Extensive Games

Oleg Arenz
o.arenz@gmx.de

Abstract

This paper provides an overview of different approaches for handling extensive games. It focuses on methods which are able to train game programs in the absence of domain knowledge and shows that such methods can be used successfully in Poker, Chess and Go. Counterfactual regret minimization is able to approximate a Nash equilibrium in heads-up limit Texas Hold'em. Search Bootstrapping can reach the strength of a strong amateur chess player by tuning the heuristic function towards the result of a tree search of a given depth. Monte-Carlo Tree Search uses random simulations to find the most promising move in Go. It can be observed that the performance of these methods heavily depends on game type.

1 Introduction

Techniques which allow machines to play games with human-like strength - or even better - can help create autonomous systems capable of fulfilling complex tasks in real world scenarios. Extensive games and games with incomplete information are particularly interesting for AI research as they demand making decisions which can not be proven best due to computation limitations or lack of information and thus reflect problems which are often encountered in practice. Additionally, as domain knowledge might be unavailable or incomplete, machine learning might be required. Several different approaches proved successful in increasing the playing strength through self learning in complex games as chess, go or poker. It could be observed that the effectiveness of these approaches is depending on the type of the game.
A common approach in the game of poker is to find a strategy for each player so that no player would profit by changing his strategy if the other players stick to theirs. Such a situation is called Nash equilibrium. A Nash equilibrium can be approximated through self play by minimizing the regret of each player independently. In chapter 2 a concept called counterfactual regret minimization is explained which decomposes regret into different additive regret terms which can be minimized independently.
In the game of chess Minimax search is the fundamental concept used by all of the strongest chess programs. Minimax requires a heuristic evaluation function which usually assigns a positive score for positions which are believed to favor the white player and a negative score for positions which are believed to favor the black player. Bigger absolute values signify bigger advantages for the corresponding side. Minimax first creates a game tree with a fixed depth and uses the heuristic to evaluate the positions of the leaf nodes. Then the score of the leaf node which is reached by always choosing the move that maximizes the minimum guaranteed score if white is to move and minimizes the maximum guaranteed score if black is to move is used as the evaluation of the root position. Alpha-beta search significantly improves Minimax by storing bounds for both the guaranteed maximum score of the black player and the guaranteed minimum score of the white player and uses this information to cut off branches which can not be reached by best play from both sides. A method called search bootstrapping can be applied to tune the heuristic function towards the result of a standard minimax or alpha-beta search of a given depth. Minimax search, alpha-beta search and search bootstrapping are presented in chapter 3.
In the game of go, random simulations are used to find the move that statistically scores best in a

given position. Chapter 4 presents such a sampling based approach called Monte-Carlo Tree Search and an improvement to it called UCT. UCT was used to create the strong go engine MoGo [5].

2 Counterfactual Regret Minimization

One way to handle the large amount of game states in extensive games is by using abstractions. Different game states are grouped together and every state of the same group is treated the same way. By reducing the number of game states which have to be considered these abstractions become solvable. The solution of an abstract game is in general better for the actual game, the fewer game states were grouped together. In [11] the concept of Counterfactual Regret Minimization was introduced and shown to solve 10^{12} game states in the game of poker, which is by two orders of magnitude larger than previous methods. Before explaining this concept, the following terms have to be formally defined:

- *Information Set:* An information set I_i is a set of non-terminal sequences of actions with the property, that for every sequence within the same set, player i is to act next and has the same actions available. All information sets are pairwise disjoint.
- *Strategies:* A strategy of player i, σ_i, assigns for each information set I_i a probability distribution over the available actions of I_i. Σ_i is the set of strategies of player i. A strategy profile σ assigns a strategy to each player. The strategy profile σ_{-i} assigns a strategy to each player except player i.
- *Utility:* The utility function $u_i(z)$ from a terminal state $z \in Z$ to the reals R assigns for each player $i \in N$ their winnings. $\sum\limits_i u_i(z) = 0$, in a zero-sum game.
- *Nash Equilibrium:*
  A Nash equilibrium of a two-player game is a strategy profile σ where

$$u_1(\sigma) \geq \max_{\sigma_1' \in \Sigma_1} u_1(\sigma_1', \sigma_2)$$

$$u_2(\sigma) \geq \max_{\sigma_2' \in \Sigma_2} u_2(\sigma_1, \sigma_2').$$

  An ϵ-Nash equilibrium in a two player game is a strategy profile σ where

$$u_1(\sigma) + \epsilon \geq \max_{\sigma_1' \in \Sigma_1} u_1(\sigma_1', \sigma_2)$$

$$u_2(\sigma) + \epsilon \geq \max_{\sigma_2' \in \Sigma_2} u_2(\sigma_1, \sigma_2').$$

