Empirical Testing of Algorithms to Find Nash Equilibria

Anthony Vo

University of Chicago

Abstract. We implement various algorithms to find Nash equilibria and evaluate their performance based on runtime and accuracy as the size of the action space grows. We find that the performance of these algorithms mostly matches what they are expected to do theoretically with some noise due to the random nature of our experiment.

Keywords: Algorithmic game theory · Empiricism

1 Background

In game theory, the Nash equilibrium (NE) of a game characterize the optimal strategies and what the result of the game will be for rational agents. It is known that finding an exact NE is PPAD-hard [2], but some algorithms are able to approximate NE without requiring an exhaustive amount of steps. This report aims to measure the speed and accuracy of these algorithms using a comprehensive testing suite in order to verify the theoretical limits of these algorithms.

Below is a list of the tested algorithms with their properties, where N is the size of the action space for one player and T is the number of iterations used to converge to a solution.

Exact Algorithms: These algorithms run in $O(2^N)$ time and find an exact NE:

- Lemke-Howson [4]

Approximation Algorithms: These algorithms can be faster than exact algorithms but have an expected regret of $\mathbb{E}(\text{Regret}) = \sqrt{T \ln N}$:

- Multiplicative Weights Updating (MWU) [1]
- Random Weighted Majority (RWM) [5]
- Follow the Perturbed Leader (FTPL) [3]

2 Methodology

To carry out these tests, we implemented each algorithm in Python and tested them on two-player zero-sum games of differing sizes. We limited games to being two-player because the Lemke-Howson algorithm does not work for games with more than two players. The zero-sum limitation was added because approximation algorithms often do not converge to a NE in non-zero-sum games. Payoffs were randomly generated using the uniform distribution on $[-1, 1]$. In order to control for the number of iterations, approximation algorithms were run with three different numbers of iterations. For each game size, twenty games were generated, and the results were averaged. The code for this paper can be found at https://github.com/tonyhieu/TTIC-31260-Project.

3 Results

Referenced figures and tables in this section can be found at the end of this paper. Algorithms are denoted by their name and, if applicable, the number of iterations used in parentheses.

We will start with the results that we expected. With more iterations, the MWU, RWM, and FTPL algorithms were better able to converge to the NE, as shown in Figure 1. As these algorithms get more time to run and analyze the game, they get better at converging to a NE.

We can also see that the graph showing external regret (Figure 5) has a curve that resembles the function $f(x) = \sqrt{\ln x}$, reinforcing the theoretical result that the expected regret is $T\sqrt{\ln N}$. Interestingly, when these algorithms are run with a smaller amount of iterations, their total regret goes down with game size. One explanation for this is that the best action in hindsight has less of an advantage over other actions as the action space increases, e.g. the number of actions in the 90th percentile in a 128-action game is almost ten times smaller than that of a 1024-action game, leading to a less dominant best action.

Figure 2 shows exploitability versus game size. Running the algorithms with more iterations led to lower exploitability, as expected, and generally, exploitability increased or stayed the same as the game size grew. One notable outlier is the MWU algorithm with 100 iterations; it greatly decreased in exploitability as the game size grew larger, performing similarly to the MWU algorithm with 1000 iterations. One reason for this is that the MWU algorithm begins with a randomly uniform strategy, distributing weights evenly across all actions. Since randomly uniform strategies perform well in large games, the MWU algorithm has an advantage over others and gets better as games get larger. It outperforms both the RMW and FTPL algorithms when given the same number of iterations.

All algorithms had a similar payoff difference (that is, the payoff between the converged-upon strategy and the NE), and the graph increases and decreases seemingly randomly. This shows that the strategies that these algorithms converge upon all have similar payoffs to one another, as their payoff differences are close together. We posit that the shape of the graph is due to random chance;

when running the same algorithms with a different seed, we find graphs with different shapes every time. Due to the noise present in the creation of these random games, some larger games will have NE that are easy to converge to while some smaller games will have NE that are harder to converge to.

