Counterfactual Regret Minimization on GPU

Jan Rudolf

January 17, 2021

0.1 Introduction

My project is parallelizing the Counterfactual Regret Minimization (CFR) algorithm [3] from non-cooperative game theory for NVIDIA graphic cards with CUDA. My motivation to use this algorithm stemmed from my part-time job at the Computational Game Theory research group at the Artificial Intelligence Center within CTU. We are using CFR implemented for CPU, so I was interested in implementing this algorithm's parallel version. We might very likely use the parallel version in the future.

0.2 Quick Game Theory Background

Non-cooperative game theory is formalizing strategic interaction between rational, self-interested players (decision makers). Rational is a player who always chooses the action with maximal expected utility. Self-interested means the player is not maximizing other players' utility but his own. Therefore we talk about non-cooperative game theory. Further, in the text, I will use game theory in the meaning of non-cooperative game theory.

The game theory distinguishes two basic types of games. In games with perfect-information, all players know or observe the actual state of the game. On the other hand, players do not know or observe the game's full state in imperfect-information games. Chess is an example of a perfect information game. Both players know the state of the game from the game's board. Poker is an imperfect information game. Every player doesn't know of the cards of his opponent. That is why the player can be in multiple different states.

Let's use a simple Rock-Paper-Scissors (RPS) game as a demonstrative example. A player in the RPS game has three actions Rock, Paper, and Scissors. Both players are choosing their actions simultaneously. Based on their decisions and the game's rules, they know who wins, loses, or a draw.

1 \ 2		**Rock**	**Paper**	**Scissors**
	σ	0	1	0
Rock	0	$(0,0)$	$(-1,1)$	$(1,-1)$
Paper	0	$(1,-1)$	$(0,0)$	$(-1,1)$
Scissors	1	$(-1,1)$	$(1,-1)$	$(0,0)$

Table 1: The table shows Rock-Paper-Scissors game in a normal-form.

Table 1 shows the RPS game as a normal-form game (NFG). NFG is a game formalism described by a triple (N, A, u), where N is a set of players, A is a set of actions, and u is a utility function. Theory and methods are the most developed for two-player games so that the set N will have a player one and player two. The utility function has the form $u : A_1 \times A_2 \mapsto \mathbb{R}_1 \times \mathbb{R}_2$. If player one chooses the action Scissors, and player two chooses the action paper, then the utility function returns $(1, -1)$, that is, 1 for player one and -1 for player two. We call zero-sum games whose utility function across players sums to 0.

The unknown in the game theory is the players' strategy. Let's denote the player's one strategy σ_1 and the player's two strategy 2 is σ_2. Strategy profile is a tuple $\sigma = (\sigma_1, \sigma_2)$. Table 1 shows the player two has a deterministic strategy $\sigma_2 = (0, 0, 1)$. Best response strategy is a strategy that maximizes a player's utility given opponent's strategy. Let's use player's two strategy σ_2 as the opponent's strategy for player 1. Best response strategy of player 1 is $BR_1(\sigma_2) = (0, 0, 1)$. Scicssors is the best reponse to Paper. The set of outcomes players come obtain are called solution concepts. Nash equilibrium it the most famous and used one. Nash equilibrium is a strategy profile σ^* for the optimal worst-case strategy, where both players are playing best responses, $\sigma^* = (BR_1(\sigma_2^*), BR_2(\sigma_1^*))$, to each other, and no player has an incentive to deviate, because the player would not increase his utility. The Nash equilibrium for RPS is the strategy profile $\sigma^* = ((\frac{1}{3}, \frac{1}{3}, \frac{1}{3}), (\frac{1}{3}, \frac{1}{3}, \frac{1}{3}))$.

An extensive-form game (EFG) is the formalism for sequential dynamic games with more than one move per player. See Figure 1 for the RPS game in the EFG formalism, which you can visualize as a game tree. An imperfect-information EFG is formally a tuple $(N, A, H, Z, \chi, \rho, \sigma, u, I)$. N is a finite set of players. A is a finite set of actions. H is a set of non-terminal history (internal nodes in the game tree). χ assigns actions to a non-terminal history. ρ assigns a player to each non-terminal history. σ is the successor function, which maps a non-terminal history and an action to a terminal of non-terminal history (to an internal node or a leaf node of a game tree). u is the utility function again, which assigns utilities to players in terminal histories (leaf nodes in the game tree). I is a set of factored histories based on an equivalence relation being a history of the same player and having the same actions.

