Fast seed-learning algorithms for games

Jialin Liu[1], Olivier Teytaud[1], Tristan Cazenave[2]

[1] TAO, Inria, Univ. Paris-Sud, UMR CNRS 8623, Gif-sur-Yvette, France
[2] LAMSADE, Université Paris-Dauphine, Paris, France

Abstract. Recently, a methodology has been proposed for boosting the computational intelligence of randomized game-playing programs. We propose faster variants of these algorithms, namely rectangular algorithms (fully parallel) and bandit algorithms (faster in a sequential setup). We check the performance on several board games and card games. In addition, in the case of Go, we check the methodology when the opponent is completely distinct to the one used in the training.

1 Introduction: portfolios of random seeds

Artificial intelligence (AI) has been invaded by ensemble methods [2, 13]. In games, some recent papers propose to do so, and in particular to combine variants of a single program, thanks to tricks on random seeds.

The impact of random seeds. We assume that an AI is given. This AI is supposed to be stochastic; even with the same flow of information, it will not always play the same sequence of moves. This is for example the case for Monte Carlo Tree Search [5, 10]. Given such an AI, we can check its performance against a baseline program (possibly itself) as we vary the random seed, i.e., we can generate K different random seeds, and for each of these seeds play K_t games against the baseline. We can then plot the success rates, sort, and compare the differences to the standard deviations. Results are presented in Fig. 1 and show for several games that the seed has a significant impact. The methodologies presented in this paper are based on this phenomenon.

Related work. Several works were dedicated to combining several AIs in the past. [11] combines several different AIs. Nash methods have been used in [7] for combining several opening books.

The work in [12] constructed several AIs from a single stochastic one and combined them by the BestSeed and Nash methods, detailed in Section 2. The application of the methodologies above to Go has already been investigated in [12]. These results were tested in cross-validation. We extend these results in Go to the case with transfer (i.e. we check the impact in terms of the success rate against other opponents, not related to the ones in learning) and we provide quadratically faster algorithms. We also perform experiments on additional games (Atari-Go, Breakthrough, Domineering, and several games from the GameTestBed platform).

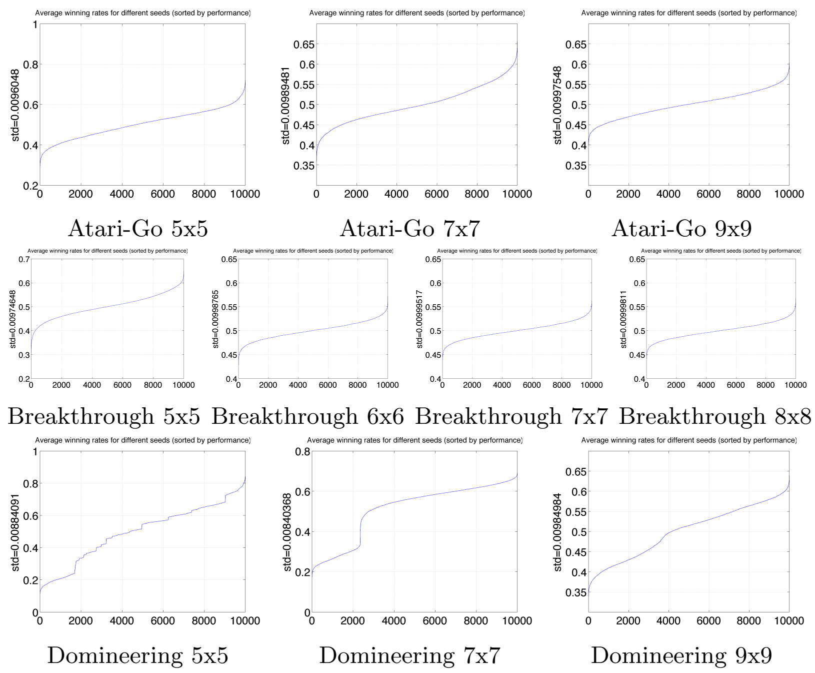

Fig. 1: Impact of the seed on the success rate. *x-axis*: index of seed; *y-axis*: success rate. For the n^{th} value, we consider the n^{th} worst seed for Black and the n^{th} seed for White, and display their average scores against all opponent seeds. The label on the y-axis shows the standard deviation of these averages; we see that there are good seeds, which obtain a success rate far above 50% - by much more than the standard deviation.