- *Regret:* If player i uses strategy σ_i^t on round t the average overall regret of player i at Time T is:

$$R_i^T = \frac{1}{T} \max_{\sigma_i^* \in \Sigma_i} \sum_{t=1}^{T} (u_i(\sigma_i^*, \sigma_{-i}^t) - u_i(\sigma^t))$$

  An algorithm is regret minimizing for player i if his average overall regret goes to zero as t goes to infinity. Regret minimizing algorithms can be used to approximate Nash equilibrium.

2.1 Counterfactual Regret

The idea of counterfactual regret minimization is to minimize regret terms for each information set independently. In order to achieve this goal immediate counterfactual regret $R_{i,imm}^T(I)$ was introduced, which can be minimized by changing only $\sigma_i(I)$. Define counterfactual utility $u_i(\sigma, I)$ to be the expected utility, if all players play according to σ except that player i is playing to reach I. For all available actions a in I define $\sigma|_{I \to a}$ to be identical to σ except that player i always choses action a when I is reached. Furthermore, let $\pi_{-i}^{\sigma^t}(I)$ be the probability that information set i

is reached if all players except player i, by playing according to strategy σ^t, would always choose actions leading to information set I. Then the immediate counterfactual regret is defined as

$$R^T_{i,imm}(I) = \frac{1}{T} \max_{a \in A(I)} \sum_{t=1}^{T} \pi^{\sigma^t}_{-i}(I)(u_i(\sigma^t|_{I \to a}, I) - u_i(\sigma^t, I)).$$

This is the player's regret by playing according to σ^t in I instead of choosing the best action, weighted by the probability that I would be reached in that round if he had tried to do so. Counterfactual regret $R^T_i(I, a)$ is defined for every information set I and for all actions $a \in A(I)$ as

$$R^T_i(I, a) = \frac{1}{T} \sum_{t=1}^{T} \pi^{\sigma^t}_{-i}(I)(u_i(\sigma^t|_{I \to a}, I) - u_i(\sigma^t, I)).$$

The following theorems demonstrate both that counterfactual regret can be minimized and that counterfactual regret minimization also minimizes regret.
Let $\Delta_{u,i} = \max_z u_i(z) - \min_z u_i(z)$ be the range of utilities to player i and let $R^{T,+}_i(I, a) = \max(R^T_i(I, a), 0)$ be the positive portion of counterfactual regret. Let $|\mathcal{I}_i|$ be the amount of information sets I_i. Finally let $|A_i|$ be the maximum amount of actions available to player i in any information set.

Theorem 1.

$$R^T_i \leq \sum_{I_i} R^{T,+}_{i,imm}(I_i)$$

Theorem 2. *If player i selects actions according to*

$$\sigma^{T+1}_i(I)(a) = \begin{cases} \frac{R^{T,+}_i(I,a)}{\sum_{a \in A(I)} R^{T,+}_i(I,a)} & \textit{if } \sum_{a \in A(I)} R^{T,+}_i(I, a) > 0 \\ \frac{1}{A(I)} & otherwise \end{cases}$$

then $R^T_{i,imm}(I) \leq \Delta_{u,i}\sqrt{|A_i|}/\sqrt{T}$ and consequently $R^T_i \leq \Delta_{u,i}|\mathcal{I}_i|\sqrt{|A_i|}/\sqrt{T}$

For both theorems proof is given by the original paper [11].

2.2 Application to Poker

Counterfactual regret minimization was applied to heads-up limit Texas Hold'em (zero-sum with four rounds of cards dealt and four rounds of betting. This variant of poker has still about 10^{18} different game states. It was shown in [11] that it was possible to solve abstractions with 10^{12} game states, which is by two orders of magnitude larger than previous methods. The resulting poker playing program was able to beat all competitors from the bankroll portion of the 2006 AAAI Computer Poker Competition [12].