The Nash equilibrium solution concept is also applicable to EFG games. The difference is that players' strategies are defined on information sets (NFG is equivalent to EFG with one information set per player). From now on, I use games in the EFG formalism.

0.3 Counterfactual Regret Minimization (CFR)

Counterfactual Regret Minimization (CFR) is an iterative, anytime algorithm, which takes a game in the EFG formalism as input and produces an average strategy that approximates Nash equilibrium. The parameter of the algorithm is the number of iterations. Since the algorithm is anytime, CFR produces valid output on every iteration. More iterations mean better approximation.

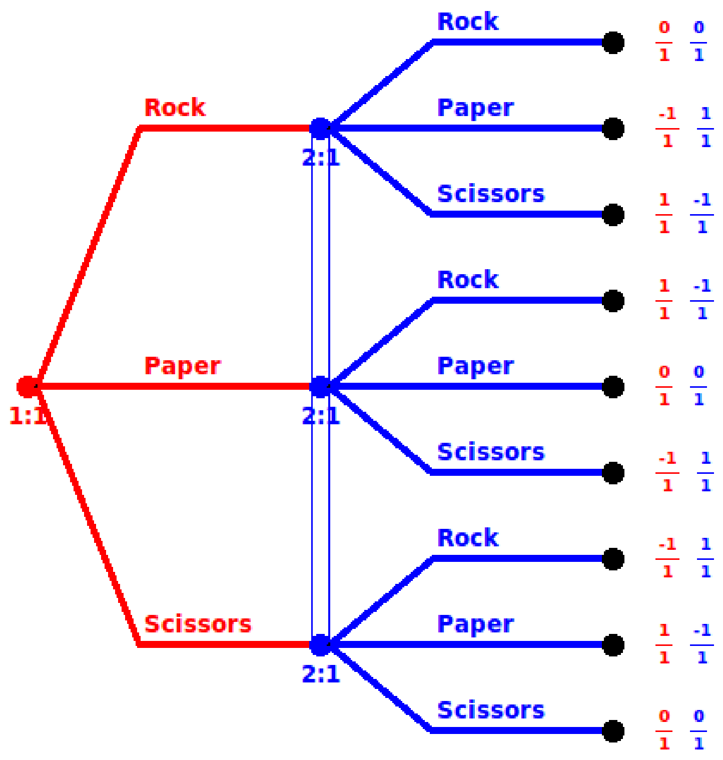

Figure 1: Rock-Paper-Scissors in the EFG formalism. Nodes (states/decision points), edges (actions), and utilities that are red (blue) belongs to the player 1 (2). There are two information sets. The player's 1 information set is 1:1. The player's 2 information set is 2:1. Note that the information set contains 3 nodes. This is because the player 2 does not know what the player 1 is going to play. The player 2 can end up in the one of those three states, which he cannot distinguish. All nodes within one information sets have the same actions and the same strategy.

The CFR algorithm maintains two vectors for each information set I throughout the iterations. The first vector is regret $R(I)$. The second vector is the average strategy $\overline{\sigma}(I)$. The sizes of the vectors are the same as the number of actions in the information set. It is valid because all nodes in an information set have the same number of actions and the same strategy. Therefore, we can talk about the strategy of the information set. CFR initializes the vectors to zero vectors, then proceeds with four conceptual steps. This description is adopted from [1].

For iteration t from T:

1. CFR computes the current strategy $\sigma^t(I)$ from regrets $R(I)$.
2. Adds the current strategy $\sigma^t(I)$ to the average strategy $\overline{\sigma}(I)$ weighted by reach probabilities.
3. It calculates counterfactual values $v^t(I)$ based on reach probabilities, the current strategy, and the utility function.
4. Updates $R(I)$ regrets from computed counterfactual values and the current strategy.

I described these steps at a high level. Let's breakdown each step more formally.

