25

Monte Carlo Tree Search and Related Algorithms for Games

Nathan R. Sturtevant

25.1 Introduction

This chapter is designed to introduce a number of recent algorithms, developed academically for game AI, primarily in board and card games. However, these algorithms also have significant potential in other video game genres, which we also explore here. This chapter is an expansion of a talk from the GDC 2014 AI Summit. We will introduce four different, but related, algorithms that can be used to create more dynamic and adaptable AI for games. With the description of each algorithm, we will provide examples of contexts where it would be most useful.

25.2 Background

To begin, we introduce a number of classifications between algorithms and other similar concepts that will be used repeatedly in this chapter.

A first important distinction is whether an approach plays strictly in an *online* manner or it also simulates actions *offline* (i.e., not player facing) before finally taking actions online. An online AI is one that gains experience and knowledge about the world strictly from making actions that are player facing. Most generally, an offline AI either ships with a static strategy or performs simulations at runtime that the player cannot see to determine the best action. In particular, we want to distinguish algorithms that require the ability to take and evaluate actions in an offline world before actually performing them in the online world.

The algorithms described in this chapter are *bandit algorithms* because the decisions they make are modeled by *n*-arm bandits (slot machines). The general *n*-arm bandit problem is to find the slot machine (a one-armed bandit) that has the best payoff. This is done by trying different bandits and looking at the resulting payoff. So, the assumption is that an action can be taken, and it will then be immediately associated by some reward or payoff. The primary difficulty in this problem is to balance exploiting the current-best slot machine with exploring to make sure that another slot machine doesn't have a better payoff. This problem describes an online slot machine because we pay each time we play a machine. In an offline problem, we would be able to simulate the slot machines offline without cost to find the best one before taking an action in the real world.

These algorithms also can be described as *regret-minimizing algorithms*. Loosely speaking, regret-minimizing algorithms guarantee that you will not regret using the algorithms instead of selecting and always playing one of the *n*-arm bandit strategies. Note that the quality of this guarantee depends on the strategies that correspond to each of the arms of the bandit. If these strategies are all poor, there is no guarantee that these algorithms will do any better.

Finally, these algorithms all use the notion of *utility* for evaluating states of the game. We use this instead of something like the chance of winning because the goal of an AI in many games is not to win, only to create the perception that it is trying to win. In doing so, the goal is usually to create a compelling experience for the human player. If we give high utility to the actions that help create a compelling experience, then in maximizing utility, the AI will be achieving the desired behavior.

Because it is simple and easy to illustrate, we demonstrate several algorithms using rock–paper–scissors (RPS) first before progressing to real-world examples that are more suited to each algorithm. To review, RPS is a two-player simultaneous game where each player chooses either rock, paper, or scissors. Paper beats rock, scissors beats paper, and rock beats scissors. RPS is usually played repeatedly. For our purposes, we assume that we get a score of 1 if we win, 0 if we draw, and −1 if we lose.

Given this background, we can now introduce our first algorithm.

25.3 Algorithm 1: Online UCB1

The first algorithm we describe, UCB1 [Auer 02], is a simple online bandit algorithm; it is deterministic and easy to implement. A naïve implementation of UCB1 is not perfectly suited for RPS, but after introducing this simple approach, we show to modify our strategies to improve the approach. Slight modifications to UCB1 have recently been proposed to give better regret bounds [Auer 10], but in practice the algorithm is quite robust, even when we break theoretical assumptions about how the algorithm should be used.

We demonstrate UCB1 by using it to play RPS. In our first approach, we assign each action (rock, paper, and scissors) to one of the arms of our slot machine, yielding a three-armed bandit. For each arm, UCB1 maintains the average payoff achieved when playing that arm, as well as the number of times each arm was played. Each time we are asked to make an action, we compute the value of each arm, $v(i)$, according to the formula in the following equation, where $x(i)$ is the average utility when playing arm i, $c(i)$ is the count of how many times we've played arm i, and k is a constant for tuning exploration and exploitation:

$$v(i) = x(i) + \sqrt{\frac{k\ln(t)}{c(i)}} \tag{25.1}$$

When asked to make an action, UCB1 plays the arm that has the maximum value of $v(i)$. The value, $v(i)$, is composed of two parts. The first part is an exploitation component, suggesting that we play the arm with the maximum average payoff. The second component is an exploration component. The more an arm is played, increasing $c(i)$, the smaller this value will be. The more other arms are played, the large the value will be. When payoffs are between 0 and 1, it is suggested that k should have the value 2. In practice, k can be tuned to achieve the desired balance between exploration and exploitation. When first starting, all arms are played once to get initial experience, although this could be preinitialized before ship. These actions are player facing, so it is important to avoid taking bad actions too many times.

