Comparing UCT versus CFR in Simultaneous Games

Mohammad Shafiei   Nathan Sturtevant   Jonathan Schaeffer
Computing Science Department
University of Alberta
{shafieik,nathanst,jonathan}@cs.ualberta.ca

Abstract

Simultaneous move games where all the player have to take their actions simultaneously are a class of games in general game playing. In this paper we analyze how UCT performs in this class of games. We argue that UCT does not converge to a Nash equilibrium in general and the situation that it converges to can be exploited. We also analyze CFR (CounterFactual Regret) and show how it can be used to exploit UCT.

1 Introduction

In General Game Playing (GGP), an agent is given the description of a game to be played through the Game Description Language (GDL) which is represented in *KIF* (Knowledge Interchange Format) [Love *et al.*, 2006]. The agent has no prior knowledge about what the game is going be to. Therefore it cannot have any special-purpose algorithms hardcoded in its nature for solving that particular game. The player should analyze the game description and play by choosing reasonable actions. The better it chooses its actions, the stronger it will be.

Games in GGP can be single agent, two player, and multiplayer. All the games are finite, discrete, and deterministic. At each step of the game, all the players are required to submit the action that they are going to take. If the game is turn taking, then the player whose turn is passed will simply submit the special *noop* (no operation) move. However, if the game is a real simultaneous move game, then all the players will have effective moves at each step of the game. This results in uncertainty in the game since none of the players know what moves other players will take. Therefore, each player should have a good model of his opponents or his actions must be robust enough considering whatever actions others will take. Although having a good model of the opponent is also beneficial in turn-taking games, it is more decisive in simultaneous move games because the player will not be even certain about result of his own action considering the indeterminism that arises from what others may do.

Different approaches have been used to create general game playing programs. Initially, most program developers tried to develop heuristics for a game by extracting features from the game description [Clune, 2007; Schiffel and Thielscher, 2007; Kuhlmann and Stone, 2006]. The heuristics were then used in conjunction with a classic search algorithm (*e.g.* alpha-beta) to play the game. Therefore, devising a good heuristic was a key factor in the success of their approaches. Inventing a good heuristic is a challenging problem since the game that is going to be played by the player is unknown beforehand. After the advent of UCT [Kocsis and Szepesvári, 2006] and its notable success in the game of Go [Coulom, 2006; Gelly *et al.*, 2006], its use in general game playing came into prominence. An appealing feature of UCT is that it does not require any special knowledge about the domain. This feature makes UCT a robust and simple way to approach the design of a general game player. UCT seemed promising at first, but like other algorithms, it was well suited to a special class of problems, *viz.* problems in deterministic domains. Although general game playing is currently focused on deterministic domains, simultaneous move games are legal. In a simultaneous move game, the presence of at least another player that is allowed to change the game state while the player is taking an action, adds indeterminism to the game and makes it challenging for UCT. Since there is no good strategy to play these games now, a non-losing strategy can be a reasonable one. Tuning our strategy to play according to the Nash equilibrium results in a non-losing strategy. CFR (CounterFactual Regret) is a way of computing approximate Nash equilibrium.

In this paper we will focus on the simultaneous move games and how they can be handled. We will review the UCT algorithm in section 2 and will analyze its use in simultaneous move games in section 3. We argue that UCT will not converge to a Nash equilibrium in general and the situation that it converges to can be exploited. We will consider the CFR algorithm in section 4 and its use in GGP in section 5. We will also present insights about how CFR performs in GGP. Finally we will discuss how CFR can be used to exploit UCT in section 6.

2 UCT

The multi-armed bandit problem is an example of an environment where an agent tries to optimize his decisions while improving his information about the environment at the same time. If we consider a K-armed bandit, we are dealing with K different slot machines whose outcomes follow different unknown distributions with different expected values. An op-

timal play with K-armed bandit is to select the arm with the highest payoff at each step of play. However, since we do not know the distribution of outcomes for different arms, the goal is to be as close as possible to the optimal play based on our past experiences. By careful tuning how much we exploit the best known arm versus exploring the other arms, we can bound our regret that results from selecting a suboptimal action (pulling the suboptimal arm). UCB1 (Upper Confidence Bound) is an algorithm to balance between exploration and exploitation. It achieves a logarithmic bound on the regret as a factor of the number of plays [Auer *et al.*, 2002]. It considers a bonus for selecting each action which is directly proportional to the number of plays and inversely proportional to the number of times that specific action has been selected till now. Therefore, actions that have been rarely selected will get higher bonus to be selected and explored.