3 Search Bootstrapping

A common method to incorporate domain knowledge in game playing programs is to create an evaluation function which tries to calculate the outcome of the game by examining only the current game state. The game of chess has three possible outcomes: White wins, Black wins or the game is drawn. An evaluation function, which could calculate the outcome perfectly would be able to solve the game by only evaluating the resulting positions for each move in the current game state. In Chess finding such a perfect evaluation function is too difficult in practice, as the amount of needed domain knowledge is too large. Instead, heuristic evaluation functions are used, which usually estimate the advantage of White by assigning a score to the position in centipawns. A score of -100 centipawns corresponds to the advantage black would have if the position was equal, besides that he would have an extra pawn. The heuristic value can be calculated by a linear combination of weighted features of the position. Important features in chess are material balance, activity of the pieces, king safety and the strength of the pawn formation. Although the strongest chess programs use heavily tuned heuristic functions, these heuristics would not suffice to find reasonable moves by only evaluating the next resulting positions. Nevertheless these chess programs can beat even the strongest human chess players by using the heuristic function for a depth-limited game tree search. Minimax and Alpha-Beta are the two fundamental algorithms for this kind of tree search. After explaining both algorithms we will look at a concept called search bootstrapping, which tunes the heuristic function towards a Minimax or Alpha-Beta search of a given depth D.

3.1 Minimax Search

Choosing the move only on the basis of the heuristic evaluations of the next positions did not give satisfying results, as the heuristic functions do not provide sufficiently accurate evaluations, especially due to tactical combination. Minimax [3] search evaluates all positions which could arise after a certain number of moves. Therefore it creates a game tree with a depth D, which corresponds to this number of moves. Minimax is able to find the path of best play by both player in respect to the leaf node evaluations. The algorithm works by propagating the leaf node evaluations to the root node layer-wise. For all nodes within the same layer the same side is to move. If white is to move for the current layer, Minimax assigns to each node within the layer the maximum of the scores of its children if black is to move the minimum is used respectively. Therefore White is called the maximizing player and Black is called the minimizing player. The value, which was propagated to root node can be used as the evaluation of the root position; the path taken to propagate this value, is the path of best play by both sides.

3.2 Alpha-Beta Search

Alpha-Beta [9] search is a crucial improvement of Minimax and is used by all of the strongest chess engines currently available. Alpha-Beta always finds the same path as Minimax but does not necessarily need to evaluate all nodes in the game tree. The fundamental idea of the algorithm is as follows: Imagine that for a given node in the game tree the side to move was able to reach the score s_1 by playing move m_1. If, by examining a different move m_2 from the same node, Minimax finds a reply leading to score s'_2 which is worse for the side to move than s_1, the move m_2 can be discarded without examining other replies. Alpha-Beta uses two additional variables than Minimax: α is initially set to $-\infty$ and updated during search to the minimum score white is able to reach, β is initially set to ∞ and updated to the maximum score black needs to allow. Alpha-Beta profits from good move ordering; if strong moves are examined earlier, the α-β-window shrinks faster leading to more cut-offs.

3.3 Search Bootstrapping

Search Bootstrapping [10] tunes the heuristic function towards the result of a Minimax or Alpha-Beta search of depth d. If the evaluation of the tuned heuristic function was able to assign to every position the same score as the Minimax search, a depth k search with the tuned heuristic would produce the same results as a depth $k + D$ search with the original heuristic. In theory a search with the tuned heuristic could be used to further tune the heuristic function, and so on. However, tuning a heuristic function to exactly match the result of a tree search is an unrealistic goal, as the search can not be applied to every possible game state in an extensive game. Furthermore, the possible parameter settings of the heuristic function are limited. Nevertheless, the concept of search bootstrapping was successfully used to train a chess engine called Meep up to the strength of strong human chess player by self-play [10].

Meep was used with two different search bootstrapping algorithms, RootStrap and TreeStrap, both algorithms can be used with Minimax and Alpha-beta search. The explanation of these algorithms will use the following definitions:

- A heuristic function $H_\theta(s)$ assigns a value to a state s, based on a parameter vector θ.
- $V^D_{s_0}(s)$ denotes the value of state s from a minimax search of depth D which started in state s_0.
- $\overset{\theta}{\leftarrow}$ notates a backup that updates a heuristic function towards some target value.