We illustrate the resulting behavior in Table 25.1 when playing against a player that always plays rock for $k = 2$. For each action, we show the number of times the action was played ($c(i)$), the average utility of that action ($x(i)$), and the value ($v(i)$) that UCB1 would compute for that action. At time steps 0, 1, and 2, UCB1 has unexplored actions, so it must first explore these actions. At time step 3, the value of paper is $1+\sqrt{2 * \ln 3/1} = 2.48$. Paper is played because this is the best value of any action and continues to be until time step 7. During this time the value of paper decreases because $c(i)$ increases, while the value of scissors and rock increases because t increases. At time step 7, UCB1 finally stops exploiting paper and explores rock to see if playing rock can achieve a better outcome.

If we use UCB1 as an AI to play RPS, it will play in a relatively predictable manner, because there is no randomization. Thus, there are many sequences of actions that will

Table 25.1 Using UCB1 to Select the Next Action and Simulate the Resulting Situation in Order to Evaluate Which Next Action Is Best

	Rock			Paper			Scissors		
Time	$c(i)$	$x(i)$	$v(i)$	$c(i)$	$x(i)$	$v(i)$	$c(i)$	$x(i)$	$v(i)$
0	**0**	**0**	∞	0	0	∞	0	0	∞
1	1	0	0.00	**0**	**0**	∞	0	0	∞
2	1	0	1.18	1	1	2.18	**0**	**0**	∞
3	1	0	1.48	**1**	**1**	**2.48**	1	−1	0.48
4	1	0	1.67	**2**	**1**	**2.18**	1	−1	0.67
5	1	0	1.79	**3**	**1**	**2.04**	1	−1	0.79
6	1	0	1.89	**4**	**1**	**1.95**	1	−1	0.89
7	**1**	**0**	**1.97**	5	1	1.88	1	−1	0.97

be able to exploit the AI behavior. In the preceding example, playing the sequence P, S, R repeatedly will always win. This, of course, may be a desirable behavior if we want to reward the player for figuring this out. Because UCB1 will keep exploring its actions, it will never completely rule out playing bad actions. Thus, it may not be wise to assign a poor action to an arm of the bandit, as it will be played regularly, albeit with decaying frequency. Finally, note that if the opponent starts playing predictably (such as playing the sequence R, P, S repeatedly), this will never be noticed by the AI and never exploited.

To combat these shortcomings, we propose a slightly more interesting way of assigning actions to the arms of the bandit. Instead of letting the arms correspond to low-level actions in the world, we can have them correspond to strategies that are played, where each strategy is well designed and safe to play at any time. For instance, the nonlosing strategy (Nash equilibrium) in RPS is to play randomly. So, this should be the first possible strategy. If we wish to discourage repeated sequences of play, we can have other arms in the bandit correspond to imitation strategies, such as playing the same action the opponent played in the last round or playing the action that would have lost to the opponent in the last round. These strategies will be able to exploit repetitive play by the opponent. Taken together, we know that UCB1 will always default to a reasonable strategy (random) if its other strategies are losing. But if an opponent is playing in a predictable manner, it will be able to exploit that behavior as well.

Sample JavaScript code is included on this book's website (http://www.gameaipro.com), and a simplified portion of the code showing the main logic for implementing UCB1 is shown in Listing 25.1. This code is generic, in that it relies on a separate implementation of functions like `GetActionForStrategy`. Thus, it is simple to change out strategies and see how the play changes.

25.3.1 Applying to Games

While the previous example makes sense for a simple game like RPS, what about more complicated games? At the highest level, for UCB1 to be applicable, the decisions being made must be able to be formulated as bandit problems, with a set of available actions or strategies that result in known utility after they are sampled in a game. Given this restriction, here are several examples of how UCB1 can be used in other scenarios.

First, consider a fighting game like *Prince of Persia*, where enemies have different styles of fighting. There may be a general well-designed AI that works well for many players. But a small percentage of players are able to quickly defeat this general strategy or might learn to do so through the game. Perhaps a second AI is a good counter for these players, but isn't as well tuned for the other players. Instead of shipping a static AI that will fail for some percentage of the players, UCB1 could, at each encounter, be used to choose which AI the human should face next. The utility of the AI could be related to how long it takes the human to dispatch the AI. If the human is always defeating a certain AI quickly, UCB1 will start sending the alternate AI more often and in this way adapt to the player. If it is taking the player too long to defeat the alternate AI, then the first AI would be sent more often instead.