2 Known algorithms for boosting an AI using random seeds

This section presents an overview of two methods proposed in [12] for building a boosted algorithm from a set of seeds: the Nash-approach and the BestSeed-approach. We propose extensions of these methods and apply them to some new games. Typically, a stochastic computer program uses a random seed. The random seed ω is randomly drawn (using the clock, usually) and then a pseudo-random sequence is generated. Therefore, a stochastic program is in fact a random variable, distributed over deterministic program. Let us define: AI is our game playing artificial intelligence; it is stochastic. $AI(\omega)$ is a deterministic version; ω is a seed, which is randomly drawn in the original AI. We can easily generate plenty of ω and therefore one stochastic AI becomes several deterministic AIs, termed AI_1, AI_2, ... Let us assume that one of the players plays as Black and the other plays as White. We can do the same construction as above for the AI playing as Black and for the AI' playing as White. We get

Algorithm 1 Approach for boosting a game stochastic game AI.

```
Require: A stochastic $AI$ playing as Black, a stochastic $AI'$ playing as White.
Output: A boosted AI termed $BAI$ playing as Black, a boosted AI $BAI'$ playing as
    White.
 1: Build $M_{i,j} = 1$ if $AI_i$ (Black) wins against $AI'_j$ (White) for $i \in \{1,\dots,K\}$ and
    $j \in \{1,\dots,K_t\}$, otherwise $M_{i,j} = 0$.
 2: Build $M'_{i,j} = 1$ if $AI'_i$ (White) wins against $AI_j$ (Black) for $i \in \{1,\dots,K\}$ and
    $j \in \{1,\dots,K_t\}$, otherwise $M'_{i,j} = 0$.
 3: if BestSeed then                                        ▷ deterministic boosted AI
 4:     $BAI$ is $AI_i$ where $i$ maximizes $\sum_{j=1}^{K_t} M_{i,j}$.
 5:     $BAI'$ is $AI'_i$ where $i$ maximizes $\sum_{j=1}^{K_t} M'_{i,j}$.
 6: end if
 7: if Nash then                                            ▷ stochastic boosted AI
 8:     Compute $(p, q)$ a Nash equilibrium of $M$.
 9:     $BAI$ is $AI_i$ with probability $p_i$
10:     Compute $(p', q')$ a Nash equilibrium of $M'$.
11:     $BAI'$ is $AI'_j$ with probability $p'_i$
12: end if
13: if Uniform then                                         ▷ boosted AI
14:     $BAI$ is $AI_i$ with probability $1/K$.
15:     $BAI'$ is $AI'_j$ with probability $1/K$.
16: end if
```

AI_1, AI_2,... for Black, and AI'_1, AI'_2, ... for White. From now on, we present the learning algorithm for Black - still, for this, we need the AI' for White as well. The algorithm for enhancing the AI as White is similar. Let us define, for $i \in \{1,\dots,K\}$ and $j \in \{1,\dots,K_t\}$, $M_{i,j} = 1$ when AI_i (playing as Black) wins against AI'_j (playing as White). Otherwise, $M_{i,j} = 0$. Also, let us define $M'_{i,j} = 1$ when AI'_i (playing as White) wins against AI_j (playing as Black). Thus, we have $M'_{i,j} = 1 - M_{j,i}$. [12] uses $K = K_t$, hence they use the same squared matrix for Black and for White - up to a transformation $M \mapsto 1 - M'$. The point in the present paper is to show that we can save up time by using $K \neq K_t$. This means that we need two matrices: M (used for the learning for Black) is the matrix of $M_{i,j}$ for $i \in \{1,\dots,K\}$ and $j \in \{1,\dots,K_t\}$; and M' (used for the learning for White) is the matrix of $(M')_{i,j}$ for $i \in \{1,\dots,K\}$ and $j \in \{1,\dots,K_t\}$. If $K_t \leq K$, M and M' have $K_t \times K_t$ entries in common (up to transformation $(M')_{i,j} = 1 - M_{j,i}$); therefore building M and M' needs simulating $2K \times K_t - K_t^2$ games.

For arbitrary values of K and K', boosted AIs can be created using BestSeed and Nash approaches, summarized in Algorithm 1. The Nash approach provides a stochastic policy, usually stronger than the original policy [12]. This can be done even if the matrix is not squared.

3 Faster methods

3.1 Rectangular learning

At first view, the approach in [12] is simple and sound: they need one squared matrix for both Black and White. However, their approach needs the result of K^2 games. With our rectangular approach, if we use K different seeds and K_t opponent seeds, we need $2K \times K_t - K_t^2$ games.

Let us now check the precision of our approach. Our algorithms use averages of rows and averages of columns. Let us define μ_i the average value of the i^{th} row of M, if K_t was infinite - this is the average success rate of AI_i playing as Black against AI playing as White. And let us define $\hat{\mu}_i$ the average value that we get, with our finite value K_t. Hoeffding's bound [9] tells us that with probability $1-\delta$, $|\mu_i - \hat{\mu}_i| \leq \sqrt{-\log(\delta/2)/(2K_t)}$. By Bonferroni correction (i.e. union bound), with probability $1-\delta$, for all $i \leq K$, $|\mu_i - \hat{\mu}_i| \leq \sqrt{-\log(\delta/(2K))/(2K_t)}$. For a requested precision ϵ, we can do as follows:

- Choose a value of K large enough, so that at least one seed i is optimal within precision $\epsilon/2$.
- Choose K_t such that $\sqrt{-\log(\delta/(2K))/(2K_t)} \leq \epsilon/2$.