For iteration t from $1, 2, ..., T$:

1. For each information set I, each action $a \in A(I)$, player $i = p(I)$:

$$\sigma_i^t(I, a) = \begin{cases} R^t(I, a)^+ / \sum_{b \in A(I)} R^t(I, b)^+ & \text{if } \sum_{b \in A(I)} R^t(I, b)^+ > 0 \\ \frac{1}{|A(I)|} & \text{otherwise} \end{cases}$$

2. For each information set I, each action $a \in A(I)$, player $i = p(I)$:

$$\overline{\sigma}_i^t(I, a) = \frac{1}{t} \sum_{t'=1}^{t} \pi_i^{\sigma^t}(I) \sigma_i^t(I, a) = \frac{t-1}{t} \overline{\sigma}_i^{t-1} + \frac{1}{t} \pi_i^{\sigma^t}(I) \sigma_i^t(I, a)$$

3. For each information set I, each action $a \in A(I)$, player $i = p(I)$:

$$v_i^{\sigma^t}(I, a) = \sum_{h \in I \cdot a} v_i^{\sigma^t}(h) = \sum_{h \in I \cdot a} \sum_{z \in Z, h \sqsubset z} \pi_{-i}^{\sigma^t}(h) \pi^{\sigma^t}(z|h) u_i(z)$$

4. For each information set I, each action $a \in A(I)$, player $i = p(I)$:

$$R^{t+1}(I, a) = R^t(I, a) + v_i^{\sigma^t}(I, a) - \sum_{b \in A(I)} \sigma^t(I, b) v_i^{\sigma^t}(I, b)$$

From EFG's definition, history $h \in H \cup Z$ is a sequence of actions starting in the game tree's root. Talking about history h is equivalent to talking about the game tree's node h. Every internal node (leaf) is non-terminal (terminal) history from H (Z).

Reach probability of history $h \in H \cup Z$, used in the 3rd CFR's step, is a product of the sequence's actions' probabilities based on their current strategy. Formally, reach probability $\pi_i^{\sigma^t}(h)$ of n length history $h = (a_1, a_2, ..., a_n) = ((I_1, a_1), (I_2, a_2), ..., (I_n, a_n))$ of player i based on the current strategy σ in iteration t is:

$$\pi_i^{\sigma^t}(h) = \prod_{j=1}^{n} \begin{cases} \sigma^t(I_j, a_j) & \text{if } i = p(I_j) \\ 1 & \text{otherwise} \end{cases}$$

the same reach probability excluding player i is denoted $\pi_{-i}^{\sigma^t}(h)$:

$$\pi_{-i}^{\sigma^t}(h) = \prod_{j=1}^{n} \begin{cases} \sigma^t(I_j, a_j) & \text{if } i \neq p(I_j) \\ 1 & \text{otherwise} \end{cases}$$

Without specifying the player, we include all players $\pi^{\sigma^t}(h) = \pi_i^{\sigma^t}(h)\pi_{-i}^{\sigma^t}(h)$. Reach probability of information set I is $\pi^{\sigma^t}(I) = \sum_{h \in I} \pi^{\sigma^t}(h)$. The term $\pi^{\sigma^t}(z|h)$ means reach probability from history h to history z, or equivalently a sequence of actions reaching the node z from node h. Reach probability of the game's tree root node is 1 (equals history of length 0). Finally, $h \sqsubset z$ for histories h and z means that the sequence h is a prefix of sequence z. So in CFR's step 3, $\pi^{\sigma^t}(z|h)u_i(z)$ is the expected utility of the history h weighted by opponent's reach probability $\pi_{-i}^{\sigma^t}(h)$.

0.4 CFR on CPU

I used the CPU implementation I have worked with before in GTLib. The CFRAlgorithm class (algorithms/cfr.cpp) implements the CFR algorithm. It is using the method runIteration called recursively from the root of a game tree. The method traverses the tree in a depth-first search manner. It computes the reach probabilities during the tree traversal and carries them as a parameter. The runIteration outputs counterfactual values to the parent EFG node. Unordered maps implement information sets, where are regret vectors, current strategies, and average strategies. There are too many components in GTLib that I do not see a reason to describe more.

0.5 CFR on GPU

First, I wanted to integrate the GPU implementation into our GTLib framework written in C++, but I ran into C++ incompatibility issues. In the GTLib, we are using too latest C++ version and advanced inheritance I cannot operate on CUDA. I solved the problem by making the GPU implementation a standalone source code. I did not want to reinvent the wheel, so I used main_gpucfr_export.cpp in GTLib to export game trees as text files. The exported game in a text file contains in a depth-first search manner generated EFG nodes from the tree's root. Every node has there its information about the player, the number of actions, parent hash, children hashes, information set hash, and node's value (for terminal nodes).