UCT (UCB applied to Trees) is an extension of the UCB1 algorithm to tree search. It gradually expands the game tree by adding nodes to it. UCT considers every interior node in the tree as an independent bandit and its children (available actions at that node) as the arms of the bandit. UCT searches the tree based on a specific strategy (*e.g.* minimax) considering a bonus similar to the one in UCB1 to balance between exploration and exploitation. When the search reaches a non-terminal leaf node, a Monte Carlo simulation from that node to a terminal state is carried out. The value that results from the simulation is then used to update the values of all nodes along the path from the root of the tree to the node that leads to that simulation. UCT is an iterative algorithm. It searches through the tree, does simulations at non-terminal nodes and adds new nodes to the tree. Tree expansion will continue until the whole tree is expanded or a memory or time limit is reached. The pseudocode for one iteration of UCT is given in Figure 1. Proofs of convergence and regret bounds can be found in [Kocsis and Szepesvári, 2006].

The first three lines in Figure 1 initialize and expand the root of the tree. Node expansion involves adding all the children of a node to the tree without expanding them. It also sets the counters and average return value of all children to zero. Tree search is done in lines 4-12. In the tree search, if we reach a node that has a child which has not been expanded yet ($counter = 0$), that child gets priority for selection over others (lines 8-9). Selection among the other children is done based on the average value of that child and a bonus to enforce exploration (line 11). When we reach a non-terminal leaf node in the tree that has not been expanded yet, it will be explored. This is the place where memory bounds can be enforced (lines 13-15). Finally if we are not in a terminal state, a simulation will be carried out and the return value will be gathered (lines 16-19). This will be used to update the average return values of the nodes in the tree along the path from the root (lines 20).

The pseudocode given in Figure 1 corresponds to single agent UCT. However, UCT has been applied in domains with more than one agent (*e.g.* computer Go [Coulom, 2006; Gelly *et al.*, 2006]) and showed notable results. It has also been used in the general game playing competition and showed superior results as well, becoming the champion for two successive years [Finnsson and Björnsson, 2008]. The single agent UCT can be modified to be used in multiplayer and simultaneous move games [Finnsson and Björnsson, 2008]. Furthermore, the behaviour of UCT has been analyzed for multiplayer games and shown to compute an (possibly mixed) equilibrium strategy [Sturtevant, 2008]. For example, by replacing line 11, the selection rule between children of a node, with

$$values[i] \leftarrow -\, traverser\,.\, children[i].\, value + C\sqrt{\frac{\ln(traverser\,.\, counter)}{traverser\,.\, children[i].\, counter}}$$

when the opponent is the active player (the player that can make a move), we will have a UCT that uses minimax in tree search.

2.1 Multiplayer Games and Opponent Modeling

In general game playing the value of the goal is defined for different players in a terminal state. Thus, the program must keep values for each player. In multiplayer games, the number of children can be as large as the product of available actions for each player. Therefore, the number of children can be very large. However, we can keep the values for different actions of different players instead of keeping them for each combination of actions. This will also enable us to do more sophisticated opponent modeling than merely considering that everybody is playing against us (paranoid assumption).

2.2 Exploration vs. Exploitation

The square root bonus added to the value of a node on line 11 is used to balance between exploitation and exploration. It is directly proportional to the number of times a state (parent) is visited and inversely proportional to the number of times a specific action (child) is selected. Therefore, by exploiting the best action repeatedly, the bonus for selecting other actions will become larger. The constant factor, C, defines how important it is to explore instead of selecting the greedy action. The higher the value of C, the more exploration will be done.

2.3 Playout Policies

Different policies can be used to do the Monte Carlo simulation. The simplest one is to select random actions for different players at each step. However, one can use a more informed approach by selecting the best action if the simulation runs into a node that has been visited before during tree search (the idea of using transition table). In addition, history heuristic about actions can be used in the playout.