We see that K_t slightly more than logarithmic as a function of K is enough for ensuring ϵ arbitrarily small asymptotically in K. On the other hand, we have no bound on K necessary for having at least one seed optimal within precision $\epsilon/2$.

3.2 Bandit methods

Bandits are a natural method for finding approximate optima quickly. Rather than computing full matrices, we consider the following approach: apply Exp3 [1], both for Black and for White, for sampling in the matrix M (evaluate $M_{i,j}$ only when you need) as a matrix game; Black is the row player and maximizes; White is the column player and minimizes. We use far less evaluations than the size of the matrix.

Finally, we can simply use UCB (separately for Black and White), which can be modified [15] for handling the infinite nature of the set of seeds; we apply this to the game of Go in 9x9 and 19x19.

4 Testbeds

We provide experiments on a list of games. First, we consider MCTS, applied to four board games, namely Domineering, Atari-Go, Breakthrough and Go. Then, we consider the randomized policy in the GameTestBed platform. Domineering is a two-player game with very simple rules: each player in turn puts a tile on empty locations in the board. The game starts with an empty board. The first player who can not play loses the game. Usually, one of the player has vertical 2x1 tiles, and the other has horizontal 1x2 tiles. Domineering can be played on boards of various shapes, most classical cases are rectangles or squares. For squared boards, Domineering is solved until board size 10x10 [3, 4]. Domineering was invented by Göran Andersson [6]. Jos Uiterwijk recently proposed a knowledge based method that can solve large rectangular boards without any search [14]. The Breakthrough game, invented by Dan Troyka in 2000, has very simple rules: all pieces can move straight ahead or in diagonal (i.e. three possible target locations). Captures are possible in diagonal only. Players play in turn, and the first player who reaches the opposite first row or captures all opponents pieces has won. There is no draw in Breakthrough - there is always at least one legal move, and pieces can only go forward (straight or diagonal) so that loops can not occur. This game won the 2001 8x8 Game Design Competition. Yasuda Yasutoshi popularized the Atari-Go variant of the game of Go; the key difference is that the first player who makes a capture wins the game. Atari-Go is also known as Ponnuki-Go, One-capture-Go, or Capture-Go. Last but not least, we provide results of experiments on the GameTestBed platform (`https://gforge.inria.fr/projects/gametestbed/`).

5 Experiments

Besides playing against the original stochastic AI, we consider the following opponent ($K' = 1$ corresponds to the original opponent, whereas $K' >> 1$ is a much stronger opponent):

- Generate K' seeds, randomly, for Black and K' seeds, randomly, for White.
- Consider the worst success rate of our boosted AI playing as White against these K' strategies for Black and consider the worst success rate of our boosted AI playing as Black against these K' strategies for White. Our success rate is the average of these two success rates (Black and White).

This is a strong challenge for K' large; since we consider separately White and Black, we have indeed K'^2 opponent strategies (each of the K' seeds for Black and each of the K' seeds for White) and consider the worst success rate. We will define this opponent as a K'-*exploiter*: it is an approximator of the exploitability property of Nash equilibria. It represents what can be done if our opponent could play the game K' times and select the best outcome. For $K' = 1$, this opponent is playing exactly as the original *AI*: this is the success rate against a randomly drawn seed. A score $\geq 50\%$ against $K' = 1$ means that we have outperformed the original AI, i.e. boosting has succeeded; but it is satisfactory to have also a better success rate, against $K' > 1$, than the original AI.

In order to validate the method, we take care that our algorithm is tested with a proper *cross-validation*: the opponent uses seeds which have never been used during the learning of the portfolio. This is done for all our experiments, BestSeed, Uniform, or Nash. For this reason, there is no bias in our results. In addition, we test our performance, in the case of Go, against another opponent; therefore, this is *transfer* learning, as explained in Section 5.