UCB1 works well here because neither AI strategy is fundamentally poor, so it can't make really bad decisions. Additionally, there are many small battles in this type of game, so UCB1 has many opportunities to learn and adapt. In some sense, UCB1 will work

Listing 25.1. An implementation of UCB1 in javascript.

```
function GetNextAction()
{
    if (init == false)
    {
        for (var x = 0; x < numActions; x++)
        {
            count[x] = 0;
            score[x] = 0;
        }
        init = true;
    }

    for (var x = 0; x < numActions; x++)
    {
        if (count[x] == 0)
        {
            ourLastStrategy = x;
            myLastAction = GetActionForStrategy(x);
            return myLastAction;
        }
    }

    var best = 0;
    var bestScore = score[best]/count[best];
    bestScore += sqrt(2*log(totalActions)/count[best]);
    for (var x = 1; x < numActions; x++)
    {
        var xScore = score[x]/count[x];
        xScore += sqrt(2*log(totalActions)/count[x]);
        if (xScore > bestScore)
        {
            best = x;
            bestScore = xScore;
        }
    }
    ourLastStrategy = best;
    myLastAction = GetActionForStrategy(best);
    return myLastAction;
}

function TellOpponentAction(opponentAct)
{
    totalActions++;
    var utility = GetUtility(myLastAction, opponentAct);
    score[myLastAction] += utility;
    count[myLastAction]++;
}
```

well for any game with these two properties. In a shooter, UCB1 might be used to decide whether to deploy a team with bold or cautious AI players. The bold AI would quickly be killed by a player holed up with a sniper rifle, while the cautious AI might sneak up on such a player. This is a natural situation where using UCB1 to balance the types of AI deployed could improve the player experience.

Oftentimes, there is hesitation to apply adaptive algorithms, as players might coerce them to adapt one way in order to exploit them in a key moment with a counterstrategy. This is less likely to be successful when all arms of the bandit are reasonable strategies. But the length of time that the AI plays in a certain way can be limited by only learning over a limited window of play or by weighting recent encounters more than earlier ones. Then, within a few encounters, the AI will be able to adapt back toward another strategy.

This approach would not work well for something like weapon selection in a role-playing game (RPG), because the AI would spend a lot of time trying to attack with poor weapons. It would also not work well when choosing individual attacks in a fighting game, because there are situations where some attacks make no sense or when multiple attacks must be balanced randomly to prevent complete predictability. (We note that sometimes this is desired, so that players can experience the joy of learning to defeat a particular opponent. But it is not always desired of all opponents.) A final shortcoming of this approach is that it learns more slowly because it doesn't consider retrospectively what might have happened if it did something different. In some games, like RPS, we can evaluate what would have happened if we used a different strategy, and we can use that to improve our performance.

25.4 Algorithm 2: Regret Matching

Regret matching [Hart 00] is another online algorithm that is just slightly more complicated than UCB1, but it can produce randomized strategies more suited to games where players act simultaneously or where the AI needs to act in a more unpredictable manner. Regret matching works by asking what would have happened if it had played a different action at each time step. Then, the algorithm directly accumulates any regret that it has for not playing different actions that were more successful. By accumulating this regret over time, the algorithm will converge to strong behavior or, more technically, a correlated equilibrium. We won't cover the theoretical justification for the algorithm here; besides the original paper, the interested reader is referred to the *Algorithmic Game Theory* book [Blum 07].

Regret matching works as follows. For each possible action, the algorithm keeps track of the regret for that action, that is, the gain in utility that could have been achieved by playing that action instead of a different one. Initially, all actions are initialized to have no regret. When no actions have positive regret, we play randomly. Otherwise, we select an action with a biased random in proportion to the positive regret of each action. Each time we take an action, we retrospectively ask what the utility of every alternate action would have been if we had taken it during the last time step. Then, we add to the cumulative regret of each action the difference between the payoff we would have received had we taken the other action and our actual payoff from the action we did take. Thus, if another action would have produced a better payoff, its regret will increase, and we will play it more often.

We illustrate regret matching in RPS, with our bandit arms corresponding to playing each of our actions: rock, paper, and scissors. Initially, we have no accumulated regret and play randomly. Suppose that we play rock and lose to the opponent playing paper. Assuming we get 1 for winning, –1 for losing, and 0 otherwise, our regret for not playing scissors (and winning) is increased by (1 − (–1)) = 2. Playing paper would have tied, so we accumulate regret (0 − (− 1)) = 1. We have not accumulated any positive or negative regret for playing rock. Thus, in the next round, we will play scissors with probability 2/3 and paper with probability 1/3. Suppose that in the next round we play scissors and draw against an opponent playing scissors. Then, our regret for not playing rock will increase by 1, since playing rock would have increased our utility by 1. Our regret for playing paper is decreased by 1, since we would have lost if we had played paper. Thus, our regrets are now 1 for rock, 2 for scissors, and 0 for paper. In the next round, we will play rock with probability 1/3 and scissors 2/3. Note that the algorithm can be improved slightly by computing regret using the expected utility of the action that was taken (according to the probability distribution that determines play) instead of using just the utility of the action that was taken. As with UCB1, regret matching can use strategies instead of actions as the bandit arms.