2.4 Updating Values

The outcome that results from a simulation will be used to update the values of the nodes along the path from the root in the search tree (line 20). Updates can simply be a weighted average. However, if the game is single-player and there is no uncertainty in the game, maximization between the old value and the new one can be used as the update rule. In the deterministic single player case, the only concern is to achieve the highest possible outcome. In addition, a discount factor can be used during the update process to favor shorter solutions over longer ones.

```
DoOneIteration()
 1  if root = NULL
 2     then root ← MakeNode                      ▷ Initializes the root of the tree
 3          ExpandNode(root)                     ▷ Adds all children of the node and sets their counters to zero,
                                                   but does not expand them.

 4  traverser ← root
 5  while traverser ≠ LEAF
 6          expandLeaf ← TRUE
 7          for i ← 1 to number[traverser . children]
 8                  if traverser . children[i]. counter = 0     ▷ Will be incremented in UpdateValues(.,.)
 9                    then values[i] ← ∞
10                         expandLeaf ← FALSE
11                    else values[i] ← traverser . children[i]. value + C sqrt( ln(traverser . counter) / traverser . children[i]. counter )
12          traverser ← traverser . children[arg max{values}]

13  if expandLeaf and traverser ≠ TERMINAL
14     then ExpandNode(traverser)
15          traverser ← RandomChild(traverser)

16  if traverser ≠ TERMINAL
17     then outcome ← DoMonteCarloSimulation(traverser)
                         ▷ Does a simulation to the end of the game and returns the outcome
18     else traverser . values ← GameValue
19          outcome ← traverser . values

20  UpdateValues(traverser, outcome)
                         ▷ Updates the values and increments the counters of the nodes along the path to the root
```

Figure 1: One iteration of the UCT algorithm.

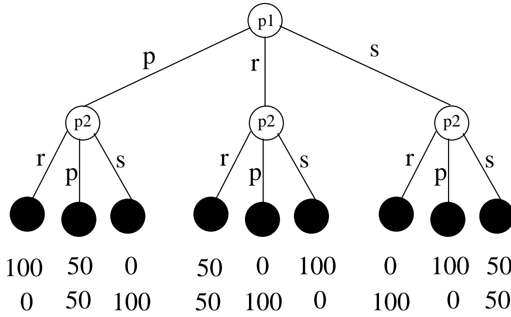

Figure 2: Rock-paper-scissors game tree for UCT. Dark nodes are terminal. Action selection is a simultaneous process, but the tree is represented in two levels to be easier to understand.

3 UCT in Simultaneous Games

The variant of UCT that we will consider for simultaneous move games will use maximization strategy for all players. It will also keep track of expected values for different actions of different players as well as the number of times that each action has been selected. Therefore it can calculate the exploration bonus for each player separately.

We consider the game of rock-paper-scissors and $C = 100$ to illustrate how UCT works (refer to Figure 2). In this example the game tree includes only one non-terminal node and three terminal nodes, therefore we will not consider tree expansion here. The game tree in Figure 2 is represented in two levels to simplify the description of the computation involved. At the beginning, since none of the actions for any player has been tried before, action selection is done randomly. For simplicity we will assume that the player will select his left most action in Figure 2 when he has to select an action randomly. The first player selects paper and the second player selects rock leading to $(100, 0)$. On the second and the third iteration (r, p) and (s, s) are selected respectively. Therefore after three iterations the expected value of actions for the first and the second players will be as follows.

$$E_{P1}(rock) = 0, E_{P1}(paper) = 100, E_{P1}(scissors) = 50$$

$$E_{P2}(rock) = 0, E_{P2}(paper) = 100, E_{P2}(scissors) = 50$$

On the fourth iteration both players select paper considering the expected values of their actions (the bonus is very small at this point). The expected values will be updated as follows.

$$E_{P1}(rock) = 0, E_{P1}(paper) = 75, E_{P1}(scissors) = 50$$

$$E_{P2}(rock) = 0, E_{P2}(paper) = 75, E_{P2}(scissors) = 50$$

The players will keep selecting paper until its expected value is down to 50 or the bonus for the scissors gets large enough

	rock	paper	scissors
rock	$50, 50$	$25, 75$	$100, 0$
paper	$75, 25$	$50, 50$	$45, 55$
scissors	$0, 100$	$55, 45$	$50, 50$

Table 1: Biased rock-paper-scissors payoff matrix.

to dominate the difference. This process will be continued and the values will be updated until all the values converge to the game theoretic expected value of the game which will be the value of the Nash equilibrium, *i.e.* 50 in this case.