3.3.1 RootStrap

The RootStrap algorithm updates the heuristic value of the root node towards a depth D Minimax search:

$$H_\theta(s_t) \overset{\theta}{\leftarrow} V^D_{s_t}(s_t)$$

The parameters are updated by gradient descent on the squared error between the heuristic value and the Minimax search value:

$$\delta_t = V^D_{s_t}(s_t) - H_\theta(s_t)$$
$$\Delta\theta = -\frac{\eta}{2}\nabla_\theta \delta_t^2 = \eta\delta_t\nabla_\theta H_\theta(s_t)$$

where η is a step-size constant. By using a heuristic function of the form $H_\theta(s) = \phi(s)^T\theta$, with the feature vector $\phi(s)$ and the weighting vector θ, the equation can be further simplified to:

$$\Delta\theta_t = \eta\delta_t\phi(s_t)$$

This algorithm needs no modifications to work with Alpha-Beta search instead of Minimax.

3.3.2 TreeStrap

TreeStrap updates the heuristic function for all nodes of the Minimax search tree, instead of only updating the root node:

$$H_\theta(s) \overset{\theta}{\leftarrow} V^D_{s_t}(s), \forall s \in T^D_{s_t}$$

where $T^D_{s_t}$ is the set of all states within the game tree of a depth D Minimax search starting in state s_t.
The following update function was used for TreeStrap(minimax):

$$\delta_t(s) = V^D_{s_t}(s) - H_\theta(s)$$
$$\Delta\theta = -\frac{\eta}{2}\nabla_\theta \sum_{s\in T^D_{s_t}} \delta_t(s)^2 = \eta \sum_{s\in T^D_{s_t}} \delta_t(s)\phi(s)$$

The TreeStrap algorithm was also modified to work with Alpha-Beta search by using a one-sided loss function and by exploiting transposition tables. This modification is omitted here for simplicity, but can be found in the original paper [10].

3.4 Application to Chess

TreeStrap($\alpha\beta$), TreeStrap($minimax$) and RootStrap($\alpha\beta$) were also tested in [10]. The algorithms were implemented in Meep, a modification of the tournament chess engine Bodo. For testing purposes the parameters of the heuristic function were initially set to small random numbers and then updated by self-play. After training a freely available program called BayesElo was used to compute maximum likelihood Elo ratings. TreeStrap($\alpha\beta$) performed best with a playing strength of approximately 2157 Elo. TreeStrap($minimax$) was able to reach approximately 1807 Elo and RootStrap($\alpha\beta$) reached about 1362 Elo. The playing strength of an untrained heuristic with random parameters was estimated to be 250 Elo.

4 Monte-Carlo Tree Search

Monte-Carlo methods in game playing do not depend on a heuristic function as they use the results of randomly simulated games to evaluate possible moves. A basic way of using the Monte-Carlo method for game playing could have the following characteristics:

- All possible moves in the root position are chosen equally often.
- The resulting positions are simulated randomly and the outcome of each simulation is used to update the empirical winning rate of the corresponding first move.
- When the algorithm stops, the move with the best winning rate is played.

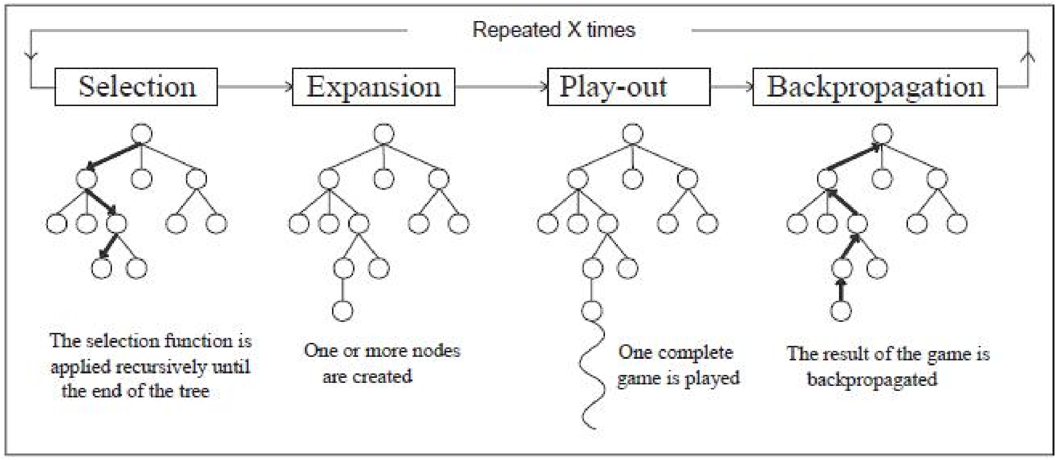
Repeated X times
Selection
Expansion
Play-out
Backpropagation
The selection function is applied recursively until the end of the tree
One or more nodes are created
One complete game is played
The result of the game is backpropagated