The code included on the book's website implements regret matching for both actions and strategies. You can play against both to observe play, and you can also try to exploit the algorithm to get a feel for its behavior. Simplified JavaScript code for regret matching can be found in Listing 25.2. The key property that the algorithm needs to run is the ability to introspectively ask what would have happened if other actions were played. Additionally, we need to know the utility that would have resulted for those actions. If this cannot be computed, then regret matching is not an applicable algorithm.

In practice, there are several changes that might be made to ensure better play. First, instead of initializing all regrets to 0, the initial values for the regret can be initialized to produce reasonable play and influence the rate of adaptation. If, in RPS, all initial regrets are set to 10, the algorithm will start adapting play in only a few rounds. But if all initial regrets are set to 1000, it will take significantly longer for the program to adapt. Related to this, it may be worthwhile to limit how much negative regret can be accumulated, as this will limit how long it takes to unlearn anything that is learned.

Finally, regret matching can be used both as an offline or online algorithm when the game has two players and the payoffs for each player sum to zero. Regret matching is the core algorithm used recursively for solving large Poker games [Johanson 07]. In this context, the game is solved offline and the static strategy is used online, although slight modifications are needed for this to work correctly.

25.4.1 Applying to Games

Once again it is natural to ask the question of how this approach can apply to more complicated video games, instead of a simple game like RPS. We provide two examples where the algorithm would work well and one example where it cannot be applied.

Our first example is due to David Sirlin in situations he calls "Yomi" [Sirlin 08]. Consider a two-player fighting game where one player has just been knocked down. This player can either get up normally or get up with a rising attack. The other player can either attack the player as they get up or block the anticipated rising attack. This situation looks a lot like RPS, in that both players must make simultaneous decisions that will then result

Listing 25.2. An implementation of regret matching in javascript.

```
function GetNextAction()
{
    if (init == false)
    {
        for (var x = 0; x < numActions; x++)
            regret[x] = 0;
        init = true;
    }
    for (var x = 0; x < numActions; x++)
    {
       lastAction[x] = GetActionForStrategy(x);
    }

    var sum = 0;
    for (var x = 0; x < numActions; x++)
        sum += (regret[x]>0)?regret[x]:0;
    if (sum <= 0)
    {
        ourLastAction = floor(random()*numActions);
        return ourLastAction;
    }

    for (var x = 0; x < numActions; x++)
    {
        //add up the positive regret
        if (regret[x] > 0)
            chance[x] = regret[x];
        else
            chance[x] = 0;

        //build the cumulative distribution
        if (x > 0)
            chance[x] += chance[x-1];
    }

    var p = random();
    for (var x = 0; x < numActions; x++)
    {
        if (p < chance[x])
        {
            ourLastStrategy = x;
            ourLastAction = lastAction[x];
            return ourLastAction;
        }
    }
    return numActions-1;
}
```

```
function TellOpponentAction(opponentAct)
{
    lastOpponentAction = opponentAct;
    for (var x = 0; x < numActions; x++)
    {
       regret[x] += GetUtility(lastAction[x], opponentAct);
       regret[x] -= GetUtility(ourLastAction, opponentAct);
    }
}
```

in immediate payoff (damage). Here, regret matching would be applied independently in each relevant context, such as after a knockdown, to determine how to play at that point. During play, the AI will appropriately balance its behavior for each of these situations to maximize its own payoff.

In these situations, the AI has the potential to balance attacks far better than a human player and, as a result, might be almost unbeatable. (Conversely, identifying the current situation properly might be too error prone to result in good play.) AI players using regret matching for their strategies can be given more personality or a preferred playing style by biasing their utility. If we want a player that likes to punch, we simply give more utility for performing punches, even if they are unsuccessful. This fools the AI into performing more punch actions, because it will maximize utility by doing so.

In this context, regret matching can also be used offline prior to shipping the game to build a single strong player via self-play. This player would not adapt at runtime but would still randomize its behavior at runtime, resulting in play that cannot be exploited.