If we compute the ratio of taking each action during the UCT iterations for the rock-paper-scissors example, it will be the same as the probabilities in the Nash equilibrium, *i.e.* $(1/3, 1/3, 1/3)$. However, UCT is not able to get the correct mixed strategy in general, even if there is only one Nash equilibrium in the game. The reason for this problem is because the balanced situation that UCT converges to is based on the model of opponents that is assumed during the iterations. In addition the probabilities (number of times each action is selected in comparison with others) that UCT assumes for each player is correlated with the probability of selecting each action for other players. Therefore the way that the player will finally behave is based on how the correlation of players' action in UCT result to a balanced situation. When UCT finds a situation (probability settings) that leads it to the achievable expected value it could get if it has followed a Nash equilibrium, it will adhere to that balanced settings which may not be a Nash equilibrium.

Rock-paper-scissors with biased payoff as shown in Table 1 is an example of a game that UCT gets into a balanced situation instead of converging to the true mixed strategy Nash equilibrium. The rules of the game are the same while the outcomes are different. In Table 1, the first row and column are the actions of players and the values in each cell are the payoffs for taking the joint actions crossing at that cell. The row player gets the first value while the second player gets the second one. There is only one Nash equilibrium in this game which is a mixed strategy with action probabilities as follows.

$$P(rock) = 0.0625$$
$$P(paper) = 0.6250$$
$$P(scissors) = 0.3125$$

One possible execution of UCT with $C = 100$ will be as follows.[1]

$(r, p), (p, s), (s, r), (p, r), (p, p), (r, s), (r, p), (s, r), (p, r),$
$(r, p), (p, p), (p, r), (p, s), (r, p), (p, p), (s, s), (p, r), (p, p),$
$(r, s), (r, p), (s, p), (p, r), (s, p), (p, p), (r, r), (s, s), \ldots$

After this sequence, the values for both players in UCT will be identical which will result in both players playing the same during UCT iterations and cycling through a balanced situation. Since the value that each player is getting is also equal to the expected value of the Nash equilibrium in the game, both players are satisfied with their outcomes and will not change their action selection. If we consider action selection ratios to define the probability of choosing each action, we will get equal probabilities for different actions $(1/3, 1/3, 1/3)$. It is clear that this probability setting is not a Nash equilibrium, because each player can increase his payoff by unilaterally skewing his action selection probability toward paper.

The balanced situation that the players arrive at is dependent on the value of C for each player. For example if we consider $C = 100$ for the first player and $C = 50$ for the second player, then the probability settings for the first and second players after approximately one million iterations will be $(0.07, 0.12, 0.81)$ and $(0.03, 0.46, 0.51)$ respectively (the first, second, and third arities are probabilities of selecting rock, paper, and scissors respectively). Therefore if UCT plays as the first player, the second player can exploit UCT by skewing his action selection probability toward rock. On the whole, the balanced situation that UCT converges to is not necessarily a Nash equilibrium and can be exploited.

[1]The purpose of this part is to give a counter example that UCT does not compute the Nash equilibrium in general.

```
DoOneIteration()
1  if root = NULL
2     then BuildTree

3  ComputeExpectedValues(root)
4  ResetAllReachingProbabilitiesToZero()
5  for each player p
6        root.reachingProbability[p] ← 1
7  ComputeReachingProbabilities(root)
8  UpdateProbabilities(root)
```

Figure 3: One iteration of the CFR algorithm.

4 CFR

When we face an unknown game, a reasonable strategy can be the non-losing one. A non-losing strategy will not lead us to a loss if we follow it during playing the game. It would be better if the opponent cannot gain any advantage by unilaterally changing his strategy against us. If our strategy has all of these properties then we are playing according to a Nash equilibrium strategy. Therefore it is convenient to find a Nash equilibrium in a game and follow it. However, if the game is very complex (*e.g.* the state space is very large) then we can hardly compute the precise equilibrium. Instead, we can use an ϵ-Nash equilibrium strategy where ϵ is an indication of how far we are from the equilibrium. Since we will not lose anything by following a Nash equilibrium, ϵ can be considered as the amount that we will lose if it happens to play against a best response to our strategy. In fact ϵ is a measure of how exploitable we will be by following an ϵ-Nash equilibrium strategy.

CFR (CounterFactual Regret) is an algorithm for finding an ϵ-Nash equilibrium in a problem. It is currently the most efficient algorithm, which can handle the largest state spaces in comparison to other available methods [Zinkevich *et al.*, 2007]. It also has the nice property of being incremental;

meaning that the longer it runs the closer it gets to the Nash equilibrium.