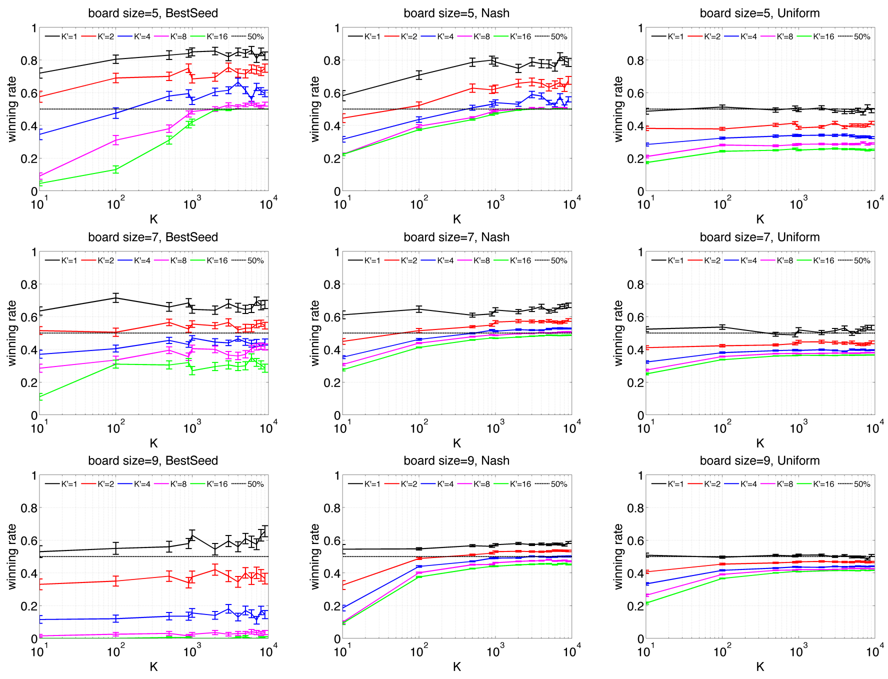

Fig. 2: Results for domineering, with the BestSeed and the Nash approach, against the baseline ($K' = 1$) and the exploiter ($K' > 1$). *x-axis*: K, number of seeds optimized for both players; *y-axis*: success rate. $K_t = 900$ in all experiments. The performance of the uniform version (original algorithm) is also presented for comparison.

Performance of rectangular algorithms in cross-validation, for some board games. All results are averaged over 100 runs. Results for Domineering, Atari-Go and Breakthrough are presented in Figs. 2, 3, and 4 respectively. Table 3 shows the numerical results when $K = 9000$ and $K_t = 900$.

In short, BestSeed performs well against the original algorithm (corresponding to $K' = 1$), but its performance against the exploiter ($K' > 1$) is very weak. On the other hand, the Nash approach outperforms the original algorithm both in terms of success rate against the baseline ($K' = 1$) in all cases and against the exploiters ($K' > 1$) in most cases (i.e. curves on the middle column in Figs. 2, 3, 4 are better than those on the right column) - however, for Breakthrough in large size the results were (very) slightly detrimental for $K' > 1$, i.e. the “exploiter” could learn strategies against it.

Performance of the bandit method in cross-validation. In this section, we present results obtained by Exp3 on the GameTestBed platform. We apply 800 iterations of Exp3 with 400 seeds for each player. The arms with frequency greater than 99% of the largest frequency are chosen as possible seeds and we

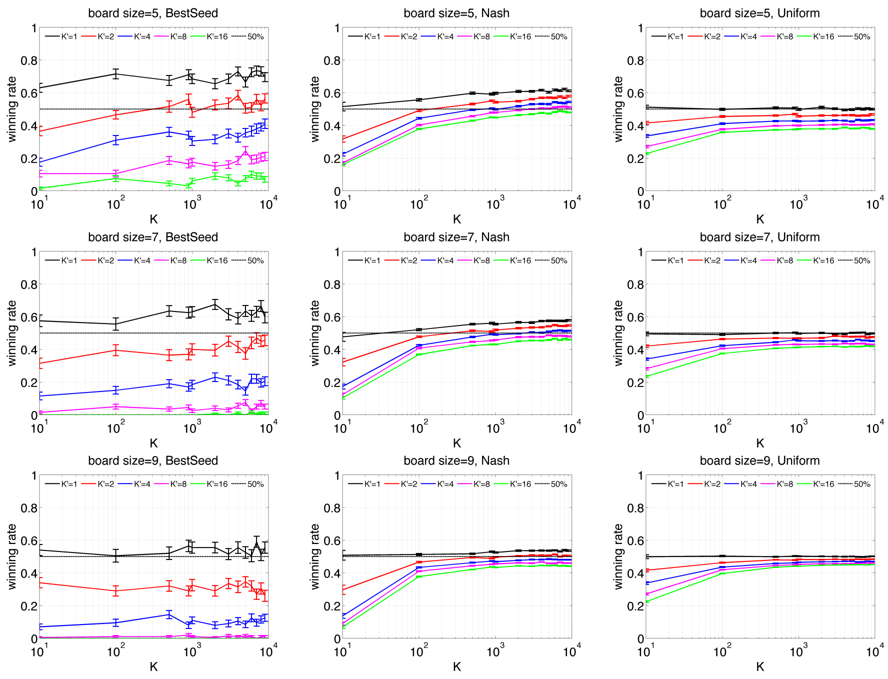

Fig. 3: Results for Atari-Go, with the BestSeed and the Nash approach, against the baseline ($K' = 1$) and the exploiter ($K' > 1$). *x-axis*: K, number of seeds optimized for both players; *y-axis*: success rate. $K_t = 900$ in all experiments. The performance of the uniform version (original algorithm) is also presented for comparison.

play them with probability proportional to their frequencies. Each learning is repeated 100 times, and each learnt AI is tested against 400 randomly drawn seeds which have never been used during the learning. Table 1 shows the results against the original algorithm, and against a stronger opponents ($K' = 16$). With the Exp3 method and most frequent arm selection, our boosted algorithm outperforms the original AI and its success rate against the stronger opponent $K' = 16$ is improved. Please note that the presented games are hard to learn: Nim is a simple game but has a brute representation which makes learning hard; and two of the games are phantom games with tricky partially observable states. The policies are the default randomized policies in the freely available code above.