The GPUCFR class (gpucfr.cu) represents the GPU implementation of the CFR algorithm. The class GPUCFR uses other classes, Node (for EFG nodes) and InformationSet (for information sets). It loads the exported EFG tree from a text file. According to the hashes, as GPUCFR is loading information about EFG nodes, it allocates the Node's and InformationSet's instances. Each instance keeps a pointer to GPU's global memory, where allocates EFG data for the game tree. Destructors free allocated memory on the host side and GPU device side.

In the GPU's global memory, the information set is an array of floats. The size of the array depends on the number of actions. The array contains:

- the number of actions
- the information set's reach probability
- a vector of the current strategy
- a vector of counterfactual values
- a vector of average strategy

EFG nodes use efg_node_t structure. The structure contains information about the player, node's reach probability, number of children, a pointer to the node's information set, a pointer to the parent's structure, and an array to children structures' pointers.

From CFR's steps 1 and 4, you can see information sets are treated independently. I exploit this with kernels rm_kernel and regret_update_kernel. Step 1 implements rm_kernel, and step 4 implements regret_update_kernel. Each thread processes one GPU's thread. I call the kernels on the array of pointers to information sets' pointers.

The tricky part was to figure out how to implement reach probabilities. CFR uses them in steps 2 for information sets and step 3 for EFG nodes. I had to compute the reach probabilities before the average strategy by the kernel rp_kernel. The rp_kernel runs on an array of EFG nodes' pointers. Each thread starts at a node and uses a while loop to traverse the tree to the root. It updates the shared values by the atomic add. The average_strategy_kernel can then run again on information sets because it uses previously computed information sets' reach probability. Finally, the cfv_kernel can execute. This kernel computes the expected value part of counterfactual values from the bottom of the game tree to the top, so the kernel is called on terminal EFG nodes.

To summarize to operation. The GPU implementation of CFR calculates its iteration in this order:

1. It computes the current strategy from regrets (rm_kernel).
2. It computes the reach probabilities for EFG nodes and information sets (rp_kernel).
3. It adds the current strategy to the average strategy (average_strategy_kernel).
4. It computes the rest of the counterfactual values (cfv_kernel).
5. It updates regrets (regret_update_kernel).

0.6 Correctness

We implemented the CPU implementation long before this semester, so we had validated the correctness before. Nevertheless, I compare the average strategy produced by the CPU implementation and GPU implementation on Goofspiel 2 after 1000 iterations. Also, I use Gambit [2] software to display further validate the correctness of the output. I use Goofspiel 2 because small enough to check quickly (see Figure 2). You can get the results from running main_correctness.cu (main_gpucfr_correctness.cpp) in the GPU (CPU) implementation, or better the prepared bash scripts gpucfr_correctness.bash.

Gambit computes the exact Nash equilibrium by linear programming. Figure 3 shows the resulting strategy under the edges. The most interesting is the player's 1 information set 1:1 and player's 2 information set 2:1. In both of them, the first action has a probability 1, and the second action has a probability 0. Table 2 shows the same probability distribution produced by the CPU and GPU implementation. It is clear that the most useful is to compare those information sets with 2 actions. Table 2 shows the resulting strategy of the information set 1:1 by the CFR's CPU and GPU implementation after 1000 iterations. Table 2 shows the same for 2:1 information set. They are both almost the same. Since CFR is producing approximation, they are both getting closer to the exact Nash equilibrium from Gambit. The CFR's CPU and GPU implementation produce the same strategies on Goofspiel 2 after 1000 iterations.

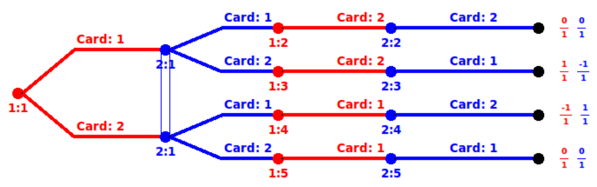

Figure 2: The figure shows the EFG game tree for Goofspiel 2. Red nodes are player's 1 desicion points. Blue ones belong to the player 2. Edges are actions. Leaf nodes are utilities (terminal histories). Player 1 has 5 information sets. Player 2 has also 5 information sets.