Since in a Nash equilibrium no player can increase his payoff (decrease his regret) in the game by unilaterally changing his strategy, we can find the Nash equilibrium in a game by trying to tune the strategies of all players to minimize their regrets knowing what others will do. CFR uses the fact that the sum of immediate counterfactual regrets is an upper bound on the overall regret of a player. Counterfactual regret is the amount that the player will regret by taking an action in a state in comparison with the expected value he could get. Since we are considering a specific state we considers that he played to reach to that state and take that action (thus it is called counterfactual) while other players played based on their probability setting for taking different actions from an initial state to that specific state. The pseudocode for one iteration of CFR is given in Figure 3. Proofs of convergence and bounds on how close it will get to a Nash equilibrium can be found in [Zinkevich *et al.*, 2007].

CFR expands the game tree at first. However, if the whole game tree is too large to fit in memory, we only expand the tree to a certain depth. Since we need return values for different players at the leaf nodes, simulations can be done to obtain these values. At each iteration of the algorithm, CFR computes the expected value for different actions of each player at each node (lines 6-10 in Figure 4) as well as the overall expected value for each player (lines 11-12 in Figure 4). It also computes the reaching probability to each node in the tree for different players. However, as CFR deals with counterfactual regret, the probability for each player is computed as that player played to reach that node while other players have played based on their probability settings (lines 1-2 in Figure 5). Counterfactual regrets are computed using the reaching probabilities and the difference between expected values for taking a specific action versus following the current strategy (lines 1-6 in Figure 6). CFR keeps track of cumulative counterfactual regret for every action of each player at each node of the tree. Action probabilities for the next iteration are computed based on the cumulative counterfactual regret. The probabilities of all the actions which have negative regrets will be set to zero as the player is suffering by taking those actions based on the current probability settings (line 11 in Figure 6). The probabilities of the actions which have positive regrets will be set according to the value that the player regrets them (line 10 in Figure 6). However, if all the regrets are zero, then the player will switch to randomization between all of his actions according to the uniform distribution.

It should be noticed that during the computation, the game is not actually being played, but the algorithm is tuning probabilities for the players to minimize their immediate counterfactual regret. The final probabilities for taking each action are computed as the ratio between the sum of probabilities over all the iterations for taking that action and the overall sum of these sums.

We consider the game of rock-paper-scissors to illustrate how CFR works (refer to Figure 7). Assume the first player's action probabilities are $(1, 0, 0)$ (the first, second, and third arities represent the probability of playing rock, paper, and scissors respectively) and the second player's action probabilities are $(0, 1, 0)$. Considering the probability settings, the expected return for the first player playing rock will be $E(r1) = P(r2) \times goal_{P1}(r1 - r2) + P(p2) \times goal_{p1}(r1 - p2) + P(s2) \times goal(r1 - s2)^1 = 0 \times 50 + 1 \times 0 + 0 \times 100 = 0$ and for playing paper and scissors will be 50 and 100 respectively. Therefore the current expected return for the first player will be $E(P1) = P(r1) \times E(r1) + P(p1) \times E(p1) + P(s1) \times E(s1) = 1 \times 0 + 0 \times 50 + 0 \times 100 = 0$. The counterfactual regret for not playing paper by the first player will be $regret(p1) = E(p1) - E(P1) = 50 - 0 = 50$ and $regret(s1) = 100$ (obviously $regret(r1) = 0$). Updated action probabilities for the first player for the next iteration will be $(0/(100 + 50), 50/150, 100/150) = (0, 1/3, 2/3)$. Similar computations will be done for the second player and his action probabilities will be updated as well before the next iteration.

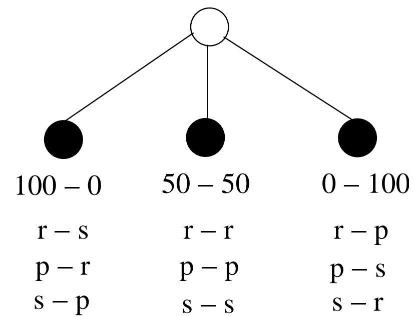

Figure 7: Rock-paper-scissors game tree for CFR. Dark nodes are terminal.