We also tested UCB with progressive widening [15]; the infinite set of arms is handled by considering, after N simulated games, the $\lceil 100N^{\frac{1}{3}} \rceil$ first arms. UCB was parallelized by pulling the arms with the 40 best scores simultaneously. We get the following results (we performed the learning once, the standard deviation refers to the success rate in cross-validation):

Table 1: Success rate for five games of the GameTestBed platform, with the Exp3 method and most frequent arm selection, against the baseline ($K' = 1$) and the stronger exploiter ($K' = 16$). In the game Morra, the AI with given random seed is still stochastic. Hence, the success rate is not greatly improved.

Game	Success rate (%)			
	Baseline		Most frequently chosen	
	$K' = 1$	$K' = 16$	$K' = 1$	$K' = 16$
Phantom 4 in a row		0.50 ± 0.00	$\mathbf{69.00 \pm 0.00}$	$\mathbf{8.75 \pm 0.00}$
Nim		0.00 ± 0.00	$\mathbf{73.50 \pm 0.00}$	$\mathbf{3.00 \pm 0.00}$
Phantom tic-tac-toe	50	0.50 ± 0.00	$\mathbf{65.50 \pm 0.00}$	$\mathbf{15.25 \pm 0.00}$
Morra		47.73 ± 0.22	$\mathbf{52.12 \pm 0.23}$	$\mathbf{48.11 \pm 0.22}$
PigStupid		40.78 ± 0.25	$\mathbf{50.04 \pm 0.25}$	$\mathbf{41.30 \pm 0.25}$

– 9x9 Go, MCTS with 400 simulations per move, after 60 000 simulated games, the seed 1125 was selected for Black and the seed 898 was selected for White, success rate 79.8%.
– 19x19 Go, GnuGo *not* MCTS[3], after only 3780 simulated games, the seed 606 was selected for Black and the seed 472 was selected for White, success rate 55.9%. This algorithm is far less stochastic than MCTS.

Performance in transfer, in the case of Go. Earlier results [12] and in Section 5 are performed in a classical machine learning setting, i.e. with cross-validation; we now check the transfer, i.e. the fact that we boost an AI, we get a better performance also when we test its performance against *another* AI.

Transfer to GnuGo. We applied BestSeed to GnuGo, a well known AI for the game of Go, with Monte Carlo tree search and a budget of 400 simulations. The BestSeed approach was applied with a 100x100 learning matrix, corresponding to seeds $\{1, \ldots, 100\}$ for Black and seeds $\{1, \ldots, 100\}$ for White.

Then, we tested the performance against GnuGo "classical", i.e. the non-MCTS version of GnuGo; this is a really different AI with different playing style. We got positive results as shown in Table 2. Results are presented for Black; for White the BestSeed had a negligible impact.

Transfer: validation by a MCTS with long thinking time. Fig. 5 provides a summary of differences between moves chosen (at least with some probability) by the original algorithm, and the ones chosen in the same situation by the algorithm with optimized seed. These situations are the 8 first differences between games played by the original GnuGo and by the GnuGo with our best seed. We use GnugoStrong, i.e. Gnugo with a larger number of simulations, for checking if Seed 59 leads to better moves. GnugoStrong is precisely defined as << gnugo –monte-carlo –mc-games-per-level 100000 –level 1>>. On these situations (Fig. 5) such that BestSeed differs from the original GnuGo with the same number of simulations, GnugoStrong played 5 games (playing both sides), all leading to the same result in each case.

[3] GnuGo does not accept MCTS for 19x19.

Table 2: Performance of BestSeed-Gnugo-MCTS against various GnuGo-default programs, compared to the performance of the default Gnugo-MCTS. The results are for GnuGoMCTS playing as Black vs GnuGo-classical playing as White, and the games are completely independent of the learning games (which use only Gnugo-MCTS). Results are averaged over 1000 games. All results in 5x5, komi 6.5, with a learning over 100x100 random seeds.