For our second example, we go from very low-level character control to high-level strategic decision making. Suppose that we are playing multiple rounds of a game like *Starcraft* against an opponent and we must decide what sort of build tree to use at the beginning of the game—optimizing for rushing or some other play styles. We can use regret matching for this purpose if we are able to introspectively evaluate whether we chose the best strategy. This is done by looking to see, after the match was complete, whether another strategy would have been better. For instance, we might evaluate the building selection and resource distribution of our opponent after 3 min of play (before either team has a chance to see the other team and adapt their resulting play). If we see that we might have immediately defeated the opponent had we chosen to follow a rush strategy, we then accumulate regret for not rushing.

To give an example where regret matching will not work well, consider again a fighting game like *Prince of Persia*, where we might be choosing what sort of AI to send out against the human player. Once the AI acts in a way that influences the human behavior, we can no longer ask what would have happened if we had sent different AI types. Thus, we will not be able to use an algorithm like regret matching in this context.

25.5 Algorithm 3: Offline UCB1

The algorithms introduced thus far primarily act in an online manner, without considering the implications of their actions beyond the feedback collected *after* every action is taken. This means that they are best used when the strategies or actions taken will always be reasonable, and the main question is how to balance these actions in order to provide a compelling gameplay experience. But this isn't always possible or desirable. In many situations, we need an algorithm that will reason to rule out bad actions and never take them. To do this, we perform offline simulations of actions in the world before deciding on a final action to take.

To discuss possible options concretely, move away from RPS and use the same example found in the introductory chapter to this section of the book—a simple role-playing game (RPG) battle. In that chapter, we discussed how a one-step search combined with a strong evaluation function would produce reasonable play. (The evaluation function should return the utility for the AI in that state.) The drawback of that approach is that we must write the evaluation function and tune it for high-quality play. The first new idea here is that it is much easier to write an evaluation function for the end of the game than for the middle of the game. So, if we play out a game to the end using some strategy (even random), we can often get a better evaluation than we would by trying to write an evaluation function after a 1-ply search. We demonstrate this using an RPG battle, where the AI is controlling a nonplayer character (NPC) spellcaster that has a fighter companion. The spellcaster will primarily use ranged attacks from a magical staff but can occasionally cast either a healing spell or an area attack such as a fireball.

Previously, we discussed how bandit algorithms can use both low-level actions and high-level strategies as the arms for the bandit. Here, we will combine these ideas together. We will use the primitive actions as the arms for our bandit using UCB1. But instead of actually taking actions online in the world, we simulate the actions internally. Then, instead of just applying a utility function to evaluate the best action, we continue by using a high-level strategy to simulate actions through the end of the current battle.

This is illustrated in Figure 25.1. The NPC must act using one of these three actions: healing, attacking with a staff, or casting a fireball. UCB1 selects an action to play and then simulates the rest of the battle using some default strategy. When the battle is over, we must compute the utility of the resulting state, for instance, returning the total health in our party after the battle finishes (perhaps adding some bonus for every party member that is still alive). This evaluation resembles what would be used in a 1-ply search, but the evaluation is much easier than before because we don't have to evaluate every

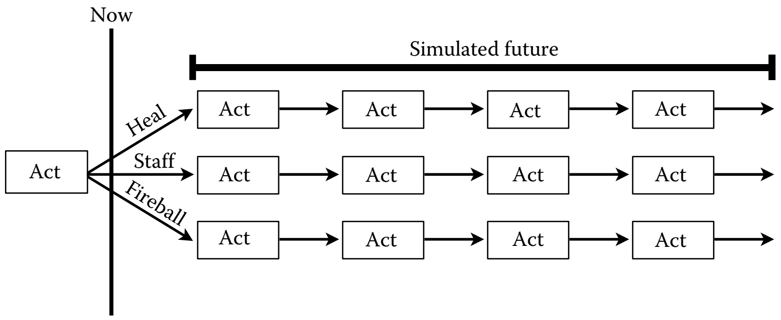

Figure 25.1

Using UCB1 to select the next action and simulate the resulting situation in order to evaluate which next action is best.

situation possible in the battle; we are restricted to only evaluating states where one team is defeated. Instead of trying to predict the outcome of the battle, we just need to evaluate if our party survived and compute the utility how many resources we have left. Suppose that casting a fireball would use all available manna and allow no other spells to be cast through the remainder of the battle. In the short term, this might seem good, but in the long term, the inability to cast healing may cause us to lose the battle. Being able to simulate the battle to its end will reveal this expected outcome.