Figure 4 shows the runtime of these algorithms versus the game size. As expected, the more iterations used, the longer the algorithm takes to run. RMW performed the best out of the three algorithms for games with larger actions spaces, while MWU and FTPL had similar runtimes for all sizes. Interestingly, the Lemke-Howson algorithm had a similar speed to running MWU and FTPL with 100 iterations. One reason for this is that, in practice, it is easier to find exact NE, as pivot paths for Lemke-Howson are shorter. Note that $O(2^n)$ is a worst case bound which occurs when the Lemke-Howson algorithm must search all nodes. When it does not have to search all nodes, the algorithm's performance is enhanced.

4 Conclusions

We have found that the practical behavior of the MWU, RWM, and FTPL algorithms is similar to what they should do in theory. Different algorithms are better at different things; e.g. MWU is the least exploitable, but RMW has the fastest runtimes. Lemke-Howson had a surprisingly good runtime compared to what it should do theoretically.

Some extensions of this paper include testing these algorithms on different types of games, as is done in [6]. Additionally, different implementations of these algorithms could lead to different results. For example, implementations using the numpy package in Python instead of the default language features have better performance because numpy is implemented in C. Additional algorithms could also be tested, such as the PNS algorithm [6] for finding exact NE.

Table 1. Average Runtime (seconds) by Algorithm and Game Size

Size	Lemke-Howson	MWU(100)	MWU(1000)	MWU(10000)	FTPL(100)	FTPL(1000)	FTPL(10000)	RMW(100)	RMW(1000)	RMW(10000)
32	0.000518	0.000624	0.005839	0.058648	0.000875	0.007853	0.060619	0.001695	0.015142	0.153288
64	0.000923	0.000714	0.006470	0.065466	0.000805	0.007726	0.076037	0.001716	0.015575	0.153479
128	0.001909	0.001292	0.011868	0.110804	0.001475	0.014065	0.139619	0.001951	0.018455	0.180385
256	0.003924	0.003248	0.029547	0.295745	0.003888	0.036540	0.368248	0.002454	0.022552	0.224808
512	0.010638	0.008915	0.071548	0.713843	0.009968	0.083838	0.828851	0.004423	0.033565	0.318554
1024	0.029609	0.046072	0.417013	4.076762	0.048904	0.431762	4.263858	0.009705	0.054933	0.500714

Table 2. Average Exploitability (Nash Gap) by Algorithm and Game Size

Size	MWU(100)	MWU(1000)	MWU(10000)	FTPL(100)	FTPL(1000)	FTPL(10000)	RMW(100)	RMW(1000)	RMW(10000)
32	0.313292	0.111267	0.035894	0.376843	0.126538	0.041721	0.366292	0.132960	0.044128
64	0.288567	0.118175	0.038427	0.365635	0.143478	0.046805	0.379917	0.144518	0.045276
128	0.222562	0.125637	0.041491	0.356094	0.155307	0.049352	0.356457	0.151152	0.050970
256	0.189806	0.130842	0.042100	0.363440	0.162681	0.052739	0.365098	0.160366	0.053127
512	0.149911	0.121854	0.044002	0.377923	0.158252	0.055608	0.372715	0.158226	0.055826
1024	0.111690	0.101141	0.045689	0.388052	0.146040	0.056967	0.383465	0.150915	0.057845

Table 3. Average Row Player Payoff Difference vs. NE by Algorithm and Game Size

Size	MWU(100)	MWU(1000)	MWU(10000)	FTPL(100)	FTPL(1000)	FTPL(10000)	RMW(100)	RMW(1000)	RMW(10000)
32	0.280385	0.282119	0.282616	0.279182	0.280712	0.282635	0.277471	0.282312	0.282632
64	0.454436	0.455470	0.455363	0.455155	0.454284	0.455319	0.452021	0.454845	0.455259
128	0.409538	0.408876	0.409137	0.408126	0.409231	0.409157	0.411580	0.408658	0.409213
256	0.284884	0.285058	0.284643	0.283932	0.285057	0.284717	0.285361	0.284726	0.284631
512	0.375477	0.375603	0.375665	0.376938	0.375147	0.375583	0.375404	0.375899	0.375558
1024	0.335886	0.335842	0.335805	0.335562	0.335731	0.335752	0.335581	0.335604	0.335803