5 Using CFR in GGP

CFR was originally designed for Poker which is an imperfect information game [Zinkevich *et al.*, 2007]. Therefore in the original CFR it dealt with the concept of an information set that the state of the game can only be defined to be among a set of states. However, the only imperfect information that arises in GGP is a result of simultaneous actions taken by different players. This simplifies the use of CFR in GGP since each information set is in fact a unique state. On the other side, it is not possible to use any abstractions while dealing with games in GGP in the same way that abstraction is used in Poker to shrink the state space. Therefore, CFR must deal with a game tree that will grow linearly as the state space grows.

In GGP the player must submit his moves before a time limit is reached, therefore deciding on the size of the tree that we must deal with is critical. The smaller the tree is, the faster it will be to do an iteration over the tree and the values will converge faster. But we will have non-terminal leaves in our tree that we need a value for. We must do simulations to acquire a value to base our computation in CFR on

[1] P: Probability, $goal_{P1}$: first player's outcome, $r2$: second player playing rock, $p2$: second player playing paper, $s2$: second player playing scissors, $r1 - r2$: a state where both players play rock.

```
ComputeExpectedValues(root)
 1  for i ← 1 to number[root . children]
 2          ComputeExpectedValues(root.children[i])

 3  for each player p
 4          for each a ∈ actions(p)
 5                  root . actionExpectedValue[p][a] ← 0

 6  for i ← 1 to number[root . children]
 7          for each player p
 8                  pAct ← root . children[i]. action[p]
 9                  prob ← Π_{op≠p} root . actionProbability[op][root . children[i]. action[op]]
10                  root . actionExpectedValue[p][pAct] += prob × root . children[i]. expectedValue[p]

11  for each player p
12          root.expectedValue[p] ← Σ_{a∈actions(p)} root.actionProbability[p][a] × root.actionExpectedValue[p][a]
```

Figure 4: Computing expected values in CFR.

```
ComputeReachingProbabilities(root)
1  for i ← 1 to number[root . children]
2          root . children[i]. reachingProbability[p] +=
                   root . reachingProbability[p] × Π_{op≠p} root . actionProbability[op][root . children[i]. action[op]]
3          if AllParentsComputationAreDone(root.children[i])
4            then ComputeReachingProbabilities(root.children[i])
```

Figure 5: Computing reaching probabilities in CFR.

```
UpdateProbabilities(root)
 1  for each player p
 2          sum ← 0
 3          for each a ∈ actions(p)
 4                  root . regret[p][a] +=
                        root . reachingProbability[p] × (root . actionExpectedValue[p][a] − root . expectedValue[p]
 5                  if root . regret[p][a] > 0
 6                    then sum += root . regret[p][a]
 7          if sum > 0
 8            then for each a ∈ actions(p)
 9                          if root . regret[p][a] > 0
10                            then root . actionProbability[p][a] ← root . regret[p][a]/ sum
11                            else  root . actionProbability[p][a] ← 0
12            else  for each a ∈ actions(p)
13                          root . actionProbability[p][a] ← 1/|a ∈ actions(p)|
14          for each a ∈ actions(p)
15                  root . cfrActionProbability[p][a] += root . actionProbability[p][a]
                            ▷ Keeps track of accumulative probabilities to extract probability of actions at last
                              Final probabilities for each player will be root . cfrActionProbability[p][a] / Σ_{a∈actions(p)} root . cfrActionProbability[p][a]
```

Figure 6: Updating probabilities in CFR.

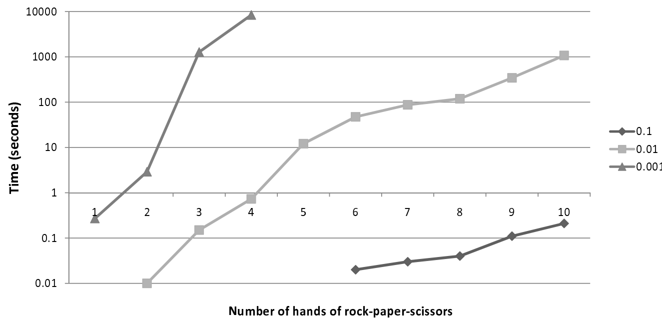

Figure 8: CFR convergence rate to different margins of Nash Equilibrium for different number of rock-paper-scissors hands.

it. The simulation trajectories are longer for smaller trees and the outcome is variable implying that we will need a higher number of simulations. Thus, although the probabilities that CFR computes will converge faster, they will be farther from the actual values that we must have converged to. The reverse is true for the larger tree. While it takes longer to converge, but we will converge to a higher quality solution. Therefore, there is a trade off between how fast we can get an stable probability setting versus how good the result will be. In addition all the simulations can be done at first or they can be done for each iteration. While the first approach is faster, the second approach will result in a better long term quality since the quality is not bounded by the simulations done at first.