Figure 1: The four steps of Monte-Carlo Tree Search. Diagram taken from ”Cross-Entropy For Monte-Carlo Tree Search”, Chaslot et.al 2008 [6].

Although this algorithm is very basic it shows two benefits of Monte-Carlo methods in game playing:

1. Besides the game rules, no further domain knowledge is required.
2. The algorithm is anytime, i.e. whenever it is stopped it can suggest a move by making full use of all simulations done so far.

However, in this basic form the method has two main drawback. The first drawback is actually a critical weakness: imagine a move m_1 which leads to a position with ten different moves available to the opponent. Further image, that for nine of those replies the opponent would be losing immediately and by playing the remaining reply the opponent would immediately win. After sufficient iterations the basic Monte-Carlo method would estimate a winning chance of about 90% for m_1. When playing against a good opponent, however, m_1 would always lose. The second drawback results from the fact, that it is not optimal to spend as much time in examining bad moves as in examining good moves.
A more sophisticated way to apply the Monte-Carlo method to game playing is called Monte-Carlo Tree Search (MCTS) [4]. MCTS can solve these drawbacks while keeping the mentioned benefits of the basic Monte-Carlo method. In the game of go, programs based on MCTS turned out to outperform programs based on Minimax. The MCTS algorithm gradually adds nodes to the game tree in an asymmetric best-first manner and stores for each node the number of visits and the overall utility. In the following explanation of the algorithm the score reached by the white player is used as the utility for all nodes and therefore the utility is independent of the side to move. Hence, the best scoring move in a position is to be understood as the move with the least utility if Black is to move. This is similar to the Minimax approach described above. However, because we store the utilities for every node independently - opposed to Minimax, where initially only the leaf nodes store a utility - we can make the correct decisions by just comparing the utilities of the child nodes. MCTS can be divided into four different phases:

1. *Select:* The selection phase starts in the root node and recursively chooses a child node until a node is reached, which has more valid moves than edges in the current game tree. The selection strategy should prefer moves which scored well in the previous iterations over moves which scored badly (exploitation). On the other hand it should prefer moves which have been visited less often over moves which have been visited more often (exploration) in order to estimate their real score better. Both goals are contradicting, which is called the exploration-exploitation dilemma in the multi-armed bandit problem [2]. A good strategy to solve this dilemma is shown in the section about UCT.
2. *Expand:* After a node was selected it is expanded by adding a child node to it. The child node represents the resulting position after making a random legal move which was not yet represented in the game tree.

3. *Simulate:* The position corresponding to the newly created child is simulated, by making random moves until a terminal game state is reached.
4. *Update:* After the simulation, all nodes visited in the selection phase as well as the node which was created in the expansion phase are updated by increasing their number of visits by 1 and increasing their overall utility depending on the result of the simulation. In case of a win by White the overall utility is increased by 1; in case of a draw it is increased by 0,5; in case of a win by Black the overall utility remains unchanged.

MCTS prefers moves which scored well over moves which scored badly and therefore creates an asymmetrical game tree, where more time is spent in analyzing good moves.
If a move allows the opponent to play a strong reply, this reply will be selected more often than other replies and the score of the initial move will therefore decrease.

4.1 UCT

The idea of applying an algorithm from the multi-armed bandit problem [2] for Monte-Carlo Tree Search was examined in [7]. The bandit algorithm UCB1 was used to formulate a new algorithm called UCT (UCB applied to trees). UCT selects the next child, based on the following formular:

$$i_{chosen} = \underset{i \in 1,\ldots,K}{\arg\max}\left(\frac{u_i}{n_i} + \sqrt{\frac{2 \ln n}{n_i}}\right)$$

where u_i is the utility of child i, n_i is the number of visits of child i, n is the number of iterations done so far (which equals the number of visits of the root node), K is the number of childs of the current node and i_{chosen} is the chosen child.