Opponent	Performance of BestSeed	Performance of the original algorithm with randomized random seed
GnuGo-classical level 1	**1. (± 0)**	.995 (± 0)
GnuGo-classical level 2	**1. (± 0)**	.995 (± 0)
GnuGo-classical level 3	**1. (± 0)**	.99 (± 0)
GnuGo-classical level 4	**1. (± 0)**	1. (± 0)
GnuGo-classical level 5	**1. (± 0)**	1. (± 0)
GnuGo-classical level 6	**1. (± 0)**	1. (± 0)
GnuGo-classical level 7	**.73 (± .013)**	.061 (± .004)
GnuGo-classical level 8	**.73 (± .013)**	.106 (± .006)
GnuGo-classical level 9	**.73 (± .013)**	.095 (± .006)
GnuGo-classical level 10	**.73 (± .013)**	.07 (± .004)

6 Conclusions

Our results (success rate of the boosted algorithm against the non-boosted baseline) are roughly for BestSeed: 73.5%, 67.5%, 59% for Atari-Go in 5x5, 7x7 and 9x9 respectively; 65.5%, 57.5%, 55.5%, 57% for Breakthrough in 5x5, 6x6, 7x7 and 8x8 respectively; 86%, 71.5%, 65.5% for Domineering in 5x5, 7x7 and 9x9 respectively. On several games in Gametestbed, we got more than 70% success rate against the baseline. We got close to 80% in 9x9 Go. Against $K' = 16$, the results were usually positive, though not always (see Breakthrough) - we believe that this would be solved with larger K, K', as proved in [12]; asymptotically, the Nash method should be optimal against all K'.

Usually, the boosted AIs significantly outperform the baselines, without additional computational cost. This does not require any source code development. The rectangle versions are faster than the original algorithms, and the bandit versions are indeed much faster.

Approximating Nash using the adversarial bandit algorithm, Exp3, does not require computing the whole matrix. The computational cost is decreased to its square root, up to logarithmic factors (see [8]) and with a minor cost in terms of precision. The success rate is significantly improved.

Our work on applying UCB with an infinite set of seeds to Go is preliminary (the parameters of progressive widening are arbitrarily chosen and the UCB parameters are guessed rather than optimized). Nevertheless, the fact that the boosted AI is significantly enhanced validates the effectiveness of our approach.

Further work. The simplest further work consists in optimizing the seeds specifically for time steps. This should provide an easy exploitation of the time structure of the game. A work in progress is the use of Exp3 with infinite set of seeds, handled by progressive widening (as we did for UCB - after N simulated games, only the first $\lceil CN^\gamma \rceil$ seeds are considered, with $C \geq 2$ and $\gamma \in (0, 1]$). Also, worst-so-far seed might be removed periodically.

References

1. P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. Gambling in a rigged casino: the adversarial multi-armed bandit problem. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 322–331. IEEE Computer Society Press, Los Alamitos, CA, 1995.
2. L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.
3. D. Breuker, J. Uiterwijk, and H. van den Herik. Solving 88 domineering. *Theoretical Computer Science*, 230(1-2):195 – 206, 2000.
4. N. Bullock. Domineering: Solving large combinatorial search spaces. *ICGA Journal*, 25(2):67–84, 2002.
5. R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy*, pages 72–83, 2006.
6. M. Gardner. Mathematical games. *Scientific American*, 230:106–108, 1974.
7. R. Gaudel, J.-B. Hoock, J. Pérez, N. Sokolovska, and O. Teytaud. A Principled Method for Exploiting Opening Books. In *International Conference on Computers and Games*, pages 136–144, Kanazawa, Japon, 2010.
8. M. D. Grigoriadis and L. G. Khachiyan. A sublinear-time randomized approximation algorithm for matrix games. *Operations Research Letters*, 18(2):53–58, Sep 1995.
9. W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13–30, 1963.
10. L. Kocsis and C. Szepesvari. Bandit based Monte-Carlo planning. In *15th European Conference on Machine Learning (ECML)*, pages 282–293, 2006.
11. V. Nagarajan, L. S. Marcolino, and M. Tambe. Every team deserves a second chance: Identifying when things go wrong (student abstract version). In *29th Conference on Artificial Intelligence (AAAI 2015), Texas, USA*, 2015.
12. D. L. Saint-Pierre and O. Teytaud. Nash and the Bandit Approach for Adversarial Portfolios. In *CIG 2014 - Computational Intelligence in Games*, Computational Intelligence in Games, pages 1–7, Dortmund, Germany, Aug. 2014. IEEE, IEEE.
13. R. Shapire, Y. Freund, P.Bartlett, and W. Lee. Boosting the margin: a new explanation for the effectiveness of voting methods. In *Proceedings of the 14th International Conference on Machine Learning*, pages 322–330. Morgan Kaufmann, 1997.
14. J. W. H. M. Uiterwijk. Perfectly solving domineering boards. In T. Cazenave, M. H. M. Winands, and H. Iida, editors, *Computer Games - Workshop on Computer Games, CGW 2013, Held in Conjunction with the 23rd International Conference on Artificial Intelligence, IJCAI 2013, Beijing, China, August 3, 2013, Revised Selected Papers*, volume 408 of *Communications in Computer and Information Science*, pages 97–121. Springer, 2013.
15. Y. Wang, J.-Y. Audibert, and R. Munos. Algorithms for infinitely many-armed bandits. In *Advances in Neural Information Processing Systems*, volume 21, 2008.