Now, we might do this once per action and then select the best action, but there is often significant uncertainty in a battle, including randomness or other choices like the selection of what enemies to target. Thus, instead of just simulating each move once, it is valuable to simulate moves multiple times to get better estimates of the final outcome. If we sample every top-level action uniformly, we waste resources simulating bad strategies and lose out on gaining more information about strategies that have similar payoffs. This is where UCB1 shines; it will balance playing the best action with exploring actions that look worse in order to ensure that we don't miss out on another action that works well in practice. It should be noted that if we are going to simulate to the end of the battle, our default strategy also must provide actions not only for the AI player but also for all other players in the battle.

We show high-level pseudocode for using UCB1 in this manner in Listing 25.3. This code just provides the high-level control of UCB1 using the definition of `GetNextAction()` defined previously in Listing 25.1. In the previous example, this function was called each time an action was needed for play. Now, this is called as many times as possible while time remains.

After generalizing this approach to the UCT algorithm in the next section, we will discuss further the situations where this algorithm could be used in practice.

Listing 25.3. Pseudocode for using UCB1 to control simulations for finding the next best move. This code uses the `GetNextAction()` method from Listing 25.1 for playing actions.

```
function SimulateUCB()
{
    while (time remains)
    {
        act = GetNextAction();
        ApplyAction(act);
        utility = PlayDefaultStrategy();
        UndoAction(act);
        TellUtility(act, utility);
    }
    return GetBestAction();
}

function TellUtility(act, utility)
{
    totalActions++;
    score[act] += utility;
    count[act]++;
}
```

25.6 Algorithm 4: UCT

UCB1 as described in the last section is a 1-ply search algorithm in that it only explicitly considers the first action before reverting to some default policy for play. In practice there can be value in considering several actions together. For instance, there may be two spells that, when cast together, are far more powerful than when used alone. But if they must be cast from weakest to strongest to be effective, a 1-ply algorithm may not be able to find and exploit this combination of spells. By considering the influence of longer chains of actions, we give our AI the ability to discover these combinations automatically. The generalization of UCB1 to trees is called UCT [Kocsis 06]; this is the final algorithm we present in this chapter. UCT is the most popular specific algorithm that falls into the general class of Monte Carlo tree search (MCTS) algorithms.

UCT extends the use of UCB1 in the previous section by building a dynamic tree in memory, using the UCB1 algorithm to direct the growth of the tree. UCT builds a nonuniform tree that is biased toward the more interesting part of the state space. The longer the search, the larger the tree, and the stronger the resulting play.

Over time, researchers have converged on describing UCT and MCTS algorithms via four stages of behavior. The first stage is selection, where the best moves from the root to the leaves of the in-memory tree are selected according to the UCB1 rule at each node. The second stage is expansion, where new nodes are added to the UCT tree. The third stage is simulation, where some default policy is used to simulate the game. The final stage is propagation, where the value at the end of the simulation is propagated through the path in the UCT tree, updating the values in the tree.

We walk through an example to make these ideas concrete. In our example, a spell-casting AI is allowed to cast two spells back to back, after which the normal battle will continue. We assume that a gas cloud can be ignited by a fireball to do additional damage. Figure 25.2 shows the root of a UCT tree for this situation with three children, one child for each spell that can be cast. The nodes in black (nodes 1, 2, and 3) are in the tree prior

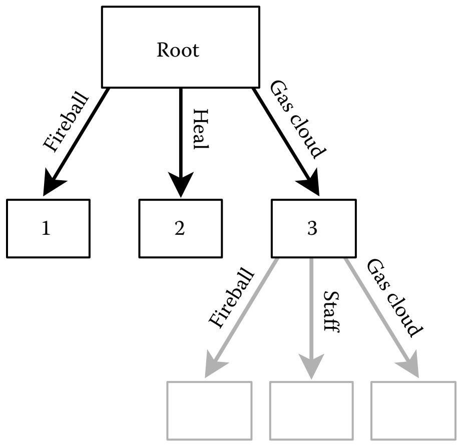

Figure 25.2

UCT selection and expansion phases.

to starting our example. The selection phase starts at the root and uses the UCB1 rule to select the next child to explore according to the current payoffs and number of samples thus far. This is repeated until a leaf node is reached. In this case we select the third spell and reach the leaf of the tree. Each time we reach the leaf of the tree, we expand that node, adding its children into the tree. Since we haven't seen these new nodes before, we select the first possible action and then continue to the simulation phase.