In Figure 8, convergence rate of the CFR algorithm for different number of repeated rock-paper-scissors game is given. The graph uses logarithmic scaling on the vertical axis. There is only one unique Nash equilibrium in the game, which is playing each action with the probability of $1/3$ everywhere. Convergence rates are give for three different margins of the Nash equilibrium, *viz.* 0.1, 0.01, and 0.001. The experiments were done on a 2.4GHz AMD machine with the time limit of 3 minutes. Points that are missed at the bottom of the graph indicate that the computation took less than 0.01 of a second. It should be considered that more repetitions of the game results in larger tree sizes. How the computation for different tree sizes scales versus time can be seen.

Our current CFR player expands its tree using as much memory as it has. For partial trees, a number of simulations are done for each leaf node to get an estimate on its value. Currently we consider one thousand simulations at most for each leaf. In addition, if the difference after doing 25 more simulations is less than 0.1 we will cut the simulations.

6 Exploiting UCT by CFR

We discussed in section 3 that UCT will not necessarily converge to a Nash equilibrium. But if we just adhere to the Nash equilibrium while playing against UCT, we are just guaranteed to get the Nash equilibrium expected value. However, if we can model the probability distribution that UCT will converge to, we can exploit UCT and gain more than what we can gain just by following the Nash equilibrium.

For example consider the payoff matrix of a simple game

	b_1	b_1
a_1	$23, 77$	$77, 23$
a_2	$73, 27$	$27, 73$

Table 2: The payoff matrix of a simple game.

shown in Table 2. There is only one Nash equilibrium for that game with the expected value of 50 for both players. The mixed strategy action probabilities are as follows.

$$P(a_1) = 0.46, P(a_2) = 0.54$$

$$P(a_1) = 0.5, P(a_2) = 0.5$$

Suppose the first player tends to select a_2 all the time. If we just follow the mixed strategy probabilities for action selection we will only get $\frac{1}{2} \times 27 + \frac{1}{2} \times 73 = 50$ points versus the potential 73 points that we could have got if we had used our knowledge about our opponent properly.

Exploiting UCT by CFR is straightforward. We set the probabilities for the player that UCT is going to play his role in the game equal to the probabilities that we assume UCT will use to actually play the game. Then we use CFR to compute the probabilities for the player that we will play his role in the game while keeping the probabilities for the opponent (UCT) fixed. Finally we use the new probabilities to play the game. However, since it is an open question that what balanced situation UCT converges to and the distribution of probabilities is not known in advance, we do not have a best response to UCT for every game. In addition, the opponent may not be even UCT. Therefore, using the best response approach can be very brittle and can suffer greatly if the assumed model is wrong.

It is desirable to exploit a known opponent but still be close to a Nash equilibrium to not be exploitable greatly. Two approaches can be taken to exploit an opponent and still do not suffer greatly if the the model is wrong. One of them is to compute both the best response and a mixed strategy Nash equilibrium and alternate between them. We can assume different probabilities for using each of the probability distributions to achieve different levels of exploitation and exploitability. Another approach is to assume that with probability p our opponent adheres to what we assumed and with probability $1 - p$ it is a general player that tries to minimize his regret and play a Nash equilibrium. Then use this new model of the opponent to compute a mixed strategy Nash equilibrium to play the game (this new equilibrium is called restricted Nash equilibrium). Different variations of p can lead to different levels of exploitation and exploitability. In Poker the latter approach shown to be superior to the former [Johanson *et al.*, 2007]. The results for using the latter approach for Goofspiel are given in Table 3.[1] Goofspiel, also known as the game of pure strategy, is a card game for two or more players. The variant that we are considering here is a two player game with three suits of cards from ace to 5 inclusive. Each player owns a suit and the third suit is on the

[1] In these experiments our UCT general game player, which competed in 2008 GGP competition, played versus our CFR general game player described in Section 5 using 1 minute start-clock and 20 seconds play-clock on a 3.4GHz Intel machine.