4.2 Application in Go

UCT was used to create the top level Go program Mogo [5]. The most notable improvements used to create Mogo are:

- Mogo uses UCB1-TUNED [1] instead of UCB1.
- Domain knowledge was used to create more meaningful simulations by using patterns to find local answers to the last played move.
- Pruning was used to reduce the tree size.

4.3 Future Work

UCT did not yield as good results in Chess as in Go. [8] identifies shallow tactical threats, which occur more often in Chess than in Go, as one reason for the bad performance of UCT in Chess. However, creating a chess program based on UCT which uses the improvements used by the strongest UCT based Go engines, might provide further insights into the strengths and weaknesses of both approaches. An interesting idea to improve UCT for chess programs is by exploiting the fact that chess endgames with only few pieces left have already been solved. N-piece endgame tablebases can be queried to get the best-play result of any chess position with N pieces left on the board. While Minimax is not able to make big use of endgame tablebases if two many pieces are remaining, UCT would profit by shorter simulations. Therefore, the simulations would be both faster (assuming that querying the tablebase is faster than simulating the endgame) and more accurate (as the N-piece endgame could not be misplayed in the simulation).

5 Conclusion

Due to their large state space, extensive games can not be solved by exhaustive search. Nevertheless strong computer players have been already developed for Poker, Chess and Go. This paper focused on approaches which can be used to increase the playing strength of computer programs by self-play. Counter-factual regret minimization is able to minimize overall regret by minimizing the regret for each information set independently. It uses self-play to approximate an Nash-equilibrium in Poker.

Search Bootstrapping is able to find good parameters for a heuristic function by shifting the heuristic towards a D-depth Minimax or Alpha-Beta Search. This method was used by the chess engine Meep which was able to reach playing at the strength of a strong club player by self-play. Monte-Carlo Tree Search simulates possible game continuation in order to find the most promising move. The go engine Mogo was able to beat all previous go engines by employing Monte-Carlo Tree Search. Interestingly, the performance of these methods heavily depends on the game they are used for. It is remarkable, that although Chess and Go are both two-player games with full information, Monte-Carlo Tree Search could not be used to create a good chess engine. On the other hand, Minimax did not lead to good go engines. The reasons for theses difference are not fully explored yet.

References

[1] AUER, P., CESA-BIANCHI, N., AND FISCHER, P. Finite-time analysis of the multiarmed bandit problem. *Machine learning 47*, 2 (2002), 235–256.

[2] AUER, P., CESA-BIANCHI, N., FREUND, Y., AND SCHAPIRE, R. Gambling in a rigged casino: The adversarial multi-armed bandit problem. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on* (1995), IEEE, pp. 322–331.

[3] BEAL, D. *The nature of minimax search.* Van Spijk, 1999.

[4] COULOM, R. Efficient selectivity and backup operators in monte-carlo tree search. *Computers and Games* (2007), 72–83.

[5] GELLY, S., WANG, Y., MUNOS, R., TEYTAUD, O., ET AL. Modification of uct with patterns in monte-carlo go.

[6] GUILLAUME, M., WINANDS, M., SZITA, I., AND VAN DEN HERIK, H. Cross-entropy for monte-carlo tree search. *ICGA Journal* (2008).

[7] KOCSIS, L., AND SZEPESVÁRI, C. Bandit based monte-carlo planning. *Machine Learning: ECML 2006* (2006), 282–293.

[8] RAMANUJAN, R., SABHARWAL, A., AND SELMAN, B. On adversarial search spaces and sampling-based planning. *20th ICAPS* (2010), 242–245.

[9] RICHARDS, D., AND HART, T. The alpha-beta heuristic.

[10] VENESS, J., SILVER, D., UTHER, W., AND BLAIR, A. Bootstrapping from game tree search. *Advances in Neural Information Processing Systems 22* (2009), 1937–1945.

[11] ZINKEVICH, M., JOHANSON, M., BOWLING, M., AND PICCIONE, C. Regret minimization in games with incomplete information. *Advances in Neural Information Processing Systems 20* (2008), 1729–1736.

[12] ZINKEVICH, M., AND LITTMAN, M. The aaai computer poker competition. *Journal of the International Computer Games Association 29* (2006).