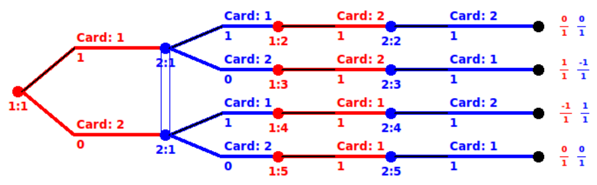

Figure 3: The figure shows the same game tree as in Figure 2. Under the edges are the probabilities of the exact Nash equilibrium solved by third party software Gambit.

Action ID	CPU	GPU
0	0.9995	0.999499
1	0.0005	0.000500501

Table 2: The table show the player's 1 strategy of information set 1:1. In programs, the 1:1 information set has hash 2533060813224638749. The CFR's CPU and GPU implementation produce the same strategies on Goofspiel 2 after 1000 iterations.

Action ID	**CPU**	**GPU**
0	0.9995	0.999499
1	0.0005	0.000500501

Table 3: The table show the player's 2 strategy of information set 2:1. In programs, the 2:1 information set has hash 2033804055022530396. The CFR's CPU and GPU implementation produce the same strategies on Goofspiel 2 after 1000 iterations.

0.7 Measurements and results

I measured the time it takes to compute 1000, 5000, and 10 000 iterations of the CFR algorithm by the CPU and GPU implementation. The games I used are Goofspiel with two (Goofspiel 2), three (Goofspiel 3), and four cards (Goofspiel 4). Table 4 shows basic statistics about these games. Goofspiel 2 is about 100 times smaller than Goofspiel 4. I did not measure the time it takes to build the game tree because it is produced by CPU in both implementations. You create the game tree only once. Also, building the game tree on GPU is not efficient. The time spent calculating iterations is the interesting part. You can run scripts for both gpucfr_time_measure.bash implementations.

Number of cards	**Number of IS**	**Number of nodes**
2	11	10
3	92	103
4	1474	1653

Table 4: The table shows sizes of tested Goofspiel's variants. Number of IS means the number of information sets in the EFG. Number of nodes means the number of EFG nodes in the EFG game tree.

Table 5 shows the numerical results of measuring the time spent calculating CFR's iterations on Goofspiel 2 by the CPU and GPU. The game is tiny, so it is not a big surprise the results are almost the same. Table 6 (Table 7) shows more exciting results in Goofspiel 3 (4). As the game's size grows bigger, the difference between the CPU and GPU implementation becomes more and more apparent. On Goofspiel 3, the GPU implementation is about five times faster than the CPU implementation. On Goofspiel 4, the GPU implementation is about 47 times faster than the CPU implementation. Figures 4, 5, and 6 display that more vividly.

Iterations	**CPU time [ms]**	**GPU time [ms]**
1000	44	41
5000	226	218
10 000	455	424

Table 5: The table shows time spent solving 1000, 5000, 10 000 CFR iterations on Goofspiel 2 by the CPU and GPU implementation.

Iterations	**CPU time [ms]**	**GPU time [ms]**
1000	236	52
5000	1 162	260
10 000	2 457	456

Table 6: The table shows time spent solving 1000, 5000, 10 000 CFR iterations on Goofspiel 3 by the CPU and GPU implementation.

Iterations	**CPU time [ms]**	**GPU time [ms]**
1000	3 663	63
5000	16 388	327
10 000	33 198	684

Table 7: The table shows time spent solving 1000, 5000, 10 000 CFR iterations on Goofspiel 4 by the CPU and GPU implementation.

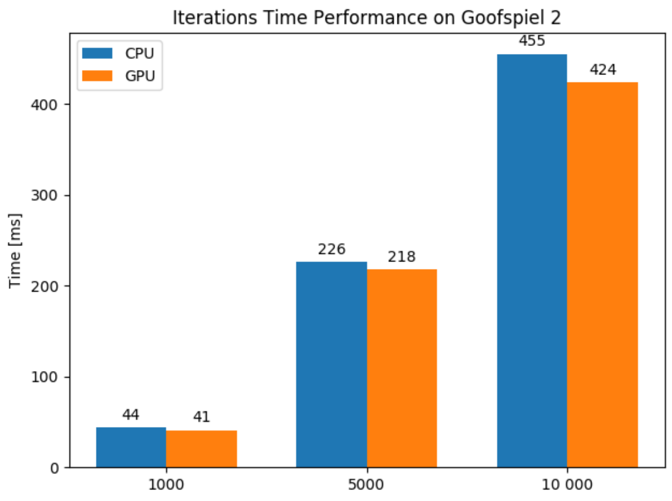

Figure 4: The figure shows time spent solving 1000, 5000, 10 000 CFR iterations on Goofspiel 2 by the CPU and GPU implementation. Goofspiel 2 is a small game, so the results are not very surprisingly similar.