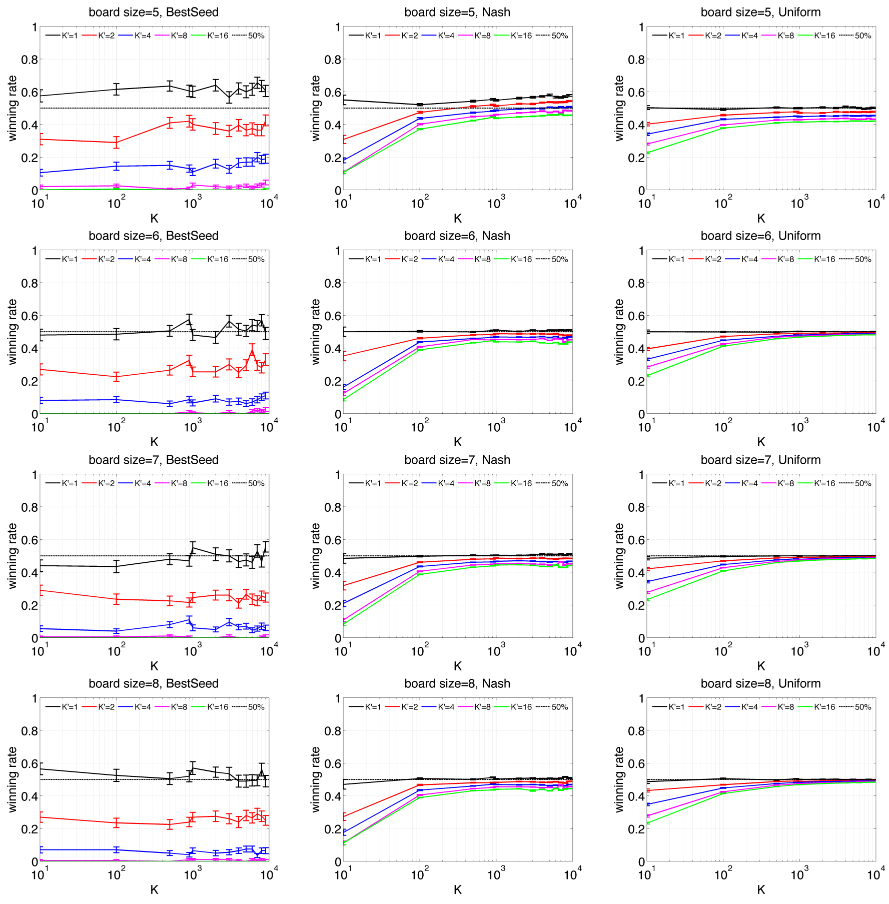

Fig. 4: Results for Breakthrough, with the BestSeed and the Nash approach, against the baseline ($K' = 1$) and the exploiter ($K' > 1$). *x-axis*: K, number of seeds optimized for both players; *y-axis*: success rate. $K_t = 900$ in all experiments. The performance of the uniform version (original algorithm) is also presented for comparison.

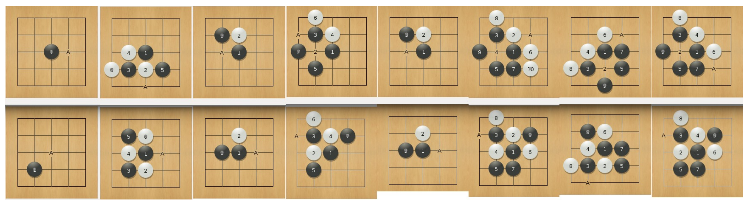

Fig. 5: Comparison between moves played by BestSeed-MCTS (top) and the original MCTS algorithm (bottom) in the same situations. GnugoStrong, used as an evaluator, prefers the moves chosen by BestSeed-MCTS for situations 1, 2, 6, 7, 8; whereas 3, 4 and 5 are equivalent.

Table 3: Success rate for Domineering, Atari-Go and Breakthrough, with the BestSeed and Nash approaches, against the baseline ($K' = 1$) and the exploiter ($K' > 1$). $K = 9000$ and K_t=900. The experiments are repeated 100 times. The standard deviations are shown after $\pm$. $K' = 1$ corresponds to the original algorithm with randomized seed; $K' = 2$ corresponds to the original algorithm but choosing optimally (after checking their performance against its opponent) between 2 possible seeds, i.e. it is guessing, in an omniscient manner, between 2 seeds, each time an opponent is provided. $K' = 4$, $K' = 8$, $K' = 16$ are similar with 4, 8, 16 seeds respectively.