In Figure 25.3 we show the simulation phase. Starting after the fireball action, we use some policy to play out the game until the current battle is over. Note that in this simulation we will simulate actions for all players in the battle, whether or not they are on our team. When we reach the end of the battle, we score the resulting state. Then, we modify the UCB1 values at the root, state 3, and state 4, updating the number of simulations and average utility to take into account the result of this simulation. If there are two players in the game, nodes that belong to the opposing player get different utilities than those belonging to the AI. Following what is done in the minimax algorithm, this is usually just

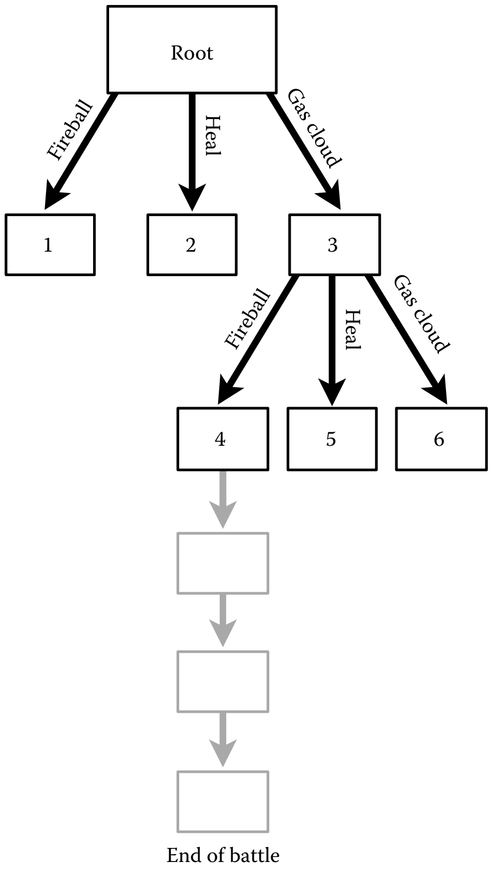

Figure 25.3

UCT simulation and propagation phases.

the negation of the score of the AI player. If there are multiple competing players, different utilities should be backed up for each player [Sturtevant 07].

This entire process should be repeated many times. The more it is repeated, the better the resulting strategy. In practice, what would usually happen in an example like this one is that initially the fireball would be preferred, as it immediately causes significant damage. But as more simulations are performed and the tree grows, the strategy of a gas cloud followed by a fireball emerges, as this combination is much more effective than a fireball followed by a gas cloud.

Pseudocode for a recursive implementation of UCT is shown in Listing 25.4. The top-level code just repeatedly calls the selection rule until the time allotment runs out. The tree selection code uses the UCB1 rule to step down the tree. Upon reaching the end, it expands the tree and then simulates the rest of the game. Finally, the counts and utilities for all nodes along the path are updated.

Listing 25.4. Pseudocode for UCT.

```
function SimulateUCT()
{
    while (time remains)
    {
        TreeSelectionAndUpdate(root, false);
    }
    return GetBestAction();
}

function TreeSelectionAndUpdate(currNode, simulateNow)
{
    if (GameOver(currNode))
        return GetUtility(currNode);
    if (simulateNow)
    {
        //Simulate the rest of the game and get the utility
        value = DoPlayout(currNode);
    }
    else if (IsLeaf(currNode))
    {
        AddChildrenToTree(currNode);
        value = TreeSelectionAndUpdate(currNode, true);
    }
    else {
        child = GetNextState();//using UCB1 rule (in tree)
        value = TreeSelectionAndUpdate(child, false);
    }

    //If we have 2 players, we would negate this value if
    //the second player is moving at this node
    currNode.value += value;
    currNode.count++;
    return value;
}
```

25.6.1 Important Implementation Details

Those who have worked with UCT and other MCTS algorithms have shared significant implementation details that are important for improving the performance of UCT in practice.

First, it is very important to look at the constant that balances exploration and exploitation when tuning UCT. If this constant is set wrong, UCT will either explore all options uniformly or not sufficiently explore alternate options. We always look at the distribution of simulations across actions at the first ply of the UCT tree to see if they are balanced properly in relation to the payoffs.

As memory allocation can be expensive, it is worthwhile to preallocate nodes for the UCT tree. A simple array of data UCT nodes is sufficient for this purpose. Although many implementations of UCT add new nodes to the tree after every simulation, the process of adding new nodes can be delayed by requiring a node to be visited some minimum number of times before it is expanded. This usually saves memory without significantly degrading performance.

After simulation, a final action must be selected for execution. This action shouldn't be selected using the UCB1 rule, as there is a chance it will sample a bad move instead of taking the best one possible. Two common approaches are to choose the action that was sampled the most or to choose the action that has the highest payoff. In some domains, these alternate strategies can have a large influence on performance, but in others, both are equally good, so this should be tested in your domain.