Table 4. Average Column Player Payoff Difference vs. NE by Algorithm and Game Size

Size	MWU(100)	MWU(1000)	MWU(10000)	FTPL(100)	FTPL(1000)	FTPL(10000)	RMW(100)	RMW(1000)	RMW(10000)
32	0.280385	0.282119	0.282616	0.279182	0.280712	0.282635	0.277471	0.282312	0.282632
64	0.454436	0.455470	0.455363	0.455155	0.454284	0.455319	0.452021	0.454845	0.455259
128	0.409538	0.408876	0.409137	0.408126	0.409231	0.409157	0.411580	0.408658	0.409213
256	0.284884	0.285058	0.284643	0.283932	0.285057	0.284717	0.285361	0.284726	0.284631
512	0.375477	0.375603	0.375665	0.376938	0.375147	0.375583	0.375404	0.375899	0.375558
1024	0.335886	0.335842	0.335805	0.335562	0.335731	0.335752	0.335581	0.335604	0.335803

Table 5. Convergence: Exploitability vs. Iterations on a 4×4 Game

Iterations	MWU	FTPL	RMW
10	0.238459	0.320668	0.649710
50	0.117864	0.255205	0.083115
100	0.105385	0.175644	0.193846
200	0.076252	0.106547	0.087403
500	0.065551	0.055451	0.052773
1000	0.037353	0.040992	0.044722
2000	0.020868	0.044845	0.042391
5000	0.011785	0.026717	0.015402

Table 6. Average Total External Regret (row + col) by Algorithm and Game Size

Size	MWU(100)	MWU(1000)	MWU(10000)	FTPL(100)	FTPL(1000)	FTPL(10000)	RMW(100)	RMW(1000)	RMW(10000)
32	31.329	111.267	358.942	34.489	128.206	406.158	36.782	126.419	429.914
64	28.857	118.175	384.273	36.882	139.886	462.859	34.705	142.532	449.025
128	22.256	125.637	414.915	36.677	154.451	492.397	35.700	150.247	508.325
256	18.981	130.842	420.996	36.665	157.223	538.926	38.337	159.984	529.250
512	14.991	121.854	440.024	36.857	160.168	575.118	37.077	158.566	559.154
1024	11.169	101.141	456.892	38.284	150.902	582.291	38.818	148.996	590.294