Domineering

Board	Method	Success rate (%)				
		$K' = 1$	$K' = 2$	$K' = 4$	$K' = 8$	$K' = 16$
5x5	Uniform	49.03 ± 1.30	41.55 ± 0.92	32.53 ± 0.60	28.95 ± 0.45	25.06 ± 0.41
	BestSeed	**82.50 ± 2.41**	**75.00 ± 2.53**	**59.50 ± 1.98**	**53.00 ± 1.20**	**50.00 ± 0.00**
	Nash	78.50 ± 2.50	67.54 ± 2.39	55.96 ± 1.60	50.00 ± 0.00	**50.00 ± 0.00**
7x7	Uniform	53.33 ± 1.41	44.33 ± 0.85	39.58 ± 0.26	37.97 ± 0.17	36.55 ± 0.13
	BestSeed	**67.50 ± 2.51**	54.50 ± 2.03	44.50 ± 1.88	41.50 ± 1.90	28.50 ± 2.50
	Nash	66.98 ± 1.39	**58.01 ± 0.83**	**52.79 ± 0.32**	**50.71 ± 0.25**	**48.72 ± 0.19**
9x9	Uniform	50.68 ± 0.58	46.68 ± 0.43	44.06 ± 0.26	42.50 ± 0.13	41.56 ± 0.09
	BestSeed	**65.50 ± 3.40**	36.50 ± 3.26	14.50 ± 2.50	3.50 ± 1.29	0.50 ± 0.50
	Nash	58.60 ± 0.61	**53.43 ± 0.46**	**50.04 ± 0.37**	**47.15 ± 0.28**	**45.11 ± 0.26**

Atari-Go

Board	Method	Success rate (%)				
		$K' = 1$	$K' = 2$	$K' = 4$	$K' = 8$	$K' = 16$
5x5	Uniform	49.95 ± 0.54	46.72 ± 0.46	43.26 ± 0.37	40.78 ± 0.30	37.85 ± 0.26
	BestSeed	**69.50 ± 2.76**	56.50 ± 2.83	41.00 ± 2.89	21.00 ± 2.49	7.00 ± 1.75
	Nash	61.16 ± 0.48	**57.91 ± 0.50**	**54.33 ± 0.40**	**51.18 ± 0.39**	**47.96 ± 0.26**
7x7	Uniform	49.76 ± 0.37	47.61 ± 0.30	45.10 ± 0.30	43.02 ± 0.22	41.84 ± 0.18
	BestSeed	**59.50 ± 3.25**	45.50 ± 3.28	20.50 ± 2.68	5.00 ± 1.52	1.00 ± 0.71
	Nash	57.79 ± 0.45	**54.66 ± 0.42**	**51.40 ± 0.33**	**47.97 ± 0.37**	**45.99 ± 0.28**
9x9	Uniform	50.16 ± 0.25	48.39 ± 0.22	47.01 ± 0.16	**46.04 ± 0.13**	**45.11 ± 0.10**
	BestSeed	**55.50 ± 3.49**	26.00 ± 3.39	12.50 ± 2.19	1.00 ± 0.71	0.00 ± 0.00
	Nash	53.61 ± 0.43	**50.46 ± 0.37**	**48.06 ± 0.24**	46.02 ± 0.22	44.15 ± 0.21

Breakthrough

Board	Method	Success rate (%)				
		$K' = 1$	$K' = 2$	$K' = 4$	$K' = 8$	$K' = 16$
5x5	Uniform	50.12 ± 0.45	47.80 ± 0.35	45.42 ± 0.23	43.18 ± 0.20	42.01 ± 0.15
	BestSeed	**60.50 ± 3.45**	42.50 ± 3.30	19.00 ± 2.75	4.50 ± 1.45	0.50 ± 0.50
	Nash	57.77 ± 0.54	**54.32 ± 0.36**	**50.75 ± 0.32**	**48.38 ± 0.29**	**45.64 ± 0.23**
6x6	Uniform	50.15 ± 0.09	**49.31 ± 0.07**	**48.86 ± 0.05**	**48.51 ± 0.04**	**48.09 ± 0.04**
	BestSeed	49.00 ± 3.71	33.00 ± 3.52	11.00 ± 2.09	2.50 ± 1.10	0.00 ± 0.00
	Nash	**50.94 ± 0.33**	47.81 ± 0.29	46.73 ± 0.22	45.13 ± 0.16	43.67 ± 0.16
7x7	Uniform	50.08 ± 0.06	**49.51 ± 0.07**	**49.03 ± 0.05**	**48.70 ± 0.05**	**48.36 ± 0.04**
	BestSeed	**55.50 ± 3.19**	24.50 ± 2.81	6.00 ± 1.64	1.00 ± 0.71	0.00 ± 0.00
	Nash	51.16 ± 0.32	48.40 ± 0.24	46.63 ± 0.18	45.13 ± 0.16	44.12 ± 0.14
8x8	Uniform	50.03 ± 0.07	**49.60 ± 0.06**	**49.07 ± 0.06**	**48.70 ± 0.05**	**48.34 ± 0.04**
	BestSeed	49.00 ± 3.50	25.00 ± 2.99	6.50 ± 1.84	0.50 ± 0.50	0.00 ± 0.00
	Nash	**50.91 ± 0.28**	48.89 ± 0.22	46.86 ± 0.19	45.65 ± 0.15	44.41 ± 0.16