UCT works best in games or scenarios that are converging. That is, the games are likely to end even under a fixed strategy or under random play. If a game isn't converging, the game simulations may be too expensive or too long to return meaning information about the game. Thus, it is common to do things like disable backward moves during simulations; in an RPG, it might be worth disabling healing spells if both parties have them available. The quality of the simulations can have a significant impact on the quality of play, so it is important to understanding their influence.

25.6.2 UCT Enhancements and Variations

There is a large body of academic researcher's work looking at modifications and enhancements to UCT and MCTS algorithms. While we can't discuss all of these in detail, we highlight a few key ideas that have been used widely.

- In games like Go, the same action appears in many different parts of the game tree. This information can be shared across the game tree to improve performance [Gelly 07].
- In some games the simulations are too long and expensive to be effective. But cutting off simulations at a shallower depth can still be more effective than not running simulations at all [Lorentz 08].
- There are many ways to parallelize the UCT algorithm [Barriga 14], improving performance.

At the writing of this chapter, a recent journal paper [Browne 12] catalogs many more of these improvements, but there has also been significant new work since this publication.

25.6.3 Applying to Games

UCT and MCTS approaches are best suited for games with discrete actions and a strong strategic component. This would include most games that are adaptations of board games and games that simulate battles, including tabletop-style games and RPGs. The last 10 years of research has shown, however, that these approaches work surprisingly well in many domains that would, on the surface, not seem to be amenable to these techniques. Within a decade or two, it would not be surprising to find that minimax-based approaches have largely disappeared in favor of UCT; chess is currently one of the few games where minimax is significantly stronger than UCT approaches. In fact, MCTS techniques have already found their way into commercial video games such as *Total War: Rome II*, as described in the 2014 Game/AI Conference. We believe that they could be very effective for companion AI in RPGs.

The main barrier to applying UCT and MCTS approaches in a game is the computational requirements. While they can run on limited time and memory budgets, they are still more expensive than a static evaluation. Thus, if simulation is very expensive or if the number of available actions is very large, these approaches may not work. But, even in these scenarios, it is often possible to abstract the world or limit the number of possible actions to make this approach feasible.

25.7 Conclusion

In this chapter, we have presented four algorithms that can be used in a variety of game situations to build more interesting and more adaptive AI behavior. With each algorithm, we have presented examples of possible use, but we suspect that there are many more opportunities to use these algorithms that we have considered. Most of these algorithms are based in some way on UCB1, a simple and robust bandit algorithm.

We hope that this work will challenge the commercial AI community to explore new approaches for authoring strong AI behaviors. If nothing else, we add four more techniques to the toolbox of AI programmers for building game AI.

References

[Auer 02] Auer, P., Cesa-Bianchi, N., and Fischer, P. 2002. Finite-time analysis of the multi-armed bandit problem. *Machine Learning* 47:235–256.

[Auer 10] Auer, P. and Ortner, R. 2010. UCB revisited: Improved regret bounds for the stochastic multi-armed bandit problem. *Periodica Mathematica Hungarica* 61:55–65.

[Barriga 14] Barriga, N., Stanescu, N., and Buro, M. 2014. Parallel UCT search on GPUs. *IEEE Conference on Computational Intelligence and Games*, Dortmund, Germany, pp. 1–7.

[Blum 07] Blum, A. and Mansour, Y. 2007. Learning, regret minimization, and equilibria. In *Algorithmic Game Theory*, ed. N. Nisan, pp. 79–102. Cambridge University Press, Cambridge, U.K.

[Browne 12] Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis S., and Colton, S. 2012. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–43.

[Gelly 07] Gelly, S. and Silver, D. 2007. Combining online and offline knowledge in UCT. *International Conference on Machine Learning. ACM International Conference Proceeding Series,* Corvallis, OR, pp. 273–280.

[Hart 00] Hart, S. and Mas-Colell, A., 2000. A simple adaptive procedure leading to correlated equilibrium. *Econometrica* 58:1127–1150.

[Johanson 07] Johanson, M., 2007. Robust strategies and counter-strategies: Building a champion level computer poker player. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada.

[Kocsis 06] Kocsis, L. and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning*, pp. 282–293. Springer, Berlin, Germany.

[Lorentz 08] Lorentz, R. 2008. Amazons discover Monte-Carlo. *Computers and Games* 5131:13–24.

[Sirlin 08] Sirlin, D. 2008. Yomi layer 3: Knowing the mind of the opponent. http://www.sirlin.net/articles/yomi-layer-3-knowing-the-mind-of-the-opponent.html (accessed September 15, 2014).

[Sturtevant 07] Sturtevant, N. 2007. An analysis of UCT in multi-player games. *Computers and Games* 5131:37–49.