are imperfect.

The traditional versions of chess-playing programs, such as *DeepBlue* and *Stockfish* were based on the minimax approach. Historically, it was always believed that chess-playing programs (which are dependent on a high-level of domain-specific knowledge found by earlier experience of humans) would always do better than an inductive approach (which starts from scratch with no domain knowledge). After all, evaluation functions use all sorts of chess-specific heuristics that depend on the past experience of humans with various types of chess positions. This trend indeed seemed to be true until the end of 2016, when *AlphaZero* outperformed[6] *Stockfish* quite significantly. This reversal was caused by increased compu-

[6]The original match was conducted on the basis of game conditions that were heavily criticized, since *Stockfish* was not provided an opening book. However, subsequent programs based on the same principles

tational power, which almost always improves inductive learning methods far more than deductive reasoning methods. Indeed, many of the exciting results in artificial intelligence in the previous decade have started to show that the classical school of thought in artificial intelligence (i.e., deductive reasoning) had serious limitations that were often overlooked in the early years (at the expense of inductive learning methods).

3.5.3 Application to Non-deterministic and Partially Observable Games

The Monte Carlo tree search method is naturally suited to non-deterministic settings such as card games or backgammon. Minimax trees are not well suited to non-deterministic settings because of the inability to predict the opponent's moves while building the tree. On the other hand, Monte Carlo tree search is naturally suited to handling such settings, since the desirability of moves is always evaluated in an *expected* sense. The randomness in the game can be naturally combined with the randomness in move sampling in order to learn the expected outcomes from each choice of move. On the other hand, a minimax tree would require one to follow each outcome of an uncertain state resulting from an agent action in order to perform the evaluation; this is because the same agent action (e.g., a dice throw) might result in different outcomes over different samples. In other words, there is no single action that yields a guaranteed outcome for a particular choice of action, and it is impossible to create a minimax tree whose branches can be deterministically selected by the two opponents.

A similar observation holds true for card games, where pulling a card out of a pack might lead to different outcomes, depending on the random order of the cards in the pack. Similarly, playing a particular card might have different outcomes depending on the unknown state of the cards in the opponent's hand. This is a partially observable setting. As a result, a minimax tree can never provide a guaranteed outcome of any particular choice of playing cards, because of the uncertainty in the state of the game, as observed by the agent. Because of this inability to fully control outcomes using the voluntary choices of the two opponents, there is simply no way of picking the best branch at a given state, unless one works with expected outcomes. This type of setting is naturally addressed with Monte Carlo methods, where repeated replay yields a score at each branch and the best scoring branch is always selected. After all, the Monte Carlo method provides an *empirically optimal choice* rather than a choice with *optimality guarantees*, which works well in uncertain settings. Furthermore, one does not gain much from the guaranteed optimality of minimax trees, because anomalous imperfections in the domain-specific evaluation function often show up in particular leaf nodes, and can drastically affect the move choice. This is sometimes less likely when one is using a statistical approach of picking a branch that leads to more frequent wins.

3.6 Summary

Multiagent search shares a number of similarities with non-deterministic environments with single agents, and similar techniques can be used in both cases. For example, a single-agent setting in a non-deterministic environment can be treated in a similar way to a

as *AlphaZero*, such as *Leela Chess Zero*, indeed outperformed all earlier minimax programs by winning the computer chess championships in 2019.