Model confidence (p)	Exploitation (CFR vs. UCT)	Exploitability (CFR vs. best response)
1	$84.5 - 15.5$	$0.5 - 99.5$
0.75	$70 - 30$	$10 - 90$
0.5	$52.42 - 47.58$	$42 - 58$
0.25	$56 - 44$	$49 - 51$
0	$50.63 - 49.37$	$53 - 47$

Table 3: Exploitation vs. exploitability in 5 cards Goofspiel with $0 - 50 - 100$ goal values.

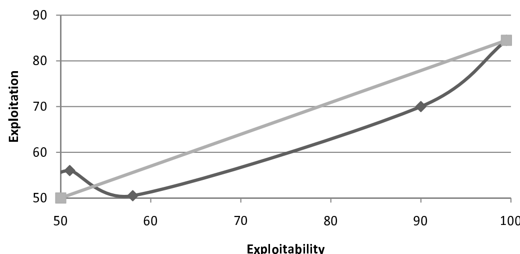

Figure 9: Exploitation vs. Exploitability.

ground in a specific order. All three suites are faced up. For convenience, we assume that the third suit is in order (from ace to 5). At each step of the game, each player selects a card in his hand and both players reveal their selected cards simultaneously. The player who has a higher card will gather the card placed on the ground from the third suit and acquire as much points as the value of the card (1 to 5 for ace to 5). Picking up cards will be done in the order they are placed on the ground (from ace to 5 in this example). If both players happen to have the same card, no one will win the card from the third suit and all the three cards will be discarded. The player with the higher points wins 100 points and the other player gets 0. A draw results in 50 points for each player.

If we consider the best response payoff ($p = 1$) and the mixed strategy Nash equilibrium, we can achieve any exploitation and exploitability tradeoff by different mixing in between. Using these two approaches, we will be in a safe margin if our model of the opponent happens to be wrong. As it can be seen in Figure 9 the line for using restricted Nash equilibrium approach is below the mixing approach for Goofspiel. Therefore it is better to use the mixing approach at least in this game.

7 Conclusion and Future Work

We analyzed how UCT plays in a simultaneous move game and gave a counter example that UCT does not converge to a Nash equlibrium, although it converges to a balanced situation which can be exploited. We showed that CFR can be used in GGP and can be used to exploit UCT if a model of it is available.

As the future work, we are working to define the characteristic of the balanced situation that UCT converges to. In addition, we are considering different ways of tree expansion to be used in CFR that can improve the quality of probabilities being computed using the partial tree.

References

[Auer *et al.*, 2002] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

[Clune, 2007] James Clune. Heuristic evaluation functions for general game playing. In *AAAI*, pages 1134–1139, 2007.

[Coulom, 2006] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games*, pages 72–83, 2006.

[Finnsson and Björnsson, 2008] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI*, pages 259–264, 2008.

[Gelly *et al.*, 2006] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, France, November 2006.

[Johanson *et al.*, 2007] Michael Johanson, Martin Zinkevich, and Michael Bowling. Computing robust counter-strategies. In John C. Platt, Daphne Koller, Yoram Singer, and Sam T. Roweis, editors, *NIPS*. MIT Press, 2007.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, pages 282–293, 2006.

[Kuhlmann and Stone, 2006] Gregory Kuhlmann and Peter Stone. Automatic heuristic construction in a complete general game player. In *AAAI*, pages 1457–62, 2006.

[Love *et al.*, 2006] Nathaniel Love, Timothy Hinrichs, and Michael R. Genesereth. *General Game Playing: Game Description Language Specification*. Stanford Logic Group, Computer Science Department, Stanford University, http://games.stanford.edu/gdl_spec.pdf, April 2006.

[Schiffel and Thielscher, 2007] Stephan Schiffel and Michael Thielscher. Automatic construction of a heuristic search function for general game playing. In *IJCAI Workshop on Nonmontonic Reasoning, Action and Change (NRAC07)*, Hyderabad, India, 2007.

[Sturtevant, 2008] Nathan R. Sturtevant. An analysis of UCT in multi-player games. In *Computers and Games*, pages 37–49, Berlin, Heidelberg, 2008. Springer-Verlag.

[Zinkevich *et al.*, 2007] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In *NIPS*, 2007.