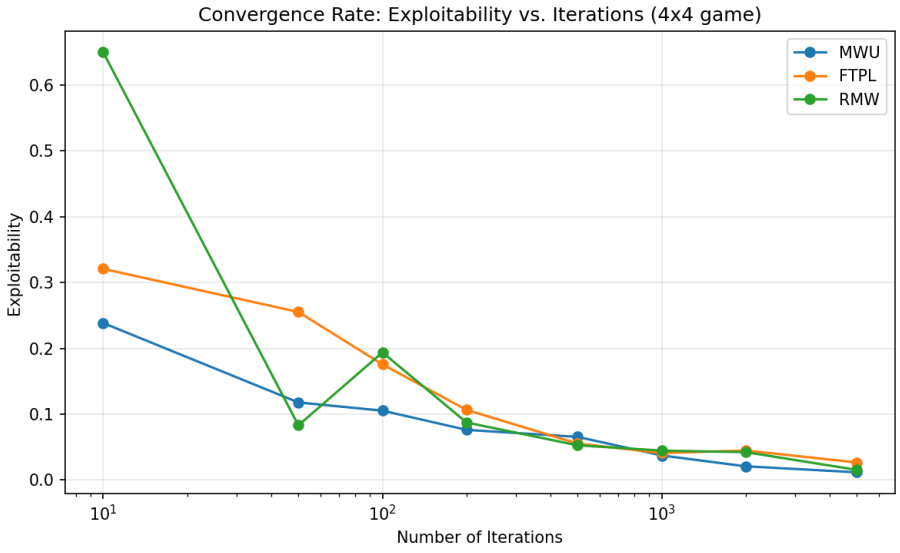

Fig. 1. Graph Showing Convergence vs. Number of Iterations

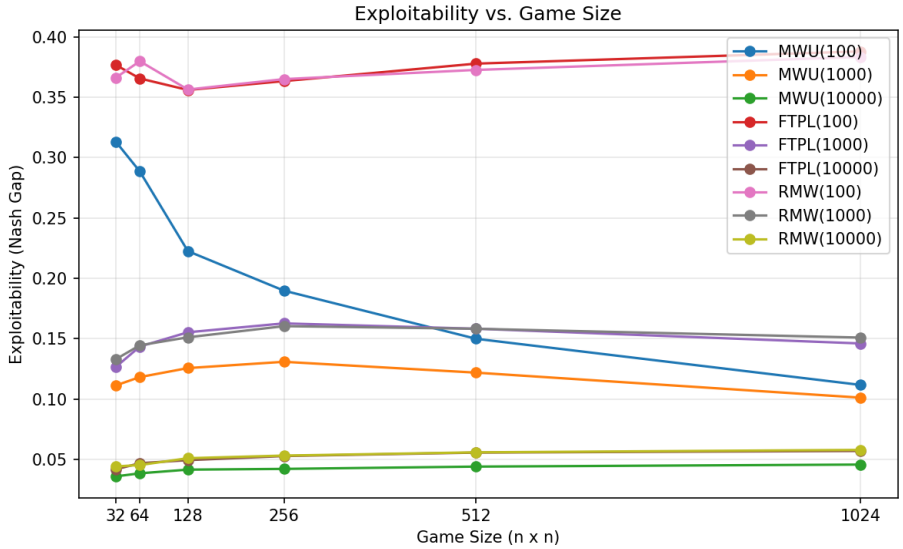

Fig. 2. Graph Showing Exploitability vs. Game Size

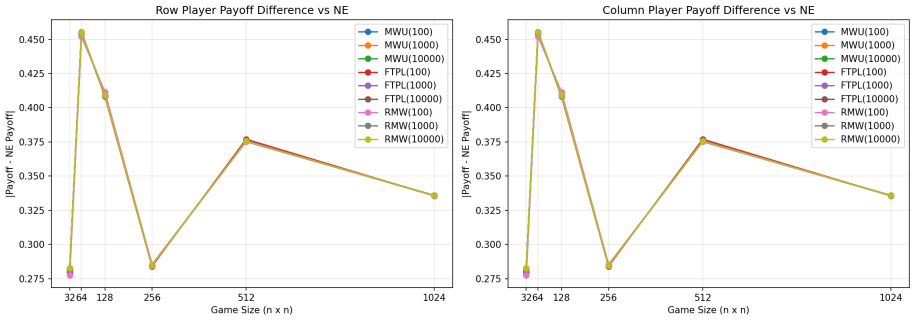

Fig. 3. Graph Showing Payoff Difference vs. Game Size

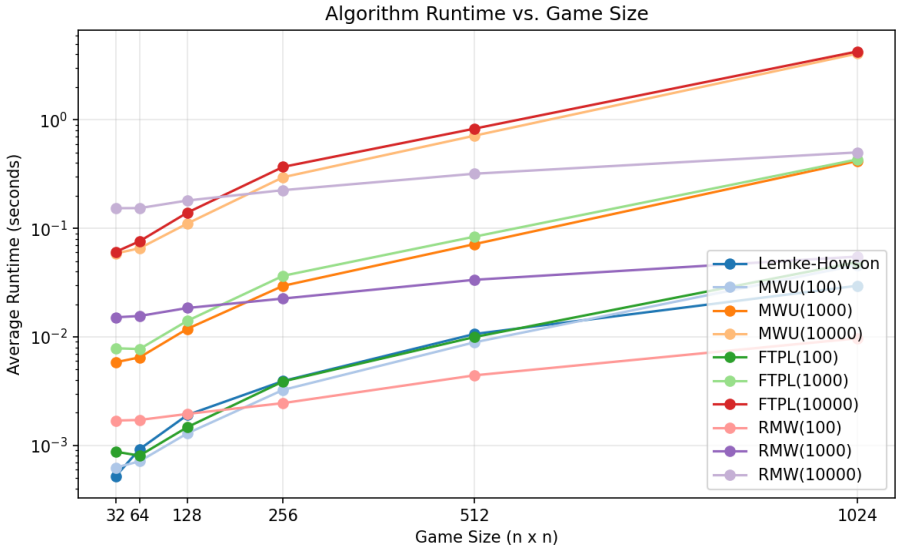

Fig. 4. Graph Showing Runtime vs. Game Size

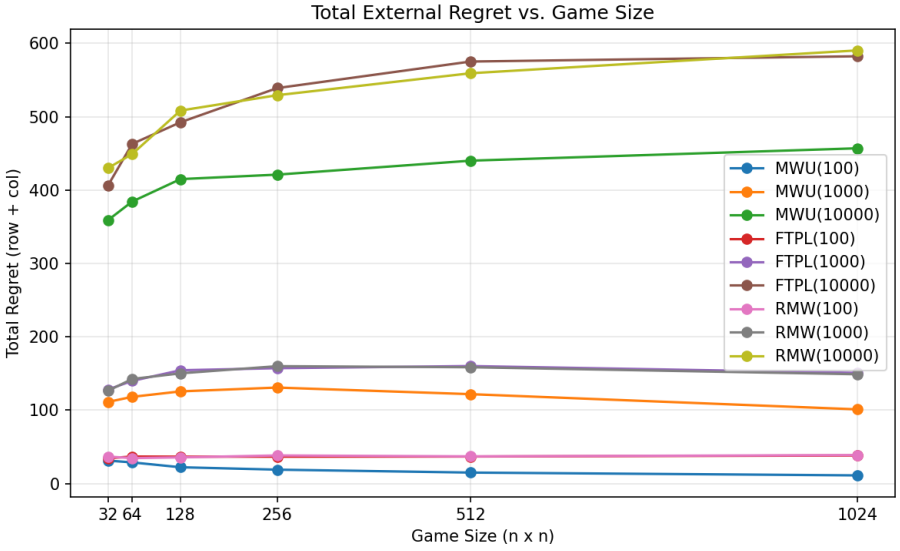

Fig. 5. Graph Showing External Regret vs. Game Size

References

1. Arora, S., Hazan, E., Kale, S.: The multiplicative weights update method: a meta-algorithm and applications. Theory of Computing **8**(1), 121–164 (2012). https://doi.org/10.4086/toc.2012.v008a006, https://doi.org/10.4086/toc.2012.v008a006
2. Chen, X., Deng, X., Teng, S.: Settling the complexity of computing two-player nash equilibria. CoRR **abs/0704.1678** (2007), http://arxiv.org/abs/0704.1678
3. Kalai, A., Vempala, S.: Efficient algorithms for online decision problems. Journal of Computer and System Sciences **71**(3), 291–307 (2005). https://doi.org/https://doi.org/10.1016/j.jcss.2004.10.016, https://www.sciencedirect.com/science/article/pii/S0022000004001394, learning Theory 2003
4. Lemke, C.E., Howson, Jr., J.T.: Equilibrium points of bimatrix games. Journal of the Society for Industrial and Applied Mathematics **12**(2), 413–423 (1964). https://doi.org/10.1137/0112033, https://doi.org/10.1137/0112033
5. Littlestone, N., Warmuth, M.: The weighted majority algorithm. Information and Computation **108**(2), 212–261 (1994). https://doi.org/https://doi.org/10.1006/inco.1994.1009, https://www.sciencedirect.com/science/article/pii/S0890540184710091
6. Porter, R., Nudelman, E., Shoham, Y.: Simple search methods for finding a nash equilibrium. Games and Economic Behavior **63**(2), 642–662 (2008). https://doi.org/https://doi.org/10.1016/j.geb.2006.03.015, https://www.sciencedirect.com/science/article/pii/S0899825606000935, second World Congress of the Game Theory Society