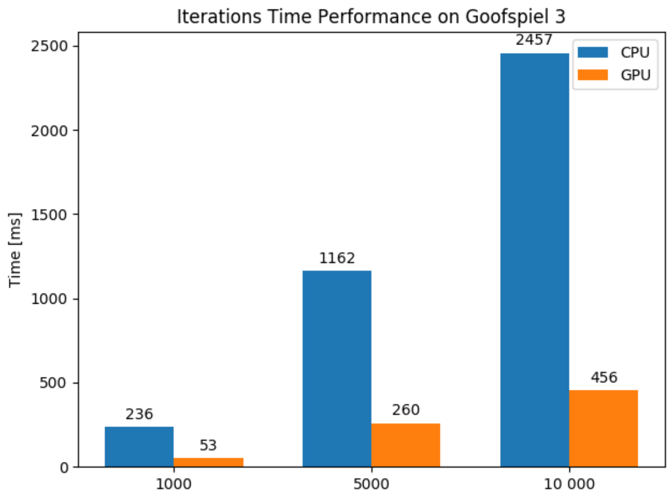

Figure 5: The figure shows time spent solving 1000, 5000, 10 000 CFR iterations on Goofspiel 3 by the CPU and GPU implementation. Goofspiel 3 is about 10 times bigger game. The results show speed up with the GPU implementation.

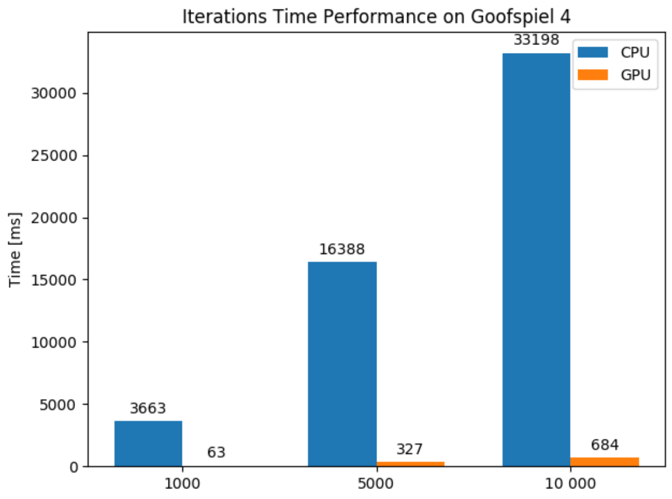

Figure 6: The figure shows time spent solving 1000, 5000, 10 000 CFR iterations on Goofspiel 4 by the CPU and GPU implementation. Goofspiel 4 is also about 10 times bigger game than Goofspiel 3. Also, Goofspiel 4 is about 100x bigger game than Goofspiel 2. The GPU implementation is much faster.

0.8 Conclusion

I described the CFR algorithm and the necessary game theory context. Then I described CPU and GPU implementation. I measured the time that it takes them to finish 1000, 5000, and 10 000 iterations on three variants of Goofspiel. Although the prototype algorithm is not very tuned, the results are promising. Furthermore, the algorithm does not look very GPU compatible. Still, my implementation shows better GPU performance than on CPU and good potential for further work, especially with augmenting the algorithm with neural networks efficiently on the CUDA level.

Bibliography

[1] Neil Burch. “Time and Space: Why Imperfect Information Games are Hard”. PhD thesis. University of Alberta, Computing Science, 2-32 Athabasca Hall, Edmonton, Alberta T6G 2E8: University of Alberta, Dec. 2017.

[2] Richard D McKelvey, Andrew M McLennan, and Theodore L Turocy. *Gambit: Software Tools for Game Theory, Version 16.0.1.* 2016. URL: `http://www.gambit-project.org/`.

[3] Martin Zinkevich et al. “Regret Minimization in Games with Incomplete Information”. In: *Advances in Neural Information Processing Systems 20.* Ed. by J. C. Platt et al. Curran Associates, Inc., 2008, pp. 1729–1736. URL: `http://papers.nips.cc/paper/3306-regret-minimization-in-games-with-incomplete-information.pdf`.